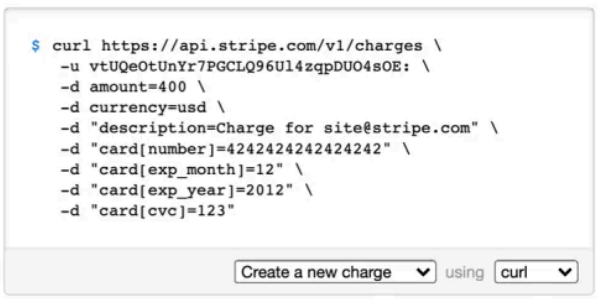
stripe

Stripe's payments APIs: The first 10 years

Michelle Bu
Payments

A few years ago, Bloomberg Businessweek published a feature story on Stripe. Four words spanned the center of the cover: "seven lines of code," suggesting that's all it took for a business to power payments on Stripe. The assertion was bold—and became a theme and meme for us.

To this day, it's not entirely clear which seven lines the article referenced. The prevailing theory is that it's the roughly seven lines of curl it took to create a `Charge`. In 2011, the code snippet featured on our landing page was nine lines long. But remove the optional `description` and `card[cvc]`, and there are visually seven lines:



A partial screenshot of Stripe.com, circa 2011. Courtesy of the Internet Archive Wayback Machine.

However, a search for *the* seven lines of code ultimately misses the point: the ability to open up a terminal, run this curl snippet, then *immediately* see a successful credit card payment *felt like* seven lines of code. It's unlikely that a developer believed a production-ready payments integration involved literally only seven lines of code. But taking something as complex as credit card processing and reducing the integration to only a few lines of code that, when run, immediately returns a successful `Charge` object is really quite magical.

Abstracting away the complexity of payments has driven the evolution of our APIs over the last decade. This post provides the context, inflection points, and conceptual frameworks behind our API design. It's the extreme exception that our approach to APIs makes the cover of a business magazine. This post shares a bit more of how we've grown around and beyond those seven lines.

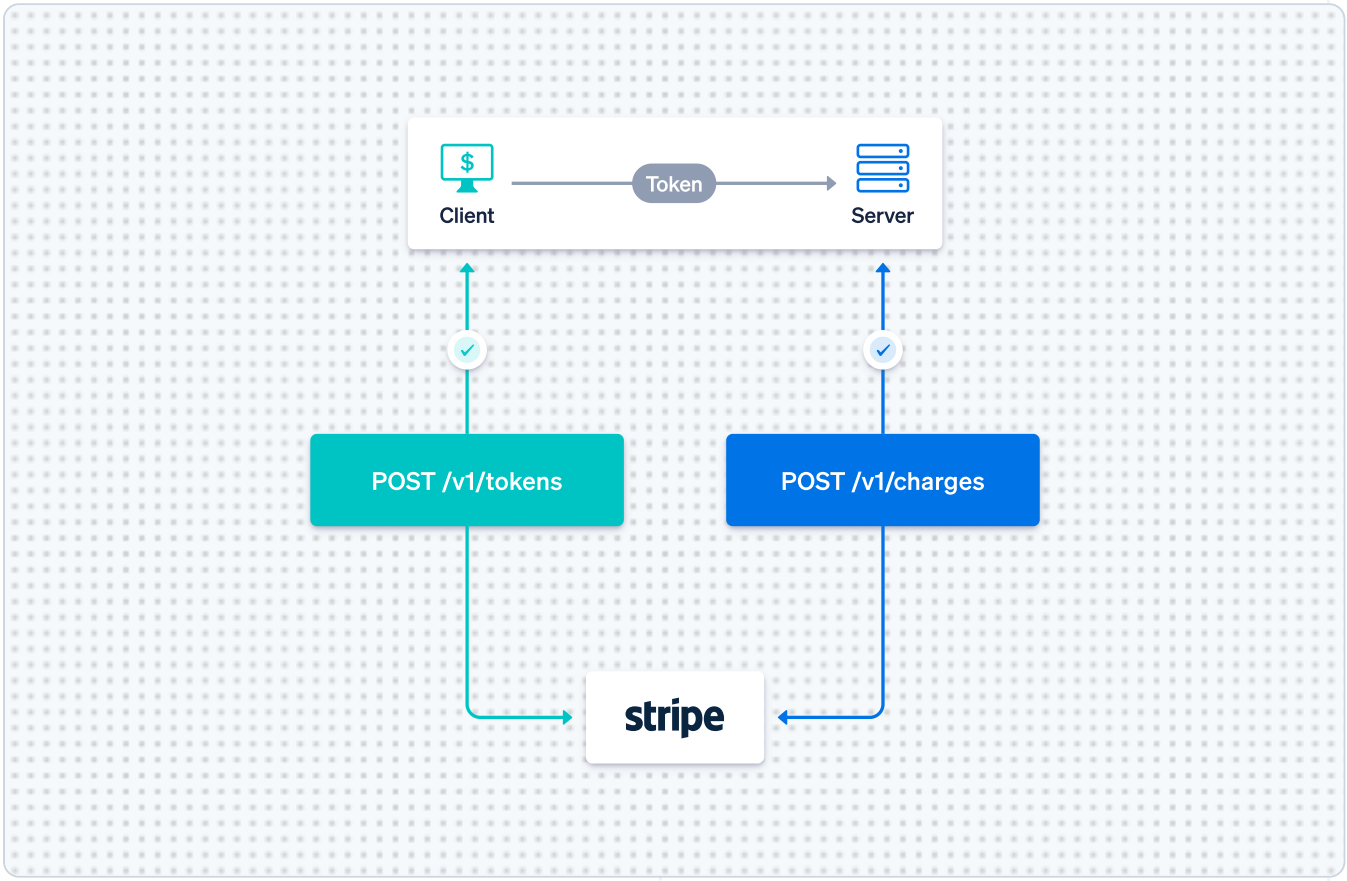
A condensed history of Stripe's payments APIs

Successful products tend to organically expand over time, resulting in product debt. Similar to tech debt, product debt accumulates gradually, making the product harder to understand for users and change for product teams. For API products, it's particularly tempting to accrue product debt because it's hard to get your users to fundamentally restructure their integration; it's much easier to get them to add a parameter or two to their existing API requests.

In retrospect, we see clearly how our APIs have evolved—and which decisions were pivotal in shaping them. Here are the milestones that defined our payments APIs and led to the PaymentIntents API.

Supporting card payments in the US (2011-2015)

We first launched the Stripe API in the US, where credit cards were—and still are—the predominant payment method. The “seven lines of code” largely sufficed, but reality was only a *tiny* bit more complicated. We also created Stripe.js, a JavaScript library to collect card payment details from the browser and securely store them with Stripe, represented as a `Token` which can later be used to create a `Charge`. This helped users avoid tedious PCI compliance requirements.



> A `Token` is created client-side and sent to the server. A `Charge` is then created server-side using that `Token`.

This payment flow follows a very common pattern in traditional web applications. The JavaScript client uses a publishable API key to create a `Token` and sends both to the server when customers submit the payment form (along with other form data about the order). The server

synchronously creates a `Charge` using that `Token` and a secret API key; orders can optionally be fulfilled based on the outcome of the payment.

The `Charge` and the `Token` became foundational concepts in our payment API.

Adding ACH and Bitcoin (2015)

When we first created `Charges` and `Tokens`, they only supported credit card payments. As we expanded to more countries and types of users, we needed to add more payment methods to the API. In 2015, we added:

- **ACH debit**, a common payment method in the US since the 1970s. ACH is used when moving money between US bank accounts, and supports both crediting and debiting bank accounts.
- **Bitcoin**, which was just gaining mindshare in the early 2010s. An increasing number of businesses were experimenting with accepting Bitcoin as a payment method.

We describe payments as "finalized" when a user has sufficient confidence the funds are guaranteed. (Of course, even finalized payments can be reversed later due to fraud or subsequent refunds.) In most cases, upon finalization, users release shipment of goods. While payments processed on card networks are initiated by the merchant and can be immediately finalized, these two payment methods are quite different from cards. Payments processed on the ACH network are finalized *days* later. With Bitcoin, customers (rather than the merchant) determine *when* a Bitcoin transaction is created. Like ACH payments, Bitcoin payments are also not finalized immediately. While the merchant will know that the customer has created the Bitcoin transaction once it is picked up by a block, it still requires 6 blocks—or about an hour—to finalize the transaction.

