

API Design Guide

From  Stoplight

What is API Design?

API design is the collection of planning and architectural decisions you make when building an API. Your API design influences how well developers can consume it and even how they use it. Just like website design or product design, API design informs the user experience. Good API design meets initial expectations and continues to behave consistently and predictably.

There is not a single approach to design APIs “the right way.” Instead, we need to lean on industry best practices where relevant and take cues from those who will use our APIs.

Choose your API Specification

Before you can communicate your API design, you need an artifact that someone else can use to understand your design. Historically, this might have been called documentation. While it’s still important to have human-facing documentation that is easy to use, more is required of modern APIs. In recent years the industry has rallied around the [OpenAPI Specification](#) (OAS).

OAS allows you to define how your REST API works, in a way that can be easily consumed by both humans and machines. It serves as a contract that specifies how a consumer can use the API and what responses you can expect.

OAS 3.0 was released in July 2017, by the OpenAPI Initiative, a consortium of member companies who want to standardize how REST APIs are described. There are various other approaches to API description:

- **OAS 2.0**, based on the Swagger definition. Still widely used, but being replaced by OAS 3.0
- **Swagger**, the best known of the approaches being replaced or augmented by OAS 3.0
- **API Blueprint** was created to foster collaboration between API design stakeholders
- **RAML**, the RESTful API Modeling Language, focuses on the planning stage of API design



While OAS 3.0 is the way forward, each of these alternative formats has tooling associated. You may find yourself converting between them, especially OAS 2.0, until the tools catch up.

Your API design requires a way to define how the API will be used. The future-thinking approach is to select OAS 3.0 to describe your API.

Why API Design-First Matters

Now that you've chosen OAS 3.0, you may be tempted to set that aside until after you build your API. While it's useful to describe existing APIs, you should also use your OpenAPI description while designing a new API.

When you design your API alongside a description, you always have the artifact to communicate what's possible with your API. The [design-first approach](#) offers a single source of truth, readable by collaborators and machines alike.

The Design-Second Oxymoron

Design-first becomes clearer when you consider the alternative. If you go straight into building your API, there's no returning to design. That's like constructing a house and then going to an architect to draw up plans. It just makes no sense.

Yet, software teams frequently make similar choices. They may output an API spec from code, which sounds efficient. Unfortunately, by the time you've built an API in code, you've lost out on a lot of the advantages of design-first approach. When your API design exists before the implementation, you can get early feedback, connect your API to tools from the start, and collaborate across departments and functions.

Do you know who will use your API? Even for an internal project, you're likely to have multiple consumers. An API spec allows you to share details about how the API will work. You can send the spec document itself or use tools to prototype your API or documentation. You could generate mock servers based on your spec, as described in another section, and have your consumers make live calls.

Your collaboration can go beyond technical teams, as well. You could get great insights from product, marketing, partnerships, and many other areas of your organization.



The Importance of Knowing Use Cases

When you understand how your software will be used you can design it better. The biggest mistake in API design is to make decisions based on how your system works, rather than what your consumers need to support. In order to design around use cases, you'll need to talk to the consumers, or at least include those who know them better.

Software is rarely built entirely by engineers. There are stakeholders throughout the organization. And while many engineers can be very product-minded, they don't always have visibility of the full picture. If your organization has a product group, that's often where the voice of the customer is most heard. Involve anyone who understands how an API will be used in discussions as you design the API.

