API Integration

Practices and Patterns

CONTENTS

UPDATED BY BRIAN BUSCH
PRODUCT MARKETING & ALLIANCES, CLOUD ELEMENTS

We all hear it so often that we almost stop hearing it: "Integration is critical to meeting users' needs."

Integration work consumes 50%-80% of the time and budget of digital transformation projects, or building a digital platform, while innovation gets only the leftovers, according to SAP and Salesforce. And as everyone from legacy enterprises to SaaS startups launches new digital products, they all hit a point at which the product cannot unlock more value for users or continue to grow without making integration a feature.

If I were to sum up the one question behind all of the other questions that I hear from customers, enterprises, partners, and developers, it would be something like: "Is integration a differentiator that we should own? Or an undifferentiated but necessary feature that supports what we're trying to accomplish?"

This Refcard won't try to answer that question for you. Rather, no matter what type of development work you do, API integration is a fact of life today, like gravity. Why? Today, experience is paramount. The average enterprise uses more than 1,500 cloud applications (with the number growing by 20% each year). Every app needs to integrate with other systems in a fluid and ever-changing application ecosystem. So instead, I'll share some of the common practices you're likely to contend with as well as some patterns to consider.

API INTEGRATION PRACTICES

Gartner will tell you there are a dozen or more integration scenarios or needs, but when you squint at all of them, a bigger distinction stands out: *internal integration* — connecting your instance of systems or applications you control — vs. *digital products*, which offer integration as a feature or service in support of products' value propositions.

Beyond that, the explosion of applications used today means there's a ton of integrations to build, so think about the integration experience you need to offer or who will do the building. Will it be centralized (in IT) and/or productized (for self-serve use), or will it be user-driven (usually for ad hoc integration needs). With these you can create a simple two-by-two matrix *(see Figure 1 on page three)*.

As they say, there's a right tool for every job — in this case, an integration platform or iPaaS. Let's consider some common integration use cases, and then look at the pros and cons of these options.

Cloud Elements

DEVELOPERS DESERVE A BETTER APPROACH TO API INTEGRATION

Built by developers, for developers - Cloud Elements is the only iPaaS you can fully embed in digital products and reuse to churn through internal requests

