Prismatic

The Beginner's Guide to

API Integrations

Table of Contents

APIs and API integrations are everywhere in modern business. This is true whether you work in a business that's integrating internal systems or a SaaS company providing integrations to your customers. (Or perhaps you are one of those customers.)

And you don't have to be a software developer or IT professional to hear the words ***API*** and ***API integration*** floating around. Or ***webhooks***, ***data mapping***, ***JSON***, and the list goes on. Today more than ever, many of us need to be least conversational in the language of integrations to do our jobs well.

Especially in a software company, regardless of your role (product, sales, partnerships, success, and of course, engineering), API integrations have a bearing on what you do and how you do it.

This guide provides a framework for discussing API integrations with your customers, partners, developers, and everyone else in your company.

What is an API integration?

An API integration is the software functionality that allows one system to transfer data to another using one or more APIs (**a**pplication **p**rogramming **i**nterfaces). It is impossible to imagine modern business processes without the millions of API integrations transferring data behind the scenes.

API integrations solve business needs

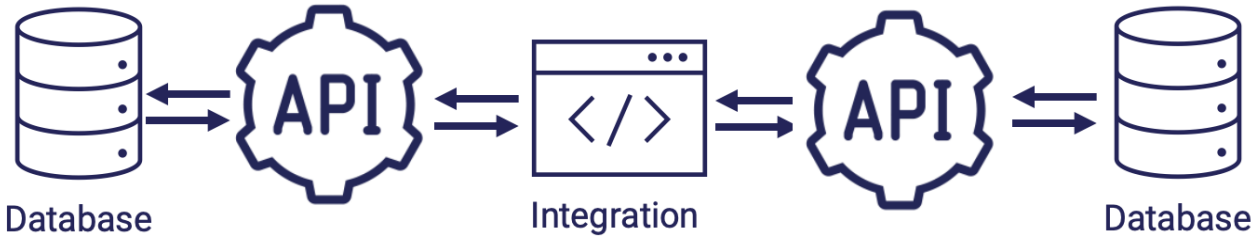
Every integration is different. Some integrations don't use APIs but may instead work with a file on a file share or something else. However, API integrations are the most common integration pattern we see used, partly because APIs make building integrations far simpler than would otherwise be the case.

An API integration is the technical answer to a business need, such as making data from one system available to another, preventing duplicate data, or automating workflows. That said, the pieces and parts of an API integration matter to the extent that they allow us to solve that underlying business need correctly.

To say that a lot is going on within an API integration is an understatement. API integrations often appear simple, but things can become complex quickly once we dig into what needs to happen.

API integrations from 30,000 feet

Before we get deeper into API integration meaning, let's first take a brief look at the big picture. Here's what an API integration might look like at the highest level:



The example API integration shown above uses an API to connect to the databases for each involved system (or application). In addition, this API integration shows data being transmitted both ways via the integration, usually termed a two-way integration. However, some API integrations may have an API on just one end of the integration and may only transmit data in one direction. Sometimes, we may even have an API data integration with three or more APIs. The above example is standard enough to be helpful as we continue through this guide.

In the diagram, we show each API connecting with a database. The API, itself an application, does not need to access the system via the UI as human users would. Instead, the API accesses the database since the entire point of an integration is to transfer data between systems – and that data usually resides in databases.

There is much more happening in an integration than shown in the example. In the following sections, we'll take the databases, APIs, and the integration itself and dig into the functionality for each.

Why do API integrations need databases?

Databases aren't exactly part of API integrations, but we've included them for context. After all, an API without an attached database is of little value to an API data integration.