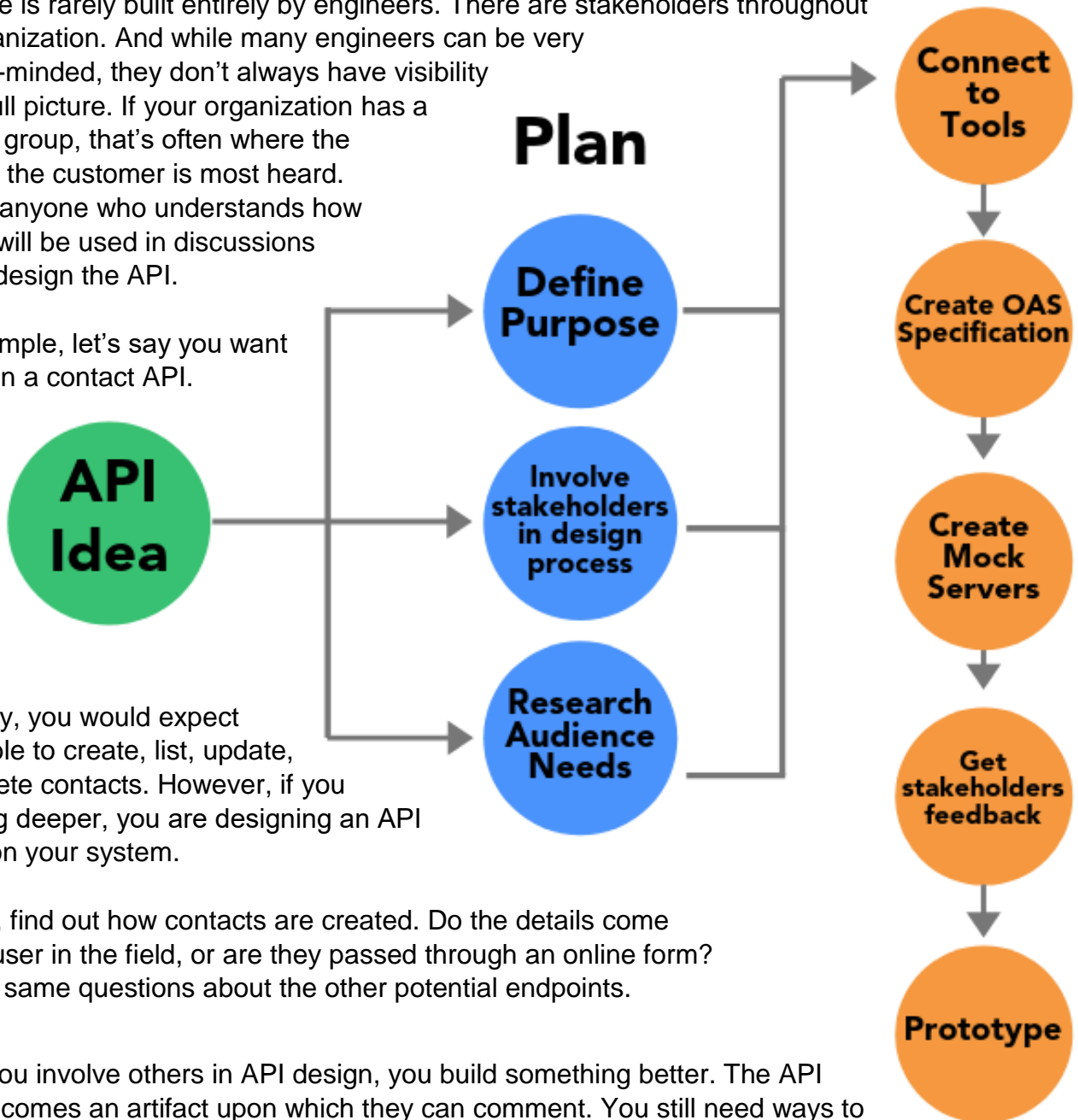
For example, let's say you want to design a contact API.

Naturally, you would expect to be able to create, list, update, and delete contacts. However, if you don't dig deeper, you are designing an API based on your system.

Instead, find out how contacts are created. Do the details come from a user in the field, or are they passed through an online form? Ask the same questions about the other potential endpoints.

When you involve others in API design, you build something better. The API spec becomes an artifact upon which they can comment. You still need ways to coordinate the cross-department conversation, but design-first makes it possible in the first place.

Design



API Design Best Practices

Armed with an understanding of your use cases, you're ready to begin your API design. Each project is different, so best practices may not always fit your situation. However, these are guidelines to keep in mind as you design your API.

While we'll go into specifics below, these are the high level tenets of good API design:

- Approach design collaboratively
- Maintain internal consistency
- When possible, use an established convention

You'll want to keep your entire team updated as you make [design decisions together](#). Your OpenAPI spec is your single source of truth, so make sure it is available in a place where everyone can see revisions and discuss changes. A GitHub repository or Stoplight's [Visual OpenAPI Designer](#) can help keep everyone on the same page.

