Powering our APIs using *attr*active DTOs

Author: Vedant Puri

Claps: 207

Date: Aug 22

Helping lay the groundwork for our new company-wide API was exhilarating. In early 2022, I was on the team that developed a proof of concept and established standards for our new APIs. Fast forward to today and youâ€™ll find over 150 endpoints spanning 12 product areas processing millions of requests daily. In this post, Iâ€™ll dive into one internal aspect of our APIs: Data Transfer Objects (DTOs). Iâ€™ll talk about why we chose attrs and how we use it. Iâ€™ll also show how we standardized the API implementation process for our developers including versioning of endpoints.

The API effort was herculean, spanning multiple engineering and product teams. It took discussions, technical design documents, and of course *some* bike-shedding! This quote from an early design document captures the vision:

> *â€œKlaviyo deserves a long-standing, consistent & flexible API that can serve developers inside and outside of Klaviyo for years to come while minimizing operational overhead for our internal developers and maximizing consistency and usability for our external developers.â€�*

Setting the scene

Our API complies with the JSON:API spec. Chad Furman from the API team wrote a great post on why we chose JSON:API and how we use it. Our implementation is in Python, using the Django Rest Framework (DRF). We leverage DRFâ€™s composable and flexible nature to customize its various components in our API.

Broadly, the implementation is as follows:

- API routes are registered on a Router object that dispatches the incoming request (body, query parameters, headers, etc.) to the respective ViewSet class.
- Custom authentication, permissioning, and rate limiting logic is plugged into DRF by configuring them on the ViewSet class. These are invoked by DRF on incoming requests.
- The different HTTP methods (GET, POST, PATCH, etc.) implemented on the ViewSet class process the incoming request by calling internal services. This usually passes through an adapter layer to massage the payload in each direction, finally returning an HTTP response.

Using Data Transfer Objects (DTOs)

In Python, it is easy to represent key-value data (e.g. JSON payloads) using plain old dictionaries, but that convenience can prove costly:

- Lack of structure: Dictionaries are loose. There are no boundaries on how they should look. It is easy to make mistakes such as typos, extra values, and insufficient values.

- Mutability: Dictionaries in Python are mutable and if youâ€™ve used the language for a while, you already know that this can lead to all sorts of nasty bugs.

Fundamentally, DTOs are objects that encapsulate only data â€” they have little to no behavior (serialization logic at most) within them. These are also known as *data-only classes* or *dataclasses*. Such classes (or classes in general) enforce a strict schema during instantiation. It is also easily possible to implement these to instantiate immutable objects. Plus, as we will see later, DTOs allow adding type hints to the attributes, which drastically improves the readability of the code.

We use DTOs to represent our API contract. Each endpoint (HTTP method) has an associated ingress DTO that represents the JSON body of the incoming request along with the query parameters and a relevant response DTO that represents the JSON body returned to the client. For example, when creating a catalog item using our API, the request and response data dictionaries are modeled as DTOs.

We discourage generic, sparsely instantiated DTOs that could be reused across multiple endpoints. Even though this adds some redundancy, it provides clear, strict schema enforcement and also results in a modular design that makes it easy to version contracts of different endpoints independently. Furthermore, this exclusive DTO-endpoint binding helps simplify the auto-generation of the public API documentation.

At the time, a few libraries already had great solutions for creating data-only classes without developers needing to write the boilerplate code that is usually required for a standard Python class. The most popular ones were: dataclasses, pydantic and attrs. I wonâ€™t go into too much detail comparing these three since there are plenty of articles out there (see Attrs, Dataclasses and Pydantic and Why I use attrs instead of pydantic).

At a high level, the first decision was between attrs / dataclasses and pydantic. The first two are similar but quite different from pydantic. Pydantic is primarily a validation library rather than a data container. Although it was tempting to use pydantic here since it fit our use case, we decided against it mostly for performance reasons. Our DTOs need to be instantiated synchronously on our API web tier on each API request, and thus every potential performance bottleneck matters. This blog post has some fascinating research and benchmarks on the performance of these libraries.

