SPEAKEASY

# How to Supercharge Your API Adoption

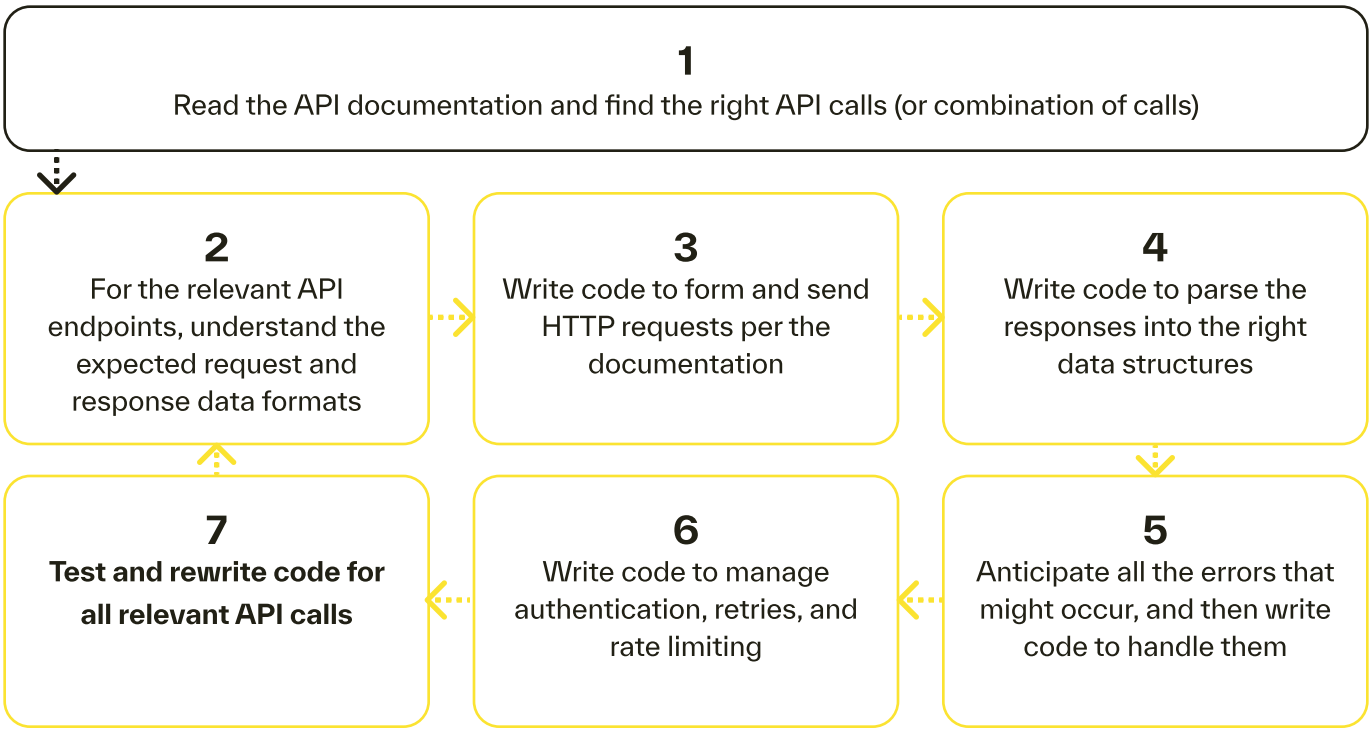
## Learn:

- Why developer experience can make or break your API initiatives
- Seven ways that SDKs can help you accelerate API adoption while reducing support costs
- Your options for creating SDKs

# APIs succeed — or fail — based on how developer-friendly they are

We're so used to the magic of APIs that sometimes we don't realize that integrating with an API is often very **manual, time-consuming, and error-prone** for API consumers.

**Here's a typical process for integrating with an API:**



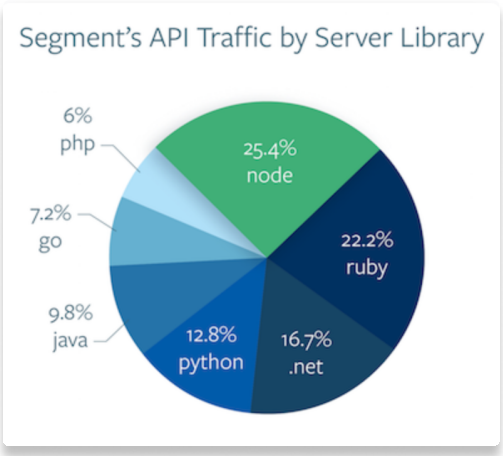

Overall, this approach tends to resemble "trial and error". API users end up frustrated and may give up. "Time to 200" is measured in days, weeks or even months