How to Design a REST API

The OpenAPI spec is focused on describing REST APIs. However, it's still possible to describe an API that violates the RESTful principles. This section is not meant to be exhaustive, but will instead help you avoid the most common infringements in REST API design.

Use HTTP verbs to communicate action. While REST guidelines can be used outside of HTTP, they are so frequently used together that it's safe to assume your API will operate over HTTP. This protocol, upon which the web is built, offers useful operations that should form the foundation of our APIs.

- GET: read existing data
- POST: create new data
- PUT: update existing data
- PATCH: update a subset of existing data
- DELETE: remove existing data

By relying upon these verbs, you can build your API to perform these actions on your fields or resources.

Use nouns for resources and avoid anything that looks like procedure calls. There's no need for endpoints like `/getContacts` when we're using HTTP verbs. Instead, your resource would be named `/contacts` and you could perform the GET action (and any others that are relevant) against that resource.



You may find yourself in a debate about naming your resources. Should they be singular or plural? When there are multiple words in a resource name, should you use punctuation or capitalization to distinguish each word? The most important thing to choose is consistency. If you use one convention with one endpoint, choose the same with another endpoint. If possible, look to maintain this consistency between your APIs, as well.

Use HTTP status codes to communicate errors and success. Just as the verbs provide a solid foundation for how your API takes action, the standard status codes share the results of those actions. For example, never send a 200-level status code along with any error message. Both machines and humans will be confused.

Here's a quick list of the most common status codes and how they should be used:

- 200: Successfully read the data you requested
- 201: Successfully wrote the data you sent
- 401: Authentication is missing or incorrect
- 403: Authentication succeeded, but the user does not have access to the resource
- 404: The resource cannot be found, client-side error
- 500: There was an error on the server-side

There are plenty of other status codes you might find useful. At a minimum, use these most common ones in the expected way.

API Design Patterns

In addition to following REST principles, you'll run into some of the same concepts others have already solved. You may have reasons to implement some of these patterns differently. In all other cases, look to these best practices for approaching your API design.

Sorting can be an expensive operation for your database, but it's one your API consumers will likely need to access. At a minimum, choose a default sort order for results (most recent first is a good choice) and be consistent with your endpoints.

Remember that your API does not have to open up your entire database to consumers. In fact, it shouldn't. You can choose the fields to enable with sorting and work to make those operations efficient. Use a query string parameter of `sort` with potential values matching the field names that are returned in your response. For example: `sort=date_added`

Another useful option for sorting is whether you want results ascending or descending. In the case of a single sorted field, use a second query string parameter of `order` to choose sort direction. If you have multiple sort fields, you're better off using a single parameter with SQL syntax for sorting: `sort=date_added DESC, name ASC`





Paging through results is something you'll need if you have a lot of data. Returning thousands of results in a single API call causes high latency for consumers and can cause issues for your systems. Instead, choose a default page size such as 250, 100, 50, or 10. Then allow consumers to adjust the page size (below a reasonable maximum) and request specific pages of results.

Commonly, you'll see `page` used as a query string parameter for the page number (starting at 1) and `limit` for the page size. Include the current page and the total pages (or total results) near the top of the response. This enables the consumer to track their current location and know what to retrieve next. Even better, include Hypermedia links (see below) to other pages of results.

If your data changes quickly, consumers might miss some results. You might consider some alternatives, such as cursor-based paging, timestamps, or streaming.

Filtering is another useful way for API consumers to control their results. In this case, you can restrict based on values in the results. While this is typically an easier query for your database, you still can control which fields to enable for filtering. With simple filter implementations, you can use one or two query string parameters for each field. For example, `city=Austin` would find all records where the city field is Austin. To look up ranges, use multiple parameters, such as `date_before=2010-01-01&date_after=1999-12-31`.

More advanced filtering will require a different approach. You can use standard search schemas, or explore other technologies like GraphQL.

Hypermedia links help show the API consumer what else is available, allowing them to “browse” your API. These are included near the top of your results, or in the relevant object within your results. Links are wrapped in a `links` or `_links` object. For example, if your API results include abbreviated versions of objects (such as contacts), you can include a link to the complete version of each contact. Links always include the full URL to the API call.