Payment is immediately finalized	Payment is finalized later
No customer action required	

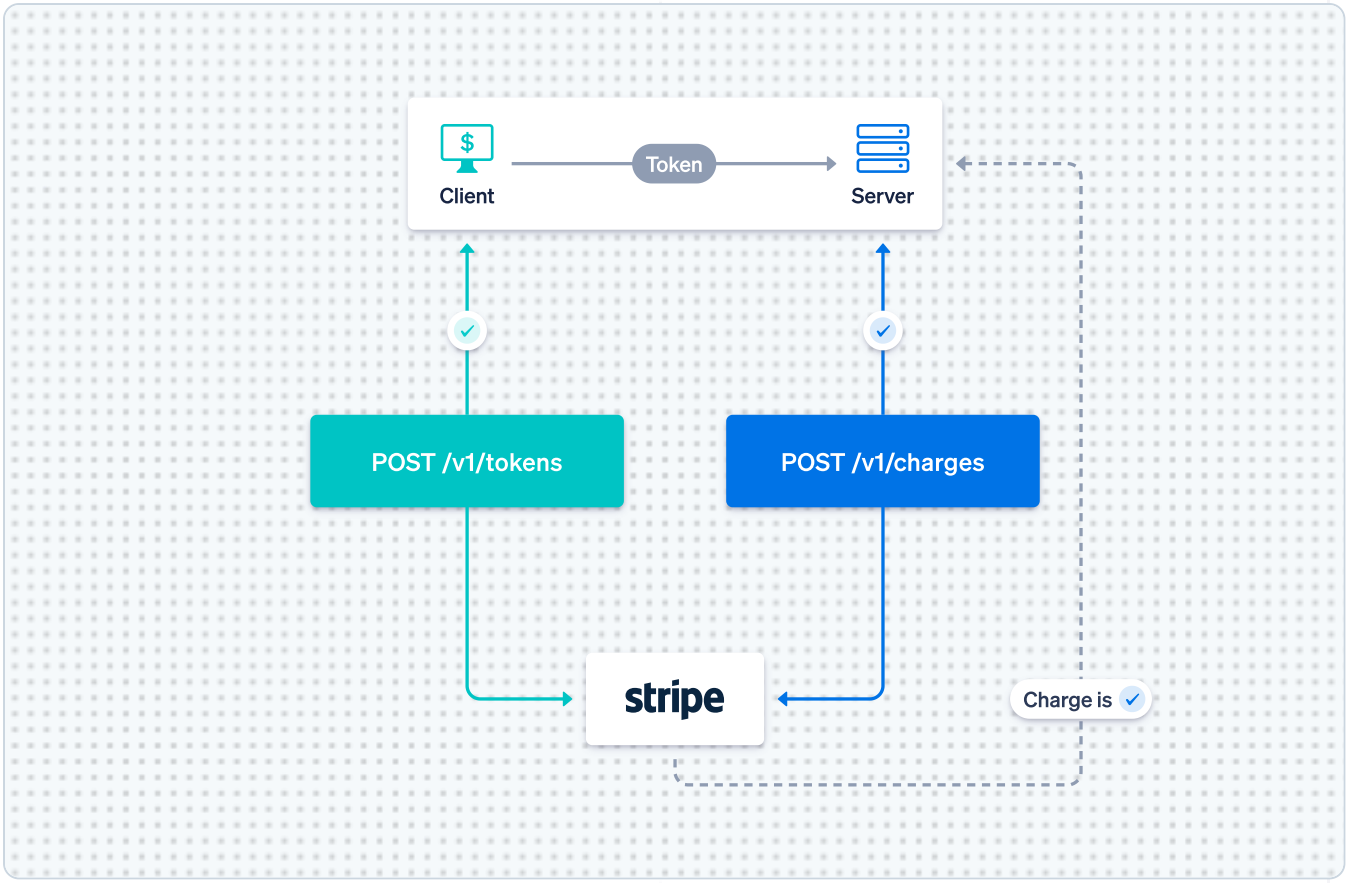
To initiate money movement	
Card	ACH debit (days) New
Customer action required To initiate money movement	
–	Bitcoin (hours) New

> The Charges API supported cards, ACH debit, and Bitcoin as payment methods.

Each of these first three payment methods differ in how the payment is initiated and when funds are guaranteed. This made the task of creating APIs that abstract over their differences quite challenging.

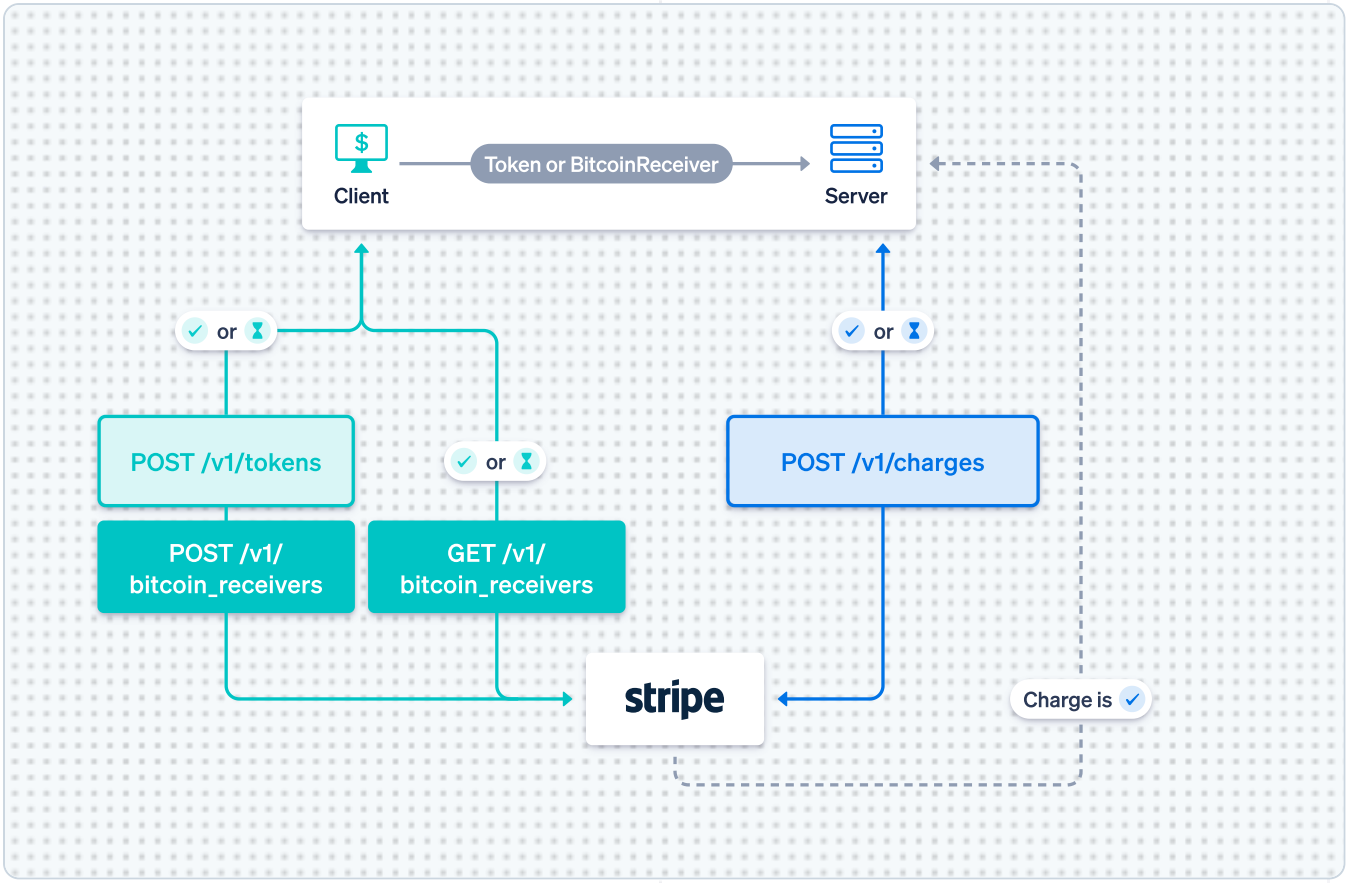
Here's what we did:

ACH debit. Since card payments and ACH debit payments both require only static information from the customer (i.e., card number or bank account number), we expanded the `Token` resource to represent both card details and bank account details. A user still created a `Charge` from either type of `Token`, but we added a `pending` state to the `Charge` to represent that an ACH debit `Charge` isn't immediately finalized and could still fail. Users ran their order fulfillment logic days later, when they received a webhook indicating that the `Charge` had succeeded.