# How do the most successful API companies become developer-friendly?

How have companies like Stripe, Plaid, Segment, and Twilio managed to thrive?
**They all offer client SDKs that make it easy to integrate with their APIs.**

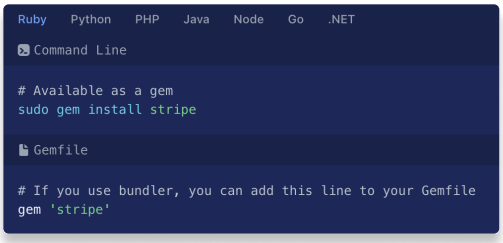
"Server data makes up 25% of our total API traffic. Because our server libraries perform batching and validation automatically, we see customers almost always preferring to send data from our server libraries rather than directly to the HTTP API. 82% of total server traffic originates from Segment-supported server libraries."

Segment



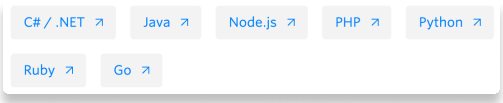

"Stripe's Server-side helper libraries reduce the amount of work required to use Stripe's REST APIs, starting with reducing the boilerplate code you have to write. [To the right] are the installation instructions for these libraries in a variety of popular server-side programming languages."

stripe




"Server-side SDKs (or Server-side libraries) make it easy for you to use Twilio's REST APIs, generate TwiML, and perform other common server-side programming tasks. These SDKs are available in a variety of popular server-side programming languages."

twilio






SPEAKEASY

# SDKs provide a far superior developer experience

### Faster development

SDKs provide pre-built functions that help developers accomplish tasks quickly. For example, an SDK for an e-commerce API might include a pre-built function and parameters for placing an order.

### More intuitive experience

Users get a much better in-IDE experience thanks to auto-completion.

### Broader feature adoption

Because the experience of integrating with an SDK can be as simple as calling an SDK method, developers are much more likely to use a wider set of features.

### Standardized data and type definitions

SDKs can ensure that data returned by an API is handled in a standard and recommended manner.

### More robust integrations

SDKs can ensure object type safety. This can dramatically reduce the chance of bugs.

### Breaking change mitigation

When an API introduces breaking changes, the SDK can in some cases be updated to accommodate those changes “under-the-hood” – while maintaining a consistent interface for the developers.

### Streamlined documentation

SDK docs focus on specific outcomes and abstract away many low-level details. This makes them more usable, easier to understand, and ultimately more effective.

# Integrating to an API with and without an SDK

This is an example of what integrating with an e-commerce API, to place an order for a new customer, might look like without an SDK:

Create the right header object

Understand and create the shape / types of all required parameters

Understand each endpoint's structure and required headers. Handle async operations elegantly

Anticipate errors and the associated HTTP status codes, interpret error messages, handle robustly

More error handling

Understand the customer data model and encode data correctly (here: JSON)

```
const fetch = require('node-fetch');

const apiKey = 'your_api_key';
const baseUrl = 'https://api.ecommerce.com/v1';
const headers = {
  'Authorization': `Bearer ${apiKey}`,
  'Content-Type': 'application/json'
};

const productName = 'Awesome Widget';
const customer = {
  firstName: 'John',
  lastName: 'Doe',
  email: 'john.doe@example.com'
};
const quantity = 2;

async function placeOrder(productName, customer, quantity) {
  try {
    // Step 1: Get product information
    const productResponse = await fetch(`${baseUrl}/products`, { headers });

    if (productResponse.status !== 200) {
      throw new Error(`Could not fetch products. Status code: ${productResponse.status}`);
    }

    const productData = await productResponse.json();
    const product = productData.products.find(p => p.name === productName);

    if (!product) {
      throw new Error(`Product '${productName}' not found.`);
    }

    // Step 2: Create a new customer
    const customerResponse = await fetch(`${baseUrl}/customers`, {
      method: 'POST',
      headers,
      body: JSON.stringify({ customer })
    });

    if (customerResponse.status !== 201) {
      throw new Error(`Could not create customer. Status code: ${customerResponse.status}`);
    }
```



SPEAKEASY

# Integrating to an API with and without an SDK [cont'd]

```
    const customerData = await customerResponse.json();
    const customerId = customerData.customer.id;

    // Step 3: Place the order
    const orderResponse = await fetch(`${baseUrl}/orders`, {
      method: 'POST',
      headers,
      body: JSON.stringify({
        order: {
          customerId,
          items: [
            {
              productId: product.id,
              quantity
            }
          ]
        }
      })
    });

    if (orderResponse.status !== 201) {
      throw new Error(`Could not place order. Status code: ${orderResponse.status}`
    }

    console.log('Order placed successfully!');
  } catch (error) {
    console.error(`Error: ${error.message}`);
  }
}

placeOrder(productName, customer, quantity);
```

Understand order data model and construct appropriate object. Manage dependencies between APIs i.e. ensure successful customer and product information is fetched

Don't forget to handle more errors!

