API Development: What Companies Need to Know (About REST, JSON:API, and GraphQL) [Part 1]

Author: Chad Furman

Claps: 63

Date: Jun 5

When starting an API standards initiative, youâ€™ll need to make some decisions. One question that is almost certain to come up is whether to build your API by following a pattern like REST or GraphQL. Klaviyo recently went through an API overhaul, and in researching this question (and many more) we gained insights that I will share with you here.

What to Expect

First, I will lay the groundwork for comparison between REST and GraphQL and discuss what REST is. I will then spend the rest of the article discussing JSON:API, the flavor of REST that Klaviyo adopted.

This article outlines the strengths of JSON:API, and where gaps both in the spec and community understanding can lead to challenges. In follow-up posts, I will talk about GraphQL as an alternative to JSON:API. Iâ€™ll also share lessons on how internal teams can organize around new API initiatives.

Starting the Discussion - REST or GraphQL?

Once it was clear that we wanted to create a new API surface at Klaviyo, the conversation shifted to what that API surface should look like. Some wanted to make RPC calls, some wanted to build a RESTful API, and others wanted GraphQL. I wonâ€™t talk much about REST vs RPC in this article, but if youâ€™re interested in that, there is a great comparison between REST and RPC in this 2016 Smashing Magazine article: â€œUnderstanding RPC and REST for HTTP APIs.â€�

We discussed at great length if we should build our API with REST or GraphQL, and what it would look like if we went one way or the other. Asking the question â€œREST or GraphQLâ€� can be so contentious that it can stir up a meta-debate about how to compare REST and GraphQL:

- Are we comparing two different architectural styles, or is GraphQL more of a framework?
- Are we actually talking about REST with Hypermedia?
- Are we talking about pseudo-REST with RPC calls?
- Is GraphQL is a subset of REST?

Ultimately, we decided that taking on a GraphQL API without first having a solid foundation in REST was not a wise path for Klaviyo. As such, we chose to refine our REST API development practices first. Since this article will focus on important considerations for REST API development, we start by clarifying what REST APIs even are.

What is REST?

REST stands for *representational state transfer* and is originally based on Roy Fieldingâ€™s dissertation â€œArchitectural Styles and the Design of Network-based Software Architectures.â€� Lately, REST has become an overloaded term widely misappropriated and misunderstood to much chagrin. An excellent article on â€œwhat REST isâ€� and â€œwhat REST is notâ€� is the aptly titled, â€œ*How Did REST Come To Mean The Opposite of REST*.*â€�*

Some key considerations for REST APIs:

- Stateless - Each request does not know about the requests before it. No state is maintained between requests.
- Uniformity - A consistent interface across resources and actions to promote re-use and simplicity

This article is focusing on a familiar subset of REST APIs that use JSON. These APIs communicate via JSON over HTTP, have a CRUD-like interface, rely on HTTP verbs and HTTP response codes, and have resources as the primary building blocks of the API design. Fortunately, the pattern is so common that around a decade ago a group started codifying it in a JSON:API specification.

What is JSON:API?

When building REST APIs, there are a lot of decisions to be made. Examples of some of these decisions include the name of pagination parameters, how errors are handled, and how to manage related resources. The JSON:API specification provides a framework for REST API development and provides solutions for common problems that REST API developers encounter. The specification itself is technical enough to answer implementation questions and also approachable enough that it can be read by product managers.

JSON:API as a REST-compatible specification solves many of the same problems that GraphQL solves. Over-fetching and under-fetching are two of the most common complaints about REST APIs. The JSON:API spec provides for sparse fieldsets through the â€œfieldsâ€� parameter which limits which fields get returned on which objects â€“ a solution for over-fetching. Likewise, JSON:API describes compound documents which support embedding related resources in a response via an â€œincludeâ€� parameter. These two JSON:API features add GraphQL-like flexibility to your API interface.

JSON:API CRUD

With a REST API, you generally have a set of CRUD operations mapped to specific HTTP verbs. These functions correspond to **C**reate, **R**ead, **U**pdate, and **D**elete actions on resources.

Create

Creating a resource is done with an HTTP POST request to the top-level URL with the resourceâ€™s name. Both top-level resources and relationships can be created.

