GOOD API Insights

Managing API Deprecation and Sunsetting

Erik Wilde

Good API Insight 2019-001

July 2019

Abstract

APIs have a lifecycle that takes them from earlier stages to the end of their productive life. Managing this lifecycle should be an integral part of API design and culture. Towards the end of their lifecycle, APIs often get deprecated, and then eventually sunset. For both of these transitions, it is useful for consumers when they can see these transitions in the API lifecycle. For both stages (deprecation/sunset) it is possible to explicitly link to documentation, and to expose the lifecycle transitions as HTTP header fields. In this insight, we describe the `Deprecation` and `Sunset` HTTP header fields and how they can be used by API providers and consumers to improve API management.

GOOD API

Table of Contents

Introduction

Digital Transformation results in an increased focus on designing and composing digital value chains, and on designing components in these value chains as APIs. The underlying idea is that APIs provide loose coupling between components, and therefore allow cultivating a dynamic and constantly evolving landscape of products exposing and using APIs. API-based value chains make it easier to quickly develop and adjust products according to the continuously evolving needs of an organization and its clients.

In such an environment, API management becomes an essential part of managing the business. When every product is represented by an API, product management always has an API management component. Like all products, API products have a lifecycle and need to be managed according to their journey throughout that lifecycle.

Products as APIs

API products are following the general lifecycle of all products, going from ideation and inception all the way to finally reaching their end-of-life. API products should be as loosely coupled as possible so that their individual lifecycles can be managed independently by product teams. In more traditional approaches, tight coupling often becomes a bottleneck, because changes even when made to one small part may still be subject to rather heavyweight and slow release processes for the larger component they are a part of.

API product management can be regarded as being not much more than traditional product management applied to API products. But there are some aspects of APIs that introduce additional constraints for API product management:

- *Rate/Type of Change:* API products ideally evolve constantly, based on feedback and investment decisions. Design and development should be done in a way so that products can be improved continuously, without disrupting existing users. But continuous feedback and development also mean that API products get deprecated and eventually discontinued.
- *Number of Products:* API landscapes in large organizations often have hundreds or thousands of products. There is a large number of cross-cutting concerns across these products, and it helps both API designers/developers as well as API consumers when these are addressed consistently. API designers/developers benefit because they do not have to reinvent the wheel and can solve well-known problems by using API design patterns and tooling. API consumers benefit because the same problem is solved in a coherent way across the API landscape, meaning that they can reuse their understanding and tooling across all APIs that they are consuming.
- *Product Dependency:* API products are special because dependencies between products result in *runtime dependencies* (i.e., if a product depends on another, that dependency will be there for the complete lifecycle of these products). This means

that responsible dependency management is crucial and should minimize the disruption caused by different products going through updates and their lifecycles at a different pace.

These special aspects of API product management mean that in API landscapes, addressing speed of change and landscape size are essential to make sure that the landscape can deliver on the promise of increasing the speed at which organizations can innovate and get products to market.

Part of responsibly managing API products in such an environment is to have a plan how to manage product evolution. And part of that evolution is to be aware of the fact that no product lives forever, and that in fast-changing environments, products will get deprecated and discontinued routinely, and quite often.

In this paper, we discuss ways in which API deprecation and sunsetting can be approached for API products, and how API designers/developers and API consumers can use these mechanisms to improve their API practice and landscape.

API Lifecycle Management

API products move through a lifecycle that should make it possible to evolve products as easily as possible. Supporting constant change [EvoArch] is one of the main goals of API management. When many of these products are managed in complex organizational settings with continuously evolving landscapes [CAMbook], it becomes important to have a strategy and a program in place that help product teams and the organization to find the most effective balance between team autonomy and organizational constraints and support.

In many cases, organizations have maturity models that guide products throughout their lifetime. While definitions and details vary, the general path goes from prototype to experimental to product to maintenance to deprecated and finally ends at sunsetting. The mechanism how APIs move through these maturity stages differs as well, it can be by self-declaration, or can be a decision that is made by a panel of product managers and architects.

Regardless of the specifics of what the stages look like, and how stage transitions are determined, what really matters in large API landscapes is how stages are communicated. It should be easy for the consumer of an API to figure out which maturity stage an API is in, and this is particularly true for the stages at the beginning ("is this API stable enough for me to use?") and towards the end-of-life ("is this an API I should build a dependency on?").