Normalized methods and enhanced API functionality shield you from API nuance

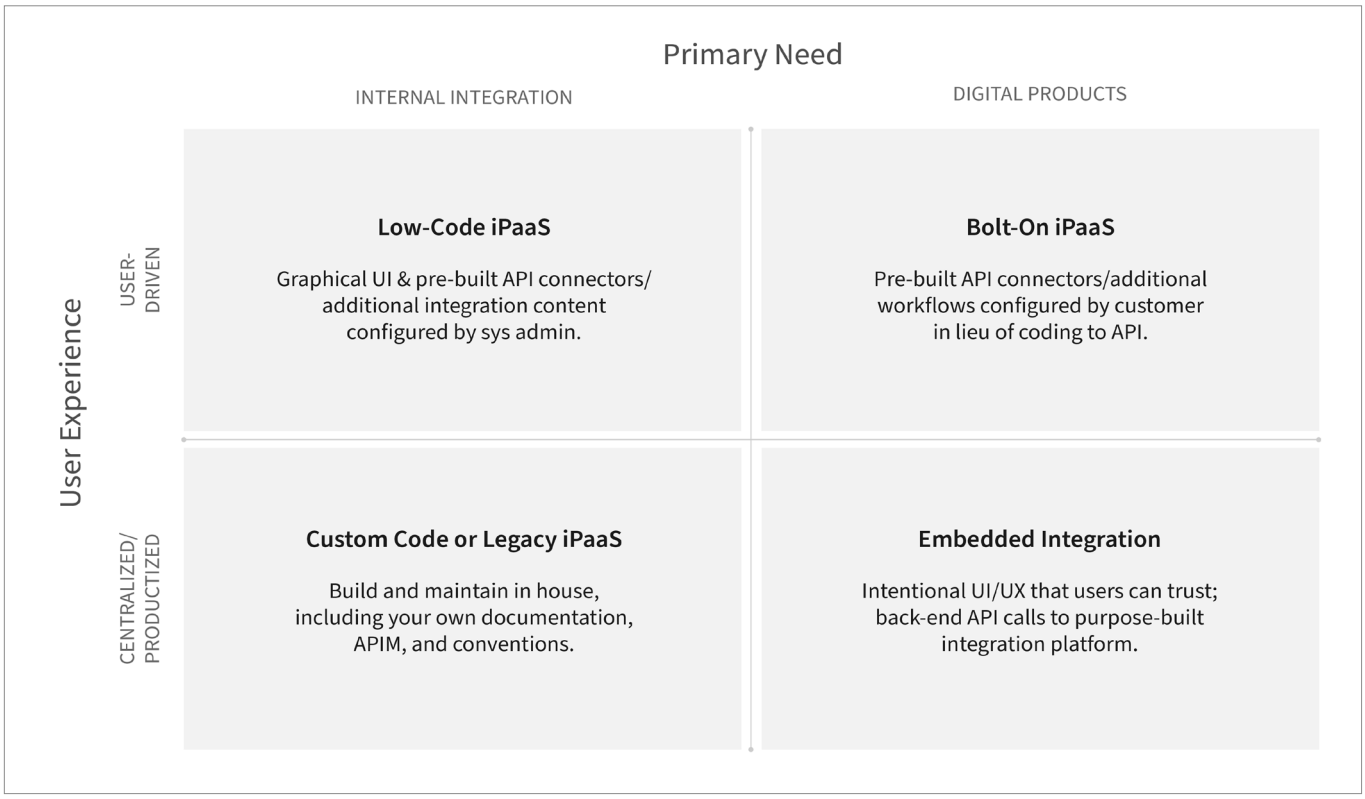
Virtualized data models power a one-to-many approach that cuts repetitive tasks

200+ pre-built API connectors (Elements) standardize auth and reduce maintenance

Extensibility for today's needs: 100% API-accessible, auto-generated instances and tooling let you create and share reusable artifacts

Request Your Free Trial Today

Figure 1: Integration practice matrix



HYPERSCALER DEPLOYMENT AND/OR CLOUD DATA LAKE

Moving applications and data from on-prem architectures to cloud services, whether to save costs or better analyze that data, typically requires re-architecting the stack itself. So it's likely that IT or the CIO's office will also need to centralize the integration work to handle the complexity. But if you had an integration platform with truly robust API connectors (e.g., scheduled bulk operations, webhooks, polling, etc.), could IT build reusable components that ad hoc integrators could use themselves?

GROUND-TO-CLOUD INTEGRATION FOR PROCESS IMPROVEMENT

Perhaps your company's enterprise resource planning (ERP) or finance and accounting (F&A) system runs on-prem still, but the CFO's office wants to take advantage of a cloud-based accounts payable (AP) automation platform like Tipalti, Tungsten Networks, or Coupa? You likely have a legacy enterprise service bus (ESB) or other middleware running on-prem and integrated with the ERP, but it will take a lot of custom code to work with Tipalti's API.

If Tipalti offers the embedded integration for your ERP, problem solved. Otherwise, look for an API-accessible, pre-built connector like those from SAP Open Connectors or Axway's Integration Builder.

CLOUD-TO-CLOUD INTEGRATION FOR DIGITAL PRODUCTS

Enterprises and startups alike generally build customer-facing applications in the cloud with modern, RESTful APIs. However, every API implements standards like OAuth differently, and there's always data mapping and transformation work to be done for point-to-point integrations. Not to mention customers rarely have the time and skills to do the integration work themselves.

A bolt-on iPaaS can be a cheap, quick way to ease the burden for users, but then you're limited by the iPaaS UX and their API connectors. Consider options that allow you to embed one-to-many integrations within your product UI/UX via API call — you don't have to build or maintain the underlying code, and you get to own the user experience.