A common usage of hypermedia is paging through results. At a minimum, supply the full URL to the API call for the next page of results. The consumer—human or machine—can easily follow the link to get the results when needed. Other links to include: previous page (prev), first, and last. You can find more details in [RFC 8288](#), which describes Web Linking.

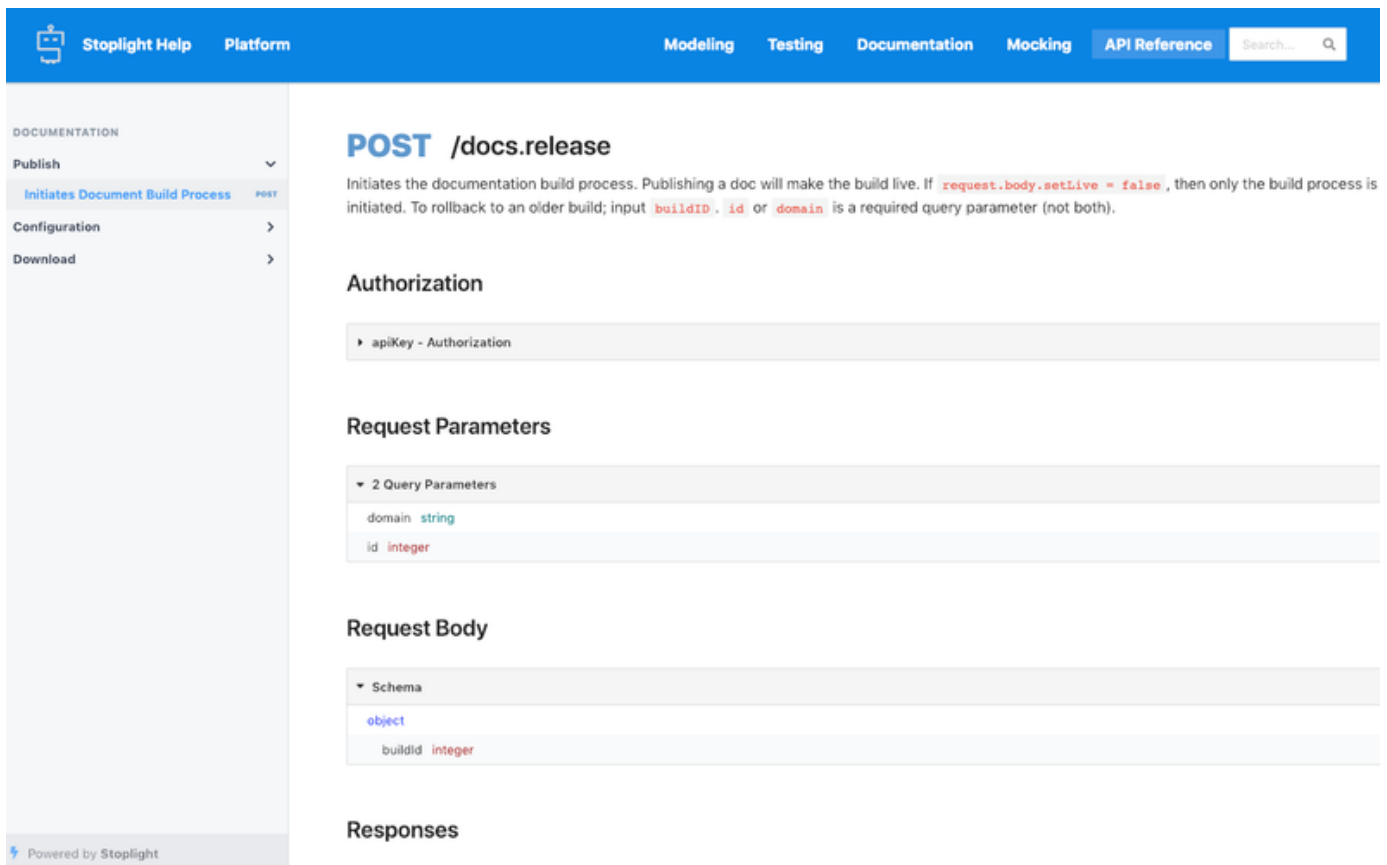


API Design Tooling

API Design Brings Powerful Tooling

When you use an OpenAPI spec to design your API, it becomes part of your workflow. That means as soon as you have even a single potential endpoint of your API described, you can begin to gather feedback and piece together how your API will be used. Rather than toiling away in an API silo, your spec allows for collaboration with colleagues and across departments. You can work the API description into your approval processes, so everyone is on the same page with its progress.