Create a Resource

In the following example, we create a book assuming that the author already exists.

```
POST /books HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "books",
    "attributes": {
      "title": "A Book Title",
      "description": "A description of the book",
    },
    "relationships": {
      "author": {
        "data": { "type": "authors", "id": "4fa99291-1d7b-4e71-8e88-96cef1
      }
    }
  }
}
```

The API would then respond with the newly created resource, including links to the relations.
Below is an example response, including some fields JSON:API considers optional.

```
HTTP/1.1 201 Created
Location: https://example.com/books/b575b7b6-f432-4060-8fde-2ba48af5e149
Content-Type: application/vnd.api+json

{
  "data": {
    "type": "books",
    "id": "b575b7b6-f432-4060-8fde-2ba48af5e149",
    "attributes": {
      "title": "A Book Title",
      "description": "A description of the book",
    },
    "relationships": {
      "author": {
        "links": {
          "self": "https://example.com/books/b575b7b6-f432-4060-8fde-2ba48
          "related": "https://example.com/books/b575b7b6-f432-4060-8fde-2b
        },
        "data": { "type": "authors", "id": "4fa99291-1d7b-4e71-8e88-96cef1
      },
 "chapters": {
        "links": {
          "self": "https://example.com/books/b575b7b6-f432-4060-8fde-2ba48
          "related": "https://example.com/books/b575b7b6-f432-4060-8fde-2b
        },
        "data": []
      }
    },
    "links": {
      "self": "https:/example.com/books/b575b7b6-f432-4060-8fde-2ba48af5e1
    }
  }
}
```

The same pattern could be used to add a chapter to this book:

```
POST /chapters HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "chapters",
    "attributes": {
      "title": "Chapter 1",
      "content": "The content of chapter 1."
    },
    "relationships": {
      "book": {
        "data": { "type": "books", "id": "b575b7b6-f432-4060-8fde-2ba48af5
      }
    }
  }
}
```

Create a New Relationship

If you wanted to add a chapter that already existed to a book (perhaps if it was in draft status and not assigned to any book previously), then you could create a distinct relationship object. This would involve a POST request to the bookâ€™s â€œchapterâ€� relationship endpoint, as follows.

```
POST /books/b575b7b6-f432-4060-8fde-2ba48af5e149/relationships/chapter HTT
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": [
    { "type": "chapter", "id": "c1e9cd8f-2a70-40ee-9614-53b34a392d00" }
  ]
}
```

Read

Read requests either list out available resources or fetch details on a specific resource using an HTTP GET request. Read requests can include url parameters to filter, paginate, and more. Below are examples of making read requests to JSON:API endpoints.

Read Many (aka List)

First, we can list out all of our books by making a GET request to the `/books` endpoint. The response includes a list of books. In this case, we only have two books, though a longer response might be paginated.

```
--- REQUEST ---
GET /books HTTP/1.1
Accept: application/vnd.api+json

--- RESPONSE ---
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "https://example.com/books"
  },
  "data": [{
    "type": "books",
    "id": "b575b7b6-f432-4060-8fde-2ba48af5e149",
    "attributes": {
      "title": "A Book Title",
      "description": "A description of the book",
    }
  }, {
    "type": "books",
    "id": "23faad83-0327-4a47-8a00-b4be6ecc1fbd",
    "attributes": {
      "title": "A Second Book Title",
      "description": "A description of the second book",
    }
  }]
}
```

Read One

We can also fetch details on a specific book with a GET request to `/books/:book_id` and in this case we receive additional information in the response including information about the bookâ€™s relationships. Depending on the API, this information may also be present in the list endpoint above; however, in our example API here, we only include it for a single resource.

Here, we see that our book has the same author as our first book and also has no chapters.

```
--- REQUEST ---
GET /books/23faad83-0327-4a47-8a00-b4be6ecc1fbd HTTP/1.1
Accept: application/vnd.api+json

--- RESPONSE ---
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json
```

```
{
  "data": {
    "type": "books",
    "id": "23faad83-0327-4a47-8a00-b4be6ecc1fbd",
    "attributes": {
      "title": "A Second Book Title",
      "description": "A description of the second book",
    },
    "relationships": {
      "author": {
        "links": {
          "self": "https://example.com/books/23faad83-0327-4a47-8a00-b4be6
          "related": "https://example.com/books/23faad83-0327-4a47-8a00-b4
        },
        "data": { "type": "authors", "id": "4fa99291-1d7b-4e71-8e88-96cef1
      },
 "chapters": {
        "links": {
          "self": "https://example.com/books/23faad83-0327-4a47-8a00-b4be6
          "related": "https://example.com/books/23faad83-0327-4a47-8a00-b4
        },
        "data": []
      }
    },
    "links": {
      "self": "https://example.com/books/b575b7b6-f432-4060-8fde-2ba48af5e
    }
  }
}
```