CUSTOM APPLICATION-TO-CLOUD INTEGRATION

Let's say that your company has, or is building, custom applications and needs to integrate them with other cloud (or on-prem) applications and data. No iPaaS will offer an API connector to your custom app out of the box, which means you'll have to write and maintain custom code, perhaps within a legacy iPaaS.

Instead, look for an extensible integration platform that allows you to create reusable, first-class API connectors that others (developers, ad hoc integrators, etc.) across the company could take advantage of to deliver their own integrations and take pressure off the centralized integration team.

API INTEGRATION PATTERNS

Integration is more complex than it looks. 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. Why?

- **Every API is unique** – Like snowflakes, researching and building integrations means peeling back layers of nuance, including SOAP vs. REST, XML vs. JSON, different auth mechanisms, workarounds for migrations when <5% of APIs offer bulk data operations, webhooks vs. polling for eventing, unique error codes, limited search and discovery mechanisms, etc.
- **Every data model is unique** – This requires developers to solve complex data mapping and transformation problems for every integration.
- **Every workflow is different** – From operations on the data itself to lookups and contingent logic, developers need the right tools to not only connect systems, but to also improve — and even automate — otherwise manual process steps.

With that in mind, below are some of the nuanced issues that developers encounter daily as they build and maintain integrations.

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.

With integrations, error codes resulting from a lack of access to size limitations help guide your application's business logic as well. While many error codes are not normalized, Table 1 is a general reference guide you can use when working with REST APIs (*see next column*).

Table 1: Error code descriptions

CODE ERROR AND DESCRIPTION					
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

AUTHENTICATION

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

BASIC CREDENTIALS

This approach is listed for educational purposes but is, fortunately, becoming less common due to its security vulnerability of man-in-the-middle attacks. This method uses the username and password in the HTTP header when you are making calls, so the security becomes dependent on the availability of SSL. This method lends itself to internal APIs for resources that aren't generally exposed elsewhere or for IoT projects.

API KEYS

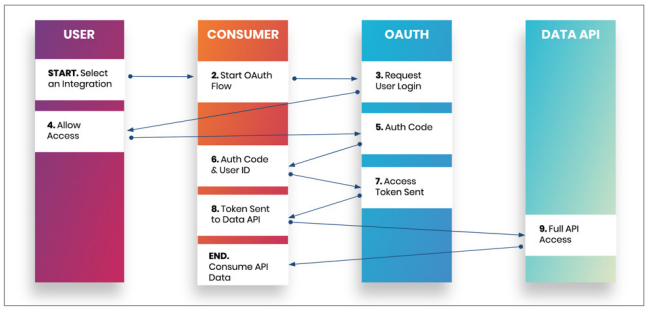
The use of API keys is still pervasive and very common for API access. It is a fast, easy way to authenticate accounts and is relatively low overhead for the provider, as the provider can easily create and purge API keys. API keys are a uniquely generated set of characters, sometimes random, and are 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

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 2: OAuth workflow



OAuth is the preferred and best-practice 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.

EVENTING

We've navigated documentation, authenticated access, and can `/GET` a "Hello World" response — now, let's bring the data to life, or at least automate it. In some cases, eventing isn't needed and you can skip ahead. However, the true power of integration is the ability to put data in motion without the bottleneck of human clicks. There are two primary ways to automate or "event" your API calls.

POLLING

Polling is very much like the kid sitting in the backseat on a road trip constantly asking, "Are we there yet?" But in our case, computers don't get annoyed, so the conversation goes like this:

> "Any changes in data yet?"
> "No, no, no, no, yes, no, no, yes, and so on."

To detect if there have been any changes in the data — create, retrieve, and delete events, specifically — polling is sending requests at a predetermined frequency and waiting for the endpoint to respond. If there is no response body, then there have been no changes. Because of this continual request for information, polling is very inefficient and can be expensive when paying for computing time, as the vast majority of requests go unanswered.