Tooling built around the OpenAPI spec can help in the very early stages of design through a live API and even as you consider versioning and deprecation. We'll cover some of the common tools you might use with your API descriptions.



The screenshot displays the Stoplight API Reference interface. The top navigation bar includes links for Stoplight Help, Platform, Modeling, Testing, Documentation, Mocking, and API Reference, along with a search bar. The left sidebar shows the DOCUMENTATION section with options for Publish, Configuration, and Download. The main content area displays the endpoint **POST /docs.release**. Below the endpoint name, a description states: "Initiates the documentation build process. Publishing a doc will make the build live. If `request.body.setLive = false`, then only the build process is initiated. To rollback to an older build; input `buildID`, `id` or `domain` is a required query parameter (not both)." The interface also includes sections for Authorization (apiKey - Authorization), Request Parameters (2 Query Parameters: domain string, id integer), Request Body (Schema: object, buildid integer), and Responses.



Generate API Documentation

Perhaps the tool most associated with API descriptions is Swagger UI and other tools for generating documentation. Before developers and architects used a definition to help them design APIs, documentation was the biggest use case. While OpenAPI allows for much more than generated documentation, that remains a huge advantage to having your API described in OpenAPI.

There are different types of documentation, but OpenAPI-generated docs thrive for API references and interactive documentation. As you add and update your API endpoints, you can automatically keep your documentation updated. You may even be able to connect these tools to your CI/CD workflow, so that as your new API hits production, so does your new API documentation.

Reading documentation is one way to determine how an API works. Live calls add another dimension to that understanding. Interactive documentation means that consumers can test requests against your API, supply their own inputs and see the response inline.

You'll want to add other types of documentation, too, such as tutorials. Look for a tool that allows you to have customized documentation alongside your generated docs. You'll also may want to match your site's style and navigation.

For an example of fully customizable, generated docs, see [Stoplight's hosted API documentation](#)

Create Mock Servers

Just as interactive documentation adds another dimension beyond simple reference, you can benefit from making calls against your API while you design. Your [OpenAPI spec can be used to create mock servers](#) that use responses you've included in your design. You can collaborate with others around real data and seek early feedback from API consumers.

Much as documentation is built and rebuilt as you update your API description, mock servers can also automatically have your latest changes. Integrate with your own API as you build it by including mock server endpoints in your code, or coordinate with API consumers and collaborators to write tests or sample code. Code you write against a mock server isn't wasted, because only the server root will change when you move to production.

Sometimes hard-coded responses aren't enough for validation. For example, you might need dates in the future, or want to randomize some of the content in your results. Some tools, such as [Stoplight's Prism-based mock servers](#) allow you to extend your API description with scripts before or after traffic reaches your mock server.

Automatically Test Your APIs

Mocking API calls before they're in production is a good idea. Once your API is live, you'll also want to make sure it's built the way you've described. That's where API testing comes in.

Your OpenAPI definition describes exactly how your API can be used and what response to expect. During testing, you create scenarios for how your API is used, then run them to make sure you get



the correct HTTP status code for the method used. If your OpenAPI document is a contact, testing makes sure you've built it true to the terms.

Testing can be built into your CI/CD pipeline, so you always know that your tests are passing. Like other software testing, you can track coverage, ensuring that errors are unlikely to slip through. You can build fully customizable tests with built-in coverage reporting with [Stoplight OpenAPI testing](#).

OpenAPI Specification (OAS) Style Guide

SPECIFICATION

OpenAPI v3

Enable or disable rules used to enforce certain stylistic or validation properties on OAS files in this project.

