# API Integration Patterns

**UPDATED BY THOMAS JARDINET**
IT ARCHITECT, MANPOWER

**PREVIOUSLY UPDATED BY BRIAN BUSCH**

CONTENTS



APIs can be deployed from anywhere — from your on-premises and SaaS applications to third parties. You first need a solid idea of which APIs you will use, allowing you to then go in depth into design and usage patterns from that perspective. The deployment architecture for your API management solution depends on this as well. Are the majority of your exchanges on-premises? Your API management will have to be on-premises. Or are they in the cloud? Same logic! And if it's both? The deployment architecture can be in the cloud, or it can be a hybrid of cloud and on-premises.

## ABOUT API INTEGRATION

Opportunities for third-party APIs are not always considered at the beginning of an API management project, which pushes the focus of deployment to the cloud. However, the diversity of APIs already available is extensive and can easily bring differentiating functionality to your business. These third-party APIs can either be called directly from a browser, from your corporate website, or from your API management solution, including for API composition.

For third-party APIs in particular, it's important to consider both the pricing model as well as the service-level agreement (SLA), seeing as if you require an SLA of 99.9% for an API, you cannot depend on an API with an SLA of 99.5%.

Once you have thought about your integration strategy, it is time to design your API integrations, respecting key foundational patterns that you should observe.

## PATTERNS FOR API INTEGRATION

It's often said that connecting to an API is easy, but secure integrations that deliver seamless, effortless, and highly performant user experiences are hard. Every API is unique: Researching and building integrations means peeling back layers of nuance, including SOAP vs. REST, XML vs. JSON, different auth mechanisms, workarounds for migrations, webhooks vs. polling for eventing, unique error codes, limited search and discovery mechanisms, etc. Further, every data model is unique, requiring developers to solve complex data mapping and transformation problems for each integration.

With these challenges in mind, let's review key patterns to follow as you build and maintain your API integrations.

### ERROR CODES

| PATTERN | Respect HTTP error codes. |
|---|---|

When creating integrations, it's important to understand what error codes you might be getting from the application provider. As you start sending requests, it's helpful to understand when things work, but it becomes equally important to understand why they don't.

**“PayPal is making it easier than ever to quickly integrate across ecommerce platforms.”**

Danny Thompson,
Web Dev Expert | PayPal Paid Partner

Learn more

PayPal Developer

With integrations, error codes resulting from a lack of access to size limitations help guide your application's business logic as well.

As the majority of HTTP error codes are standardized by RFCs, it is important to respect them and to have an exhaustive, precise list at your disposal.

The following table describes the main HTTP error codes, but it's advisable to rely on a complete and up-to-date list, such as the one from Mozilla:

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

**Table 1**

| ERROR CODE DESCRIPTIONS | | | | | |
|---|---|---|---|---|---|
| 1XX | Informational | 302 | Temp. Found | 410 | Gone |
| 2XX | Success | 303 | See Other | 411 | Length Req. |
| 3XX | Redirection | 304 | Not Modified | 412 | Precondition |
| 4XX | Client Error | 307 | Temp. Redirect | 413 | Entity Too Big |
| 5XX | Server Error | 400 | Bad Request | 414 | URI Too Long |
| 200 | Executed | 401 | Unauthorized | 415 | No Media Type |
| 201 | Created | 403 | Forbidden | 417 | Failed |
| 202 | Accepted | 404 | Not Found | 500 | Internal Error |
| 204 | No Content | 405 | Not a Method | 501 | Not Implemented |
| 301 | Permanent Move | 406 | Not Accepted | 503 | Unavailable |

However, simply mastering the error codes is not enough. Consumers do not have to guess all the error codes you implement in your API. It is advisable to document the errors implemented directly in the definition of your swagger description file.

Here's an example of describing responses, which shows how to document your error codes:

```
Responses:
      '200':
      description: OK.
      '400':
      description: Bad request. Parameter should be
a string.
      '401':
      description: Authorization failed.
      '404':
      description: Not found.
      '5XX':
      description: Internal error.
```

And of course, at least for debugging reasons, you should add an error structure — i.e., with a dynamic error message and code contained in a dedicated structure:

```
Responses:
      400':
      description: Bad request
      content:
      application/json:
      schema:
      $ref: '#/components/schemas/Error'
```

## GRANULARITY AND THE KISS PRINCIPLE

| PATTERN | Keep your API simple and readable; aim for medium-grained resources. |
|---|---|

Granularity and the KISS principle work together hand in hand. KISS, for "Keep it simple st*!" will drive you to define one granularity policy — one that will be the simplest for the end user: the developer who will consume it.