A new `pending` state was added to the `Charge` to represent payments that finalize asynchronously.

Bitcoin. As Bitcoin didn't fit into our abstractions, we had to introduce a new `BitcoinReceiver` API to facilitate the client-side action we needed the customer to take in the online payment flow. Particular to Stripe, a "receiver" was a temporary receptacle for funds. It had a very simple state machine that described the status of the receiver: a boolean, `filled`, that was either true or false. Once the receiver was filled, the user could create a `Charge` using that `BitcoinReceiver` object instead of a `Token` object. This would virtually move the funds from the receiver to the user's balance. If a user didn't create the `Charge` within a certain time frame, the money in the receiver would be refunded to the customer. Like ACH debit Charges, Bitcoin Charges started in the `pending` state and succeeded asynchronously.



> We introduced the BitcoinReceiver resource to represent that the customer needed to take an action to complete the payment.

With ACH debit and Bitcoin, the integration grew more complex. It now involved dealing with asynchronous payment finalization, and in Bitcoin's case, it involved managing two state machines to complete payment: `BitcoinReceiver` on the client and `Charge` on the server.

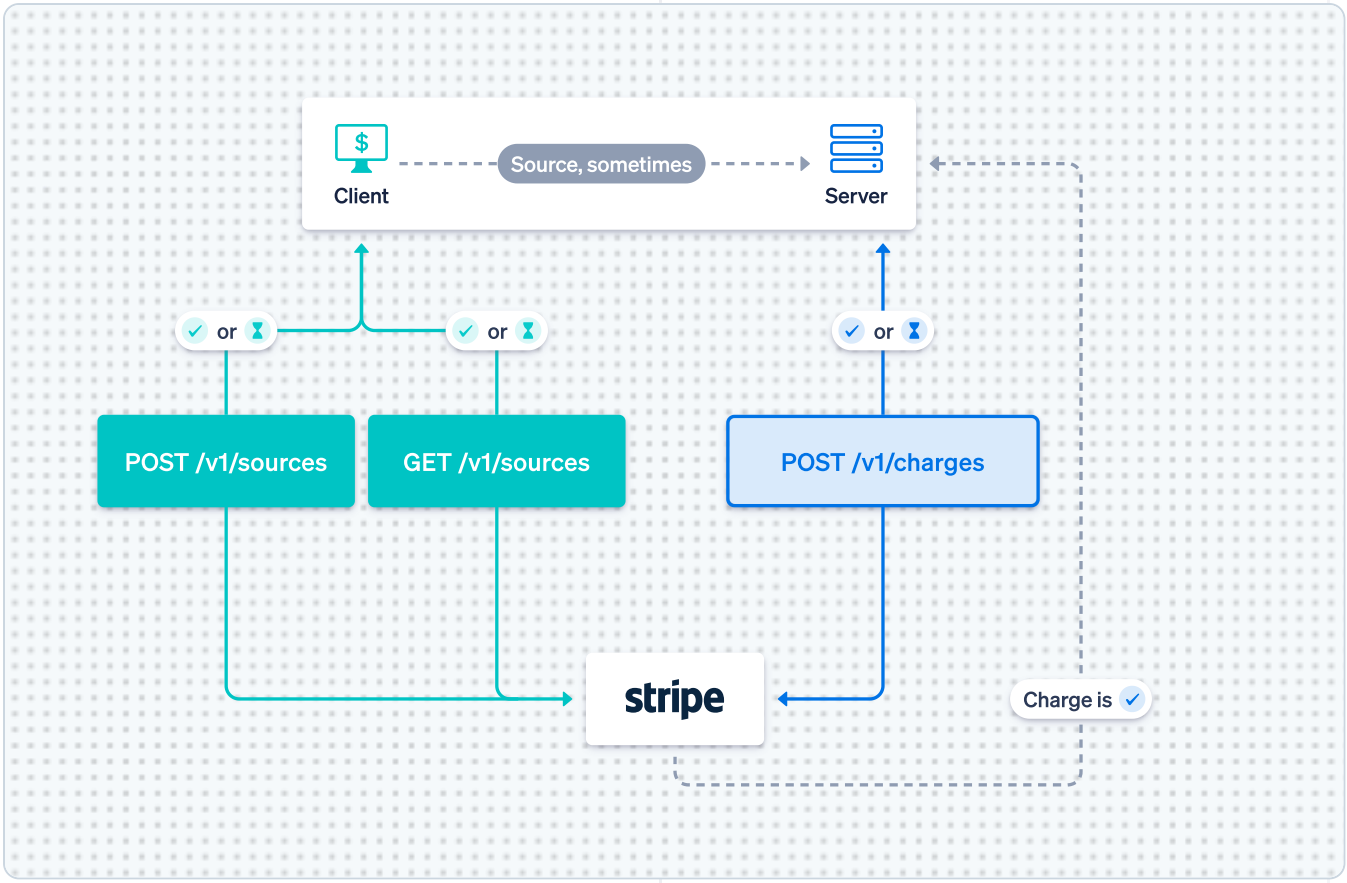
Seeking a simpler payments API (2015 - 2017)

Over the next two years, we added more payment methods. Most of them were more like Bitcoin than cards—they required customer action to initiate a payment. We discovered that it wouldn't be developer-friendly to introduce a brand new `BitcoinReceiver`-like resource for each of these—it would simply introduce too many new Stripe-specific concepts to reason about in the API. We aspired to design a simpler payments API and began exploring how to unify these payment methods on one integration path: the Sources API.

Payment is immediately finalized	Payment is finalized later
No customer action required To initiate money movement	
Cards	ACH debit SEPA direct debit New Bacs debit New BECS debit New
Customer action required To initiate money movement	
iDEAL New Alipay New giropay New Bancontact New WeChat Pay New Przelewy24 New Cards with 3D Secure New	Bitcoin Multibanco New Paper checks New

> The Sources API was designed to be a single client-side API that could represent multiple payment methods.

We combined the two client-side abstractions we'd previously designed (`Tokens` and `BitcoinReceivers`) into a client-driven state machine called a Source. Upon creation, a Source could be immediately `chargeable` (e.g., for card payments) or `pending` (e.g., for payment methods that require customer action). The server-side integration remained a single HTTP request that used a secret key to create a `Charge`.



> We combined the functionality of `Tokens` and receivers into a single client-side API: `Sources`.

