White Paper: **January 2022**

Introduction to API Product Management

A guide for product managers working on their first API product.

vmware®

Table of contents

What is an API?

An Application Programming Interface (API) is a tool that takes an input (the request) and gives an output (the response). APIs are incredibly common in modern software development. You may be most familiar with public APIs, such as those provided by Google Maps or Twitter, that allow people outside the company to access information. Just as important, but less publicized, are APIs developed for internal use, so that multiple applications owned by the same company can easily communicate with one another. APIs have become a widely used way of building systems of smaller software applications that communicate with one another to deliver value through reuse and permutation.

Yelp and Google Maps

Yelp uses a Google Maps API to display nearby restaurants. Yelp inputs a location, and Google Maps outputs the names and addresses of restaurants near that location.

Health Insurance

An insurance company uses a claims service to display claims to users in a web app. The web app inputs a user identifier, and the claims service API returns a list of claims for that user, with claim numbers, date of service, and cost.

Microservices

To understand why APIs are so prevalent, you need to understand microservices. Microservices architecture is an architectural style that consists of independently deployable services that, taken together, comprise a software application or system that delivers value to users and businesses. Many modern apps have a microservices architecture, with a user interface that talks to those services to present information to users and allow them to act on it or modify it. Microservices architecture has become popular because it supports domain-driven design, continuous integration and deployment, and autonomous teams. APIs allow for communication between all these services and user interfaces, while maintaining each service as an independently deployable unit of software (it's worth noting that there are other methods for communication between services, such as event streaming). As a product manager for an API, you're a product manager for one of these independent units of software that work together with other software to deliver value.

Terminology

What's the difference between an API and a service? What about services vs. microservices? And what about web services? There's a lot of confusing language in this area, and the terms have become blurred, especially when non-technical people start talking strategy. The main things to know are that there are many types of services, but an API is specifically intended to be consumed easily by other applications and developers. If you want to dig into this more, this article explains the relationship between APIs and web services.

Why build an API?

Building an API, or integrating with one, is roughly the same whether your API is internal or external-facing. The motivations for building, however, may be different based on the intended use case.

Generally speaking, an API is a published contract, that exposes an interface to application functionality, for use by a consuming application. This interface allows the flexibility to hide/modify the underlying business logic or implementation details, without impacting the previously published interface.

For users outside your company – if you provide a service that users outside your company find valuable, exposing that service via an API allows others to integrate with you. In this scenario, you're more likely to charge for access to the API as part of your business model. Examples include FitBit's Web API or Twitter's API.

For users inside your company - the benefit of internal-only APIs is that they provide centralized business logic or data sets in bounded contexts. This information can then be used by other internally-developed applications. Creating APIs around core business domains allows for more flexible development of other applications, and less risk in duplicating business logic across multiple software applications. This is why APIs have become so popular – for large and complex domains (i.e. any enterprise organization), APIs allow for easier management of information and more consistency in how information is used, while providing the flexibility that individual teams need to operate independently to deliver value to users. Without APIs, attempts to standardize information result in major roadblocks to development, and result in workaround versions of the same information popping, which inevitably become out of sync with the source of truth.

How APIs work

When you communicate with an API, you send it a request, and the API sends a response. Responses can vary in size from very short to extremely verbose. Regardless, data from an API is returned in a structured format so a computer can parse it. Luckily this format is also human-readable.

API Formats

JavaScript Object Notation, or JSON, is the most popular format in use today. It's built on JavaScript, which is ubiquitous on the web, and uses keys and values to structure responses. A key is an attribute of the object being described. A value is the corresponding detail for a key.

In this example, our keys are name, role, and location. The corresponding values appear after the : in each line of the response

```
{
     "name": "Jane Smith",
     "role": ["developer", "contractor"],
     "location": "Austin"
}
```

The other common format for API responses is XML, but you're unlikely to encounter it these days, as JSON has become ubiquitous for newly-developed APIs. Read more here. You can also read more about API formats in Zapier's guide to APIs.

Methods

The ways of interacting with an API are referred to as methods. The most common ones you'll encounter are POST, GET, PUT, DELETE. These methods map to the common terms used for interactions with applications you're probably familiar with: Create, Read, Update, and Delete, or CRUD.