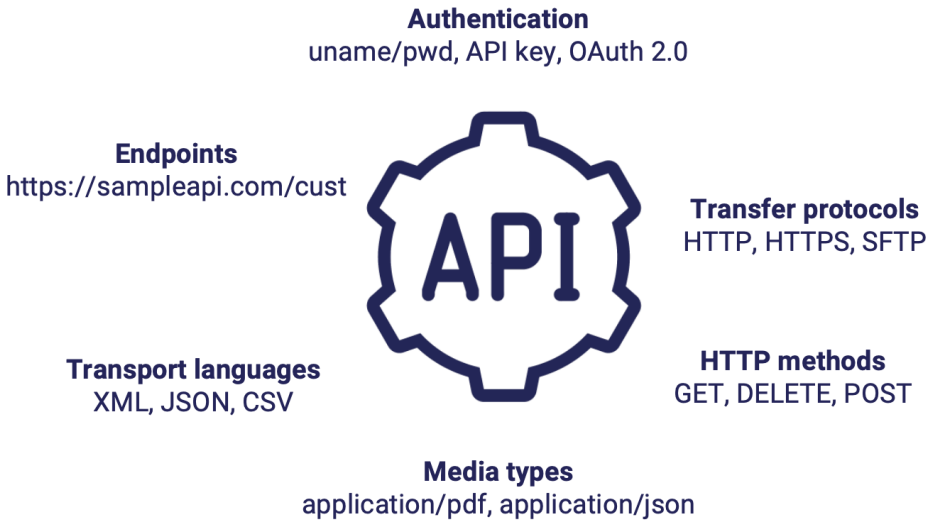
The relationship between the API and database is defined before we bring the integration into the picture. As part of that relationship, the database restricts the API regarding which records it can work with, what it can do with them

(create, read, update, and delete), the database views it can use, and everything else defined for a database user. Then, the API further restricts the integration regarding what it can to. Here's what that looks like:

APIs are data doorways

An API sits between a database and an integration to facilitate data transfers. For API integrations, it may be simplest to think of the API as a doorway to the database. Some APIs only permit data to be read from the underlying database, while others allow new information to be written. Common types of APIs are REST, SOAP, XML-RPC, and GraphQL. In addition, we have webhooks, often called reverse APIs, because they push the data rather than wait for it to be pulled – as is usually the process with APIs.

Technically, the API is what the integration communicates with. Practically, however, we can't separate the API from the integration because each API comes with a set of constraints or rules that we need to follow when building an API integration. Though we can do many things within an API integration, we cannot override those rules. Here are the things constrained by an API:



Auth (authentication) controls access to the API

Auth is the process of verifying the identity of a user requesting access to an API or other software. Auth is how an API ensures the request comes from a legitimate requester. The requester may be a human user or a system. Most API integrations use common auth types or some variation of them. Common auth types are as follows:

- ✓ **Basic authentication** uses the classic username and password approach. This is no longer common in modern SaaS apps, but we still see it for many legacy systems, including FTP or older HTTP-based apps.

- ✓ **API key** auth methods are the original type used with APIs. An API key is a single string used both for identification and authentication. Auth based on API keys is often referred to as token-based auth and is common in modern SaaS apps.

- ✓ **OAuth 2.0** is ubiquitous. In general, it is set up so that a user clicks a button in App A, and App A sends the user over to App B to ask if the user wishes to enable sharing of something with App A. The user clicks the button to agree to this data sharing, and App A is granted permission to access App B on the user's behalf.

Transfer protocols keep data moving

These are network protocols that support transferring data between systems. Think of transfer protocols as moving sidewalks. The data sent from or received by an API is moved from the integration to the API through a transfer protocol. The API defines transfer protocols. Most APIs support a single transport protocol, but some are configured for multiple transfer protocols.

Almost any application layer network protocol could be used for an API integration. But, HTTP and HTTPS are the transfer protocols best suited for the task, so they are used most of the time.

For transferring files (instead of data collections), we might use FTP, SFTP, or FTPS. However, these protocols are not generally used with APIs. However, it is possible to have an integration connecting to an API on one end with HTTPS while using SFTP to connect to an FTP server on the other end of the integration.