API

	Enabled?
api-servers OpenAPI <code>servers</code> must be present and non-empty string.	<input checked="" type="checkbox"/>
contact-properties Contact object should have <code>name</code> , <code>url</code> and <code>email</code> .	<input type="checkbox"/>
info-contact Info object should contain <code>contact</code> object.	<input checked="" type="checkbox"/>
info-description OpenAPI object info <code>description</code> must be present and non-empty string.	<input checked="" type="checkbox"/>
info-license OpenAPI object info <code>license</code> must be present and non-empty string.	<input type="checkbox"/>
license-url License object should include <code>url</code> .	<input type="checkbox"/>
openapi-tags OpenAPI object should have non-empty <code>tags</code> array.	<input type="checkbox"/>
openapi-tags-alphabetical OpenAPI object should have alphabetical <code>tags</code> .	<input type="checkbox"/>
tag-description Tag object should have a <code>description</code> .	<input type="checkbox"/>





Use Linting to Spot Errors

As you design your APIs using OpenAPI, you'll need to conform to the spec's schema. You can use linting tools to validate your JSON or YAML as you write. An accurate API definition is important so that you can feel confident that other tools will interpret your API the way you expect.

Linting tools come in command line, editor plugin, and built-in varieties. It helps you spot errors before you commit them to your repository. Since the OpenAPI spec becomes your source of truth, you want it to be right!

More advanced linting tools can also help you design consistent APIs. For example, have you decided to use plural terms for your resources? If you have an API style guide, you may be able to use a linter to catch that singular endpoint before it goes live. Consistency leads to a better developer experience and a greater likelihood that your API won't need major changes.

Regardless of how much your tools help you, it's a good idea to become familiar with structure and elements of your OpenAPI documents.



Understanding the OpenAPI Specification

The industry has selected OpenAPI as the way forward, so let's understand it. From a technical standpoint, it is a YAML or JSON file that follows a specific [document structure](#). You should be able to describe any REST API using a document that adheres to the OAS 3 schema.

The primary sections of an OAS 3 document are:

- Info: meta-data about the API, including its name and version.
- Paths: relative endpoints, their operations, and responses.
- Security: the scheme used to authenticate calls, such as API Key or OAuth.
- Servers: one or more servers that can be reached with the paths.
- Components: schemas to describe reusable elements, such as error messages or responses.
- Tags: labels that can be used for grouping related paths
- External Docs: meta-data for human-readable documentation

While not all of these sections are required in an OpenAPI definition, they can be used together to flexibly describe an API with minimal repetition. Promoting re-use means you can avoid the tedium and potential human error of find-and-replace updates.

OpenAPI Versions: OAS 2 vs OAS 3

While OAS 3 is the most recent version of OpenAPI, it replaced OAS 2, previously known as Swagger. The newer version provides a simpler way to describe APIs, while also offering more flexibility. Because there were a lot of legacy Swagger documents, it's important to have a compatible community-owned version. But API practitioners wanted to move the Spec forward with OAS 3.

One of the biggest differences between OAS 2 and OAS 3 is the components object. For example, responses were their own distinct object in OAS2, whereas they are now organized under components. Other reusable objects now part of components include schemas for security schemes, parameters, and request bodies.

There are a handful of other components, some of which didn't directly exist in OAS2. Two notable new components are callbacks and headers. Callbacks can be used with Webhooks and other asynchronous technologies. Headers, while describable in OAS 2, are now able to be reused more easily.

Should Definitions Use JSON or YAML?

Through the OpenAPI Initiative, the industry has agreed upon this new approach. However, the format to use for definitions is still up for much debate. Both JSON and YAML are supported by OAS 3. They each have advantages for both human and machine consumers.



In terms of readability, YAML is clean and easy for most to decipher. It uses whitespace, colons, and newlines—a common writing syntax. By contrast, JSON has a lot of curly braces, quotes, and commas. Yet, when pretty printed, it can be similarly readable. JSON is also very easily consumed by machines. The syntax is still relatively lightweight and helps modern languages quickly parse data.

The whitespacing of YAML describes the nesting of data. When accurately written, it can be quickly parsed. However, consistent spacing becomes difficult for human editors. Once a machine understands the data, outputting YAML is straightforward, but manual writing can become an effort in fighting indentations.

There are good and bad things about both YAML and JSON. In addition to reading and writing issues, you'll find some tools support only one or the other. It's best to be familiar with both and plan to convert between them when needed.

Get Familiar With API Design

Whether wrestling with data formats or spinning up mock servers, there are tools to improve your API design experience. Start from scratch or import an existing description, then start building and sharing with your team. Stoplight's [Visual OpenAPI Designer](#) provides a design-first suite of tools to help you build great APIs.

