WHITE PAPER–APRIL 2020

Principles of Designing and Developing an API Product

vmware®

Table of contents

Introduction

We are Caroline Hane-Weijman (engagement director), David Edwards (software engineer) and Francisco Hui (product designer)—a cross-functional product development team that worked together at VMware Tanzu Labs, the product development consultancy arm of VMware. At Tanzu Labs, we work with our clients as an integrated team, sitting side by side, to build and deploy digital products while enabling our clients to learn lean, user-centered, agile software practices that they can use as key capabilities on an ongoing basis within their organizations. (*Learn more* about the way we work).

The genesis of this white paper was an internal customer engagement report that we'd written. In this instance, many of our peers were referencing our report regularly and, as a result, we were encouraged to make this available to everyone. So here we are, sharing the experiences we encountered and the learnings we accumulated when designing and developing a product offering for a real company where the interface was a set of application programming interfaces (APIs) for external developers to consume to meet the business and user needs of their company.

One last thought: We believe that one of the reasons that our original report was somewhat popular was not so much due to the rise of microservices and the fact that APIs are the medium of communication between microservices, but rather the approach we took of bringing a balanced team to bear on this project. (*Learn more* about balanced teams.) Every word in API screams engineering, so "leave it to the engineers to figure it out" is the usual MO for these kinds of projects. At Tanzu Labs, we see things differently. Engineers are people, too, with their preferences, frustrations and needs. Applying lean principles and user outcomes the same way we do with every project is how we work. That means that, at the very minimum, a balanced team comprised of a product manager, a designer and an engineer is as essential on a front-end application as it is on a project where the end product is a set of APIs.

We hope you find this information useful.

Part 1: Resourcing and designing your API

Through APIs, we can access the power of billions of dollars of functionality built by others to radically accelerate our productivity, business expansion and customer offerings. API-driven development is understandably a primary focus for today's enterprises and start-ups to drive innovation and speed to market.

APIs define how software communicates with other software systems. This is similar to how user interfaces define how a user accesses the functionality of an application, but the end user for an API is a computer system, not a human. APIs create Lego blocks of software functionality. They can be used by developers internal to your company to leverage discrete, common functionality, as well as by developers external to your company to use for their own business and user needs. Often, development teams creating an API assume that because their system does not have a graphical interface, development won't require the kind of product and design practices you would expect to use to create an application for human users. In fact, that's not the case at all. When developing APIs, the same practices and tools for lean product development apply, but there are inherent differences that need to be considered for how you apply these practices.

Key takeaways

We'll start with the TL;DR. We'll elaborate on these and other learnings throughout this white paper:

- Treat product and design roles and their practices with as much significance as you would a product with a presentation layer. The same lean and user-centered design disciplines apply to ensure you are iteratively building a product valued by the business and users.
- Don't assume things take less time because you don't have a presentation layer. They don't.
- Consider all levels of users when defining features and designing the API experience. This includes the business and their users that the API features provide value to/for, the developers working for said business that are using your APIs, and the computer system that ultimately consumes your APIs.
- The ultimate consumer of your API will be a computer system, not a human. This means that some usual intuitions such as minimizing the number of steps in a workflow don't apply. It's more important to make the building blocks as intuitive as possible for developers to build this computer system.
- Define your versioning strategy carefully as developing APIs is like developing a language; changing APIs can break the computer system that is built on top of it.
- Consider creating either a client library for the API, or a dummy application that consumes the API, in parallel with the API itself. This allows you to create quicker feedback loops to test the usability of your APIs.

Importance of a balanced team

We've come across many teams working on APIs where staffing a product manager and designer has been deemed unnecessary. The same lean and user-centered design principles apply to products without a presentation layer as one with a user interface; API products are still solutions to business and end-user needs. A product manager and designer will help you validate the priority of the business problem you are solving and whether your API design is going to solve this problem in a desirable experience, while minimizing waste and keeping the team productive.

A product manager (PM) should lead the team to discover and deliver an API product that creates meaningful value for the business and users, with minimal waste using lean principles. For APIs, the set of businesses and users may also be very complex, and PMs will need to understand the needs and business impacts to facilitate decision-making in service of shipping successful features. As with all products, PMs should articulate the product vision and strategy, establish an outcome-oriented product roadmap that translates to a prioritized backlog, establish and track measurable objectives for your API usage, de-risk product direction, and help ship software. These are all in favor of effective communication and keeping the team productive and motivated.

Designers are too often pegged in a design agency role to make things pretty. However, designers contribute a set of capabilities far beyond visual design to develop a desirable, useful and usable experience.

These user-centered design (UCD) capabilities include conducting exploratory user research, ethnographics, pattern mapping, prototyping, conducting user validation sessions, mapping information architecture and so on. UCD activities are just as valuable for APIs as for other products with presentation layers, and designers play a critical role in focusing on these essential UCD capabilities for an API team.

Because the direct users of APIs are developers, we found the developers on our API team could easily slip into the habit of making decisions based on what they would want themselves. We quickly discovered that every engineer on our team had a different take on what a great RESTful API should be, and we were stuck in never-ending design discussions. Having a product manager and designer lead the team through objective processes of user exploratory interviews, design validation sessions and facilitated team discussions, we were able to move forward more quickly with more confidence that our decisions reflected our users' needs and not our own assumptions.

API design

Although an API benefits from the same UCD practices as any other system, the form and focal points of design are different. In designing a graphical user interface, your team would consider things such as visual language, consistency, information layout, intuitiveness of controls and usability. All of those have analogs in an API. Additionally, there are design considerations unique to the nature of APIs (such as versioning) and principles from graphical interface design that do not apply (such as minimizing the number of steps in a workflow).

We followed a *RESTful API* architecture. REST is an architecture style for designing networked applications that provides a convenient and consistent approach to requesting and modifying data. REST APIs use HTTP requests to GET (retrieve), PUT (update), POST (create) and DELETE data. It simplifies each call into a single-action Lego block. Although this approach has become standardized, we interestingly discovered that our team members had different ideas for how to design RESTful endpoints. We ended up having discussion fatigue after endless back and forth on what a proper RESTful solution would be. Based on our experience, we suggest quickly choosing an endpoint design principle, creating mocks, getting feedback from developers and then iterating, incorporating user feedback to break design deadlocks.

In summary, the key considerations of API design that determined the usability of our product were:

- Resources, requests and responses
- Endpoints
- Error responses
- Versioning

We will explain these in more detail. To help anchor the discussion, we'll use the following business use case: A product team working for a Payment Company is building an application that processes payments, developing features that easily allow online retailers (e.g., Sock Company) to take recurring payments from customers (Sock Buyer). When the Sock Buyer purchases a pair of socks online, they provide personal information and payment information to Sock Company. The APIs we developed communicate the relevant information from Sock Company's software system to Payment Company's software system to process the payment.

Resources, requests and responses

To communicate between software systems, an API essentially functions as a collection of questions/commands (requests) and answers (responses) that allow you to access the data (resources) to accomplish what you need. For example, Sock Company needs to interact with our API to do tasks such as collecting a consumer's payment information on sign-up and updating that consumer's subscription information so they can be billed on the appropriate interval. Our API would then include resources such as consumer, subscription and payment method.

It isn't always obvious what resources you should include in your API. Is it better for our API to have a subscription resource or a payment resource? Both? Is consumer or customer the better name for the person placing a subscription? Your resources make up the language available for users to ask questions and send commands, so you need that language to be clear, unambiguous and powerful enough to accomplish their needs.