Read Extensions

Read Related Resources

Read requests can also be used to fetch data on resource relationships specifically. In the following example, we see that a GET request to `/books/:book_id/author` is the same as a GET request to `/authors/:author_id` when author_id is the author of book_id

```
--- REQUEST ---
GET /books/23faad83-0327-4a47-8a00-b4be6ecc1fbd/author HTTP/1.1
Accept: application/vnd.api+json

--- RESPONSE ---
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "type": "authors",
  "id": "4fa99291-1d7b-4e71-8e88-96cef139e1c2",
  "attributes": {
    "firstName": "Dan",
    "lastName": "Gebhardt",
    "twitter": "dgeb"
  },
  "relationships": {
    "books": {
      "links": {
        "self": "https://example.com/authors/4fa99291-1d7b-4e71-8e88-96cef
        "related": "https://example.com/users/4fa99291-1d7b-4e71-8e88-96ce
      },
      "data": [
        { "type": "books", "id": "b575b7b6-f432-4060-8fde-2ba48af5e149" },
        { "type": "books", "id": "23faad83-0327-4a47-8a00-b4be6ecc1fbd" }
      ]
    }
  },
  "links": {
    "self": "https://example.com/authors/4fa99291-1d7b-4e71-8e88-96cef139e
  }
}
```

Include Related Resources

If we wanted to include those resources in the response to the original request, JSON:API offers an include parameter.

```
--- REQUEST ---
GET /books/23faad83-0327-4a47-8a00-b4be6ecc1fbd?include=author HTTP/1.1
Accept: application/vnd.api+json

--- RESPONSE ---
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "type": "books",
    "id": "23faad83-0327-4a47-8a00-b4be6ecc1fbd",
    "attributes": {
      "title": "A Second Book Title",
      "description": "A description of the second book",
    },
    "relationships": {
      "author": {
        "links": {
          "self": "https://example.com/books/23faad83-0327-4a47-8a00-b4be6
          "related": "https://example.com/books/23faad83-0327-4a47-8a00-b4
        },
        "data": { "type": "authors", "id": "4fa99291-1d7b-4e71-8e88-96cef1
      },
 "chapters": {
        "links": {
          "self": "https://example.com/books/23faad83-0327-4a47-8a00-b4be6
          "related": "https://example.com/books/23faad83-0327-4a47-8a00-b4
        },
        "data": []
      }
    },
    "links": {
      "self": "https://example.com/books/b575b7b6-f432-4060-8fde-2ba48af5e
    }
  },
  "included": [{
    "type": "authors",
    "id": "4fa99291-1d7b-4e71-8e88-96cef139e1c2",
    "attributes": {
      "firstName": "Dan",
      "lastName": "Gebhardt",
      "twitter": "dgeb"
    },
    "links": {
      "self": "https://example.com/authors/4fa99291-1d7b-4e71-8e88-96cef13
    }
  }]
}
```

Sparse Fieldsets

If we wanted to limit the number of fields returned, we can use JSON:APIâ€™s fields parameter to request a sparse fieldset. For example, if we only wanted the title then we could add `?fields[books]=title`. This would result in only the title being returned. Note that relationships are also considered fields, and since author and comments were not specified, theyâ€™re not included in the response.

```
--- REQUEST ---
GET /books/23faad83-0327-4a47-8a00-b4be6ecc1fbd?fields[books]=title
Accept: application/vnd.api+json

--- RESPONSE ---
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "type": "books",
    "id": "23faad83-0327-4a47-8a00-b4be6ecc1fbd",
    "attributes": {
      "title": "A Second Book Title"
    }
    "links": {
      "self": "https://example.com/books/b575b7b6-f432-4060-8fde-2ba48af5e
    }
  }
}
```

Filter, Pagination, and Sorting

Lists of resources can be filtered, paginated, and sorted using other URL parameters. Here, weâ€™re asking for the first page of results from the books list endpoint, with a max of 10 (thereâ€™s only 2). We also specify we only care about books with the substring â€œTitleâ€� anywhere in their title and weâ€™re sorting by title in reverse order.

```
--- REQUEST ---
GET /books?page[number]=1&page[size]=10&filter[title]=%2ATitle%2A&sort=-ti
Accept: application/vnd.api+json

--- RESPONSE ---
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "https://example.com/books"
  },
  "data": [{
    "type": "books",
    "id": "23faad83-0327-4a47-8a00-b4be6ecc1fbd",
    "attributes": {
      "title": "A Second Book Title",
      "description": "A description of the second book",
    }
  },{
    "type": "books",
    "id": "b575b7b6-f432-4060-8fde-2ba48af5e149",
    "attributes": {
      "title": "A Book Title",
      "description": "A description of the book",
    }
  }]
}
```