Below is an overview of design paradigms under KISS:

- Design your API with your clients and developers in mind; do not copy/paste your own data model (at least for security reasons...).
- Design your main use cases first, then others after.
- Design using common terms, not business verbiage.
- Design with the idea in mind that developers should have only one way to implement functionalities.

With this mindset, fine- and coarse-grained APIs do not align well. Fine-grained APIs will give you extra effort for no gain, as you'll need to discuss how a resource should be categorized, regarding all the others resources you have or should expose — all of this just to get a longer URL.

Meanwhile, coarse-grained APIs will give the impression that your API is a mess because you'll have resources at the same level, even if there are no relations between all those resources. To demonstrate these differences, let see some examples for a pet shop.

Don't:

```
//Fine grained
Get /animals/mammals/omnivores/cats

//Coarse grained
Get /cats
Get /payments
```

You must then aim for medium-grained APIs. Typically, an API resource whose structure contains no more than two levels, for example, is a suitable level of granularity.

For a pet shop, do:

```
Get /animals/cats
```

Or:

```
Get /animals/42
{
       "id":"42",
       "animalType":"cat"
       "cat":
{
              "mustacheSize":"7",
              "cuteness":"maximum"
       }
}
```

With all this in mind, anyone should be able to use your API without having to refer to the documentation, which should be one of your ultimate goals.

## URLS

| PATTERNS | Use nouns instead of verbs, establish a hierarchical structure, maintain naming conventions, include versions, and employ proper HTTP methods. |
|---|---|

### SINGULAR AND PLURAL NOUNS

Contrary to the SOAP philosophy, REST principles are designed around resources, and nouns should be used in most cases.

Do:

```
Get /orders
```

Don't:

```
Get /Getorders
```

Finally, using plural nouns instead of singular allows you to handle collections of resources and single resources at the same time.

Do:

```
Get /orders
Get /orders/042
```

Don't:

```
Get /Getorders
```

In this way, you can keep a consistent and easy-to-use structure.

### HIERARCHICAL STRUCTURE

Your resources should have a structure, and you should use the URL path to get the details you need. For example, let's say a customer is defined by their own ID, name, and address. As done in object-oriented programming, you would have a customer class containing addresses classes — you handle API resources the same way. So based on our example, to get a customer's addresses, you call the customer number "id" and then `addresses`:

```
Get /customers/042/addresses
```

### CONSISTENT CASE

You must define your naming conventions — in particular, a "case policy" that determines a consistent letter case across *all* resources, as users shouldn't have to read your documentation every five minutes to know which case is required for a specific resource. For parameters, snake_case and camelCase are often used. While there's no true reason to favor one over another, developers often prefer camelCase because it is deeply used in Java, while snake_case is easier to read by non-techies. Meanwhile, spinal-case is preferred for URLs as some servers don't respect case.

Here are examples of each case:

- camelCase – `Get /customers?countryCode=US`
- snake_case – `Get /customers?country_code=US`
- spinal-case – `Get /vip-customers`

### VERSIONING

Version must be contained in URLs in order to help developers migrate easily from one version to another. For those who didn't have the time to migrate, you also risk breaking client integrations for their applications. Versioning is crucial, and the simplest method is to include it in the URL. This way, API calls are logged on your servers, so you'll be able to see easily which version of the API was called, even if you don't have access to the consuming application.

You cannot be sure that everyone is able to migrate quickly to the latest version, so maintain backward compatibility and support at least two previous versions:

```
Get /v1/customers
Get /v2/customers
```

### HTTP METHODS

You must use HTTP methods (i.e., verbs) for their designated purposes. From the beginning, HTTP was designed around resources, and it describes HTTP methods in order to execute specific actions for a resource.