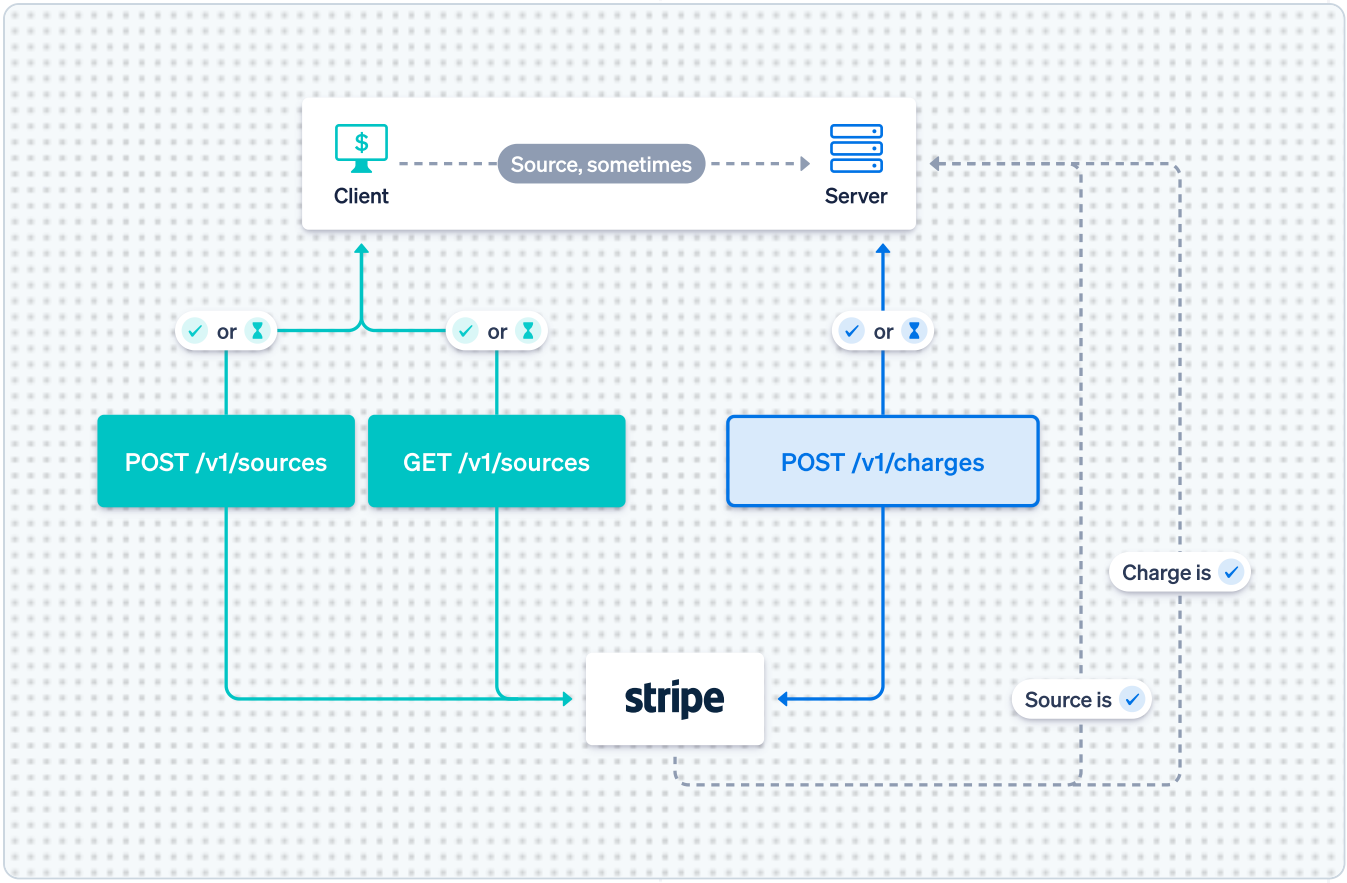
The payment flow for every payment method relied on the same two API abstractions: a `Source` and a `Charge`. This seems conceptually simple at first glance, as it resembled a card integration in the U.S. However, once we understood how this flow integrated into users' applications, we discovered many rough edges.

For example, when users added a payment method that doesn't finalize immediately, they could no longer fulfill their customers' orders immediately after the `Charge` was created. Instead, they'd have to wait until the `Charge` transitioned to `succeeded` before shipping goods. This usually involved adding a webhook integration that listens for `charge.succeeded` and moving fulfillment logic there.

`Sources` and `Charges` were still more complex for other payment methods—and integration issues could lead to lost revenue. For example, with iDEAL, the predominant payment solution in the Netherlands, the customer initiates the payment after they're redirected

to their bank's website or mobile app. If the client-side application creates a `Source` and the browser then loses connectivity with the server, the next request to create a `Charge` wouldn't make it through, even though the customer believes they paid. (The browser could lose connectivity for any number of reasons: the customer closes their tab after they pay on their bank's site, the payment method requires a redirect that the customer never returns from, or the customer has a flaky internet connection.) Because the server never created a `Charge`, we'd refund the money associated with the `Source` after a few hours. This is a conversion nightmare.

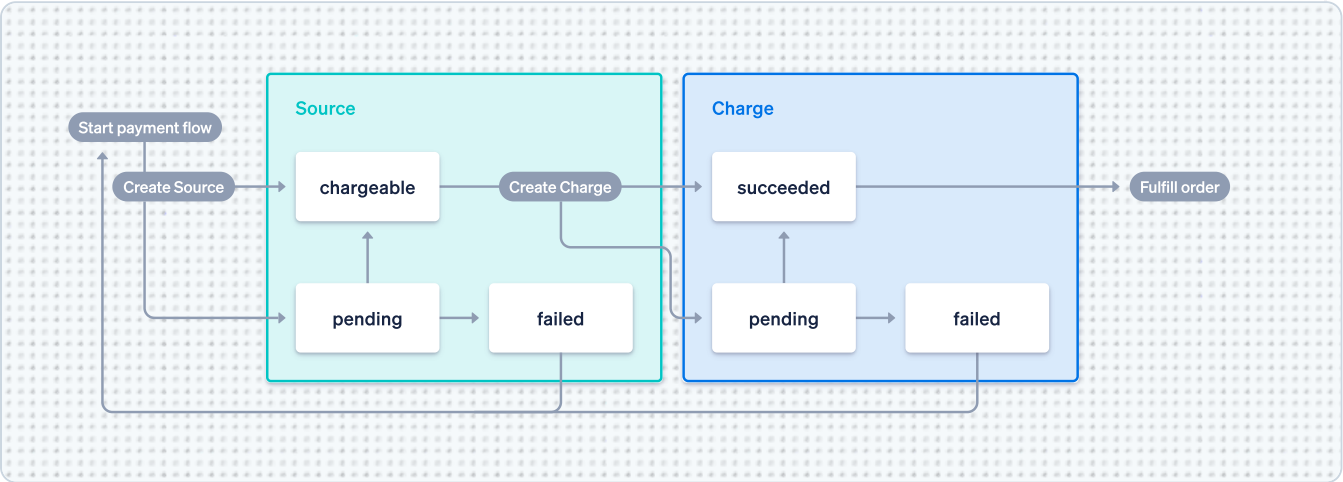
To reduce the chance of this occurring, we recommended that users either poll the Stripe API from their server until the `Source` became `chargeable` or listen for the `source.chargeable` webhook event to create the `Charge`. But, if a user's payment application goes down and they use `Sources` and `Charges`, these webhooks aren't delivered and the server won't create the `Charge`. We'll return the customer's money and users have to get them back on their site to pay again. Even if the user implements and maintains this best practice correctly, there's still complexity around the different possible states of `Sources` and `Charges` and the paths and requirements for different payment method types.



> There are many ways to actually create a `Charge` from the `Source`, depending on the payment method.

Some `Sources` —like cards and bank accounts—are *synchronously chargeable* and can be charged immediately on the server after the online payment form is submitted, while others are *asynchronous* and can only be charged hours or days later. Users often built parallel integrations using both synchronous HTTP requests and event-driven webhook handlers to support each type. This means users now have multiple places where they're creating a `Charge` and fulfilling their order. The code branching factor deepens for payment methods like OXXO, where the customer prints out a physical voucher and brings it to an OXXO store to pay for it in cash. Money is paid entirely out-of-band, making our best practice recommendation of listening for the `source.chargeable` webhook event absolutely *required* for these payment methods. Finally, users must track both the Charge ID and Source ID for each order. If two `Sources` become chargeable for the same order (e.g., the customer decides to switch their payment method mid-payment) they can ensure they don't double-charge for the order.

This effort demands more bookkeeping and conceptual understanding from developers than “seven lines of code” did. Our users *needed* to grok all of these edge cases in order to build a functioning Stripe integration. Imagine the confusion caused by reasoning about these two state machines, with varying definitions of each state depending on the payment solution. Developers must manage the success, failure, and pending states of two state machines—whose states may differ across different payment methods—in order to complete a single payment.



Users must manage two different state machines that span client and server to complete a payment.

Let’s refer back to the table of payment methods. You may notice that cards are the only payment method in the top left quadrant: they finalize immediately and don’t require customer action to complete a payment. This means we built support for new payment methods on top of a set of abstractions that were designed for the simplest payment method of them all: cards. Naturally, abstractions designed for cards were not going to be great at representing these more complex payment flows.

	Payment is immediately finalized	Payment is finalized later
No customer action required		
To initiate money movement		
	Cards	ACH debit SEPA direct debit

	Bacs debit
	BECS debit
Customer action required	
To initiate money movement	
iDEAL	Bitcoin
Alipay	Multibanco
giropay	Paper checks
Bancontact	
WeChat Pay	
Przelewy24	
Cards with 3D Secure	

> Global payment methods aren't different; cards are!

Introducing additional states and expanding on the definition of resources that were created for a specific, narrow use case resulted in a confusing integration and an overloaded set of API abstractions. It's as if we were trying to build a spaceship by adding parts to a car until it had the functionality of a spaceship: a difficult *and* likely doomed proposition. `Charges` and `Tokens` were foundational in the API because they were the first APIs we had, not because they were the right abstraction for global payments. We needed to fundamentally rethink our payments abstractions.