Update

Update actions in JSON:API use the PATCH method to the respective `/resource/:id` endpoint. PATCH is used because it supports partial updates, whereas PUT only supports full-on replacement.

```
--- REQUEST ---
PATCH /books/23faad83-0327-4a47-8a00-b4be6ecc1fbd HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "books",
    "id": "23faad83-0327-4a47-8a00-b4be6ecc1fbd",
    "attributes": {
      "title": "A Second Book Title (Revised Edition)"
    }
  }
}

--- RESPONSE ---
HTTP/1.1 200 OK
```

```
Content-Type: application/vnd.api+json

{
  "data": {
    "type": "books",
    "id": "23faad83-0327-4a47-8a00-b4be6ecc1fbd",
    "attributes": {
      "title": "A Second Book Title (Revised Edition)"
      "description": "A description of the second book"
    }
    "links": {
      "self": "https://example.com/books/b575b7b6-f432-4060-8fde-2ba48af5e
    }
  }
}
```

Delete

An HTTP DELETE request can be issued to a `/resource/:id` endpoint to delete that resource. No response content is necessary.

```
--- REQUEST ---
DELETE /books/b575b7b6-f432-4060-8fde-2ba48af5e149 HTTP/1.1
Accept: application/vnd.api+json

--- RESPONSE ---
HTTP/1.1 204 No Content
```

Summary of JSON:API CRUD

Above, we have seen examples of Create, Read, Update, and Delete operations in JSON:API. We included both request and response, as well as some variations. We briefly touched on filtering, pagination, and sorting. See the spec for details and other interesting topics like error handling, meta information, and optional recommendations.

Considerations for Choosing JSON:API

Pros

- **Bikeshedding** â€” JSON:API enables API development efforts to avoid several bikeshedding conversations, including â€œWhat qualifies as REST?â€�
- **Familiar** â€” JSON:API reinforces the traditional, ubiquitous JSON-based API, whereas GraphQL is a relatively radical and unfamiliar overhaul.
- **Caching** â€” JSON:API leverages the HTTP standard, unlike GraphQL, and as such benefit from established caching methodologies .
- **Consistency** â€” The JSON:API spec gives guidance on how to shape your return data and can help avoid drift between endpoints.

- **Relationships** â€" First-class relationship support between resources is written into the JSON:API specification.

The â€œWhat is RESTâ€� Debate

Discussing what is and is not a REST API can sap time away from actually building your APIs. Are you returning JSON and accepting JSON, or do you want to support XML and HTML responses also? Are HTML fragments what REST means by â€œhypermedia?â€� If one of your endpoints is more like an RPC task, is it no longer a REST API? At some point youâ€™ll have to make a decision and stick with it.

The standards that JSON:API proposes are compatible with most interpretations of REST and come with several added benefits.

Familiar and Intuitive

Standing up a REST API is a known problem with a known solution. Many patterns exist, including MVC and CRUD, which map specific functions to specific route patterns with HTTP verbs. Moreover, CRUD over HTTP is a very intuitive pattern. Need to retrieve something? Make a GET request to the `/something/:id` endpoint. Need to save a new object? Thatâ€™s going to be a POST to `/something`. An update to a thing? Well, itâ€™s probably going to be a PATCH (but maybe itâ€™s a PUT or POST depending on the API) to `/thing/:thing-id`. Delete? DELETE!

REST APIs are the dominant API standard on the web so much so that companies offer REST APIs alongside their GraphQL APIs, e.g Shopify, Github, and even Facebook. (Facebook, interestingly enough seeing as they invented GraphQL, only uses GraphQL to power their website and mobile apps. They donâ€™t expose their GraphQL API to third parties.) Developers and product teams know how to build REST APIs, they know the pitfalls, and theyâ€™re familiar with common tooling like Postman and Swagger. Thereâ€™s no new terminology to learn (like the difference between a Query and a Mutation or what a â€œschemaâ€� is and how you â€œstitchâ€� them). Developers looking for speed often stick with the devil they know, and for most developers that devil is a REST API.

Leverage Widely Available HTTP Caching Layers

As your API traffic scales, caching can play a crucial role in your performance. If youâ€™re receiving 1000 requests a second for `/books/1` and you stick that in a cache with a 60-second TTL, youâ€™re going to get one request to your server every 60 seconds for 60,000 requests worth of traffic. Those 60,000 requests worth of traffic are now resolving in milliseconds while that one request a minute is possibly taking half a second or more to resolve as it queries your database and so on.