When a user requests information about a resource, your API sends them a response. For example, if Sock Company sends us a request asking about one of their consumers, we might answer with this response:

```
{
  “username”: “johndoe”,
  “email”: “johndoe@example.com”,
  “emailConfirmed”: true,
  “subscribedOn”: “2019-01-01”
}
```

How much information do you put in a response? Does the same information occur in different kinds of responses? For example, the response above doesn't say anything about johndoe's registered payment methods. Should it, or should users have to ask about payment methods specifically?

There are costs and benefits to both, so the right answer will depend on your users' specific needs. Regardless, the responses your API sends comprise the information layout of your interface, so they need to be consistent, intuitive to your users and structured in a way that helps them accomplish their goals.

Key learnings when designing resources, requests and responses

- Consistency is key.
- When users can both create and later fetch a particular resource, responses should echo all the original inputs along with additional system-generated fields (such as its creation date, etc.).
- For future iterations, taking away fields is worse than adding fields, as it hampers backward compatibility. When in doubt, start with fewer fields.

Endpoints

An endpoint is the address to which users send requests to retrieve information. Most of the time, an endpoint corresponds to a particular resource. The way you design your endpoints also indicates the hierarchy and structure of your resources and data, similar to how you choose to organize folders and files in a file system. For example, Sock Company might ask for a list of John Doe's registered payment methods by sending a GET request to this endpoint:

https://our-payment-api.com/consumer/johndoe/payment-methods

The consumer resource is the top-level resource, and the payment method is nested under consumer/johndoe to access John Doe's payment information.

However, this isn't the only way to structure such an endpoint. You could instead provide an endpoint, where payment method is non-nested and is a top-level resource:

https://our-payment-api.com/payment-methods/johndoe

Here, users would indicate whose payment methods they want by providing the consumer's username as a separate parameter in the request, rather than making it part of the endpoint itself.

Again, there are costs and benefits to both approaches. Embedding information in the endpoint shapes how users think about the structure of your data by forcing them to think about certain resources in terms of others. If done well, this can help users build an effective mental model of your system—but in other situations, the constraints it introduces result in inconsistencies that hinder user understanding. In our project, we ended up using the non-nested approach, because our users often needed to query payment method data in different ways. (For instance, rather than getting all payment methods for a given consumer, they might need to get all confirmed payment methods regardless of who they belonged to.) The non-nested endpoint allowed us to accommodate all of these queries in a consistent way.

Just like a button on a webpage, it needs to be intuitive to a user what will happen when they send a particular request to a particular endpoint—and you shouldn't trust that your intuition matches that of your users. The only way to know for sure what a user expects an endpoint to do is to ask—hence the need for user research.

Key considerations for designing endpoints

- Top-level resources vs. nesting sub-level resources
- The number of nested resources for one endpoint
- Consistency in endpoint paths
- Whether the endpoint is intuitive, flexible and accessible

Error responses

Error responses arise when you've made a request where the receiver is unable to provide you with the response you're expecting. The error response includes an HTTP status code, which specifies the type of error, and the response body. Error response code design can become intricate as *different error codes* signify both different types of errors and different severity levels of each error. It's important to keep in mind that a computer system will be reading the error code initially, therefore the error code should signal the behavior of the system and how you wish to respond as a business. For example, is there something wrong with the system (urgent) or with the format of user input (not urgent and can be handled with messaging)? We opted to start with generic error code responses and return everything as a 400 ("Bad Request: The server cannot or will not process the request due to an apparent client error") except for authorization errors, which we specified as a 403. The intent was to gather feedback from developers and iterate on the granularity of the error response codes.

Versioning

Humans are adaptable—if you decide to roll out some changes to a user interface, as long as they're intuitive and clear, human users will figure out what to do in a minute or so. But humans are not the direct users of an API—computer systems are. And computer systems are not nearly that smart.

This means that in terms of versioning/product iteration, developing APIs is like developing a language. On the spectrum of developing a language (versioning extremely important) to developing an internal app for internal users (versioning less important), APIs are closer to that of developing a language.

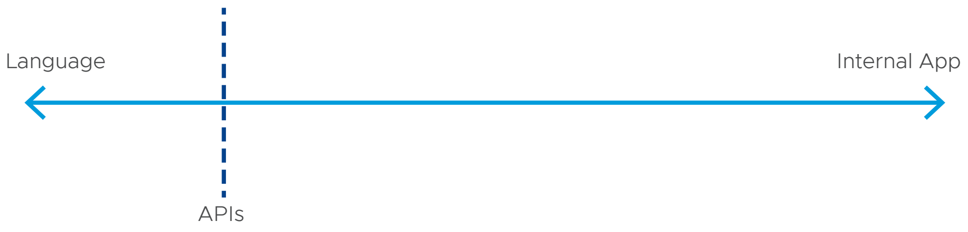


FIGURE 1: Spectrum of versioning complexity from harder to easier.

Developers will be using APIs like Lego blocks to build their own computer system. Changing a block in the structure they've built could break the structure.

For example, if you release a version of your API that includes an emailConfirmed field in the response for customers, one of your users might build a system that pulls that field from that response and uses it for some logic. If you then decide to put that field in a different response and remove it, your user's system will continue looking for it in the original location, and break when it isn't there.

On the other hand, systems don't usually break if there are fields in a response besides the ones they use, so adding fields (starting to provide something that, worst case, no one uses) poses a lower risk for breaking changes than removing fields (taking away something that a developer might be relying on).

In conclusion, start with less and add more as feedback dictates. Collapsing two endpoints is easier than breaking apart a single endpoint, and adding fields from responses is easier than removing fields. Start with fewer fields in responses and highly focused endpoints.

Putting it all together for a usable API

Just like any other product, making the product easy to use is a key consideration of design. But because the direct user of your API will be a computer and not a person, the nature of usability is a little different.

For example, consider this workflow: A customer visits the Sock Company website. They select socks, check out, create an account, save their profile and payment information, and then click Buy.

The API calls involved in this workflow likely do not correspond directly to the customer's actions; Sock Company developers may use multiple API calls to create a profile, save payment information and trigger a transaction with the payment API business. That's fine.

The developer is not the one clicking on every step for every customer—they build a system that automates this. Therefore, although you might try to minimize the number of clicks needed for a particular workflow in a human-driven interface, that intuition doesn't necessarily apply to an API—having to make multiple calls is not necessarily slower. It's more important to make the building blocks as intuitive as possible to help developers build their automated system.

Overall, developers agreed that focused, single-action, intuitive, consistently designed endpoints were desired. As previously mentioned, APIs are like Lego blocks for developers. Small, focused endpoints give developers more flexibility for how they want to design their own system/interface. We originally had an endpoint that consolidated creating a consumer profile, payment, subscription and triggering a transaction all in one call. This was poorly received. So we broke each action up into separate endpoints.

Part 2: The process of designing APIs

Discovery and framing: User research

We typically begin a product engagement with the discovery and framing (D&F) process: a set of activities that helps a team understand the users and evaluate possible solutions, and minimizes product and development risk for the first release. We typically begin by exploring the problem space through user and stakeholder interviews, and market research to validate and narrow down the problem space for the first release (discovery phase). We ideate and prototype our ideas for that first release, test those prototypes with real users, and start development when we have a higher level of confidence in the initial set of features (framing phase).