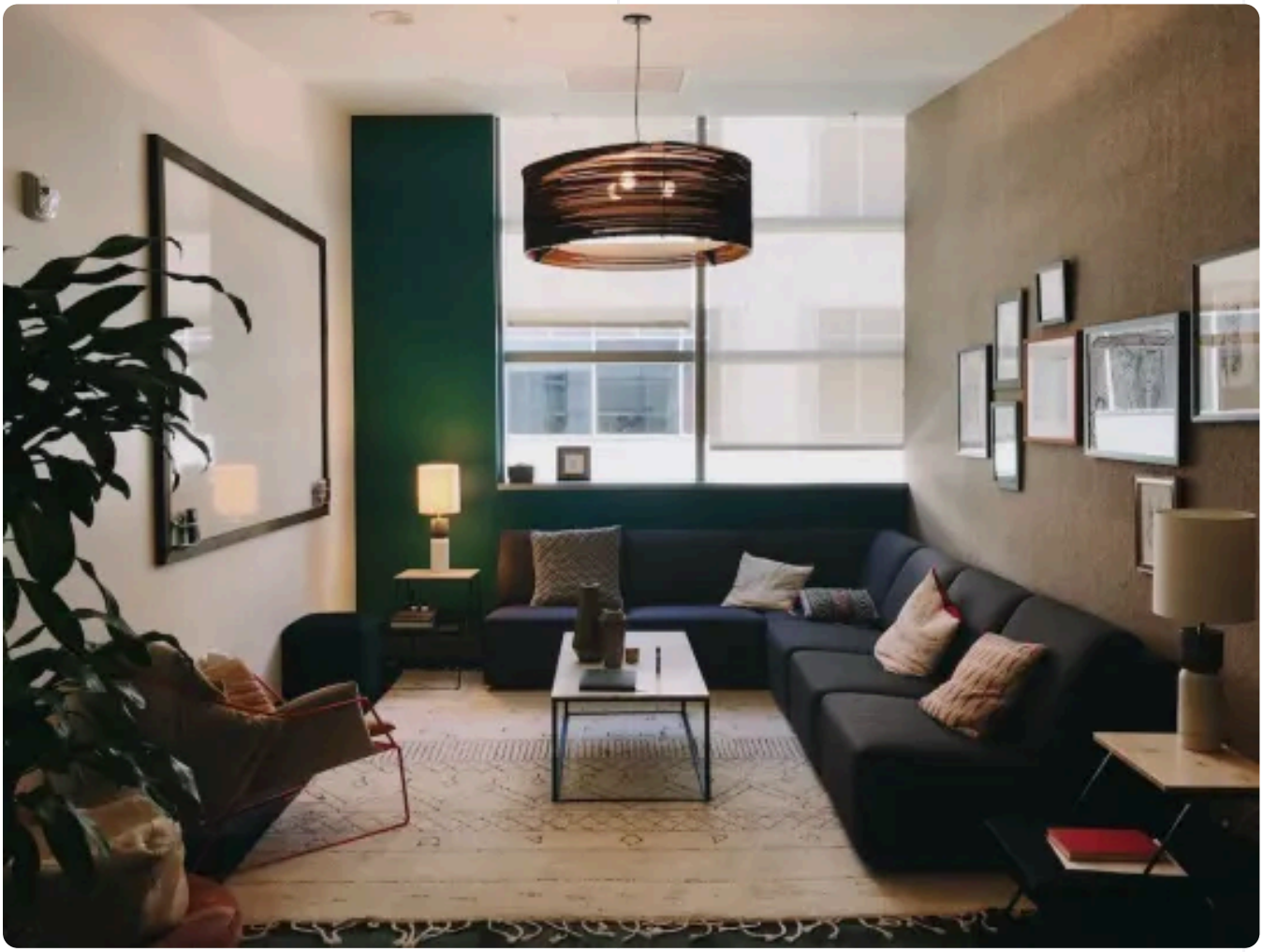
Designing a unified payments API (*late 2017 - early 2018)*

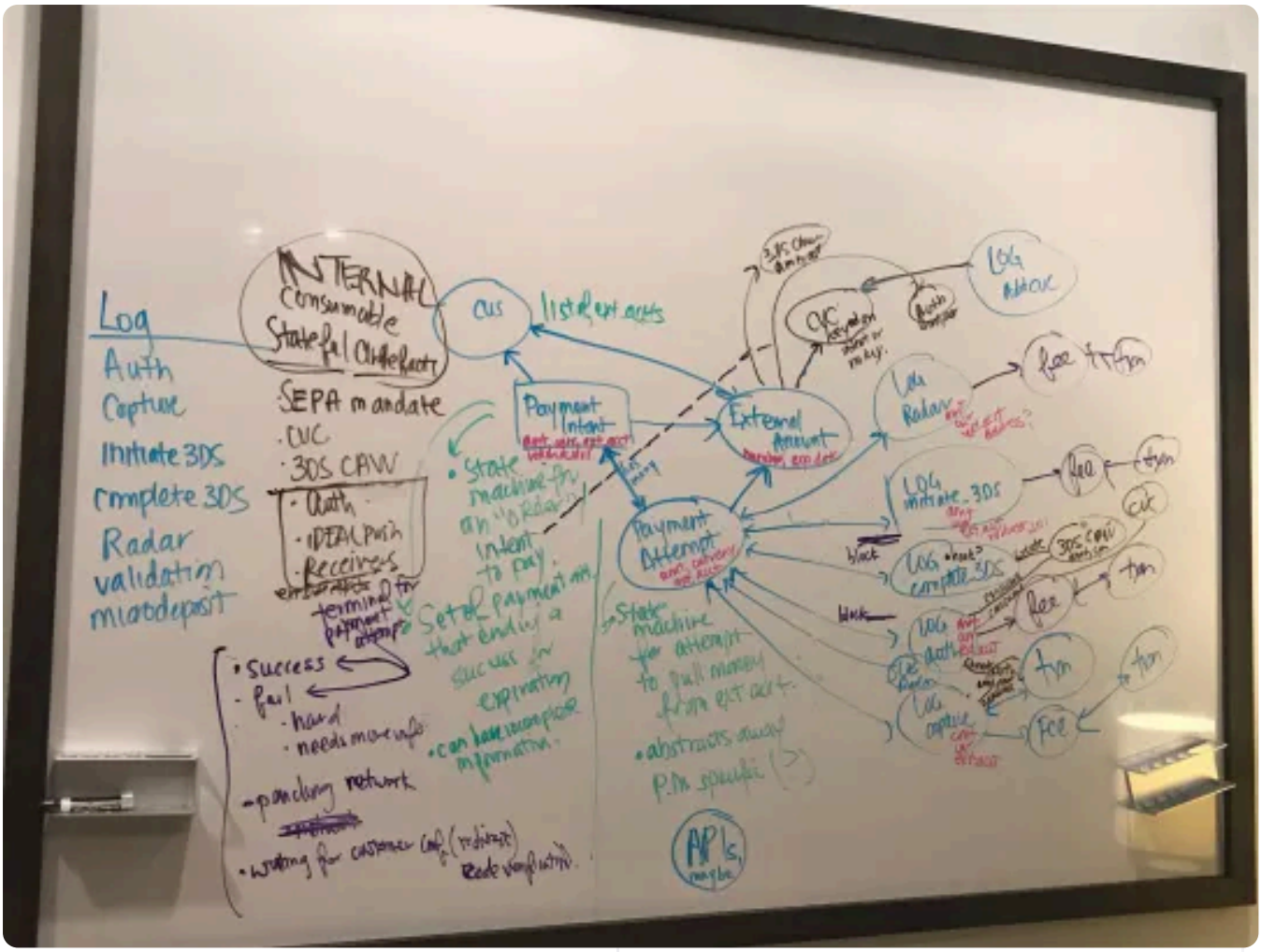
We were able to start designing the APIs we wanted when we set aside further changes to `Sources` and `Charges`. It was much easier *because* we had a chance to learn from users over the years, and deeply understood the issues they encountered with our existing integration paths. We also accumulated payments domain expertise, having had

years of experience iterating on our APIs. Taken together, our API design had a better chance to not repeat past mistakes.

We locked ourselves in a conference room for three months with the goal of designing a truly unified payments API. If successful, a developer would only need to understand a few basic concepts in order to build a payments integration. Even if they hadn't heard of the payment method, they should be able to just add a few parameters to a few specific points in their integration. To enable this, the states and guarantees of our APIs had to be extremely predictable and consistent. There shouldn't be an array of caveats and exceptions scattered throughout our docs.