We settled on attrs since it is performant, feature rich (compared to data classes), flexible, and also simple to use. Plus, since attrs is not part of the standard library (unlike dataclasses) incorporating new features does not require a Python version upgrade. Personally, I really like their decorator style pattern instead of inheritance that pydantic uses. They philosophically favor composition over inheritance which makes it more transparent and easy to customize for our use case. attrs attaches methods to the class and once the class generation decorator executes, it is a plain old Python class.

> â€œIt does *nothing* dynamic at runtime, hence zero runtime overhead. Itâ€™s still *your* class. Do with it as you please.â€� â€” attrs docs

The Klaviyo API DTO

Generally, it is considered good practice to wrap third party libraries when reasonable. First, it facilitates uniform usage of, for example, desired settings and defaults of a library with many options. Second, it creates a central spot to apply universal changes. Third, it offers the opportunity to abstract the details of the library thus providing flexibility to swap it out for an alternative, in which case the wrapper serves as an adapter. For all those reasons, we wrapped attrs in the tooling we give our developers.

Before unraveling that tooling, letâ€™s look at a toy example. Imagine a simple API for Books in a library. Users of this API would want to query for books using search parameters. Hereâ€™s how one of our developers would write the query request DTO for the API:

```
from app.views.apis.v3.dtos import api_dto, field
from app.views.apis.v3.validation import common_validators

@api_dto(ApiResourceEnum.BOOK, enable_boolean_filters=True)
class BookQueryDTO:
    id: str | None = None
    title: str | None = field(
        default=None,
        external_desc="Title of the book you are querying for",
        example="Harry Potter and The Sorcerer's Stone",
        filter_operators={FilterOperators.CONTAINS, FilterOperators.EQUALS
        validator=common_validators.max_len(100)
        sortable=True
    )
    author_id: str | None = field(
        default=None,
        filter_operators={FilterOperators.EQUALS},
    )
    page_cursor: str | None = None
    return_fields: list[str] | None = None
    sort: str | None = None
```

The example above knits together a few important pieces that we will discuss in the upcoming sections:

- The `@api_dto` decorator
- The attrs `field` wrapper
- The `common_validators` module for request validation

Our @api_dto decorator: Wrapping attrs @define

Our `@api_dto` decorator is implemented with code like this:

```
from attrs import define, resolve_types

def api_dto(
    resource: ApiResource,
    enable_boolean_filters: bool = False,
    non_dto_sort_fields: list | None = None,
    min_max_page_size: tuple[int | None, int | None] | None = None,
) -> Callable:

    # ...
    # Arg validation
    # ...

    def inner(py_dto_cls: type) -> ApiDtoClass:
        generated_attr_dto = resolve_types(
            define(frozen=True, kw_only=True, auto_attribs=True)(py_dto_cl
        )

        setattr(generated_attr_dto, "__api_dto__", True)
        setattr(generated_attr_dto, "__resource__", resource)
        setattr(generated_attr_dto, "__boolean_filters_enabled__", enable_

        if non_dto_sort_fields:
            setattr(generated_attr_dto, "__non_dto_sort_fields__", non_dto

        if min_max_page_size:
            setattr(generated_attr_dto, "__min_max_page_size__", min_max_p

        # ...
        # More Validation to ensure proper setup
        # ...

        return generated_attr_dto

    return inner
```

It applies the attrs define decorator to the passed in class, with a predetermined configuration:

- `frozen=True`
  These DTOs should be frozen, to enforce immutability throughout their lifecycle in the API request. This also makes the object hashable, which is beneficial for caching requests and responses.
- `kw_only=True`
  Since these DTOs would likely have multiple attributes, for the sake of clarity, these must be instantiated only using keyword arguments.
- `auto_attribs=True`
  This is a nice feature of attrs that circumvents the need to assign each attribute to a field. It also enforces type annotations.

One more important detail here is that the define decorator by default generates a slotted class (`slots=True`), thus these DTOs have a light memory footprint, one more factor that helps with scale.

Even though attrs has several other parameters in the define decorator, we havenâ€™t needed them so far for our API DTO, and our wrapper shields our internal developers from having to think about them.

Finally, we `resolve_types()` on this decorated class to allow string type hints for forward references. This ensures the type of each attribute is defined and ready to use for serialization / deserialization.