Finally... place the order

**The API consumer needs to construct all this code themselves**

- From reading API docs to figure out which APIs should be called
- To understanding what the response data structures should look like and which data needs to be extracted
- And how to handle auth, what error cases might arise and how to handle them...

# SDK integrations are much friendlier to your users

Now here's the SDK version of this code:

```
const { EcommerceClient } = require('ecommerce-sdk');

const apiKey = 'your_api_key';
const client = new EcommerceClient(apiKey);

const productName = 'Awesome Widget';
const customer = {
  firstName: 'John',
  lastName: 'Doe',
  email: 'john.doe@example.com'
};
const quantity = 2;

async function placeOrder(productName, customer, quantity) {
  try {
    await client.placeOrder(productName, customer, quantity);
    console.log('Order placed successfully!');
  } catch (error) {
    console.error(`Error: ${error.message}`);
  }
}

placeOrder(productName, customer, quantity);
```

Notice how much simpler and concise it is? These much shorter, standardized code blocks can be shared with every API consumer, minimizing the work required to integrate with the API.

Authentication is handled automatically with the developer just needing to copy in their key. Pre-built functions mean the developer doesn't need to parse through pages of API docs to stitch together the required calls and associated data extraction themselves. Error handling and retries are built-in.

**Overall, a far easier and superior experience.**

# How can you create SDKs and improve your developer experience?

## Create your own SDKs

Best for those with spare engineering resources, and where ultimate control over every aspect of the SDK is paramount.

**Pros:**

1. **Maximum, fine-grained control:** since the SDKs are built by hand, they can be completely customized

**Cons:**

1. **Roadmap sprawl**: Building core product is hard enough. Adding SDKs creates an explosion of work that has to be prioritized and maintained. This can pulls engineering focus and velocity away from key differentiators
2. **Organizational overhead**: you'll need a strong release process — every time the API changes, you must ensure the SDK is versioned, updated, re-published
3. **Very expensive to create and maintain**: writing SDKs requires engineering time. Don't forget that SDKs get bugs too. You'll also need to write docs for your SDKs, and may need PM resources to help prioritize / triage SDK features
4. **Need to hire experts in every supported language**: this can add up quickly if you want to support multiple languages. And these languages may not be used elsewhere in your core product
5. **Maintenance burden:** oftentimes, SDKs fall into disrepair when the original maintainer of the SDK leaves the company

## Use open-source generators

Best for hobbyists or those content to hack around rough edges. And where language idiomaticness is not a top concern.

### Pros:

1. **Lower upfront cost**: no commercial license to pay, and apparently lower engineering cost to create SDKs
2. **Can embed into an automated workflow**: with the right tooling around the open-source generators, SDKs can be automatically re-generated whenever the API spec changes

### Cons:

1. **Code quality**: SDKs are an extension of your product. A Go SDK for your API should look and feel idiomatic to a Go developer — likewise for Python, Ruby, C#, PHP, etc. The OpenAPI generator code can be noticeably uneven here, and you may need to write your own modifications.
2. **Bugs / Lack of Support:** While the OpenAPI generator has a strong and vibrant community of maintainers, be aware that there are over 3.6K open issues today — and no clear prioritization or timeline on fixes.
3. **Hidden setup and maintenance costs**: Be prepared to dedicate significant engineering time to integrate the OpenAPI generator into an automated SDK creation workflow, and to mitigate some of the issues noted above.

## Use Speakeasy

Best for companies that want a production-ready solution today, with minimal setup and maintenance cost.

### Pros:

1. **Offer SDKs today:** SDKs can be created and published in just a few minutes
2. **Idiomatic, rich, type safe SDKs:** Users love our SDKs for their language idiomaticness and "batteries-included" features like retries, pagination, and authentication. Type safety minimizes integration errors.
3. **Fully managed workflow:** Speakeasy takes care of the entire SDK workflow to save you significant time. We validate your OpenAPI spec, use AI to suggest fixes, create SDKs, and publish to package managers.
4. **Always up to date**: We automatically generate SDKs every time your spec changes - no extra work required.
5. **Fully supported:** Speakeasy handles bugs and feature requests so your engineering team doesn't have to
6. **Customizable code output**: control SDK code output via Speakeasy extensions in your spec
7. **Standards-compatible**: Speakeasy works with OpenAPI and JSON schema -- and has been battle-tested on over 4,000 APIs
8. **Generated SDKs are yours to keep**: we don't own the SDKs -- you do.

### Cons:

1. **Not 100% customizable**: Can't change every aspect of SDK code

**Want to learn more?**

- **Book a meeting**, **email us**, or **join our public Slack** today
- Read customer case studies, docs, and more at **www.speakeasyapi.dev**