In order to make this easy for developers, the best way is to make stages part of the configuration of the API product. This way, the API can use tooling that picks up this information, and the deployment pipeline or a configuration resource of the API itself can expose this information. This way it becomes possible that the information about the API gets exposed through the API.

This information about the maturity stage then can be picked up by API consumers and by the API landscape. At the landscape level, this allows easy insights into questions such as how many APIs there are in the experimental stage, or how many APIs there are that have or will be deprecated or sunset. For consumers, this information is useful when they start using an API, but it also can be useful at runtime when they are using an API. For example, if an API starts emitting information of an upcoming deprecation or sunset, and the API consumer recognizes this information, this can raise warnings in logs or someplace else.

Web APIs and HTTP

In this paper, we are specifically looking at how to signal the most relevant events, which are deprecation and sunset, and in the most widely used type of APIs, which are Web APIs based on the Hypertext Transfer Protocol (HTTP) [RFC7231]. HTTP is an application-level message-based protocol that has a rich model of interaction metadata, most importantly through its header fields. There are over 200 types of HTTP header fields [WebConcepts] and they represent many different kinds of information that can be represented in HTTP interactions.

HTTP header fields are designed to be optional, meaning that they can be safely ignored while still keeping basic HTTP interactions operational. However, it is possible that certain interactions can only succeed when certain header fields are present and used in the interaction. But the general model underlying header fields makes it possible to add new ones which can be useful for HTTP peers supporting them, but which also can be safely ignored without compromising the ability to interact.

In the following two sections, we describe two HTTP header fields, `Deprecation` and `Sunset`. Their semantics and usage will be described in the respective sections, but there also is an underlying design idea that applies to both HTTP header fields.

The `Deprecation` and `Sunset` HTTP header fields signal information about the fact that an API is or will be deprecated, respectively that an API will be sunset. This information is represented in header fields which start appearing in HTTP responses when the service decides that it is time to signal this information to consumers. It is up to the consumers to act on this information. They are also free to ignore it and keep consuming the API optimistically. However, consumers can benefit from using this information, for example by alerting developers of the consuming service that a dependency is or will be deprecated, or that it will be sunset.

In order for this model to work best, it is important for the Web API using `Deprecation` or `Sunset` to document the potential usage of these header fields. This means that even though during its normal operational phase the API will not use these header fields, it is documented from the very beginning that when deprecation or sunsetting happen, they will be signaled with these header fields. This published policy allows consuming services to listen for these header fields and to act on them when they start appearing.

HTTP Header Field Scope

One typical scenario is that an API as a whole is deprecated or sunset. Looking at it from the HTTP resource perspective, that could mean that all its resources should start using `Deprecation` or `Sunset` HTTP header fields. This is the most pure approach to make sure that each resource is entirely self-contained. In this approach, consumers of resources do not even need to know that these are part of an API: They simply treat them as self-contained resources that carry all relevant information.

An alternative approach is to design an API where for example only select resources (such as the API home document) will use `Deprecation` or `Sunset` HTTP header fields. This means that the context of the header field is larger than just the resource it appears on (the home document). As a result, only clients knowing this particular API design will be able to pick up this information, whereas clients treating every API resource as entirely self-contained will not. This means that there is a risk that some clients will miss this information, and may therefore not be able to benefit from the information about deprecation or sunsetting.

This is simply an API design decision: Exposing the HTTP header fields *on all resources* is a more pure approach eliminating potential problems with visibility for certain clients. Exposing the HTTP header fields on select resources only reduces chatter, may be easier to implement, but may mean that some clients fail to pick up the information from those select resources.

Last but not least, it is also possible to combine these approaches: An API may publish its complete deprecation or sunset (of the API as a whole) only on its home resource. But it may still use the `Sunset` header field on select resources that during normal API operations have a limited lifetime (for example because by definition they only exist for a certain period of time). The effective sunset of one of these limited-lifetime resources in this API then is the minimum of the sunset as advertised on the limited-lifetime resource itself, and of the API as a whole as advertised on the home resource.