HTTP methods define supported actions

HTTP methods (or HTTP verbs) are the specific commands that an API data integration can use to interact with an API that uses HTTP or HTTPS as the transfer protocol. As with the other things we discuss in this section, the API defines acceptable HTTP methods.

In general, HTTP methods are dependent on the type of API. For example, most REST APIs accept the same HTTP methods (`GET`, `DELETE`, `PUT`, `PATCH`, and `POST`). But most RPC or SOAP APIs only implement `POST`. GraphQL APIs, as a rule, can support both `POST` and `GET`. However, most integrations with GraphQL APIs use `POST` because `GET` requests can grow too large to work correctly.

Here's an example of a HTTP method for a specific endpoint:

```
HTTP GET https://www.samplapi.com/customer/89344
```

Transport languages are the medium of data exchange

Transport languages (aka data exchange formats or data interchange formats) describe the data transferred between systems. A transport language can also be referred to as an interim data format since the source system and receiving system store the data in formats (in a database, usually) that are different from the transport language.

The most common transport languages for API integrations are XML and JSON. Formatting, tags, and syntax are the most obvious ways in which transport languages differ.

Here are a few lines in XML:

```
<task>
  <task_ID>438983</task_ID>
</task>
```

Here is the same data, but in JSON:

```
{
  “task”: {
    “task_ID”: 438983
  }
}
```

The bottom line with transport languages is that an API is set up to use at least one of them. It may use one (or more) for imports and another for exports, or it may use the same one for both. If an API receives a request in an unsupported transport language, the API returns an error (since it doesn't know how to process the request).

Media types identify files and other data collections?

These indicate the nature and format of a document, file, or assortment of bytes. Formerly called MIME types, they are usually included in HTTP headers. When data is sent from an integration to an API that uses HTTP (or HTTPS) as the transfer protocol, we'll want to include the media type.

Media types describe the transport language and any binary (non-human readable) files encoded with the data transfer. Since there are hundreds of media types for binary encoding, explicitly setting media types lets the API know what to do when receiving the encoded data.

A media type is comprised of a type/subtype. For example, the media type for the JSON data format is `application/json`, and the one for a PDF file is `application/pdf`.

Endpoints are where APIs and integrations interact

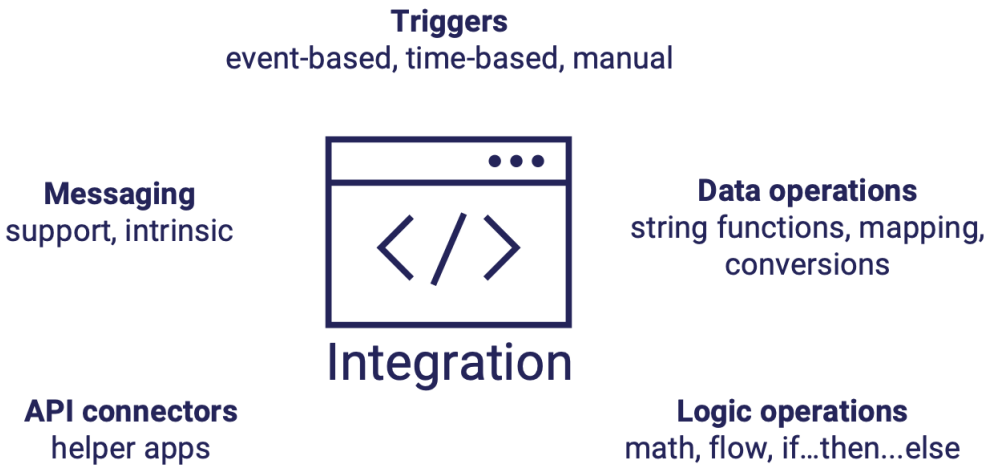
An endpoint is where requests made to an API or webhook are fulfilled. Sometimes, as with REST APIs, a single API has many different endpoints, each for working with a different record type. For example, to list all customers, the integration might call `https://sampleapi.com/customers`. But, to retrieve the details of a single customer record, the integration might instead call `https://sampleapi.com/customers/{cust-id}`.