REST APIs, being built around the HTTP specification, allow both the API developer and API consumer to benefit from standard HTTP caching techniques. Developers of client-side apps calling REST APIs get a lot of benefit from browser-side caching because the URL (and query params) work well as a cache key. The URL works well as a cache key because the route structure maps to a consistent response payload in a REST API.

Consistency in Request and Response Patterns

The most common complaints about REST stem from the routing and response patterns. You have to decide how to name your endpoints, what you return, and how you accept data. JSON:API provides us a strict specification on what these request and response objects should look like, including HTTP status codes and error responses.

Additionally, adopting JSON:API allows us to benefit from the specification around sparse fieldsets and compound documents. REST API endpoints historically have challenges related to over-fetching and under-fetching. Through sparse fieldsets and included resources via compound documents, JSON:API allows us to solve these challenges and fetch the data that we want in a single network request (assuming the API supports these features, as there are performance concerns â€” more on this below).

The consistency of request and response patterns enables flexible, reusable solutions that are easier to maintain for API developers and easier to consume for end users.

Relationships

When working with related data, questions around whether to include the related resource ID in the response are just the tip of the iceberg. Actually including the related resource in the response, filtering on related resource properties, and adding and deleting relationships are additional considerations which begin to show design complexity.

JSON:API has first-class support for resource relationships. Arguably, JSON:APIâ€™s relationship support is the most important and most well-documented feature of the specification. While JSON:API is not a panacea and API developers have to program relationships into API resources, the structure proposed by the specification goes a long way to guiding the implementation of these relationships and promoting their use.

Cons

We found that JSON:API addresses a lot of the common pain points of developing and consuming REST APIs, but there are a few pain points that JSON:API does not solve.

Namely:

- **RPC** â€” REST APIs gain RPC endpoints over time for use case-driven functionality. Managing the RPC creep is nigh impossible.
- **Typing** â€” The only real type enforcement is through the documentation. Not all REST APIs have an OpenAPI spec.
- **Discoverability and Documentation** â€” API functionality becomes harder to document and harder to find as the API grows.
- **Versioning** â€” No clear specification on managing multiple versions of functionality, deprecation strategies, and backwards compatibility.
- **Implementation Details** â€” Pagination, filtering, and other fine-grained details are left up to the implementation.
- **Performance** â€” The added flexibility of filtering, sorting, and included relationships can raise performance challenges

The REST vs RPC Debate

The question of how to address RPC-like things was quite consuming and, at one point, was going to steer us away from adopting JSON:API altogether. REST APIs are known for CRUD-like functionality, and many developers see this as a contrast against RPC-style calls. When youâ€™re working with a domain entity (something with an ID that you create and delete), itâ€™s obvious how youâ€™d create, update, read and delete. However, it is less clear how an operation that isnâ€™t intuitively CRUD-able fits into a standard CRUD API pattern. This leads to lots of confusion and often ad hoc endpoints that diverge from the CRUD pattern. An example might be a â€œsubscribeâ€� call to subscribe a user to an email list, for example.

The article from Smashing Magazine, â€œUnderstanding RPC vs REST for HTTP APIs,â€� talks about how Slack has an RPC API alongside its REST API. In this article, the author questions â€œWhy am I trying to jam all of these actions into a REST APIâ€� because sometimes actions as a side effect of resource creation do not fit â€œnicelyâ€� into REST. They then refer to creating a â€œkicksâ€� or â€œbansâ€� collection as an example of a square-peg/round-hole.

In Chapter 5, section 5.1.5 of Roy Fieldingâ€™s dissertation, we read â€œThe central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface.â€� The purpose of this is so that â€œthe overall system architecture is simplified and the visibility of interactions is improved.â€� Itâ€™s recognized that â€œthe trade-offâ€� is that â€œinformation is transferred in a standardized form rather than one which is specific to an applicationâ€™s needs.â€� The dissertation goes on to specify that constraints are needed to guide REST API development so that we can benefit from a simplified, uniform interface. One of these constraints is resource identification â€” i.e. a â€œkicksâ€� and â€œbansâ€� collection.

The Klaviyo Solution: Allowing RPC calls

Trying to avoid RPC calls entirely may be impossible, and being prepared for this reality ahead of time can save you some difficulty. Sometimes it may be possible to get creative with the definition of a resource (e.g. the â€œsubscribeâ€� RPC call could be a â€œSubscriptionâ€� resource). Other times, however, you may not have necessary information (for example, on an unauthenticated endpoint) or you may need to trigger multiple side effects and in these cases an RPC call might be the way to go.