Method	Description	CRUD Analogy	Example
POST	Asks to create a new resource	Create	Create a new entry for a recently added person
GET	Asks to retrieve a resource	Read	Retrieve a list of people with their name, roles, and location
PUT	Asks to edit an existing resource	Update	Modify the address for a person
DELETE	Asks to delete a resource	Delete	Remove a mistakenly added person

Endpoints

In order to use any method, you'll have to send a request to the API via an endpoint. An API is accessible via a URL, and an endpoint is a more granular location at that URL that returns certain information (the end point of the URL). For example, our team directory may look like the following:

- URL: `https://teamdirectory.com`
- Endpoint for getting people: `https://teamdirectory.com/people`
- Endpoint for getting houses: `https://teamdirectory.com/teams`

An API may have just one endpoint, or it may have thousands. It all depends on how much information the API provides, and how developers and apps will need to access that information.

Resources

The bulk of the content of an API response is comprised of resources—discrete chunks of information that a developer wants to interact with. A developer is usually interacting with this information in order to provide information to a user—see the Design section below for more on how this impacts the structure of your API. Resources can be just about anything that can be described with data, including people (demographics like name, location, etc.), products (like items for sale in an ecommerce store), or tweets (as in Twitter's API).

Data types

There is specific terminology for the data types you'll encounter when working with API requests and responses. Below are some common ones you'll encounter. For a full list, consult documentation for the language of your API.

- **string:** An alphanumeric sequence of letters and/or numbers
- **integer:** A whole number, positive or negative
- **number:** A number (float or double), possibly with decimal value, positive or negative
- **boolean:** "true" or "false"
- **array:** A list of values
- **date:** ISO8601, timezone, UTC
- **currency**
- **null:** Not a data type of its own, but worth mentioning in the context of data types. "Null" means "empty value"—any field that is allowed to be nullable may have "null" as the value, rather than a string, integer, number, etc.
- **object:** An object is a collection of key-value pairs in JSON format. While not exactly a data type itself, it's a term you'll encounter when talking about data in APIs.

Status codes

Something else that comes along with an API response is a status code. The status code provides a summary of the status of the response itself. This helps developers understand if the communication worked as expected. Below are some common status codes— these may be familiar to you from general application development, as they are standard for communication over HTTP.

Status Code & Name	Meaning	Action Required
200 OK	Successful request	None
400 Bad Request	Incorrect syntax	Check the request and make sure it's formatted properly
401 Unauthorized	Authorization is required	
403 Forbidden	Requestor does not have access rights to the content	Ensure your credentials are correct, or get additional permissions
404 Not Found	Cannot find the requested resource	Verify your request is valid, or reach out to the API team for more information. What you're looking for may not be available via this API
500 Internal Server Error	The API server encountered an unexpected condition	The API server encountered an unexpected condition

Authentication

Most APIs require some type of authentication to access. This is obvious for data that should have access restrictions, but even for publicly available information, authentication provides protection for the API. Storing and transmitting data requires servers and network connections, neither of which are free. Authentication ensures that the team responsible for those costs can manage how many requests they receive, and predict how much they'll need to invest in infrastructure to support those requests.

Building an API

Some special considerations come into play when building an API product that a Product Manager should take into account.

Discovery

A key difference between user-facing products and APIs is that APIs have two special types of users.

Machines: The applications that will make requests of your API can be personified and thought of as users. This perspective can be helpful for writing user stories (see How to Write User Stories without Users).

Developers: The developers who are building the applications that will make requests of your API are also users of your product. This perspective is especially useful when writing documentation and providing support for your API.

Design

When designing the API, keep in mind what you learned from talking to your users. A product designer can help facilitate design decisions among the team by gathering feedback from users, and talking with developers on the team about design options. Product Managers can help by bringing the broader business perspective, especially when it comes to what is in scope of this API versus another.

Consider end-user workflows

Think about the end user in order to structure endpoints and resources in a way that makes sense for the domain. A well-designed API that understands users all the way out to the end-user can also guide developers toward making their apps better for their users.

Naming things is hard

Invest time in naming things well. Clear and consistent naming reduces the need for documentation and support, and establishes a common domain dictionary for your team and your customers. Include input from stakeholders, subject matter experts, and your users in the naming process through activities like user research, domain-driven design (event storming), and stakeholder interviews.

Consider the broader context and adjacent domains

Deciding what should be an endpoint, versus a new resource at an existing endpoint, versus an entirely new API, is an ongoing discussion and discovery process that occurs at all levels of an organization. To start with, take a user-centered approach: if the new information is in a similar domain or workflow to one you're already working in, you should probably keep it within the scope of your API. If it feels distinct to your users, it may make sense to belong elsewhere. You can use card sorting to help you design a model that makes sense for your domain and your users.

Backlog management

Writing user stories

Use machine personas to write acceptance tests from the perspective of consuming applications. In this example, MyInsurance is a user-facing mobile app that displays insurance information to customers. MyInsurance needs to call a variety of APIs to gather data for display to the customer, of which the Claims Service is one.