You may have noticed that once this class object has been generated, the next few lines set a few attribute values on the class (not instance):

- The `__resource__` attribute refers to an `ApiResource` object.
  This object stores the type of the resource that this DTO models, among other things. This is then used by serialization and documentation tooling. Each domain has an enum that holds all its `ApiResource` objects. Then on the decorator, the enum is provided, e.g. ApiResourceEnum.BOOK.
- The `__api_dto__` is a flag that indicates this class was generated using this decorator. This works as a watermark that is verified during the DTO registry to make sure all API DTOs are being generated from this decorator.
- The `__boolean_filters_enabled__` attribute is a switch to allow the fields within the DTO to be filtered using AND / OR / NOT boolean operators.
- `__non_dto_sort_fields__` and `__min_max_page_size__` help parse and process request query parameters for this DTO.

Our API DTO field: Wrapping attrs field

attrs allows attaching metadata to an attribute, which as it turns out, is quite nifty! We use it to store information for that attribute which is used by different parts of the API: documentation, filtering, redaction, etc.

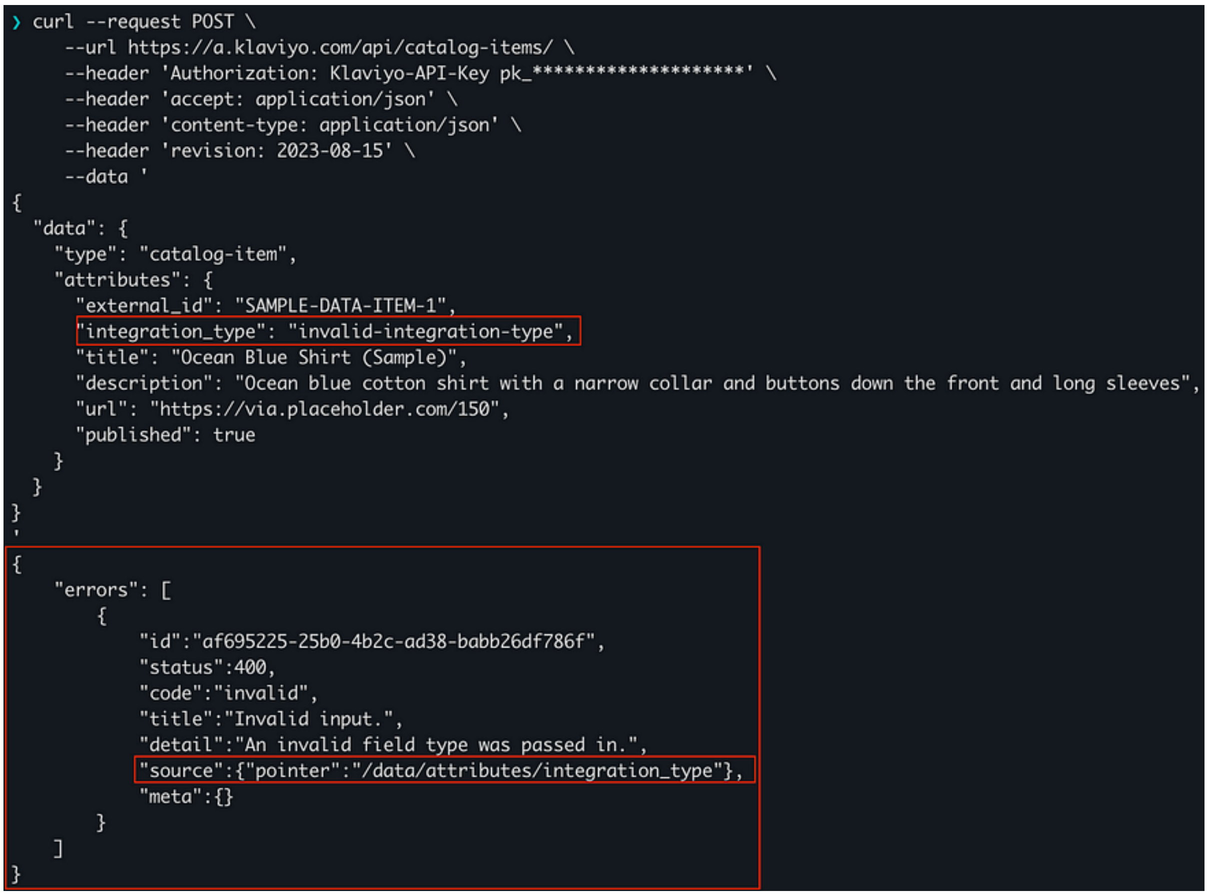
Instead of relying on developers to set values in this dictionary free-form, we added a simple wrapper around the attrs field function. This wrapper provides a consistent interface for setting additional keyword arguments like filter_operators, sortable, external_desc, and more (see below). Here is a snippet that represents our field wrapper:

```
from attrs import field as attrs_field

def field(
    *args,
    filter_operators: set[FilterOperators] | None = None,
    non_filterable: bool = False,
    sortable: bool = False,
    accept_multiple_query_param: bool = False,
    external_desc: str | None = None,
    example: Any | None = None,
    data_classification: DataClassification = DataClassification.DEFAULT,
    meta: bool = False,
    **kwargs,
):
    # ...
    # Parse and validate args
    # ...

    # ...
    # Construct field metadata in a standardized fashion (fixed keys, inte
    # eg. metadata["__external_desc__"] = external_desc
    # ...


    return attrs_field(*args, **kwargs, metadata=(metadata or None))
```