RPC calls at Klaviyo are the exception rather than the rule. Where possible, we try to model RPC calls as a â€œjobâ€� â€” for example with async calls like bulk uploads. An API does not need to support all CRUD operations on a job resource if all you need to do is to CREATE the job (i.e. execute it) as a way of triggering your RPC call. At this point, your RPC call fits nicely into the CRUD pattern along with the rest of your resources. The added benefit here is that you can have a record of jobs that youâ€™ve created (if you want) and a way to poll their status via GET requests and to cancel them via DELETE requests.

In most cases, API users are likely CRUDing a domain object or an aggregate. In cases where this isnâ€™t possible for one reason or another, jobs and/or RPC calls can fit the bill. It takes discipline, practice, and collaboration to adhere strictly to have a uniform API interface and ultimately what the business needs takes precedence over technical purity.

URL Params

Another point that came up during our discussions was that of URL length relative to request parameters. With the tight coupling to the HTTP standard comes an inherent limitation around

URL length. Edge leads the charge with a limit of approximately 2000 characters. Apache HTTP server has a limit of 4000 characters. Firefox is approximately 64k, and Chrome is 2MB! Some web application firewalls (WAFs) and other server and client software have other URL length limitations. So while you may be tempted to have really intricate query patterns embedded into your URL params seeing as how the HTTP spec does not put any limitations on the length of URLs, you will be sad to hear that the reality is much different. Keep your URL length below 2048 characters or risk disappointing your users.

This becomes an issue, primarily, when trying to filter large GET requests. When asking for GET `/books`, you may want to add `?filter=or(eq(some_field,somevalue), eq(some_other_field,some_other_value))&sort=desc(another_field)&page=ddd07 eeda-45d1â€"8cc3-f347574a666a` and now youâ€™re already at almost 150 characters. Not a big deal, but combine this with a long base URL, version information, additional filtering and sorting criteria and this can be a concern.

The Klaviyo Solution: Expect your URLs to be hard-capped at 2048 chars and create jobs that run complex queries

If you need to support complex queries, an alternative to cramming your criteria into the GET params is to create (i.e. POST) a job resource which runs async and then poll that resource with GET requests. When the job finishes, the GET response contains the results that the caller needs either directly, as related resources, or as a link to a file the caller can then download and parse.

Lack of Typing

One problem that frontend developers have when trying to interface with an API, especially when using a typed language, is knowing what datatypes theyâ€™re receiving in responses from the API as well as what datatypes their code is expected to send. Unfortunately, nothing in REST or JSON:API defines a standard for typing. Without a rigorous typed schema, itâ€™s tough to know if you are working with a string, int, undefined, null, false, etc. Strictly typed languages are a boon to large-scale development projects by embedding domain knowledge into the application while also providing a means to detect errors early on in the development cycle.

You can specify types in the documentation, though youâ€™ll have to add additional tooling to ensure that 1) the documentation actually matches the types expected by the server and returned in responses, and 2) that the types sent by the API consumer are actually the types that the server expected to receive (and vice versa â€" the types sent by the server may not be the types the client expected for various reasons).

The Klaviyo Solution: OpenAPI

OpenAPI is a way for API developers to specify an APIâ€™s structure resource schemas in a machine readable format. Generating the OpenAPI spec from the API serverâ€™s code can provide API consumers with a way to validate the types sent to and returned from the API. Below is an example of an OpenAPI specification, truncated for brevity.

```
openapi: 3.0.0
info:
  version: 1.0.0
  title: Books API
  description: A Books API based off the OpenAPI Example

servers:
  - url: https://example.com/v1
```

```
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
  schemas:
    Chapter:
      type: object
      required:
        - id
        - book_id
        - title
        - content
      properties:
        id:
          type: integer
        book_id:
            type: integer
        title:
            type: integer
        content:
            type: string
security:
  - BasicAuth: []

paths:
  /books:
    get:
      description: Returns a list of books
      parameters:
        - name: limit
          in: query
          description: Limits the number of items on a page
          schema:
            type: integer
        - name: offset
          in: query
          description: Specifies the page number of the books to be displa
          schema:
            type: integer

      responses:
        '200':
          description: Successfully returned a list of books
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  required:
                    - id
                    - title
                    - description
                    - author_id
                    - chapters
                  properties:
                    id:
                      type: integer
                    title:
                      type: string
                    description:
                      type: string
                    author_id:
                      type: integer
                    chapters:
                      type: array
                      items:
                        $ref: '#/components/schemas/Chapter'
```

Example OpenAPI spec for a book. Several endpoints are excluded for brevity â€" OAS specs can be many thousands of lines.