This process requires a balanced team of a product manager, product designer and engineer so that the problems and solutions are considered through the lens of the business priorities, user needs and technical feasibility. Including the three disciplines from the start of the project allows the team to deeply understand the problem space and build a solution that works.

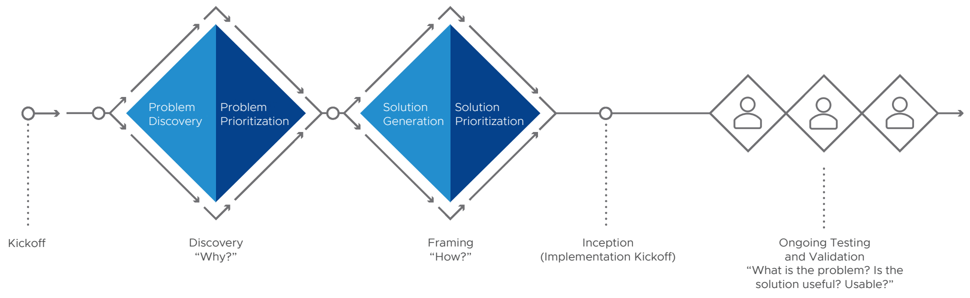

Problem Discovery
Problem Prioritization
Solution Generation
Solution Prioritization
Kickoff
Discovery "Why?"
Framing "How?"
Inception (Implementation Kickoff)
Ongoing Testing and Validation "What is the problem? Is the solution useful? Usable?"

FIGURE 2: The D&F process.

Identifying users and approach

As in Part 1, we'll use a hypothetical business case to help anchor the discussion: A product team working for a Payment Company is building an application that processes payments, developing features that allow an online retailer (e.g., Sock Company) to take payments from a customer (Sock Buyer). When the Sock Buyer purchases a pair of socks online, they provide personal information and payment information to Sock Company. The APIs we developed communicate the relevant information from Sock Company's software system to Payment Company's software system to process the payment.

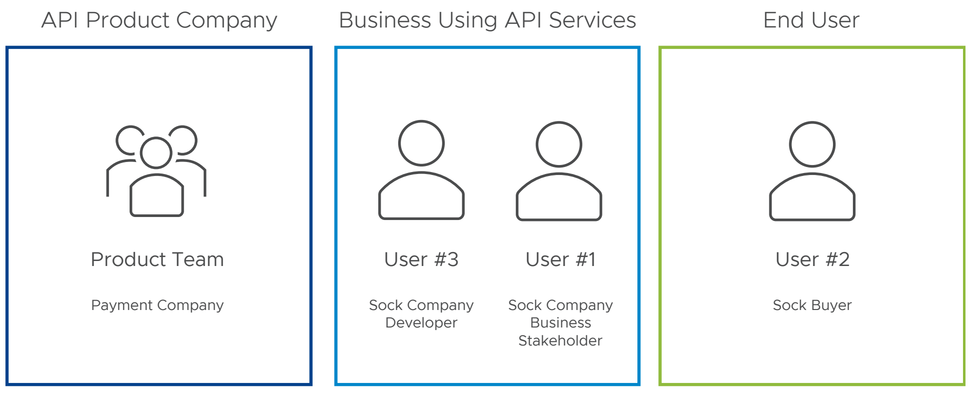


FIGURE 3: User and stakeholder map for example API use case.

The functionality that we developed as a team was defined by the needs of the business and its end customers. However, the direct users of our API are the developers working for said business. We needed to first define feature priorities by solving for the business user and end-user needs (e.g., online Sock Company business and Sock Buyer needs). Then, we needed to consider the developers consuming the APIs to define the API design and experience (e.g., developers working for online Sock Company). There were therefore three sets of users that defined the feature priority and experience of our API.

We recommend approaching the D&F process in two phases to address our multiple users' needs. Assuming a three-week D&F:

- Weeks 1–2 – Define features and priorities based on exploring the needs of the business consuming the APIs (User #1) and its end customers (User #2). Recommend a balanced team of a PM, a designer and an engineer.
  - Conduct stakeholder and business user interviews to identify the business priorities, needs and constraints
  - Develop user personas that represent groups of business users to illustrate the users' motivations and needs
  - Develop service blueprints and workflow diagrams for target user personas
  - Identify features and prioritize based on target user personas
- Weeks 2–3 – Design the API experience based on exploring the needs of developers working for this business (User #3), and do technical discovery and setup. Recommend an expanded balanced team of a PM, a designer and two to three engineers.
  - Interview developers (engineers on the product team should be part of these interviews)
  - Based on features and developer interviews, create API mocks of endpoints with requests and responses (using a tool such as Stoplight with *Postman*)
  - Validate initial mockups with developers

At the end of D&F, the team can then review both feature prioritization and mockups of APIs.

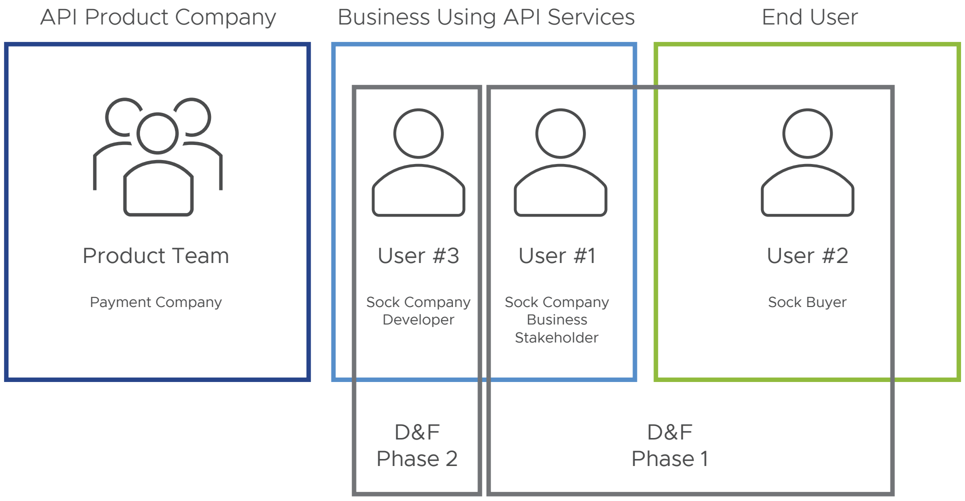


FIGURE 4: Target user groups for D&F phases 1 and 2.

Phase 1: Understanding product needs

For the initial phase of D&F, we primarily interviewed business users (e.g., head of product and pricing at Sock Company) that would be paying for our functionality and could represent the needs of their own customers. Similar to how we create personas of end users, these stakeholder interviews helped us create pseudo-personas of business segments to understand their needs and the opportunity they represented for our own business.

Once we understood who the primary user was, creating workflow diagrams helped us understand how an action starts from the customer, touches the business and reaches our payment API. We used this to narrow down the initial feature set based on capturing the most common business use cases, feasibility and ability to capture the market. For example, one prioritized feature was the ability to take fixed amount payments every month, as that served the biggest use case for businesses. Another feature that followed was the ability to offer customers installment plans for their purchases.

Once we had a list of features, we stack ranked these features with a card-sorting exercise and identified our first set of release markers that would allow us to quickly deliver value and learn from our beta users.