A team of five people—four engineers and a PM—walked through every payment method we supported and we could imagine supporting in the future. We iterated on an API design that would be able to model all of them. We ignored all existing abstractions and thought about the problem from first principles.





We did early work on our unified payments API in a conference room named Lynx.

It's hard to remember now exactly what happened each day, but some rules and routines really helped us:

- **Close laptops**. When working together in the same room, we found the fastest way to be fully present and attentive was to close our computers. When we did, we felt more listened to and could more clearly and easily explain our reasoning to each other.
- **Pace your questions**. Start each session with a set of questions you want to answer. Write down any new questions that arise in a working session for the *next* session. Try to avoid discussing them in the moment. In the time between sessions, you'll get some distance from those questions, collect new information, and meditate more on the topic. End each session with clear answers and questions to explore in the next session.

- **Use colors and shapes**. Early on, lean on simple representations for complex, nascent concepts, rather than try to give them concrete names. We exhausted the available set of marker colors and drew many shapes on the whiteboard. This tack helped us avoid anchoring on specific definitions for the concepts that we were trying to shape—and helped us avoid naming bikesheds prematurely.
- **Focus on enabling real user integrations.** In API design, it's common to get caught up with pursuing perfect invariants, airtight theories, or intellectually pure solutions, but none of that is useful if it doesn't enable a real user integration. One of our primary design tools was writing hypothetical integration guides to validate our concepts and to make sure we didn't introduce old or new pits of failure. We wrote these for every payment method we could list—and even for some payment methods we made up, like sending cash via carrier pigeon.
- **Question *every* assumption underpinning existing APIs**. We specifically designed the first API to make card payments extremely easy, and it grew relatively organically from there. We needed to reason from first principles at every turn. Looking back, we probably could have done it even more.
- **Invite domain experts as guests.** Import know-how for discussions with a specific topic in mind. Elevate the conversation with expertise.
- **Make decisions quickly knowing you might change your mind.** New observations or data would either further reinforce our initial decision or lead us to make a better choice. In every case, it was more efficient to make *a* decision early and avoid stasis, even if we later reversed that decision.

We frequently felt like we were brute-forcing the problem space, but the enemy of any large design project is not making

decisions quickly enough because no option feels perfect.

Introducing PaymentIntents and PaymentMethods (2018)

We ended up with two new concepts: PaymentIntents and PaymentMethods. By packaging these two concepts, we finally managed to create a single integration for all payment methods.

PaymentMethods, like the original `Tokens`, represent static information about the payment method that the customer wants to use. It includes the payment scheme and the credentials needed to move money, like card information or the customer's name or email. For some methods, like Alipay, only the payment method name is required because the payment method itself handles collecting further information after you redirect to their site. Unlike a `Source`, there is no state or data specific to the particular transaction type captured on a PaymentMethod object—you can think of it as an object that specifies *how* to process a payment request.

PaymentIntents, on the other hand, capture transaction-specific data such as how much to charge and is the stateful object that tracks the customer's attempt to pay with various payment methods. Combine a PaymentMethod (the "how") and a PaymentIntent (the "what") and payment can be attempted. If one payment attempt fails, the customer can try again with a different PaymentMethod.

A PaymentIntent has the following states, summarized quickly here:

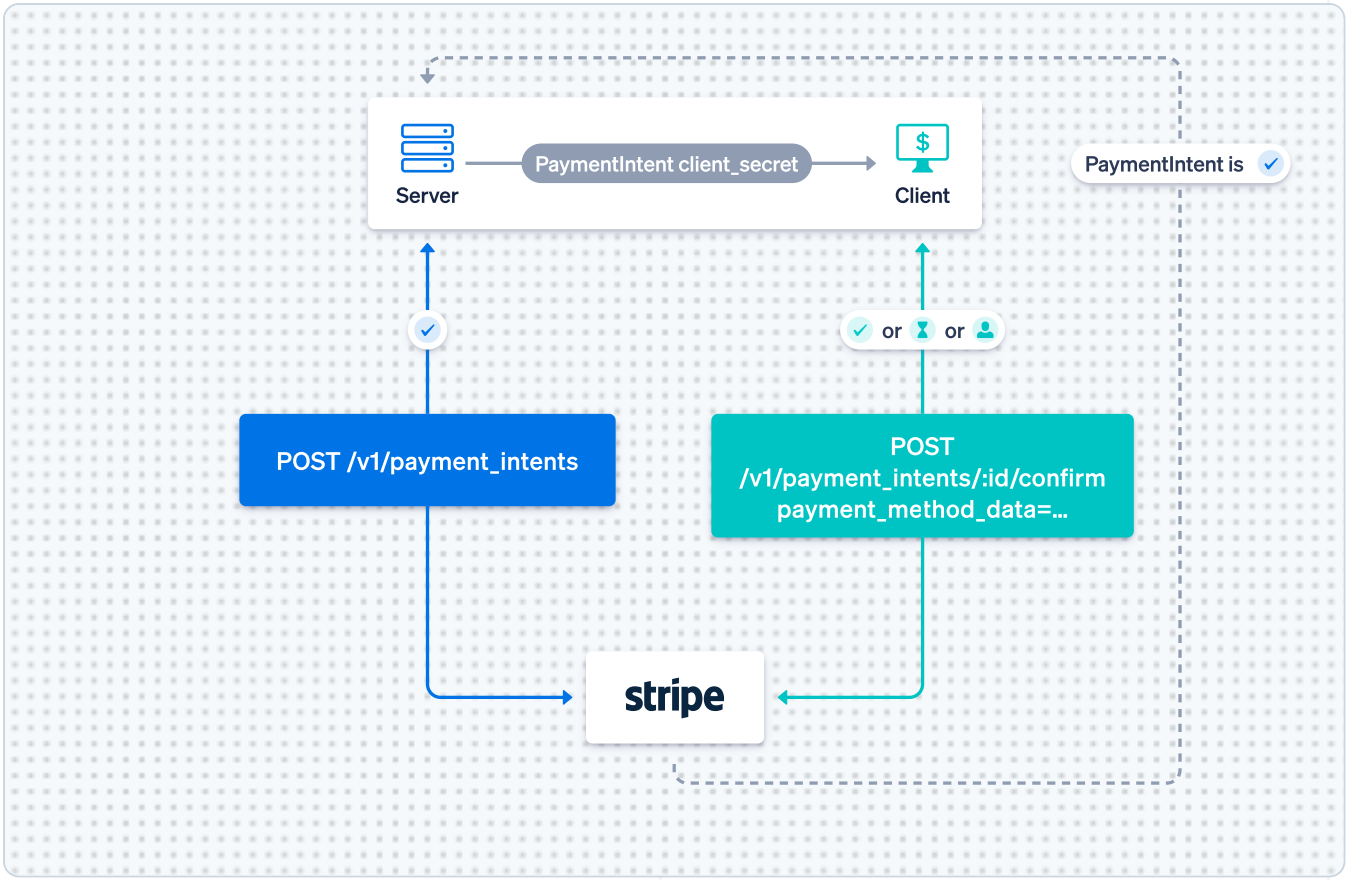
- **requires_payment_method:** Specify the PaymentMethod to use.
- **requires_confirmation:** "Confirm" basically means "make money go!" Sometimes you want to pause between collecting payment method

details and actually making the money go, and this (optional) state makes that possible.

- **requires_action:** Please perform the specified action. This can be anything from a generic `redirect_to_url` (self-explanatory) to a very payment-method-specific action like `oxxo_display_details`, which provides information for you to generate an OXXO voucher.
- **processing:** You're waiting on us to process the payment.
- **succeeded:** The payment has been finalized. Funds are guaranteed.
- **failed:** There's no failed state because if a single payment attempt fails, the PaymentIntent goes back to the `requires_payment_method` state so that the customer can try again with a different payment method. This is convenient because the same object created server-side can be used repeatedly on the client.

With `Charges` and `Sources`, a "best practice" payments integration for cards, iDEAL, and ACH debit required managing two webhook handlers (one that is time-sensitive and in the critical path to collecting money correctly), dealing with three different times a `Charge` could succeed, handling two paths to failure, and dealing with two stateful objects.