Discoverability and Documentation

We had to figure out how we wanted to present our documentation to our API users. As the developer of the API, you want to give users great documentation. Unfortunately, great documentation is not trivial to create. The API developer has to write and maintain these specs and documentation. And as discussed, nothing forces the specs, documentation and underlying APIs to remain in sync with each other. As the consumer of the API, it is necessary to know how the API works in order to interface with it.

Standards like JSON:API support limited discoverability for end users. More often than not, however, the only way for the API consumers to discover what features an API supports is to read through the documentation. This can be problematic when the documentation is stale or the API is large enough or the docs are presented in a difficult layout.

Having live, interactive documentation is great for an API consumer. Some of the best documentation is in the form of an interactive website which includes the documentation for how the API works alongside a tool that allows API consumers to make actual API calls and receive live responses directly on the page. For an API developer, though, interactive documentation is a challenge to create, maintain, and vet for security concerns.

The Klaviyo Solution: OpenAPI + Dedicated Engineers

Klaviyo has dedicated our Velocity and Developer Experience teams to integrate OpenAPI. From the OpenAPI spec, Klaviyo generates our developer portal and our SDKs. Keeping these tools up to date and solving for various edge cases is a lot of work, but we feel they are a necessity for successful long-term REST API development.

Versioning

Versioning was a hotly debated topic for us.

As your REST API changes over time, protecting your API consumers from backwards incompatible changes becomes a challenge, especially when you donâ€™t want to have to maintain separate versions of the same functionality indefinitely. We wanted to avoid the common solution where you have /api/v1/books and /api/v2/books both existing in the same code base.

Some of the questions we had to answer were:

- How long to keep the old version of the code before deprecating it?
- How to communicate the deprecation strategy and new functionality to your end users?
- How do you keep your system DRY while managing breaking changes?

REST APIs do not inherently communicate information about versioning. The JSON:API spec has a version, the API itself can have a version, and the resources exposed by the API can also have versions, though JSON:API doesnâ€™t give guidance on the latter two.

The Klaviyo Solution: A Revision Header

We decided to have a 12-month deprecation window for our API revisions, and to allow API consumers to specify their revision through a parameter when making an API request. These revisions map to separate viewsets and DTOs in order to allow us to expose new functionality without affecting previous revisions. Deprecated viewsets and DTOs only change when a security concern is surfaced in their implementation. This solves for both the API version changes as well as the resource version changes while enabling us to make breaking changes where necessary and only having to maintain the latest version of each codeset.

Implementation Details

Another pain point that we had to figure out was how to implement everything. We found that JSON:API does leave some things to the imagination:

- How to handle RPC calls?
- How and when to implement cursor-based pagination vs offset-based pagination?
- How to implement complex filtering logic across resources with disjoint data storage layers?
- How should â€œincludeâ€� actually work under the hood?

JSON:API doesnâ€™t attempt to prescribe a solution. The spec does reserve a â€œpageâ€� key which can be used for cursor (i.e. page[cursor]=â€¦) or offset (page[number], page[size]) pagination. Thereâ€™s also a â€œfilterâ€� key and an â€œincludeâ€� key â€” but the implementation is entirely up to us.

The Klaviyo Solution: A custom JSON:API Framework

Early on we thought of choosing the Django REST Framework JSON:API package, but we decided this could end up being too restrictive and so we set out to build our own JSON:API framework. We created a decorator strategy for linking resources, a router for registering viewsets, adapters for linking viewsets to existing services, and made Attrs (and cattrs) first-class citizens.

For pagination, we made the call that all of our resources would offer cursor-based pagination by default and the resources which have infrastructure in place for handling parallel queries could optionally offer offset-based pagination. The benefit of defaulting to cursor-based pagination is that this pagination strategy allows API consumers to request exactly the â€œnext pageâ€� regardless of changes to the underlying data. Additionally, offset-based pagination enables users to make multiple requests for separate pages in parallel and cursor-based pagination protects our systems from this additional load. When an API consumer has to wait for an API call to finish in order to have the next cursor, it slows down the rate at which the consumer can paginate through the API.

For filtering, we implemented our own grammar using the pyparsing library. Our list endpoints specify query DTOs where we define which filter operators each field supports. The base viewsets map these DTOs onto the request and parse the filters into a dictionary before handing them off to the underlying endpoint viewset as necessary. From there, the viewset can use a utility library to map the filter dictionary onto a Django ORM model or to optionally translate the filters into syntax compatible with their respective data stores as needed.

While a custom JSON:API framework is an option that is working for us, this is a large undertaking and requires a team to build and maintain. In case you, the reader, are looking for a cheaper / faster approach and arenâ€™t as concerned with the restrictions you may encounter, I advise you to check out existing JSON:API implementations like the Django Rest Framework JSON:API package.