This structures the metadata in a predictable, clean and robust way. There are a few interesting arguments in that wrapper:

- `filter_operators` is used to specify the possible filter operators for this field in an API request. We have our own filtering grammar (implemented using pyparsing) that parses the JSON API filters and validates the request using the operators specified here. This kwarg is just the tip of the iceberg and I think that our API filtering grammar deserves a post of its own.
- `external_desc` and `example` fields are used by an in-house tool that generates the OpenAPI spec documentation. This streamlines documentation updates with DTO code changes (our API contract). Developers simply configure the new information on the DTO field using this kwarg, and docs are updated with that information!

Validation Toolbox



```
❯ curl --request POST \
    --url https://a.klaviyo.com/api/catalog-items/ \
    --header 'Authorization: Klaviyo-API-Key pk_********************' \
    --header 'accept: application/json' \
    --header 'content-type: application/json' \
    --header 'revision: 2023-08-15' \
    --data '
{
  "data": {
    "type": "catalog-item",
    "attributes": {
      "external_id": "SAMPLE-DATA-ITEM-1",
      "integration_type": "invalid-integration-type",
      "title": "Ocean Blue Shirt (Sample)",
      "description": "Ocean blue cotton shirt with a narrow collar and buttons down the front and long sleeves",
      "url": "https://via.placeholder.com/150",
      "published": true
    }
  }
}
'
{
    "errors": [
        {
            "id":"af695225-25b0-4b2c-ad38-babb26df786f",
            "status":400,
            "code":"invalid",
            "title":"Invalid input.",
            "detail":"An invalid field type was passed in.",
            "source":{"pointer":"/data/attributes/integration_type"},
            "meta":{}
        }
    ]
}
```

Example of a request failing a validation check.

As mentioned earlier, we use DTOs to represent the JSON body in a request. We added in a layer that would give us the warm fuzzy feeling of rejecting an invalid payload even before it makes its way to the internal service boundary!

This validation would occur synchronously on the API web servers, and thus, we needed to be cautious about the extent of validation for these DTOs. For example, making a database call was not something we wanted to do here; that would happen at the internal service boundary. The idea was to have lightweight validation that would be just enough to reject bad payloads from using resources deeper in the stack unnecessarily.

attrs makes it easy to validate these data classes by specifying a validator function as a kwarg on the field. These validators are run on object instantiation (deserialization of raw JSON into request DTO in this case). Our internal developers have access to a lean wrapper around these validators that produces consistent error messages. Using a decorator to define the error message, we can now relay back an HTTP Response with a 400 status. Normally, we add rigorous validation to DTOs representing requests and not so much on those for responses. This is because we have control over the generation of the latter and can ensure correctness using automated testing.

A Python module in our API codebase encapsulates generic DTO validators available for use by all teams. Of these validators, many are just wrappers around attrs validators while others build off those. These form a toolbox that is used when implementing DTOs. Many teams end up writing their own validator modules, specific to their domain, built off of these base validators. If a validator is generic enough to be useful to other teams, it makes its way into the base validator module.