Phase 2: Designing a usable experience

For the second phase of D&F, we interviewed eight developers. These exploratory interviews were useful to decide what we knew and needed to validate, or didn't know enough about and wanted to understand. The output from these interviews gave us a set of guidelines that developers use to evaluate APIs in the same domain space, how they think about solving business problems, and their mental model for defining the objects and resources needed to build specific features. We used all of the insights from these interviews to generate ideas for how to structure our endpoints.

We did a combination of exercises during the developer interviews:

- Conducted exploratory interviews:
  - Focused on recent behavior and interaction with other APIs in a similar domain (e.g., how do they evaluate, compare or use an API?)
- Prompted developers with a business scenario and asked them to write down:
  - Resources they anticipated existing/being created.
  - Expected endpoints based on specific actions. We recommend also doing the reverse: Show mocked endpoints and ask developers, "What do you expect to happen with this endpoint?"

See *Appendix I* for exploratory interview questions.

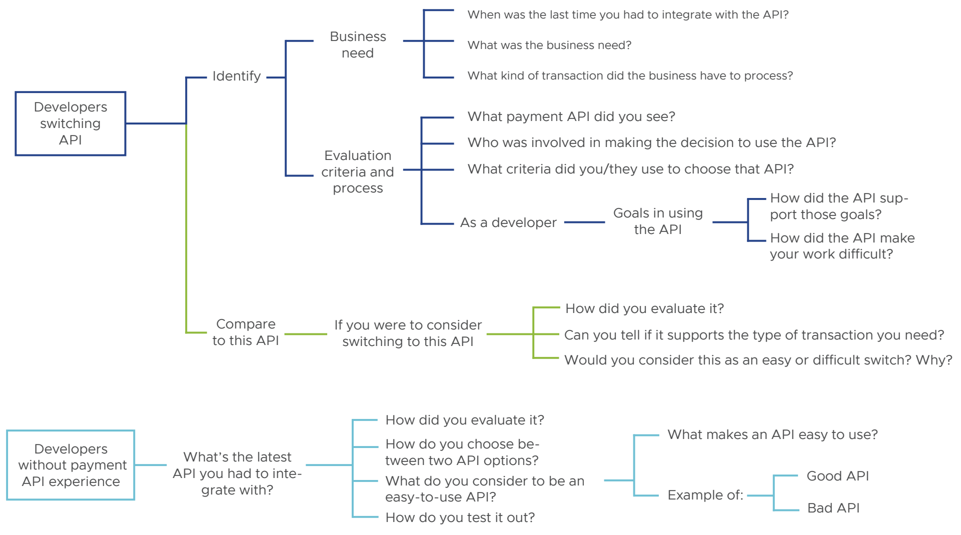

Developers switching API
Identify
Business need
When was the last time you had to integrate with the API?
What was the business need?
What kind of transaction did the business have to process?
Evaluation criteria and process
What payment API did you see?
Who was involved in making the decision to use the API?
What criteria did you/they use to choose that API?
As a developer
Goals in using the API
How did the API support those goals?
How did the API make your work difficult?
Compare to this API
If you were to consider switching to this API
How did you evaluate it?
Can you tell if it supports the type of transaction you need?
Would you consider this as an easy or difficult switch? Why?
Developers without payment API experience
What's the latest API you had to integrate with?
How did you evaluate it?
How do you choose between two API options?
What do you consider to be an easy-to-use API?
How do you test it out?
What makes an API easy to use?
Example of:
Good API
Bad API

FIGURE 5: Questions for exploratory user interviews with developers on their API experiences.

Once we had completed this round, the result was five sets of endpoint design principles that we dot-voted on to pick our direction:

1. Nested resources: consistent pattern, only consumer top-level resources
2. Nested resources: consistent pattern, both consumer and subscriptions top-level resources
3. Nested resources: inconsistent patterns
4. Non-nested/all top-level resources
5. Non-nested/all top-level resources, except for payment method, which was nested under consumer

Refer to Part 1 of this white paper for details on nested, non-nested and top-level resources design principles.

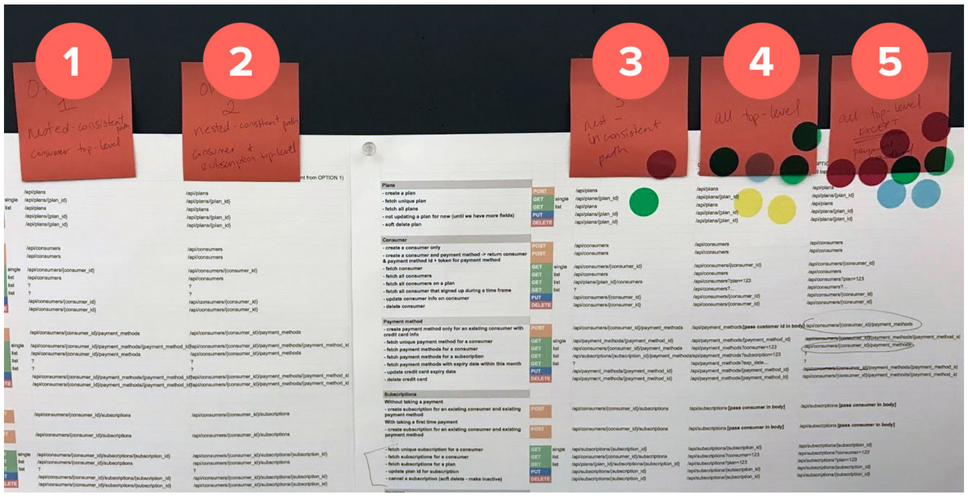


FIGURE 6: Material we used for a dot-voting exercise to align as a team on the initial API endpoint design.

Validating solutions

Once we had a direction for how to design the endpoints, we conducted another set of user interviews, this time to validate that our API designs were intuitive and matched their mental model. We used a similar task of showing the developers an endpoint, asked them what they expected to happen, showed them our mock response and observed their reactions. This allowed us to identify any outstanding pain points and see how usable our solution was to them. Although there isn't a presentation layer to the interface, the endpoints and responses are a representation of expectations, and user testing the task gave us the confidence to move forward with our solution.

Key artifacts

While all projects are different, we recommend considering these key activities as you design your API:

- Create developer and company personas – This helped us identify the right segment of users.
- Map user journeys and pain points – This helped us define a criterion to generate and evaluate solutions.
- Identify scenarios and use cases – This helped us define tasks.
- Prioritize features with a card-sorting exercise – This helped us define the backlog.
- Interview developers for their API development experience – This helped us define our API design.
- Create prototypes of APIs and conduct usability tests – This helped us test assumptions before building.

In this section, we covered the key methods you can use to identify and illustrate the conditions in which your solutions are evaluated. These UCD methods allow the team to have a deeper understanding of how to build a solution so that it has a higher chance of meeting the business and user needs.

Part 3: Designing and developing an API product

There were many things we learned about the similarities and differences of managing our product backlog for an API product in comparison to one with a graphical interface to help us be more effective. We iterated on our model of writing user stories and doing acceptance based on a lot of feedback from the engineering team to do what worked best for us.