`GET` retrieves data about a resource. For a collection:

```
Get /customers
200 OK
[{"id":"42", "name":"John Doe"},
{"id":"43", "name":"Elisa Re"}]
```

For a single instance:

```
Get /customers/42
200 OK
{"id":"42", "name":"John Doe"}
```

**POST** creates a new resource. For a collection:

```
POST /customers {"name":"John Doe",
"countryCode":"US"}
201 Created
Location: https://myapi.com/customers/42
```

**PUT** replaces or creates a resource. For a single instance:

```
PUT /customers/42 {"name":"John Si",
"countryCode":"US"}
200 OK
Full replacement:
200 OK
Creation:
201 OK
```

**PATCH** makes partial updates to a resource. For a single instance:

```
PATCH /customers/42 {"countryCode":"UK"}
200 OK
```

**DELETE** deletes any existing resource. For a single instance:

```
Delete /customers/42
204 OK
```

## AUTHENTICATION

Getting the right access to the right data underpins any integration project, yet authentication can be one of the hardest parts. At its core, authentication is the ability to prove your application's identity. There are several different ways applications can grant access to developers to create integrations.

| PATTERN | Follow industry-accepted authentication and identity protocols regarding your business needs. |
|---|---|

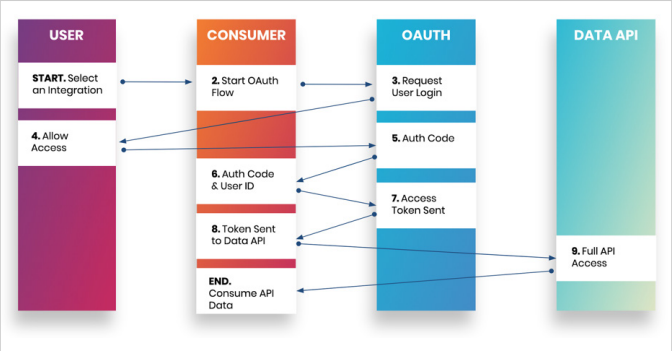
### API KEYS

The use of API keys is still pervasive and very common for API access. It is a fast, easy way to authenticate accounts and is relatively low overhead for the provider as they can easily create and purge API keys. API keys are a uniquely generated set of characters, sometimes random, and are often sent as a pair, a user, and a secret. When receiving API keys, it's best to copy and store them in a password manager and treat them like any other password.

### OAUTH 2.0 AND OPENID CONNECT

OAuth is a bit different in that it is token-based. There are three major components in OAuth: the user, the consumer or integrating application, and the service provider. In this flow, the user grants the consumer (i.e., the application you want to be integrated) access to the service provider by an exchange of tokens through the OAuth endpoint before accessing data from the API.

**Figure 1:** OAuth workflow




OAuth is the preferred approach to authentication because it allows users to decide what level of access the integrating application can have while also being able to set time-based limits. Some APIs have shorter time limits for which the use of refresh tokens is needed.

Open ID Connect is a standardization of OAuth 2.0 that adds a layer of normalized third-party identification and user identity. Not all API providers need it, but it is recommended if you want fine-grained authorization controls and manage several identity providers.

### WEBHOOKS

Webhooks allow you to get updates as they occur in real time as they are *pull*-based instead of *push*-based. In this method, when updates occur from the application, they are sent to a URL that you've specified for your application to use. This push-based refreshing of information is what gives applications the ability to update in real time and create dynamic user experiences.

Additionally, it is the preferred pattern since the implementation is just a URL, as opposed to polling where you need to create a polling framework that manages the frequency and scope of the information you wish to receive.

For example, Marketo's polling framework is 239 lines of code compared to just five lines for a similar webhook.

**Figure 2:** Polling vs. webhooks

## QUERYING

| PATTERN | Use universally conventional keywords and characters in your parameters to query, search, and paginate. |
|---|---|

We've automated the movement of data so that our integration is live and fluid, but now it's flowing in like a river when all we really need is a small piece of the data that is useful to what our application is doing. We need to set **query parameters** to filter out all the data that we don't need. Querying is the last part of the API endpoint, or path, that directs the call to pull only the data you want:

```
GET /widgets?query_parameter_here
```

The ability to search against databases as well as the target apps is a crucial step. This allows you to test and confirm responses and see any difference in the data structure between what is returned and what is in the documentation. Querying allows you to modify the key-value pairs of the request. At the end of the URL path, the query starts with a question mark (`?`) and is separated with an ampersand (`&`). For example:

```
GET /widgets?type=best&date=yesterday
```

To test an endpoint, there are multiple options depending on which language you prefer, but the majority of API documentation will reference commands in `curl`. To test an API, open your Terminal and copy/paste the `curl` command to see what response you get back. Now try adjusting the key-value pairs and fine tune to just the information your application needs from the target endpoint. When adding parameters to a `curl` command directly, be sure to include a backslash (`/`) before the question mark of the query as `?` and `&` are special characters.