Other types of APIs, such as GraphQL APIs, have a single endpoint for everything. For example, we might connect an integration with `https://mygraphqlapi.com/` to access all data available via the API.

For most API integrations, the integration sends a request to an endpoint to send or receive data. For webhooks (aka reverse APIs), the integration doesn't send a request to the endpoint but waits for data to be sent from the webhook.

API integrations have a lot going on

Now that we've looked at common rules for APIs and how they constrain API integrations, we'll dig into the integration proper. An integration is an application, often comprised of several modules, that connects systems for data exchange. As a result, it can encompass just about anything that can be done with code. That said, what happens in integrations can be grouped as follows:



Triggers get things started

A trigger is what tells the integration to run. Triggers are usually event-based or time-based (scheduled), though other types of triggers can exist. It's not uncommon for triggers to also be set up so they can be invoked manually (in case of a network outage, need for troubleshooting, or something else).

Event-based triggers are usually tied into webhooks. For example, an order record is created in System A. This sends data to a webhook which the integration is watching, letting it know that the integration should run.

Time-based triggers cause an integration to run at a certain time of the day, week, month or after a certain amount of time has elapsed. For example, a time-based trigger can be set up to run an integration every Thursday at 06:45 local time, or it could be set to run every 2.5 hours.

API connectors are the go-betweens

An API connector is the functionality needed to connect an integration to an API. API connectors are generally built to run on an enterprise iPaaS or embedded iPaaS (more later on these tools that help you build integrations more easily, without writing code from scratch). However, even one-off integrations that are coded from scratch and don't run on an enterprise iPaaS or embedded iPaaS have code equivalent to the API connector.

If the API is the locked door to a building, the API connector is the person who inserts the (auth) key, opens the door (to access the endpoints), and deposits or picks up packages (using actions). Auth, endpoints, and actions are defined or constrained by the API, but the API connector uses them.

For those integrations with multiple APIs, the integration includes an API connector for each.

Messaging keeps everyone in the loop

Messaging is a crucial capability of most integrations. This messaging (also called notifications) can be intrinsic to the integration and its actions or driven by the framework that supports the integration.

- ✓ **Intrinsic messaging** includes notifications that are built directly into integrations. For example, an integration may have a notification built into it to send an email, SMS, or Slack message if an order contains more than 1000 items. This might be done to ensure that the order fulfillment team gets a bit of warning that they've got a larger-than-average order to fulfill.

- ✓ **Support messaging** includes any notification sent by the system running the API integration because something worked or something failed to work. For example, if an integration is supposed to run every Tuesday at 14:15 local time and fails to run, a support notification might be sent to IT or support personnel. In the same way, a manager might be set up to receive a message when an API integration runs successfully.

Much like the functionality in an integration, the number and types of API integration notifications can vary tremendously. In general, we need enough messaging to ensure that no one who relies on the integration is left wondering what state the integration is in or why something didn't occur as expected.

Data operations handle system differences

The primary function of an API data integration is to transfer data from one system to another. However, because integrations may need to work with data from different databases, using differing schemas, and encoded in various formats, the API integration may need to perform any number of operations on the data it receives before it is ready to send.

Data operations include text manipulation, such as string joins or splits, text case changes, regex matching, and find and replace functions.

One of the simplest text manipulations is the string join. Let's say that in the sending system for the API integration, a user has a first name `user_fname` and a last name `user_lname`. The destination system needs `fullname`. We'll use text manipulation in the integration to join `user_fname`, a space, and `user_lname` to get `fullname` for the destination system.

Another very common data operation is called data mapping. We see this used all the time for integrations – because data schemas in separate systems are almost always different. Data mapping allows us to take a data element from the source system, such as account_number, and map it to a data element with a non-matching name in the receiving system, such as acctno. We can do this with a single data element or many of them.