API Deprecation

Good API design often means to design APIs for extensibility and evolvability, so that they can safely evolve without having to alert existing consumers about extensions being added. When following this approach, it becomes easier for API developers to evolve and improve the API, without having to worry about disrupting existing API consumers every time they make changes.

This paper is not intended to go into the details of how to design APIs for extensibility and evolvability, but it recommends to make these issues part of your initial design decisions, and to document these design decisions from the very beginning.

The following example shows how it is possible to explicitly link to an API's or a resource's deprecation policy. This often might be documentation about when deprecation is considered, and how it will be managed. Ideally, this should include SLA-style information, such as how far in advance a deprecation is announced, and for how long afterwards APIs and resources will remain operational. The `deprecation` link relation type [draft-dalal-deprecation-header] can be used to link to a deprecation policy.

```
Link: <https://developer.example.com/lifecycle>; rel="deprecation";
type="text/html"
```

Some APIs might prefer to not send these links in all responses for all resources. In that case, it may make sense to only send those for select resources of the API, such as the home document [draft-nottingham-json-home], or to include these links in the home document (or other descriptive resources for the API).

Whatever your choice is how to design APIs for extensibility and evolvability, there always will be a limit of how far you can take extensibility and evolvability. At some point every API reaches the point where it makes more sense to design a new one, rather than pushing the limits of extensibility and evolvability too far. When that point is reached, it makes sense to design and deploy a new version.

When that state has been reached, there now are two APIs operating in parallel: the old one that had reached the limits of its extensibility and evolvability, and the new one that improves on the old design. From the operational perspective, this may be the point in time where you want to start moving existing consumers from the old API to the new one, and where you want to encourage new consumers to use the new API instead of the old one. Oftentimes, the goal is to have some transition period where the two versions are operated in parallel, and to get to a point where you can safely turn off the old API, saving on operational and maintenance cost.

Somewhere during this process, you start marking the old API as deprecated, meaning that it is officially not the recommended version anymore. Existing consumers should migrate to the new one, and new consumers should not start using the old one at all. Oftentimes, this status of the API lifecycle is indicated in its documentation, clearly marking the API as being deprecated. In addition, this information can be exposed in the API itself.

The HTTP `Deprecation` header field [draft-dalal-deprecation-header] can be used to indicate that an API will be or has been deprecated (depending on whether the timestamp is in the future or the past). An API that has been marked as deprecated starts emitting this header, which can then be picked up by consumers listening for it. This way, the deprecation information is not just shown in the human-readable documentation, but instead it becomes machine-understandable in the API itself.

The following example uses the HTTP `Deprecation` header field, as well as the `deprecation` link relation type.

```
Deprecation: Sun, 11 Nov 2018 23:59:59 GMT
Link: <https://developer.example.com/lifecycle>; rel="deprecation";
type="text/html"
```

In this example, the deprecation date is in the past, meaning that the resource is deprecated. When seeing a deprecation header announcing an upcoming or effective deprecation, visiting the linked resource representing the deprecation policy may be helpful, because it may have been updated with information about the deprecation. As mentioned above, APIs may choose to not expose the link to the deprecation policy on every resource, and may instead just expose it on select resources such as the API's home document [draft-nottingham-json-home].

It should be noted that this section mainly portrays deprecation in a scenario where an API gets replaced by a new design. The very same model applies to an API that is not going to be replaced by a new one, but simply reaches the end of life in its API lifecycle. In this case, the same considerations apply: Indicate the fact that the API will be or has been deprecated, and then (possibly later) follow-up with an indication when the API will be sunset.

In either case, whether transitioning to a new API or not, the next step in API lifecycle management is to decide and announce when the deprecated API will be sunset. While deprecation is important for consumers to understand that they should start to look for replacements, sunsetting is even more critical as this means that a resource will stop responding, and thus may break consumers that depend on it. Similar to the HTTP `Deprecation` header field, there is an HTTP `Sunset` header field for announcing sunsets.

API Sunsetting

APIs and their individual resources typically are not operated forever. In the previous section, we have discussed how to use the HTTP `Deprecation` header field to announce that a resource is not the recommended resource to use anymore. Once a date is know when an API or a resource will go permanently offline, the HTTP `Sunset` header field [RFC8594] can be used to alert consumers of the upcoming sunset.