Key takeaways

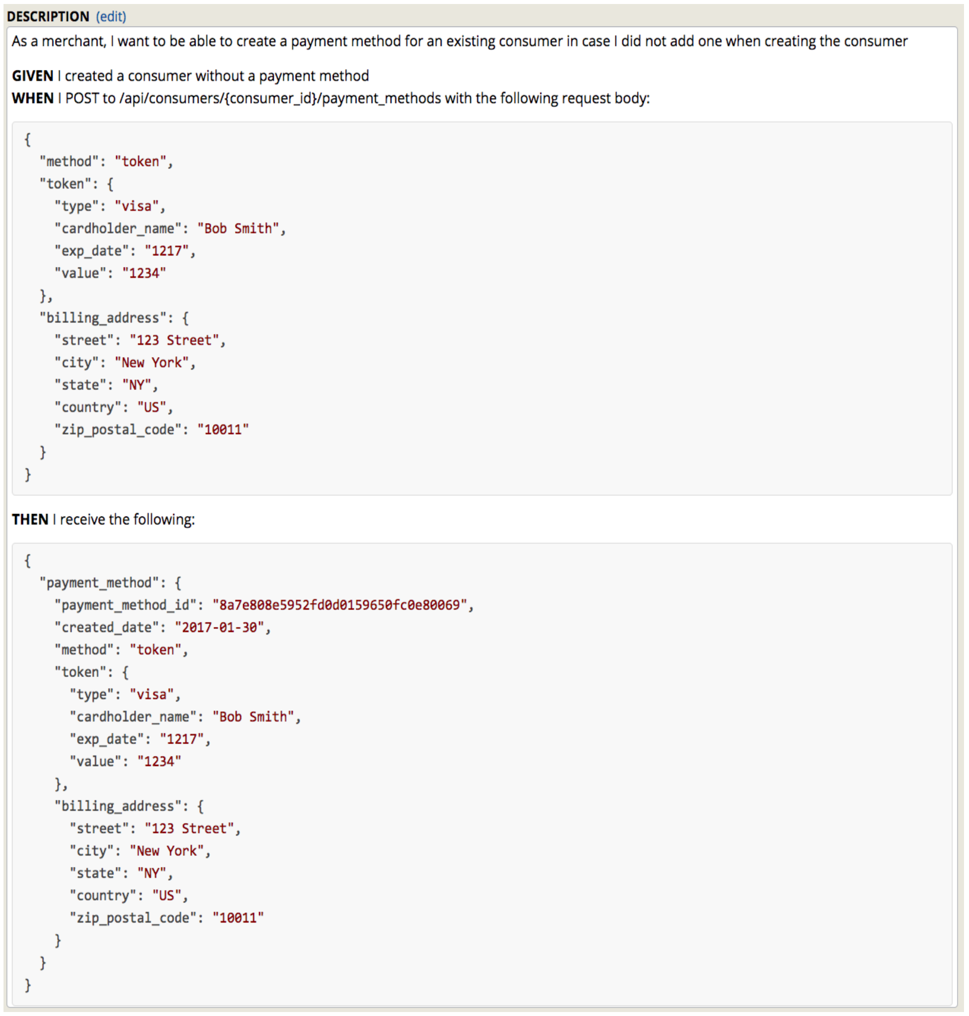
Some of the things we learned were:

- There is a fine line between product/user experience design decisions and implementation details for API products. It was best to start with a higher level of abstraction and then iterate with the team to strike the right balance between what and how.
- Providing exact API requests and responses for each user story, with example data, was extremely valuable for the team to be more productive and consistent once the API design direction was clearer.
- As with most backlogs, small user stories were key. We recommend treating each new field in a request/response as a new user story, and creating separate stories for each field validation and each type of error response.
- We recommend using an API design tool, such as Postman, to simplify acceptance of user stories.
- What is true for projects with graphical interfaces is also true for APIs: Acceptance criteria should always be from the API consumer's perspective.
- We recommend creating either a client library for the API, or a dummy application that consumes the API, to more effectively test your features.

Backlog management

There is a fine line between product/user experience design decisions and implementation details for API products. Our team had frequent discussions regarding what should be defined in the user story by the PM and what should be left as an implementation detail for the engineering team. We found it best to start with a higher level of abstraction, focusing on the what and why, and iterating on how much of the how needed to be defined. Be patient in your iteration/sprint planning meetings—when you review the upcoming backlog as a team—to figure out the right level of implementation details that works for you.

Providing exact API requests and responses, with example data, for each user story was highly valuable for the team. This ensured consistency and saved time for our developers.



DESCRIPTION (edit)

As a merchant, I want to be able to create a payment method for an existing consumer in case I did not add one when creating the consumer

GIVEN I created a consumer without a payment method
WHEN I POST to /api/consumers/{consumer_id}/payment_methods with the following request body:

```
{
  "method": "token",
  "token": {
    "type": "visa",
    "cardholder_name": "Bob Smith",
    "exp_date": "1217",
    "value": "1234"
  },
  "billing_address": {
    "street": "123 Street",
    "city": "New York",
    "state": "NY",
    "country": "US",
    "zip_postal_code": "10011"
  }
}
```

THEN I receive the following:

```
{
  "payment_method": {
    "payment_method_id": "8a7e808e5952fd0d0159650fc0e80069",
    "created_date": "2017-01-30",
    "method": "token",
    "token": {
      "type": "visa",
      "cardholder_name": "Bob Smith",
      "exp_date": "1217",
      "value": "1234"
    },
    "billing_address": {
      "street": "123 Street",
      "city": "New York",
      "state": "NY",
      "country": "US",
      "zip_postal_code": "10011"
    }
  }
}
```

FIGURE 7: API user story example.

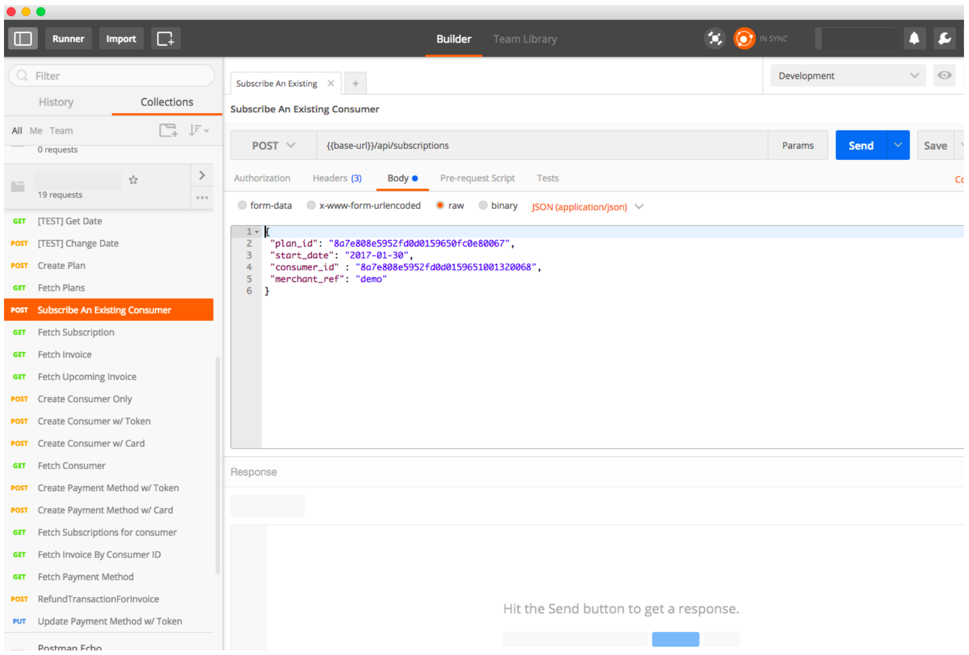
That said, the first set of user stories were intentionally more high level, with just a bulleted list of fields. This was because we did not have a clear sense of the request/response design, and we did not want to dictate how the engineers should design the objects upfront. After the first few stories, the design stabilized, and we felt more comfortable transitioning to exact JSON requests/responses. For less-technical PMs, using a *JSON-formatting tool* can be very helpful.