And, also in the realm of data operations, we have data format conversions. We touched on transport languages a bit earlier. One of the things that we might need to do within an integration is to convert between transport languages.

Here's what an example of exported XML data looks like from the sending system:

```
<task>
  <task_ID>438983</task_ID>
  <taskowner>Jeong Kim</taskowner>
  <project>Oak Valley Factory</project>
  <type>Normal</type>
</task>
```

We then take the XML data above and run it through a data format conversion to JSON. It comes out looking like this:

```
{
  “task”: {
    “task_ID”: 438983,
    “taskowner”: “Jeong Kim”,
    “project”: “Oak Valley Factory”,
    “type”: “Normal”
  }
}
```

This is far from an exhaustive list of possible data operations, but it includes several things common to API integrations.

Logic operations support complex functions

Logic operations within an integration can be at two levels. The first level is called flow logic. These operations deal with the integration flow, including decisions and branching.

An example of flow logic for an API integration might say that since data was received from the input API in CSV, certain operations need to be performed on that data to prepare it for output. Other data operations would be necessary if the data were received as XML. In brief, flow logic starts `as if… then… else` statements, though it can become far more complex as the scenario requires.

The second logic level pertains to the API integration data, and we’ll call it math logic. In addition to the text operations we covered above, the integration may also need to perform calculations with pieces of the data itself.

Let’s say we have an integration where one system exports high-level order data to the other. Here are the data elements (fields) for the order record from the source system: `order_id`, `order_amount`, `order_currency`, `order_date`, and `customer_id`.

In this example, we’ll say the currency is US dollars. But the destination system needs the currency in euros. As a result, the interface would use the `order_amount`, `order_currency`, and `order_date` to calculate the cost in euros on the specified date. Then, it would update the order record accordingly, changing the value of `order–currency` to euros and the value of `order_amount`.

There is no limit to the types of logical operations that might be necessary within an integration, but they could encompass anything required to prepare the input data for output to the destination system.

Is there more to an integration?

We've talked mainly about the input side of the integration (database to API, API to integration, and what happens with the API connector and other pieces inside the integration). We could go into the same detail concerning the output side of things, but the output side of an API integration (if we have APIs on both ends of the integration) is doing the same things as the input side, just in reverse order. The integration uses the API connector to talk with the API, which is either receiving or sending data in response.

How to build an API integration

The exact process for building an API integration varies from one integration to the next. However, there are some big-picture things that you should do every time:

1. **Get needed people on board.** Yes, you'll probably need devs, but you'll also need onboarding, support, end-users, and technical assistance from your tech partners.

2. **Gather integration requirements.** Every integration starts with a business need. What do you need to do to address that business need? And what does that mean for API owners and those who will do the work to hook everything together?

3. **Act on those requirements.** Create the integration. Refine and adjust as you go, considering the business need you are solving. Test thoroughly to ensure that things are going to work in production.

4. **Deploy the integration to your customers.** Whether you are deploying the integration internally for a handful of users or externally for hundreds of customers, this is where you meet those users' expectations.

5. **Keep the integration going.** Support (keeping it running) and maintenance (making changes to get in front of issues, improving functionality, and the like) are long-term commitments. But that's what keeps your customers happy.

What is the cost of an API integration?

API integration costs can vary widely. Traditionally, a simple API integration might require 1 to 2 months from start to finish, with medium-complexity integrations requiring 3 or 4 months and complex integrations taking anywhere from 6 to 9 months or more. Given the above, simple integrations could cost $10,000 or more, with complex integrations costing over $100,000.

API integration types and tools that can help

Let's look at two buckets of API integration scenarios and the tools that can help cut the time and cost of building for each:

- ✓ **Internal integrations:** These are API integrations that companies build to enable workflow automation within their businesses. An increasing number of internal API integrations today are built with an enterprise iPaaS (aka traditional iPaaS or iPaaS). The important thing to remember about an enterprise iPaaS is that it is a general-purpose platform used by ***businesses*** to create integrations for ***internal use***.