Tweaking these pairs will decrease the size and overhead your application might expect. You can always increase the amount of data — especially in testing — to see where you might run into errors.

### FILTERS

To filter, you must use `?` to filter resources and use `&` as separator between parameters:

```
GET /widgets?type=best&date=yesterday
```

### SEARCH

Using the way Google Search works (i.e., with the search keyword) is often the preferred pattern for searching resources — for example, when you search the word test: www.google.com/search?q=test.

To search a specific resource:

```
GET /customers/search?type=vip
```

To search multiple resources:

```
GET /search?q=customers+vip
```

### FIRST, LAST, AND COUNT KEYWORDS

Use `first`, `last`, and `count` to get the first item, the last item, and the number of items in a collection:

```
GET /customers/first
200 OK
{"name":"John Doe", "status":"VIP"}

GET /customers/last
200 OK
{"name":"Johnny Doe", "status":"VIP"}

GET /customers/count
200 OK
{"5"}
```

### SORTING AND ORDERING

An important piece to getting the information you need is how the data will be presented at first glance, which becomes especially helpful in testing and presenting data through a UI. Many APIs will support `sort`, `sort_by`, or `order` parameters in the URL to change the ascending or descending nature of the data you're working with.

For example, a `GET /widgets?sort_by=desc(created)` can give us the freshest widgets in inventory.

## PAGINATION

| PATTERN | Use pagination methods for web front ends and mobile usage. |
|---|---|

Pagination requires some implied order by a field, or fields, like unique id, created date, and modified date. There are a few different types of pagination you can implement based on your needs.

### PARTIAL RESPONSE

In some cases, you don't need to get all fields that define your object. Partial responses are especially useful in mobile use cases — for example, where bandwidth optimization is a priority.

Use the `fields` keyword to get the specific fields you want:

```
GET /customers/42?fields=name,status
200 OK
{"name":"John Doe", "status":"VIP"
}
```

### OFFSET

This is easiest to implement and is, therefore, very common. The `LIMIT` is the number of rows that will be returned, and `OFFSET` is the starting point of results. For example, a path of `/widgets?limit=10` would return 10 rows for the first page. Then the second page could be `/widgets?limit=10&offset=10`, which would return 10 rows, starting with the 10th row, and so on.

The downside of using offset pagination is it becomes less performant with large datasets. For example, if the offset is 1M rows, it will still have to run through 1M rows before returning the limit requested.

### KEYSET

A keyset is helpful to get around large datasets and uses the filter value of the last page of results as the starting point for the next page of results using an additional limit URL parameter. For example, the first call could be `GET /widgets?limit=10` with a unique identifier such as date created or modified. The second call would be `GET /widgets?limit=10&created:lte:2019-09`, the next would be `GET /widgets?limit=10&created:lte:2019-10`, and so on.

### FIXED DATA PAGES

Fairly straightforward, when adding into the query, you select which page of data you would like returned. For example, to return the fourth page of results, you would use this query: `GET /widgets?page=4`.

This method is preferred if the application you're integrating with has known pages and you would like the data returned sequentially. However, it becomes harder if you're not sure what exactly is on that fourth page.

### FLEXIBLE DATA PAGES

Similar to fixed data pages, you could still call the fourth page of widgets, but now you can specify the size of the page. Calling `GET /widgets?page=4&page_size=25` allows you to further dictate what you will get back in terms of page size.

This can be very helpful if you're building a UI and need the results to be a certain size.