As with most backlogs, we advocate for small user stories. We suggest treating each new field in a request/response as a new user story. This made it easier for our developers to keep their tests focused and to reprioritize on a more granular level. In particular, we strongly recommend creating separate stories for each field validation and each type of error response. We experienced the pain of trying to combine these into the original user story for creating a field or endpoint. By splitting these out, the team was more easily able to deprioritize certain complex validations and error responses. For example, we chose to deprioritize authorization error responses (403s) because we only had one beta user for the initial release and wanted to get feedback as early as possible. Focusing on smaller user stories allowed us to ship the first set of features to this beta user much more quickly.

Acceptance

It's important to write acceptance criteria from the consuming system's point of view. We had requirements that some other system or service be called, and it is often tempting to drop these requirements into the acceptance criteria verbatim (i.e., then a request is sent to the email microservice). This can make stories impossible to accept, because back-end calls cannot be observed. Instead, we had the acceptance criteria describe the literal process by which the PM would accept the story (i.e., then I receive an email from the email microservice about the charge to my account). This also helped highlight stories that cannot be accepted, and drew early attention to test utilities that we wanted to build into our acceptance environment. For example, suppose the email microservice is only accessible in staging and prod, not in the acceptance environment. We built a fake email service that exposed the emails sent through an API instead of sending actual emails, so that the PM's "I receive an email from the email microservice" step could consist of asking the fake email service what emails were sent.

We used a human interface for accepting our API stories. *Postman* is a user-friendly, web-based API design tool. On our API project, it served as a front end for the purpose of acceptance. Alternatives include *Advanced REST Client* and *Insomnia*. We saved our services and endpoints to be able to easily access them for acceptance and demos.



GET	[TEST] Get Date
POST	[TEST] Change Date
POST	Create Plan
GET	Fetch Plans
POST	Subscribe An Existing Consumer
GET	Fetch Subscription
GET	Fetch Invoice
GET	Fetch Upcoming Invoice
POST	Create Consumer Only
POST	Create Consumer w/ Token
POST	Create Consumer w/ Card
GET	Fetch Consumer
POST	Create Payment Method w/ Token
POST	Create Payment Method w/ Card
GET	Fetch Subscriptions for consumer
GET	Fetch Invoice By Consumer ID
GET	Fetch Payment Method
POST	RefundTransactionForInvoice
PUT	Update Payment Method w/ Token

Subscribe An Existing Consumer

POST {{base-url}}/api/subscriptions

Authorization | Headers (3) | Body | Pre-request Script | Tests

form-data | x-www-form-urlencoded | [✓ raw] | binary | JSON (application/json)

```
{
 "plan_id": "8a7e808e5952fd0d0159650fc0e80067",
 "start_date": "2017-01-30",
 "consumer_id" : "8a7e808e5952fd0d0159651001320068",
 "merchant_ref": "demo"
}
```

FIGURE 8: Our APIs saved in the Postman tool for testing and acceptance.

Although we didn't do this on our project, we recommend creating either a client library for the API or a dummy application that consumes the API, in parallel with the API itself. This would have let us dogfood our own system. As it was, our intuitions and assessments of our design decisions were often informed by what the API was like to use through Postman, which isn't a lifelike use case.

Between the two—client library or dummy consumer app—the client library probably wouldn't have been the best for our use case. Our consumers were going to be working in a variety of different languages, and the client library we built would only support one of them.

On the other hand, if all the consumers of the API were going to be using the same language (for example, if the API was to be used internally in a company that did all their development in Java), then the client library could actually have served as the front end of the system, and we could have treated the API itself as an internal detail.

Part 4: Developing, architecting, testing and documenting your API

So far, we've talked about what developing an API can be like for the product management and design practices, and the critical value those practices lend to the process. In this final installment, we want to talk about our learners in the other corner of the balanced team triangle: engineering.

Engineering is fundamentally the art and science of weighing trade-offs. When developing an API, all of the principles and concerns you've learned building other kinds of products are still in play, but some of the scales tip differently when you weigh the costs and benefits of different techniques. Certain kinds of tests become much cheaper, certain architectural concerns become much more costly, and the cost of having poor documentation skyrockets compared to products whose direct users are humans.

Testing

Because APIs are built to be consumed by other programs, it's especially easy to write feature tests for them.

By feature test, we mean a test that exercises the application through the same interface as an end user. These tests usually bring a large slice of the system online for the test, but they aren't necessarily end to end. For a web application, this means a browser-driven test using a tool such as Selenium or Capybara.

Because feature tests are written in terms of actions the user takes, they often serve as a useful reference point for conversations between developers, designers and PMs. However, browser-based feature tests are expensive in a number of ways. Having to drive a browser usually makes them quite slow, and they tend to develop nondeterministic or brittle behavior over time. Getting the necessary browser and browser driver on your CI system can be another source of pain.

For APIs, the costs of feature tests are greatly reduced. The system is designed to be consumed by other computer programs—and isn't your test suite just a computer program consuming the API? This means that writing a feature test against an API likely won't be brittle, won't require any special tool such as a browser driver and will be fairly fast.

This difference in trade-offs meant that, on our project, we leaned on feature tests more than we would have for a browser-based product. We captured the happy path of each feature in a test that exercised the system through the API, the same way the PM would when accepting that feature. Engineers used the story's acceptance criteria more or less verbatim to write tests with something like *REST Assured* (which we used) or *MockMvc*.

But we took it further. We designed every feature test to be runnable against either a local instance of the application or against a deployed instance (for example, our acceptance environment). This meant that our CI system could deploy a new artifact out to an environment, and then immediately run the feature test suite against the new deployment to ensure all its features (including the brand-new ones) were behaving well. With just a quick glance at our CI monitor, the PM could see that all features were working in a given environment, which indicated whether it was safe to do acceptance or a demo there.

To make every feature testable we have to build some test utilities in nonprod environments. For example, we added a test-only API endpoint that allowed the caller to change the system's understanding of the current date. This allowed us to write tests such as the following:

- Given I have signed up for a monthly subscription
- After a month goes by
- Then my primary payment method gets billed

Rather than writing a feature test with a Thread.sleep(ONE_MONTH) line, we could hit the test endpoint to jump into the future. But those utilities weren't just useful for our automated tests. Our PM had the same needs as our feature tests and used the test utilities to immediately accept stories that otherwise would have required waiting a month to see the desired behavior.

The fact that we could run our feature test suite against our deployed test environments gave us a lot of confidence that each environment was behaving well. Whenever the product managers encountered behavior they didn't expect, our first step was to open the relevant test and compare it to what they had done, anchoring the discussion in concrete expectations. If you can't feature test it, the product managers can't accept it.

Using feature tests in this way, and designing them to run both locally and against deployed environments, isn't actually specific to an API product. You can do the same thing with web apps—the tests are just more costly because a browser driver is involved, which means you'll probably want to invest in other less-costly techniques instead.

In terms of testing techniques that are API-specific, we did experiment with a couple of tools for validating the JSON schema of API responses. Eventually, we settled on a method that was simple to keep up to date and had the added bonus of adding generated examples to our API documentation. (See the API documentation section.)

System design

The following concepts are not specific to API projects, but we wanted to highlight their importance in the context of APIs.