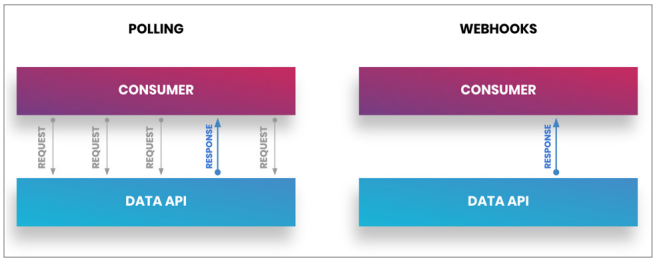
Finally, because of the intervals in timing, polling is immediately out of date once the API call returns, batching any new updates that would have happened since the last call.

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 3: Polling vs. webhooks



QUERYING

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 of 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 applications 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`. "cURL" stands for Client URL and is a command line tool that becomes particularly valuable when retrieving data from other applications. To test an API, open up your Terminal and copy/paste the `curl` command to see what response you get back. Simple enough, right?

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 add 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. Simplicity is the act of demystifying something without losing the magic, and the same could be said for APIs. You can always increase the amount of data — especially in testing — to see where you might run into errors.

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

There might be way too much data, or data that just keeps going and scrolling. This is where pagination comes in. Pagination is the ability to put that giant pool of responses from every customer you've had since 1994 into human-readable pages without seizing up your computer. Pagination requires some implied order by a field, or fields, like unique id, created date, modified date, and so on. There are a few different types of pagination you might encounter.

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`.

This 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 data sets. 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 data sets 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 either a unique identifier like 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 are integrating with has known pages and you would like the data returned sequentially. However, it becomes harder if you aren't 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.

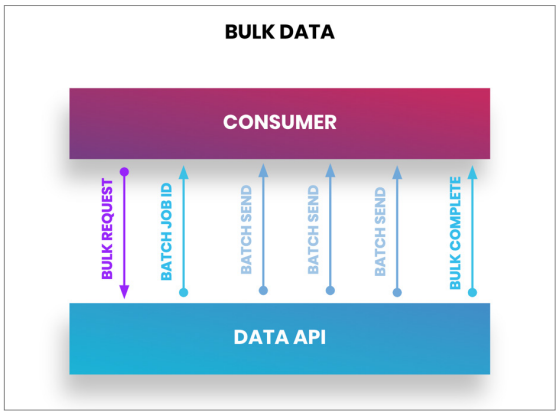
BULK

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, or ERP) from one provider to another.

Bulk actually operates in several batches of data sets 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 data set, 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 data set back, be sure to check any API rate limits that apply, as you may exceed them with large data transfers.

Figure 4: Bulk data transfer



ADDITIONAL RESOURCES

**Books from the "Foundations of RESTful Architecture" Refcard:*

- Richardson, L., & Amundsen, M. (2015). *RESTful web APIs*. O'Reilly.
- Allamaraju, S. (2010). *RESTful web services cookbook*. O'Reilly.
- Masse, M. (2011). *REST API design rulebook: Designing consistent RESTful web service interfaces*. O'Reilly.
- Biehl, M. (2016). *RESTful API design*. CreateSpace.
- Webber, J., Parastatidis, S., & Robinson, I. (2010). *REST in practice*. O'Reilly.
- 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.

WRITTEN BY BRIAN BUSCH,

PRODUCT MARKETING & ALLIANCES, CLOUD ELEMENTS

Brian Busch is an API enthusiast, leads Product and Alliances Marketing for Cloud Elements, and has been involved in launching and scaling new products and services throughout his career, most recently with Kapost (now Upland Kapost) and before that, Captricity (now Vidado.ai).

He holds degrees from Boston College and an MBA from UC Berkeley-Haas. @brbusch

DZone, a Devada Media Property, is the resource software developers, engineers, and architects turn to time and again to learn new skills, solve software development problems, and share their expertise. Every day, hundreds of thousands of developers come to DZone to read about the latest technologies, methodologies, and best practices. That makes DZone the ideal place for developer marketers to build product and brand awareness and drive sales. DZone clients include some of the most innovative technology and tech-enabled companies in the world including Red Hat, Cloud Elements, Sensu, and Sauce Labs.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC 27709

888.678.0399 919.678.0300