Before that happens, in a way similar to the deprecation policy mentioned in the previous section, an API should have a well-defined sunset policy and document how sunsetting is going to be managed. Typically, this policy should at least say how far ahead consumers will be notified, and how they will be notified. This information is made available in a sunset policy that can be identified by linking to it using the `sunset` link relation type [RFC8594], as shown in the following example:

```
Link: <https://developer.example.com/lifecycle>; rel="sunset";
type="text/html"
```

It should be noted that both the example shown in the previous section and the one shown here link to the same resource. That makes sense if there is a resource about the general lifecycle policy, which describes both deprecation and sunsetting. If that is the case, it alternatively is possible to link to that resource using both link relation types, as shown in the following example:

```
Link: <https://developer.example.com/lifecycle>; rel="deprecation
sunset"; type="text/html"
```

Sunsetting means that a resource will stop responding. Once that date is known, it can be published in a `Sunset` HTTP header field:

```
Sunset: Wed, 11 Nov 2026 11:11:11 GMT
```

It is up to consumers to decide how to use this information. It may help them to create alerts or warnings that might help users to adjust to the used resource being sunset. Also, this date should always be treated as a hint (i.e., it is not guaranteed that the resource will not be sunset earlier or later than the advertised date), but it still can help consumers to be more proactive in terms of managing their dependencies.

It also should be noted that sunsetting is not just for managing the lifecycle of a complete API, where an API and consequently all of its resources will be sunset at some point in time. It also can be used at the resource level, where individual resources of an API are being sunset, and are using a `Sunset` header field to expose this fact.

Resource Sunsetting

Resource lifecycle management can be part of the normal operations of an API. For example, consider the case where an API supports creating orders, and it supports a model where an order can be "pending" so that customers can update orders before finalizing them and getting them fulfilled. In many cases, pending orders will have a well-defined lifetime during which they either have to be updated (which may reset the lifetime counter) or completed, which will then get them fulfilled.

In such a scenario, the API can use an HTTP `Sunset` header field to expose the lifetime of pending orders, allowing API consumers to easily and clearly see when the pending order is going to expire. Of course, there would be other possible ways of designing this particular resource lifecycle, but using the `Sunset` header field is one possible design choice.

Another possible scenario are resources that by regulation or legislation only can be kept for a certain time. For example, there may be rules that some information has to be deleted because of privacy considerations. Accessing this kind of information through an API then would also mean that these resources have a well-defined time when they will expire. An API design in this context can also choose to use the HTTP `Sunset` header field to expose this information.

Supporting API Deprecation and Sunsetting in API Products

When the deprecation and sunsetting strategies outlined above are incorporated into API designs, they need to be supported in API products, and thus in implementations. There are a variety of ways in which these capabilities can be implemented. The goal of this section is not to give implementation advice for specific languages or frameworks. Instead, the goal is to highlight certain implementation patterns that make it easier to implement and use the API design in API products.

One important consideration is to think about how deprecation and sunsetting are working in the API. If they are part of a service's logic, then they should be implemented programmatically. For example, if some resources by definition have a limited lifetime, and the API design exposes this lifetime using a `Sunset` header field, then this header field should be controlled by the implementation (which will have logic to compute the lifetime of each affected resource).

If, on the other hand, deprecation and sunsetting are managed as part of general API lifecycle management, then it may make more sense to control them through configuration. For example, if a service supports deprecation and sunsetting through the header fields discussed here, then it should not be necessary to make any changes to the implementation when it is decided to deprecate or sunset the service. Instead, there should be a simple configuration where API managers can configure the time when these events happen, and the lead time when warnings should start appearing, and that should cause the service to start exposing the respective header fields.

For example, the following information could be added to a service's (fictitious) configuration to tell it when to start exposing deprecation and sunsetting information. It would be up to the service's implementation to interpret the information and behave accordingly, but the behavior itself is controlled by the configuration.

```
Deprecation-Time: Sun, 11 Nov 2018 23:59:59 GMT
Deprecation-Warning: Tue, 11 Sep 2018 23:59:59 GMT
Sunset-Time: Wed, 11 Nov 2026 11:11:11 GMT
Sunset-Warning: Fri, 11 Sep 2026 11:11:11 GMT
```

Most importantly, it should be noted that incorporating support for API deprecation and sunsetting into an implementation early on can make it easier to support these concepts in the API product throughout the API lifecycle. Depending on how the API design plans to expose this information (see the section on scope), this information can be injected into the appropriate places in the API as soon as this is triggered through configuration settings. Supporting this early on means that it does not have to be added to the API product when it enters later stages of its lifecycle, which may be helpful because it avoids having to change a codebase that is about to be retired.