Ports and adapters

The presentation layer (whether it's a browser-based UI or an API) doesn't have to affect the underlying system. Using a loose-coupling architecture such as the ports and adapters pattern (sometimes referred to as hexagonal architecture) is one way to prevent changes in other parts of the system from rippling up to the front end, or vice versa. This is desirable in any kind of system, but especially so in an API because making changes to the user interface is a much bigger deal.

If your internal modeling is coupled to how you present information to your callers, then the necessity of keeping the API stable can prevent you from refactoring and improving the internal design of your system—and that's a recipe for disaster. You need to be sure that you can quickly iterate and modify your system internals while keeping the exposed interface constant.

Separate data classes

Because an API is often simply serving JSON objects, which may correspond to service-layer objects and persistence-layer objects (e.g., SQL tables), it may be tempting to use a single class to represent all of these objects. However, we learned (the hard way) that these data models change at different times for different reasons. Having dedicated data classes to represent your API request and response bodies, and not using those classes directly in other layers of the system, can make it easier to iterate on system internals without changing the visible API. Over time, we evolved our system to a place where the only point of coupling between how we supplied resources to our callers and how we modeled those resources internally was a simple (and easily testable) converter class.

As a quick note: In our experience, this concept does not compromise well. For a time, we had a few data classes that were shared by some parts of the system, but duplicated by others. This was very confusing and might have actually been more damaging than if we had gone with a completely coupled system.

It does mean more work upfront—work whose value is not immediately obvious—but it will be worth it later on.

Dumb controllers

We aimed to keep as little logic as possible in our controllers. Business logic of any kind was restricted to the inner service layer. The controller was responsible only for translating API-layer data objects to service-layer objects and handling the bare minimum of validation (that is, validating that the response was well-formed enough that we could call the service layer, but not any business-rule validation, such as "dates must be in the future.") This also allowed us to iterate quickly on the request and response bodies.

Our system ended up being shaped roughly like Figure 9 (each shape represents a component of the system and contains a nonexhaustive sample of the sorts of things that lived there).

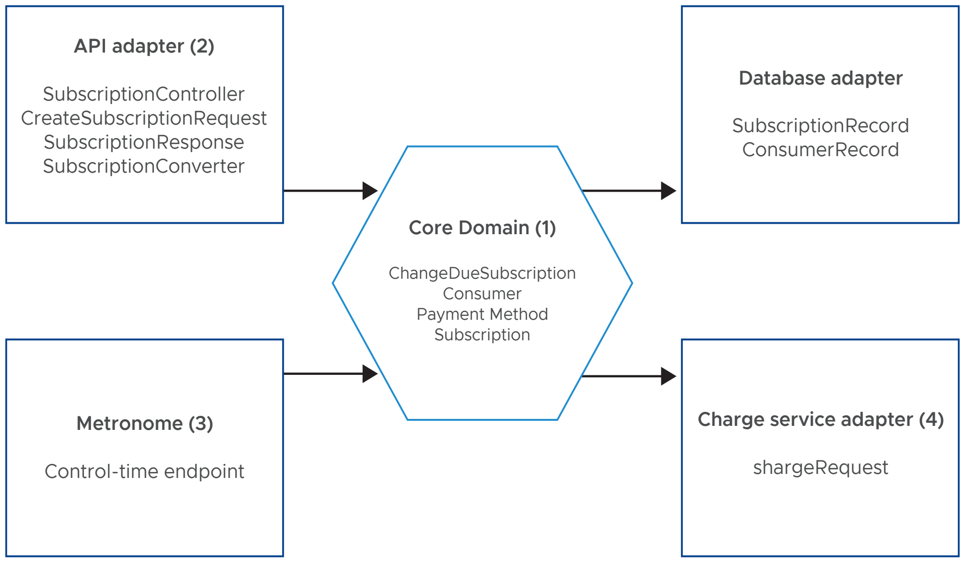


FIGURE 9: Shape of our system.

1. Our core domain component was where we did all the complex modeling and rule handling of our payment processing—stuff that had nothing to do with the fact that we were an API. This component contained use-case objects such as ChargeDueSubscription, which you could invoke to check for any due subscriptions and charge them appropriately, and model objects such as Consumer and Subscription, which were optimized to work nicely with the way we modeled our business rules.
2. Our API adapter layer defined the endpoints of our API (in classes such as the SubscriptionController), including the expected structure of requests to those endpoints and the structure of their responses. Importantly, we didn't reuse our Consumer and Subscription model objects as the definition of the request and response structure. Instead, we used separate classes, such as CreateSubscriptionRequest and SubscriptionResponse. Converter classes, such as SubscriptionConverter, translated CreateSubscriptionRequest into the appropriate Subscription objects in a cleanly testable way.

   This made it cheap to refactor our underlying domain model while still keeping our API stable. The converter classes were malleable, and their tests clear and simple to update, so we didn't have to stress about backward compatibility when we identified an improvement we wanted to make in the model.

3. The metronome component periodically invoked our core domain's ChargeDueSubscription use case with the current time. This was the component where we added our control-time endpoint, by which our test suite and product manager could jump the system's understanding of the current time into the future to ensure that various features worked appropriately.
4. External integrations were carefully tucked away from the rest of the system. For example, once we determined that payment needed to be made, we invoked an external service to process the payment. This integration was strictly encapsulated to prevent changes in the charge service's API from forcing us to make backward-incompatible changes to our own API.

API documentation

API documentation is a critical component of creating a good user experience for developers consuming APIs. We landed on a solution that allowed us to automatically generate documentation based on tests, and also input human text and diagrams. This meant that the heart of the documentation never went stale (it was always updated on every deployment), while our product manager could also easily contribute to the documentation (by committing to our version control repository).

We used *Spring REST Docs* to generate API documentation from certain feature tests. It integrates with REST Assured or MockMvc and generates snippets of AsciiDoc that can be incorporated into a larger document. We served this as part of our application, much like what Swagger UI does. The advantage of this over Swagger UI is that the documentation is guaranteed to be accurate by merit of being test output. If a field is present in the request or response but not in the documentation—or vice versa—it will result in a test failure.

One big advantage of the snippet generation (and AsciidoctorJ's include functionality) is that you can include snippets in a human-written document. The PM wrote an introduction and overview of our system, including a sequence flow diagram, and then included the snippets at the end. The result was an easily navigable, readable and complete API document with provably accurate examples.

Pro tip for the PM writing the intro context: Download a text editor such as *Atom* with AsciiDoc tools (asciidoc-preview and language-asciidoc) to easily preview the text as you are writing documentation. Then copy and paste the text into the repository to commit the changes.

Wrap-up

We hope this provides some tactical tips as a product development team developing APIs. There are many resources now available that provide best practices (explore *Google's sites*), and tools are evolving. If you have questions and/or feedback, don't hesitate to reach out.

The authors

Caroline Hane-Weijman, engagement director

As an engagement director, east area lead for Tanzu Labs, Caroline is responsible for leading customer success for our most strategic financial accounts in the company, partnering with our clients across our product and service portfolio to drive toward business and technology transformation outcomes. Prior to that, Caroline co-led the product management team at Tanzu Labs New York. Prior to Tanzu Labs, Caroline served as product manager at the fintech start-up LearnVest, during which they were acquired by Northwestern Mutual, and as a business management consultant at McKinsey and Company, working with Fortune 500 companies across industries and functions. Caroline has a mechanical engineering degree from the Massachusetts Institute of Technology, with a focus and strong passion for international development.

linkedin.com/in/chaneweijman/

medium.com/@cmhw

David Edwards, software engineer

David has worked with a wide range of clients and systems in his seven years with Tanzu Labs, from Rails apps for scrappy start-ups to trading platforms for one of the world's largest banks. He attributes most of his skill as an engineer and consultant to his *background in endangered language revitalization*, which is far more similar to agile software consulting than you think. He *spoke at SpringOne* in 2018.