We also have a module that maintains Python closures that produce validator functions. The idea here is that sometimes different teams may end up implementing similar validators that have the same validation logic, just different â€œarguments.â€� Having this module helps DRY up that redundancy. A trivial example of this closure would look like this:

```
def divisible_by__validator_closure(divisor: int) -> Callable:
    if not isinstance(divisor, int):
        raise ValueError(f"divisor must be of type int, got {type(divisor)

    if divisor == 0:
        raise ZeroDivisionError("Cannot use 0 as a divisor")

    @api_custom_validator
    def generated_validator_fn(instance, attribute, value):
        if value % divisor != 0:
            raise ValueError(f"{value=} is not divisible by {divisor=}")

    return generated_validator_fn

# Example use:
# divisible_by_two_validator = divisible_by__validator_closure(2)
```

That *wraps* up (sorry, I couldnâ€™t resist) how DTOs are created for Klaviyoâ€™s API. JSON:API relationships are also modeled within these DTOs, but for the sake of brevity, we wonâ€™t cover them in this post.

Registry of DTOs and ViewSet metaprogramming

So far in this post, we uncovered what DTOs represent in our APIs and how they are created in a standardized manner. But, how does each version of an API endpoint know which DTO to use? Furthermore, once that is resolved, how is the inbound raw JSON converted to this DTOs (and similarly in the other direction)?

To answer those questions, letâ€™s understand how our ViewSet classes are implemented and versioned. Using the Books API example from above, this is what a Klaviyo API ViewSet would look like:

```
class BooksViewSet(BaseApiViewSet):
    @api_revision(
        "2020-01-01",
        ingress_dto_type=BooksListQuery,
        egress_dto_type=BooksResponse,
    )
    def list(self, request: Request, request_dto: API_DTO) -> JsonApiRespo
        ...

    @api_revision(
        "2023-06-01",
        ingress_dto_type=BooksListQuery,
        egress_dto_type=BooksResponse,
    )
    def list(self, request: Request, request_dto: API_DTO) -> JsonApiRespo
        ...

    @api_revision(
        "2020-05-05",
        auto_deprecate=False,
        ingress_dto_type=BookCreateQuery,
        egress_dto_type=BookResponse,
    )
    def create(self, request, request_dto: API_DTO) -> JsonApiResponse:
        ...
```

There are a few interesting things going on in the example above and we will be taking a look at how itâ€™s bootstrapped under the hood (including the magic that makes it possible to have methods with the same name in the same class, without any real overloading).

There are roughly three steps in preparing the Klaviyo API ViewSet class:

1. The `@api_revision` decorator populates a global registry of all methods for all ViewSets to specific revisions, keyed by class name. For example:

```
{
    "BooksViewSet": {
        "list": [Revision(...), Revision(...)],
        "create": [Revision(...)],
    },
    "FooViewSet": {
        "list": [Revision(...), Revision(...), Revision(...)],
        "create": [Revision(...)],
        "retrieve": [Revision(...), Revision(...)],
        "update": [Revision(...)],
        "partial_update": [Revision(...)],
        "destroy": [Revision(...)],
    },
    "BarViewSet": {...}
}
```

The `Revision` objects in the registry above are simple dataclasses:

```
@dataclass
class Revision:
    """A single API method revision's information, defaults are set in the
    revision_date: str
    func: Callable
    auto_deprecate: bool
    deprecation_date: str
    removal_date: str
    ingress_dto_cls: Type[API_DTO]
    egress_dto_cls: Type[API_DTO] = None
```

This means each method revision has a DTO bound to it and is stored in that global registry.

2. The `BaseApiViewSet` is built using the `ApiV3Metaclass` metaclass. The metaclass reads this global registry and attaches a mapping of method name to all endpoint revisions as a class attribute on the respective ViewSet.