- ✓ **External integrations:** SaaS companies often build API integrations into their product to connect that product to the other systems their customers use. To build these integrations, SaaS companies can use an embedded iPaaS. The important thing to remember about an embedded iPaaS is that it is a purpose-built platform ***software companies*** use to create native product integrations for their ***customers***.

Enterprise iPaaS vs embedded iPaaS

Both enterprise iPaaS and embedded iPaaS have been specifically designed to help teams build integrations. Both tools simplify and streamline the assembly of everything we've covered in this guide: enterprise iPaaS for internal integrations and embedded iPaaS for integrations between your SaaS product and all the other apps in your customer's ecosystem.

Teams use an enterprise iPaaS or embedded iPaaS because these platforms can substantially cut down on the time and effort needed to build API integrations. It's common to see API integrations that previously took 3 months or more to be completed within 2 weeks. That's a time savings of more than 80%.
For a SaaS team, an embedded iPaaS is much more than an API integration

framework (something that helps SaaS teams put all the pieces in the correct order to make things work). As noted, it can save you substantial time and elevate your integrations from black-box, behind-the-scenes, and bolted-on functionality to first-class product features.

How SaaS teams benefit from embedded iPaaS

If you and your SaaS teams are responsible for designing, building, and supporting a B2B SaaS product, using an embedded iPaaS for your product's API integrations is a great way to realize several benefits. Let's look at those benefits:

- ✓ **Save engineering time.** Use an embedded iPaaS to develop integrations with fewer engineering resources, shift integration onboarding and support to non-devs, and use ready-made infrastructure. Engineers can spend most of their time on your core product (where they need to be).

  > *"Prismatic took something that previously took us months to build and turned it into a matter of days."*
  >
  > *Brian H., CEO*

- ✓ **Increase win rate and sales velocity.** With an embedded iPaaS, you can meet many integration requirements upfront, define high-level requirements to set the scope of integrations, include new API integrations in the initial product onboarding, and even use integrations to offset functional gaps in your product. You can reduce the friction that keeps contracts from closing on time by providing more answers upfront.

  > *"Prismatic allows us to move much faster and create integrations that would not have otherwise been feasible."*
  >
  > *Trevor D., CTO*

- ✓ **Provide a great UX for integrations.** We've already noted that an embedded iPaaS can help make integration a first-class part of your product, but it can also let customers enable, configure and support their integrations, as well as allow you to ensure that integrations have flexible configuration options for multiple customers. Customers will see your API integrations as a welcome extension of your product.

> *"The embeddable marketplace and user configuration wizard offers an enhanced user experience that we cannot deliver ourselves."*
>
> *Justin B., G2 Reviewer*

- ✓ **Improve your customer service.** An embedded iPaaS lets you leverage an infrastructure designed for integrations, empower non-engineers to solve most integration issues, and allow customers to self-service their integrations. The faster customers have their issues addressed, the faster they can stop thinking about API integrations and get back to more important matters.

> *"Our customers now have an integration marketplace where they can connect and configure their integrations themselves which has reduced time and mistakes for our internal support teams."*
>
> *Adam Jacox, VP of Engineering at Hatch. Read Hatch's full story here.*

- ✓ **Reduce customer churn.** Finally, using an embedded iPaaS makes your product central to customer productivity, provides customers with a consistent end-to-end integration experience, draws their attention to the value of integrations, and increases the cost of switching to another product. Customers may come for your product, but they'll stick around because your API integrations have made your product indispensable.

> *"Customer retention and annual revenue per user or subscription, both of those have significantly changed since we've got the Prismatic embeddable solution baked into our product. Our churn month over month has gone down, I'd say, almost 3%."*
>
> *Frank Felice, CRO at Sisu Software. Read Sisu's full story here.*

If you want to see how Prismatic's embedded iPaaS can help your team (devs and non-devs alike) build reusable, productized API integrations for your customers, schedule a demo, and we'll make it happen.

Prismatic