Francisco Hui, product designer

Francisco has spent the last 10 years applying user-centered design to building products and design practices. Over the last couple of years, he has focused on enabling organizations, teams and individuals as a product design manager at Teachers Pay Teachers and at Tanzu Labs New York. He writes about continuous learning and the future of work on FranciscoHui.com.

twitter.com/franciscohui

Appendix

I: Usability research plan

Value of research

1. Gather additional validation for open questions, where the team doesn't have a strong opinion.
2. Use additional insights to inform decision/directions.
3. Apply a user-centric approach to gather information.

Goals for developer interviews

- Learn/validate/invalidate:
  - A good API experience for developers
  - Context in which they're consuming APIs
- Decide how much information to include in our API responses.
- What kind of endpoints should we have?
  - Which endpoints can be consolidated?
  - Which endpoints belong together vs. separated?
- How do merchants and developers organize and trace data and objects?
- How do they receive notifications for transactions?
  - Notifications in real time/one at a time vs. scheduled in bulk?
  - Webhooks vs. xml/text/csv files?
- What is the process for financial reconciliation?
  - How do they reconcile (e.g., reports, notifications, what is the process)?
  - What data elements are necessary for reconciliation?

Areas of focus

- All vs. individual ID for each object
- Do we need upcoming invoice request for MVP?
- When requesting a refund for a transaction on an invoice, should we specify the transaction ID or can we just use the invoice ID?
- Do we need to support fetching all invoices for a consumer more than is this an endpoint? How does a dev accomplish this?

Define types of users to interview

- 3–5 developers that have integrated with an API to accept payments
- Gift card for 45-minute interview
- Create screening form:
  - Have you used a payment API before?
  - Which ones?
  - What did you use the API for?
  - Name/email
  - Available to talk this week

Interview questions for developers with payment API experience

1. Q&A: Description of past payment API experience (20 minutes)

- Describe past use cases:
  - What payment APIs have you used?
  - What kind of transaction did you use it for?
  - What systems did you integrate the API with? Before: CRM, billing; after: accounting, finance
    1. What systems need to know 'why'?
    2. What was the process for financial reconciliation?
    3. What was the process for integration and use?
       - Setup experience
       - Maintenance
       - Ongoing interactions
  - Why did you or your company choose these APIs?
    1. Which other APIs did you consider?
    2. Which criteria did you use to chose? Recommendation from current client/friend industry reviews
       - Did you look at the documentation?
       - How did that inform your choice?
       - What are you looking for? (Endpoints? Features?)
- Which payment API was easiest to use?
  - What made it easy to use?
- Bad payment API experience?
  - Why?
  - Have you experienced changes in the APIs? Versioning?
- How have you used documentation when integrating with/using APIs?
  - How has it affected your experience?
  - What makes for good documentation?
  - What makes for bad documentation?
- When you used the payment API, how were you made aware of a transaction being made?
  - What data did you need to see? Why?
  - Ways of getting data?
    1. Notifications in real time /one at a time vs. scheduled in bulk
    2. Webhooks vs. xml/text/csv files
    3. Reports

2. Scenario: Developer response to example use cases (20 minutes)

You're a developer at Netflix. Bob Jones, who has never used Netflix before, wants to sign up for the Standard $9.99 monthly service. He picks the service, enters his name, billing/shipping address, email and credit card information. He will be billed starting today (October 6) as soon as he signs up and will then be automatically billed monthly (November 6, December 6, etc.).

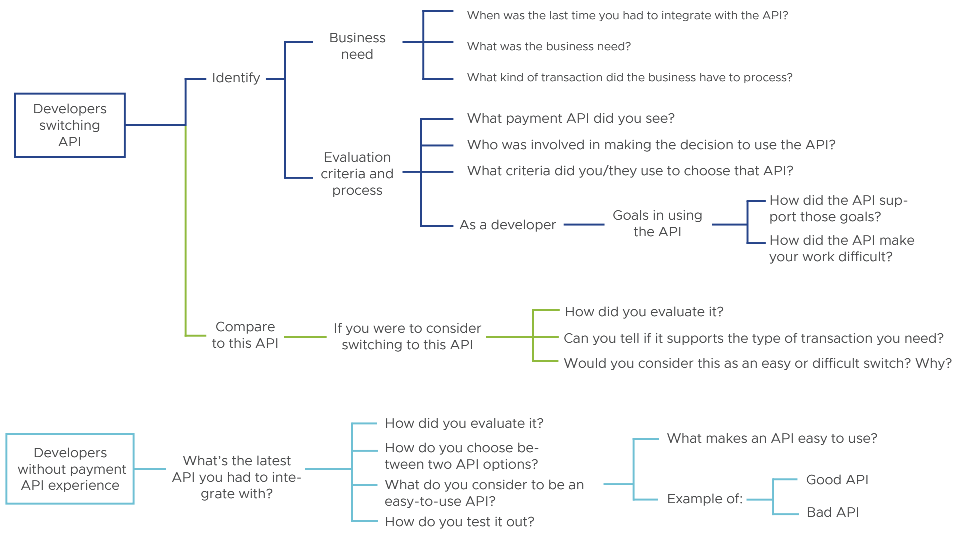
Prompts

- What objects would you want to create in the database to capture the information, charge the customer automatically, and know what charges will be upcoming for services and what the status is?
  - Option 1: Developer names objects most intuitive to them and organizes them
  - Option 2: Developer picks from a stack of labels for each object (i.e., plan, subscription, membership, tier) and organizes
  - Option 3: Developer only organizes using chosen objects
- Will you ever create a consumer without subscribing them?
- What do you expect to happen when you subscribe a new consumer with payment method and subscription info, and the latter two fail? If immediate payment fails?
- In the scenario that Bob Jones signs up and he provides all of his consumer and payment information and purchases: You want to send that information to the API to create a consumer object, a subscription object and a payment method, and get back a token value for subsequent payments.
  - What type of endpoint would you expect to use for this action?
  - What type of information would you expect to give to the API?
  - What type of information do you expect to be included in the response?

3. Validating an API design (5 minutes)

- Pull up APIs in Postman, react to working endpoints.

Appendix II. Usability research plan diagram


Developers switching API
Identify
Business need
When was the last time you had to integrate with the API?
What was the business need?
What kind of transaction did the business have to process?
Evaluation criteria and process
What payment API did you see?
Who was involved in making the decision to use the API?
What criteria did you/they use to choose that API?
As a developer
Goals in using the API
How did the API support those goals?
How did the API make your work difficult?
Compare to this API
If you were to consider switching to this API
How did you evaluate it?
Can you tell if it supports the type of transaction you need?
Would you consider this as an easy or difficult switch? Why?
Developers without payment API experience
What's the latest API you had to integrate with?
How did you evaluate it?
How do you choose between two API options?
What do you consider to be an easy-to-use API?
How do you test it out?
What makes an API easy to use?
Example of:
Good API
Bad API