The metaclass looks roughly like this:

```
class ApiV3Metaclass(type):
    def __new__(mcs, class_name, bases, attrs):
        # attributes to attach to the class
        attrs_to_build = dict()

        # collection of our api methods and revisions, we want to structur
        # to make incoming requests as fast as possible to route at runtim
        # { method_name -> [Revision(...), Revision(...), ...] }
        revision_list_by_viewset_method = defaultdict(list)

        # Create the revision methods on the class based on the revision f
        # @api_revision decorator
        for (
            viewset_method_name,
            revisions,
        ) in _funcs_and_revisions_by_class_and_method[class_name].items():
            revision_list_by_viewset_method[viewset_method_name] = sorted(
                revisions,
                key=lambda revision: RevisionDate(revision.revision_date),
                reverse=True,
            )

        attrs_to_build["revisions_by_method"] = revision_list_by_viewset_m

        # ...
        # More setup
        # ...

        return super(APIV3Metaclass, mcs).__new__(
            mcs, class_name, bases, attrs_to_build
        )
```

Each viewset then has a `revisions_by_method` attribute that looks like:

```
{
  "list": [Revision(...), Revision(...), Revision(...)],
  "create": [Revision(...)],
  "retrieve": [Revision(...), Revision(...)],
  "update": [Revision(...)],
  "partial_update": [Revision(...)],
  "destroy": [Revision(...)],
}
```

There is an interesting Python interpreter detail that makes the decorator work seamlessly with the metaclass to make this setup possible:

In Python, the class body is executed **before** the class is set up using the determined metaclass. There are more details about this process here, but for our scenario this means that the decorator executes first (populating the global registry) followed by the execution of the metaclass `__new__` method, which uses this global registry to create a class attribute that stores revisions by method.

The decorated methods never really get attached to the class, and only exist as a reference in the `Revision` object. This is why it is possible to have methods with the same name!

3. The base method (from `BaseApiViewSet`) looks up the method to call based on version header

The `BaseApiViewset` class holds a trivial implementation to all the ViewSet action methods (list, create, retrieve etc.). This simple piece is actually what brings it all together:

- (Recap) The router dispatches the request to the respective ViewSet class. Since the decorated methods were never attached, the only implementation of these methods that exists is from the base method, which gets invoked here.
- The base method parses the request header to get the revision date for the endpoint being requested. It gets the specific `Revision` object from the `revisions_by_method` lookup on the ViewSet class. Recall that this `Revision` object holds the DTO and function reference to the specific version of the endpoint.
- Finally, the serializer structures the JSON into the DTO bound to that revision and passes it to the function, executes the function and serializes the response back into JSON on the way out!

All API ViewSets inherit from the `BaseApiViewset` and work using this machinery.

Serialization

Our API uses cattrs to accomplish the serialization / deserialization to and from JSON. This is a convenient Python library that helps structure unstructured data (like dictionaries) and vice versa. This library is powerful and provides a wide range of possible conversions. (Even though it may not sound like a huge deal, I thought that the ability to convert raw values to Enums was convenient and pretty neat.)

cattrs is very well integrated with attrs, which made it an easy choice for our APIs. I also like the exception handling in cattrs using ExceptionGroups: It is useful in the serializer layer where we would need to pinpoint (either externally or internally) exactly where the DTO failed to create and why.

The usage of cattrs is internal to the API system. We have a specific `APIConverter` class that has some default hooks registered to it and a registry where other teams can contribute to handle specific rare edge cases. The default hooks are useful beyond just providing a variety of transformations. In some cases a registered hook in here can act as a global validator for a particular type. For example, dates and datetime types can accept a variety of formats and this converter takes care of validating those (as opposed to each DTO having to validate it) and raises validation error if something is wrong.

Overall we have a vanilla usage of cattrs and it has proven to be effective.

Conclusion

Summary of what I described:

- Our APIs represent the request body, query parameters and responses using DTOs.
- We wrapped attrs to simplify and standardize its use for our developers.
- We employed Python patterns like decorators and metaclasses to simplify implementation at scale, version the API, and bind DTOs to endpoint versions.
- Supplementing attrs with cattrs simplified the serialization and deserialization of our DTOs.

Weâ€™re happy with the decisions we made in early 2022. As compared with, say, each endpoint directly accepting and producing JSON or dictionaries, our DTO-based approach helps achieve the vision mentioned at top:

> *â€œKlaviyo deserves a long-standing, consistent & flexible API that can serve developers inside and outside of Klaviyo for years to come while minimizing operational overhead for our internal developers and maximizing consistency and usability for our external developers.â€�*