With PaymentIntents and PaymentMethods, the integration is the same across all payment method types: start by creating a PaymentIntent on your server for the amount and currency to collect for an order. Pass the secret embedded on the PaymentIntent to the client. Collect the customer's preferred payment method and confirm the PaymentIntent using the secret and payment method information. The PaymentIntent instructs what to do next when it's in the `requires_action` state. Actions are standardized and predictable per payment method; for example, the 3D Secure authentication flow is managed via a set of actions. Lastly, listen for the `payment_intent.succeeded` webhook or wait for the PaymentIntent to enter the `succeeded` state to know when funds are guaranteed and when to fulfill a customer's order. This is wholly managed by one predictable state machine. Importantly for conversion, the sole webhook handler that users must implement isn't in the critical path to collecting money.



A PaymentIntents integration.

Launching PaymentIntents and PaymentMethods (*2018 - 2020*)

The design of a set of APIs that would work across all payment gateway methods globally with a single integration was the hard but fun part. The implementation of a beta, production-ready version of the API was also relatively straightforward. But launching a new payment API that replaces a foundational, established API doesn't stop at just writing the code to spec—rolling out this change took *almost two years*.

Connecting the design to reality

Introducing a new set of abstractions to an existing public API is much harder than updating internal interfaces. No matter the size of the company, sufficient tenacity and planning can drive teams to upgrade

their dependencies. However, for an API *product*, there's no forcing developers to migrate, nor breaking their integration.

A great API product stays out of the developer's way for as long as possible.

If it is possible to make small changes to an existing API to accommodate new use cases, try that first so developers don't have to rewrite their integration. In our case, we already knew from experience that just adding more parameters and states to the existing API resources wasn't working. Even if the resource had the same name, the payment flow would look completely different.

That said, the alternative—building new, entirely independent APIs which required developers to migrate everything at once—also felt daunting. After talking to many users, we identified common patterns in their integrations. One integration *created* Stripe objects in the payment solution. Other integrations *consumed* Stripe objects for analytics, support, or reporting—potentially syncing these objects to their own database. For some users, these integrations were even owned by different teams. Given a core feature of Stripe's APIs is that developers don't have to touch their integration for years, we had to figure out a way to motivate users to migrate their payment flow. One way to do this was to make sure that any changes to the payment flow don't break their other integrations.

To accomplish this, we decided to layer over the legacy APIs and create a `Charge` object for each payment attempted by the PaymentIntent. This way, users could migrate their payment flow to the PaymentIntents API while their analytics and reporting integrations still chugged along on an unchanged `Charge` resource. (This is also a good reason to not just reuse the `Charge` abstraction with changes to conceptually behave more like PaymentIntents. Lots of users and extensions make

assumptions about what a `Charge` means, and changing its state machine drastically would break those assumptions.)

We didn't like how cluttered the `Charge` resource had become over the last seven years, so this was not ideal. Between 2011 and 2018, the `Charge` resource grew from having 11 properties to 36 properties and `Charge` creation grew from accepting 5 parameters to 14 parameters! To make sure we don't make the problem worse as we add more payment methods, we introduced payment_method_details, a polymorphic, typed hash on the `Charge` that contains payment-method-specific data. This approach helps us keep the top-level `Charge` resource simple, while making payment details easy to find and identify for details such as a partner reference ID or a payment-method-specific verification status:

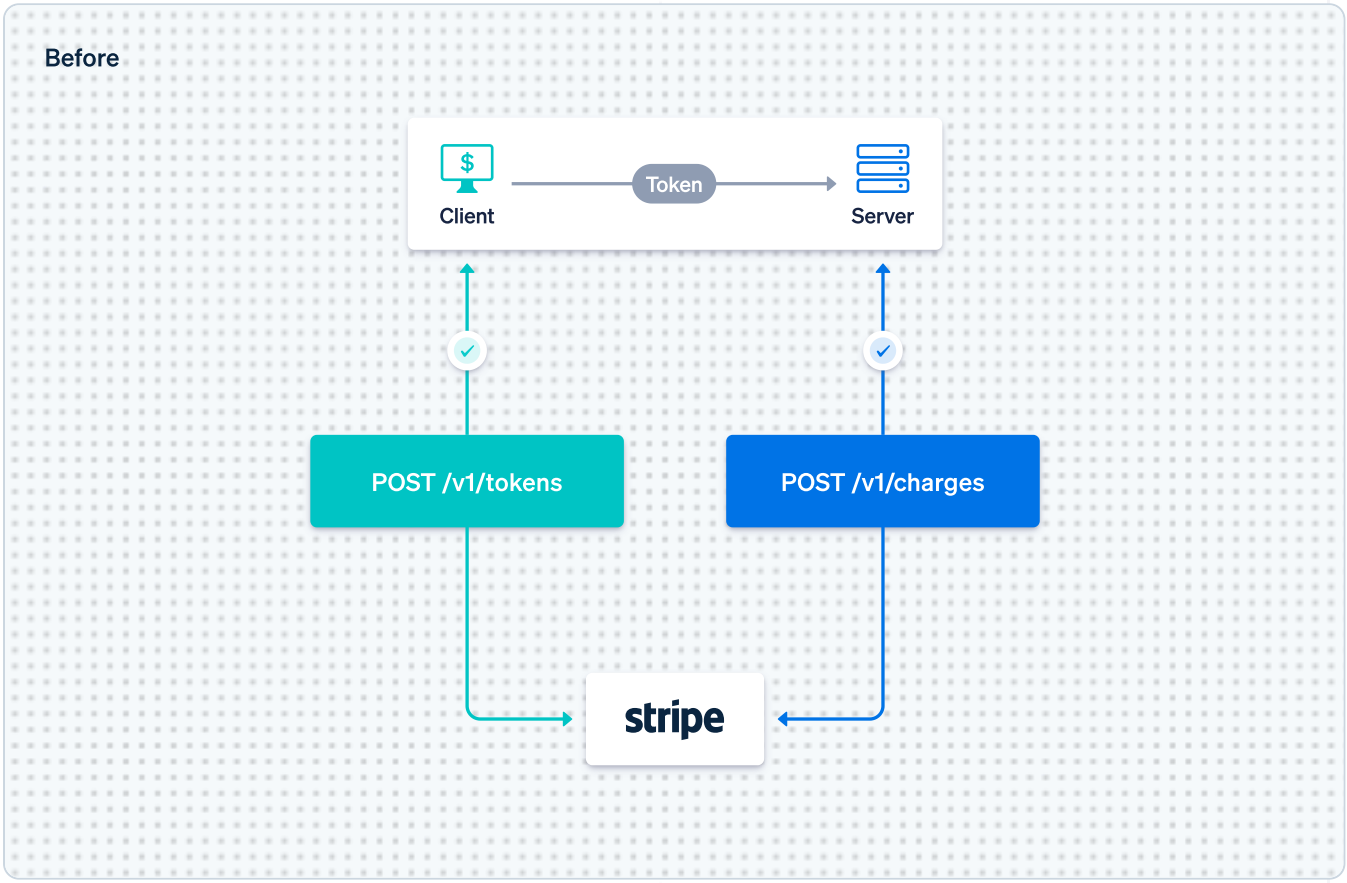
```
{
  payment_method_details: {
    type: PaymentMethodType,
    [PaymentMethodType]: {
      // Payment-method-specific details about the transaction.
      // For cards, maybe it's the CVC verification information.
      // For OXXO, maybe it's the voucher information.
    }
  }
}
```