Example
User story
As MyInsurance
I want Claims Service to send me claim statuses
So that I can display claim status to my users

Acceptance criteria
Given I send a GET request to endpoint /v1/claims with a user id
When a successful response is returned (200)
Then I see a list of claims for the user id
And I see a status for each claim

Acceptance testing

When testing delivered stories, you'll call the endpoint with the request described in the story, and verify the response is being returned as expected. You have a few options for testing, each with its own benefits:

UI spec: If your team is using a tool for generating documentation like Swagger UI, you can test the API through a user interface in your browser by browsing to the correct endpoint and trying it out.

Postman: Postman provides a UI for testing your API, as well as saving certain calls for later testing. They offer a free version and a variety of paid tiers.

In-browser – You can also call the API URL directly in a browser. You'll need to know the URL of the API, and write the query string yourself (see below for more on query strings). Firefox formats JSON by default. For Chrome, use a simple extension like this one to make it easier to read. Note, however, that this only works for GET requests.

Writing query strings

A query string is appended to the end of an API URL in order to return a subset of results, based on filters the API can accept. They look like this: `http://teamdirectory.com/people?key=value`, where the ? and everything after it is the query string.

Examples

If you had a team directory service that manages a list of team members, you might send a GET request with the following: `http://teamdirectory.com/people?location=Austin` which would return all people that are in the `location` of `Austin`.

Using the same directory service, you might send a GET request with the following: `http://teamdirectory.com/people?location=Austin&lastName=Smith` which would return all people that are in the location value Austin and have a `lastName` value `Smith`. Note that both criteria would have to be true for the order to return in the list.

Query strings can also be used when dealing with paginated responses. If there is a lot of data to return, API responses may be broken up into chunks. You can specify which page of results you want returned via a query string. For example, `http://teamdirectory.com/people?page=2&size=100` would return the second page of results for `people`, with 100 results per page.

Making changes

Changes to an API must be approached with more caution than updates to user-facing applications. If Facebook moves a button in their UI, users may be frustrated, but they can figure out how to work around the change. If your API moves a resource from one endpoint to another, the application making the request isn't smart enough to go find the new endpoint. Therefore, PMs on API products must put more thought into future-proofing their product than is commonly needed on user-facing applications. This doesn't mean you should design the whole thing up front, only that you should invest more time in validating your direction than is strictly necessary for other products.

When you do need to make changes to how or what data is transmitted, communicate this to the teams using your API, well in advance. Your timeline for making a change may need to depend on the other teams' ability to implement the changes in their code to account for the updates to your API. Be especially careful about breaking changes – changes to the API that will break current integrations, or break how information is requested and displayed in applications that are using your API.

Versioning

Versioning allows you to make breaking changes while mitigating adverse impacts to consuming applications. Teams that aren't ready to move to the new version can continue on as they have, while you make changes in the new version. However, this means supporting multiple versions of the API in production, and having a strategy for when you will deprecate the old version. Think of your API's communications with other applications as a business agreement. If you change the terms of the agreement without notice, your customers will be frustrated and lose trust in your API. Even for internal APIs, this can have severe consequences, not the least of which is lost value to your business and your end users.

Documentation

The deployment process for getting the code of an API into production isn't any different from other apps, but there are special considerations for developer experience of using your API. In addition to making the API available and letting people know it's out there, you need to have documentation that tells developers how to integrate with and make requests of your API. Many APIs are documented poorly or not at all. This impacts adoption and retention of your API, so thinking about documentation throughout the development process is key.

This documentation is most commonly provided as a website, though a Google Doc or PDF can work in a pinch (or if you're still validating your MVP). Documentation should include:

- **Authentication information** tells developers how to authenticate to your API if authentication is required.
- **Endpoints and their definitions** help developers understand what your API can provide, and which endpoint to call for the data they need
- **Data format and example responses** tell developers what to expect from the response, and how to understand terms that may be unique to your domain
- **Code snippets that developers** can copy and paste into their code for ease of use

Additional nice-to-haves:

- An **interactive console** to experiment with available endpoints
- A **quickstart guide** that helps developers complete a request and response quickly
- An **FAQ page**
- **API status information** to help with troubleshooting

These items commonly come up when discussing quality API documentation, but be sure to get feedback from the developers using your API, like you would with any product.

Beyond product-market fit

There are some business considerations unique to API products you need to keep in mind. You should think about these before you get to production,
but their importance becomes much greater once you've validated product-market fit.