Finally, it also is possible to outsource header field control entirely and manage it through an API management solution. One typical approach would be to have an *API gateway* in place and to use the gateway's features to control `Deprecation` and/or `Sunset` header fields. In this scenario, the API implementation does not support these header fields at all, but for API consumers they are still made available as part of the overall API management strategy and infrastructure.

Consuming APIs Responsibly

So far this paper covers how to use deprecation and sunsetting in API designs, and how to implement the respective HTTP header fields in API products. But it also is important how to *consume APIs* that are using these mechanisms.

First of all, from a consumer point of view, ideally the usage of the respective HTTP header fields is well-documented in the API. This is important because in most cases, the HTTP header fields will not show up in the API until the API lifecycle reaches later stages. Clear documentation of the API lifecycle will make it easier for developers to understand that an API will at some point start exposing `Deprecation` and/or `Sunset` HTTP header fields.

In most cases, consumers of APIs will want to use these header fields as indications that the consumed API has advanced in its lifecycle. In most cases this is information that is helpful for the developer consuming the API, and thus should be surfaced in places where they are likely to see it, such as logs, dashboards, or through warning/alert channels. An exception to this is when individual API resources use deprecation and/or sunsetting as part of their regular resource lifecycle, in which case this information can be handled programmatically and should not be surfaced to developers.

In most cases, mitigation will require the intervention of a developer, either by migrating from a deprecated API to a new one, or by searching for a replacement for an API that is being sunset. The main value of deprecation and sunsetting is that it warns consumers ahead of time, so that they have more time to adjust to the changes in the consumed API.

Summary and Conclusions

In this insight, we present ways in which deprecation and sunsetting can be represented in API and resource design using the `Deprecation` and `Sunset` HTTP header fields. The main takeaways from this insight are as follows:

- *Applicable to APIs and Resources:* Both header fields are applicable to individual resources or entire APIs. It is up to the API design to decide which scope they are used for, and how they are used in resource interactions.
- *Design and Document and Implement Support:* Both header fields are intended for later stages of resource and API lifecycles. Making them part of the initial design,

documenting their usage, and implementing support for them helps both producers and consumers.

- *Support via Implementation, Configuration, or Infrastructure:* Developers have different ways how to control when these header fields show up. They can be triggered by application logic, can be controlled by configuration, or may be injected by API management infrastructure.
- *Support by Consumers:* Implementations consuming APIs that expose deprecation and/or sunsetting should make sure that they use effective channels to alert product maintainers when resources or APIs expose deprecation and/or sunset information outside of the expected lifecycle of individual API resources.

In summary, deprecation and sunsetting of APIs or individual resources should be part of API design right away. By using the `Deprecation` and `Sunset` HTTP header fields and link relations, this information can be surfaced so that it is available to API consumers in a machine-understandable way, and the respective documentation can be made more easily discoverable.

Bibliography

- *[CAMbook]* Mehdi Medjaoui, Erik Wilde, Ronnie Mitra, and Mike Amundsen, "Continuous API Management: Making the Right Decisions in an Evolving Landscape", O'Reilly, December 2018
- *[draft-dalal-deprecation-header]* Sanjay Dalal and Erik Wilde, "The Deprecation HTTP Header Field", Internet Draft draft-dalal-deprecation-header
- *[draft-nottingham-json-home]* Mark Nottingham, "Home Documents for HTTP APIs", Internet Draft draft-nottingham-json-home
- *[EvoArch]* Neal Ford, Rebecca Parsons, and Patrick Kua, "Building Evolutionary Architectures: Support Constant Change", O'Reilly, September 2017
- *[RFC7231]* Roy Fielding and Julian Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", Internet RFC 7231, June 2014
- *[RFC8594]* Erik Wilde, "The Sunset HTTP Header Field", Internet RFC 8594, May 2019
- *[WebConcepts]* http://www.webconcepts.info/concepts/http-header/