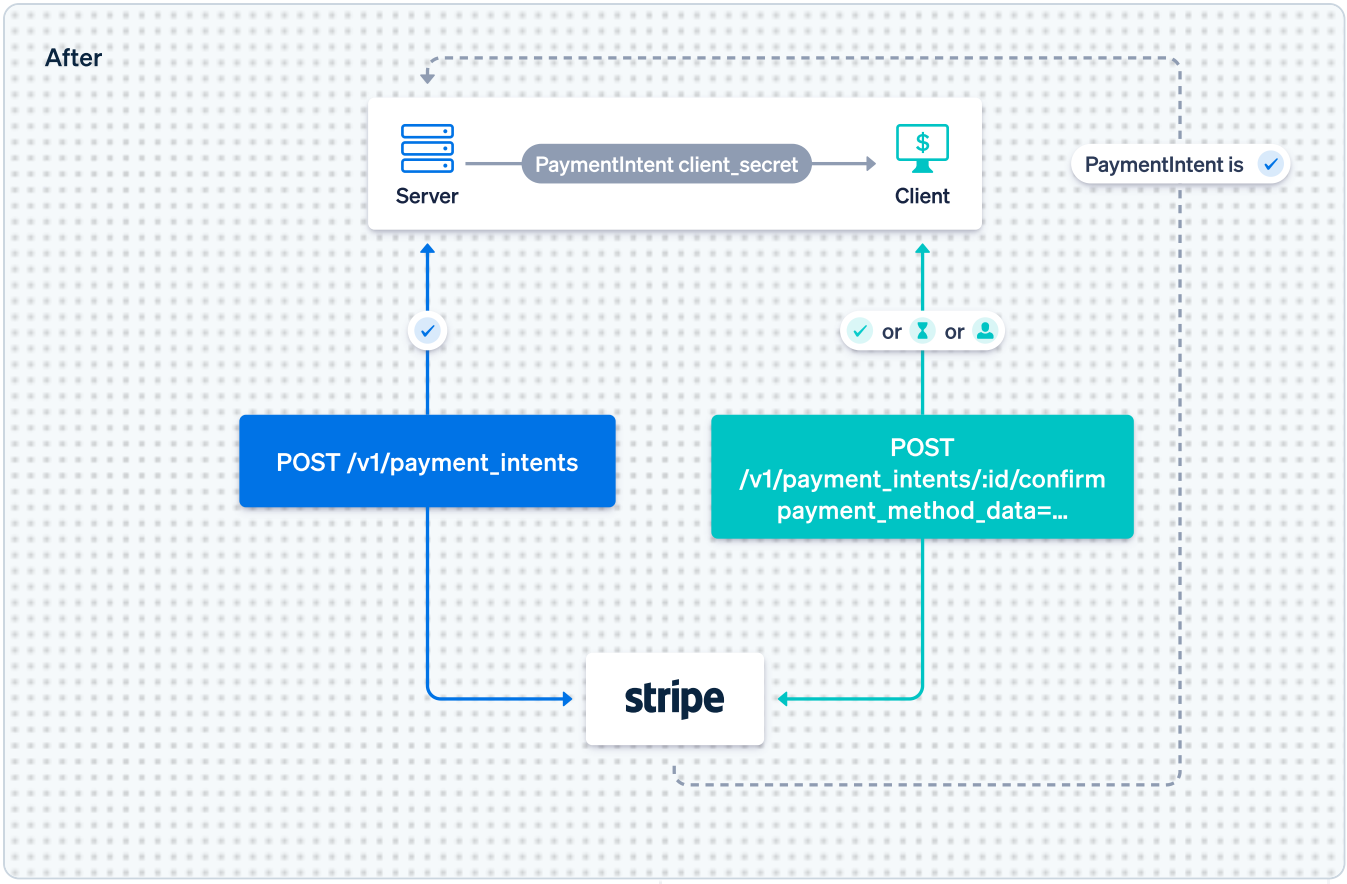
Over time, we've standardized this design pattern and have applied it to other resources in the API.

Layering over the Charges API is just one example of a design compromise we had to make for the sake of migration. There were many other smaller challenges, but ultimately they all had *some* least-bad solution we could pursue, so it wasn't too dire. The *hardest* part of realizing the PaymentIntent migration was not a technical challenge, but a perception challenge: The new APIs didn't feel like "seven lines of code" anymore.

Keep it simple, Stripe

In normalizing the API across all payment methods, card payments became more complicated to integrate by introducing webhook events and by flipping the order of the client and server requests in the payment flow. These choices are not intuitive for those familiar with card payments, nor are they easy to implement for developers building traditional web applications.





Compared to a simple card payments integration on Charges, a PaymentIntents integration requires flipping the client and server API calls and dealing with a webhook.

This change to card payments was a challenge for one of our most important types of users: the eager developer at a startup who wants to get up and running with card payments for checkout as soon as possible. Before, their seven lines of code pasted in a terminal would result in a successful charge. This new payment processing flow relies on asynchronous events, so the magic becomes much less tangible.

PaymentIntents is also objectively a harder integration for users who *only* care about accepting card payments in the US and Canada. We flipped the order of the client and server calls, which is difficult for traditional web applications to handle, and webhooks are often more than a little bit annoying to set up, test, and debug. (We later developed the Stripe CLI to make developing with webhooks simpler for users.)

The power-to-effort curve looks different between the Charges integration and the new PaymentIntents integration. Each incremental PaymentMethod is cheap to add to a PaymentIntents integration. However, speed is key for startups who want to get started quickly. With

`Charges`, getting cards running was intuitive and low-effort—a compelling combination for startups.

A PaymentIntents integration requires more effort up front, but each incremental payment method requires little incremental work to understand and add. On the other hand, a Charges integration is very low-effort for cards in the US and Canada, but becomes tedious and unpredictable for each subsequent payment method.

Our first attempt at launching PaymentIntents without overwhelming existing users was to show both the PaymentIntents and Charges integration guides in our documentation, switching which one we showed first depending on the user's location. The idea was that *most* users in the US did not need these non-card payment methods, and thus would feel overwhelmed by the idea of payments as a state machine. In reality, this branching between two completely different integrations was *tremendously* confusing.

Many US businesses *do* want to go global, and folks aren't always coding from the locale of the business they want to run. If a developer for a EU-based business ended up following the Charges integration guide, they'd eventually realize that they would have to start from scratch. This happened a few times, and was always a costly and painful experience. It was not user-centric thinking to assuage our own worries about this big API change by recommending two incompatible integration paths.

Our ultimate solution to this problem was to add a *convenient packaging* of the API that caters to the hypothetical user that would turn away from our APIs if they had to use webhooks up front. We called the default integration the "global payments integration" and named the new integration "card payments without bank authentication." We put the implications of this integration front and center in the documentation: with this simpler flow, you won't be able to easily add new payment methods.

The way this conceptual packaging actually manifests in the API is a special parameter called `error_on_requires_action`. This parameter tells the PaymentIntent to error if further action is required to complete the payment. A user who wants a simple payment flow like `Charges` won't be able handle any actions required by the PaymentIntent state machine.

```
# Our packaging made PaymentIntents seven lines of code.
curl https://api.stripe.com/v1/payment_intents \
  -u sk_test_xxx: \
  -d amount=1099 \
  -d currency=usd \
  -d confirm=true \
  -d payment_method="{{PAYMENT_METHOD_ID}}" \
  -d error_on_requires_action=true
```

The parameter name makes it *very* clear what users are choosing. Additionally, this approach allows us to easily track how often users choose this integration path, which would not be possible if we'd just recommended that U.S. users *ignore* PaymentIntent states they couldn't handle. Someday that eager developer will have the time to build out a webhooks integration or will need to add a new payment method. When that day comes, it's clear what they need to do: remove the parameter from the integration to start handling the `requires_action` state. Developers using this *packaging* of PaymentIntents don't have to change the core resources at play, even when they upgrade to the global integration.

> Our simple packaging of PaymentIntents for U.S. and Canadian card payments requires the same amount of effort to integrate as `Charges`.

With this packaging, we were able to provide a low-effort integration similar to `Charges` for users who had no interest in doing a global-payments-ready integration up front.

Keeping things simple doesn't just mean reducing the number of resources or parameters.

Two overloaded API abstractions are not simpler and are definitely not more flexible and powerful than three or four clearly-defined abstractions. Keeping things simple means making sure your APIs are consistent and predictable—and that you're creating the right packages to gradually reveal the power of your API as your users need it. It also means not underestimating your user. It's tempting to abstract away too much in service of "keeping things simple," but users will often quickly discover that they need more control.

An API product is more than just the API

There has—and will always be—many lines of code propping up the vaunted “seven lines of code.” It’s reliably the case with APIs. They don’t happen without a lot of work that isn’t designing or building the actual API. Much of the effort required is unglamorous and tedious, like tracking down every piece of documentation, support article, and canned response that references the old APIs, reaching out to folks who have made community content and asking them to update it, and planning and recording many tutorials for users and user-facing teams.

There’s also the teams that appear on the periphery, but are instrumental in the success of APIs. There’s the documentation and developer products that supplement the integration experience. Stripe CLI’s launch made webhooks much less daunting. A redesign of the information architecture of our documentation made relevant guides easier to find. Stripe Samples allows developers who prefer to learn by example rather than prose to just start with some working code. A redesign of the payments view in the Stripe Dashboard allows developers to more easily debug and understand the PaymentIntent state machine.

The care, choices, and effort of Stripes past and present from across the company contributed to our most recent two-year effort to design and launch our new payments APIs. The more we grow, the more we realize that we must continue to build and rebuild deliberately and thoughtfully. These are still early days. Come join us.