Date limits

Rate limiting puts a cap on the number of requests that can be made in a given period of time. Too many requests in too short a time can take down your service, impacting all applications that are using what your API provides. Rate limiting can prevent malicious or unintentionally noisy requests from taking things down completely, by simply dropping requests once the limit is reached and picking them back up when enough time has passed.

Downtime

If your API goes down, there's a cascading effect. Any application that relies on the data you provide will be impacted and may even go down entirely as well, if not be simply unusable. This impacts the end users of that application. In a world of microservices passing data around and apps that are being used around the world, the importance of uptime has never been greater. Downtime is always something to be avoided, but as a link in the chain that is closer to the source, the responsibility of APIs to remain operational and performant is more critical because the impact is often greater than a single user base.

Costs

Storing and transmitting data costs money. Just like user-facing applications have to consider server space based on the number of users they have, your API needs to consider server space based on the number of requests you receive. If your API provides value to other companies, you can pass this cost on to the consumer, usually via a monthly charge. However, if your API is internal, this will factor into the total cost of ownership of your product. When it comes to data storage, consider what you need to store to provide your service, and how you'll offload any data you don't need to be responsible for.

Data freshness

Consuming applications can end up with out-of-date data if they don't request updates for data that changes. There are a few ways to mitigate this, and your approach will depend on how fresh the data needs to be, how many requests you are willing to process, and what technologies are involved. Some customers may be able to work with stale data if it increases API availability (i.e. serving stale data is preferred to not serving data at all). Knowing your customer's needs will help you weigh these tradeoffs.

- **Polling:** Polling is when the consuming application makes requests at regular intervals for new data. This doesn't scale well and is very inefficient. More requests, means more servers, which means more money. This could work for infrequently updated data.
- **Long polling:** With long polling, your API doesn't respond to requests until something has changed in the data. This saves you processing costs, but there are some drawbacks. How many requests can you hold on to? What happens if the connection breaks?
- **Webhooks:** With webhooks, an API can POST data to the consuming application, which also listens for requests instead of just making them. This allows your API to send a request to update information when data changes. This works well for things that are batched, such as getting all changes once per day. Unfortunately, not all consuming applications can act as servers.

Measuring success

User feedback

An API product has a chain of customers. The developers who integrate with your API are your customers. The business those developers make software for is your customer. And the customers of that business are also your customers. Getting feedback all the way through that chain will help you ensure that your API is providing value all the way through to the end user and ensure its adoption.

Key metrics

In addition to qualitative feedback about the API's value, you can measure an API product by quantitative measures of success.

Metric	Definition	Why Measure
Conversion rate	The rate at which people who visit your portal/docs actually end up using your API	Indicates value of your API and ease of adoption
Use cases covered	How many of the desired use cases your API addresses	Indicates if your API is finding a market (i.e. meeting user needs)
Retention	The rate at which you keep customers once they've started using your API	The rate at which you keep customers once they've started using your API
Request volume	Number of requests per time period	Indicates engagement with the API
Time to first call	How long it takes someone to go from "I'm going to use your API" to completing their first request and response	Indicates how difficult it is for someone to realize the benefit of integrating with your AP
Customer support time	Amount of time spent on customer support work	Indicates clarity and usability of your product and its documentation
Uptime	Percentage of time your API is up and running	The rate at which you keep customers once they've started using your API
Latency	Amount of time it takes for your API to respond to a request	Indicates responsiveness of your product

Resources

- Principles of Designing and Developing an API Product whitepaper by Caroline Hane-Weijman, David Edwards, and Francisco Hui
- How to Write User Stories without Users by Amanda White
- API resources on the Tanzu Developer Center
- Agile Architecture presented by Matthew Parker at SpringOne Platform 2018 (there's a corresponding article as well)
- What's an API? by Justin Gage: A quick article explaining what APIs are using real-world examples
- Decoding REST APIs for Product Managers: Short article that uses the common restaurant/server/customer analogy to describe how APIs work
- An Introduction to APIs by Zapier: A series of lessons on APIs, good if you want a deeper dive
- The UX of DX: User Testing in the Invisible World of APIs by Jenny Wanger
- The Best Developer Experience KPIs by Jenny Wanger
- OpenAPI Initiative

Examples of API docs

Intercom, Stripe, and GitHub are all well-known and praised for the quality of their API documentation. Keep in mind that all three use integration as part of their business model. For internal-only products, you may not need this level of documentation, but the more users you have, the more value you'll get out of self-service documentation.

About the author

Becki Hyde is a Staff Product Manager at VMware Tanzu Labs, where she works with enterprises to implement Lean Product Management, User-Centered Design, and eXtreme Programming at scale