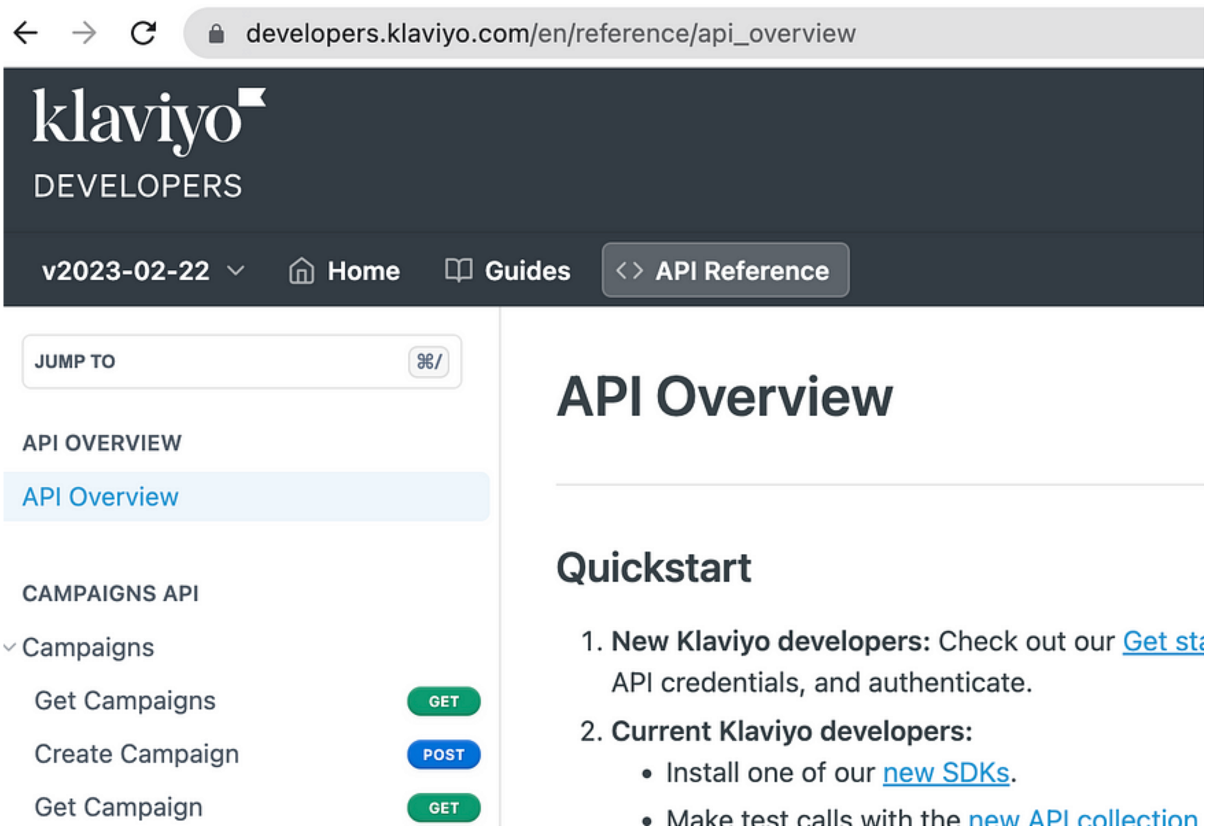
Performance

With the flexibility of filtering, including additional resources, different forms of pagination, and sorting on various fields comes load on the back-end systems. Allowing users to sort on fields which are not indexed or to join across multiple tables (and in some cases multiple databases) can result in expensive database calls and additional in-app processing to form the result structure. This often limits which of these features endpoints choose to support. While the JSON:API standard provides for these features in the specification, actually adding them to your system is often blocked not only by development time but also by underlying infrastructure.

Summary

A REST API with CRUD resources is familiar, flexible, and time-tested. The JSON:API specification provides a rigorous guide to REST API development. OpenAPI is a valuable tool for creating good documentation and ensuring accurate typing.

In my next post, Iâ€™ll be covering the pros and cons of GraphQL before continuing on to pain points common to both (pagination, rate limiting, authentication, and similar). Iâ€™ll conclude with the organizational side of things and how to get a new API initiative off the ground at your company.



API Overview

Quickstart

1. **New Klaviyo developers:** Check out our Get sta
   API credentials, and authenticate.
2. **Current Klaviyo developers:**
   - Install one of our new SDKs.
   - Make test calls with the new API collection

Klaviyoâ€™s developer portal