## BULK

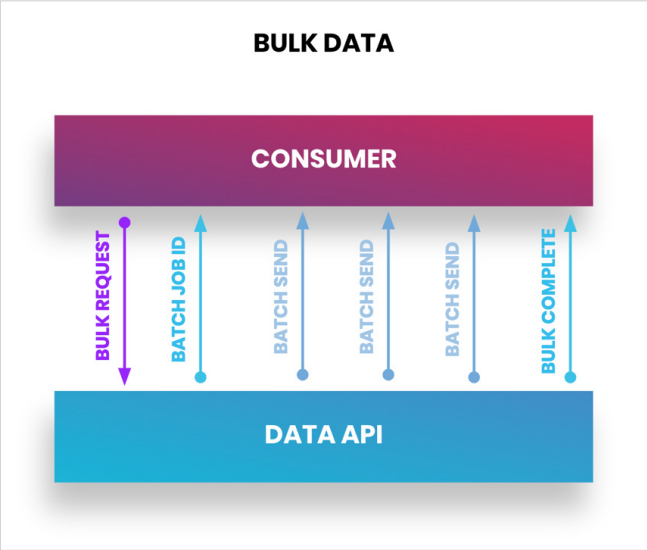
| PATTERN | Use bulk data transfer for large volumes of data. |
|---|---|

Up to this point, API integration has focused on specific sets of data, but what happens when you need to move a large amount of data from one system to another? Some applications expose this ability with Bulk APIs. Bulk APIs allow you to update, insert, and delete a large number of records all at once. This can be particularly useful when transferring large systems of record (e.g., marketing automation, CRM, ERP) from one provider to another.

Bulk operates in several batches of datasets when a request for a bulk job is sent to the application provider. The application provider will send batches over asynchronously to complete the job. Depending on the size of the dataset, the job will also send you a unique identifier to check the job status and close the job once complete.

Be sure to double-check the file type that a bulk API will provide — either CSV, JSON, or XML. Additionally, if you're not getting the full dataset back, be sure to check any API rate limits that apply, as you may exceed them with large data transfers.

**Figure 3:** Bulk data transfer




## SDKS

| PATTERN | Propose an SDK to help developers to use your API. |
|---|---|

Many API providers offer SDKs to help users easily integrate them, which includes setting up the API's interface, methods, error handling, access security, etc. Users have to manage the changes in their code as the API is modified; therefore, it is crucial that an SDK updates at the same rate as the API itself — not to mention the support for version upgrades, debugging, and developer errors.

An SDK also allows you to encrypt data and strengthen security requirements (e.g., for storing passwords). For the medical and financial sectors in particular, an SDK can help enforce user compliance with relevant regulations. Below are several practices for developing an SDK:

- Mutualize in the form of a batch in the same HTTP call as much as possible because the opening of sessions and connections is very consuming. You can set up a queuing system for when the calls do not necessarily need to be received at the same time.
- Compress payload; it can be easy to forget this when calling an API — you might as well not forget it in the SDK.
- Fill in the User-Agent header with the SDK version number and the SDK language, which will provide information on the adoption of new versions of your SDK.
- Set up API analytics to measure usage and performance and to see what improvements could be made to the SDK.
- Offer the most popular languages, especially those that the API's target developers use the most.
- Document the SDK comprehensively, with a complete changelog, but also for its parameterization.

## ADDITIONAL RESOURCES

The following are related resources for further reading:


- Richardson, L. & Amundsen, M. (2015). *RESTful web APIs*. O'Reilly.
- Biehl, M. (2016). *RESTful API design*. CreateSpace.
- Louvel, J. (2013). *Restlet in action: Developing RESTful web APIs in Java*. Manning.
- Kalali, M. & Mehta, B. (2013). *Developing RESTful services with JAX-RS 2.0, WebSockets, and JSON*. Packt Publishing.
- Jacobson, D., Brail, G. & Woods, D. (2011). *APIs: A strategy guide*. O'Reilly.
- Lauret, A. (2019). *The design of web APIs*. Manning.
- Higginbotham, J. (2021). *Principles of web API design: Delivering value with APIs and microservices*. Addison-Wesley Professional.


**WRITTEN BY THOMAS JARDINET,**
*IT ARCHITECT, MANPOWER*

As an IT architect with seventeen years of experience, I oversee business projects by defining their architectures, whether functional, applicational, or technical, studying the best path forward. As a supporter of flattened organizations, I also accompany them on the organizational side, and above all, I seek both an intellectual and human exchange.

DZone
3343 Perimeter Hill Dr, Suite 100
Nashville, TN 37211
888.678.0399 | 919.678.0300

At DZone, we foster a collaborative environment that empowers developers and tech professionals to share knowledge, build skills, and solve problems through content, code, and community. We thoughtfully — and with intention — challenge the status quo and value diverse perspectives so that, as one, we can inspire positive change through technology.