PRODUCTION READY GRAPHQL

Public Schemas Review & Analysis

Introduction

One of the best ways to learn how to build GraphQL APIs is to look at examples out there.

I have analyzed three examples of GraphQL APIs and shared some thoughts on what's great, some tradeoffs that were made, and some things that could potentially be improved. Enjoy!

- Marc-Andre

Shopify's Public API

Shopify was one of the earliest adopters of GraphQL and one of the earliest adopters of GraphQL **as a public API.** This makes it an amazing pick for schema reviews as there is a ton to learn from their API. Let's get started.

Documentation

Reference URL: https://shopify.dev/docs/admin-api/graphql/reference

The first thing to look at when exploring a new API is the documentation. Shopify's documentation is great, however, it is guilty of documenting mostly in terms of GraphQL concepts.

The main menu for the GraphQL API offers us "QueryRoot", "Queryable objects", "Mutations", and "Custom scalars". This is the first thing we see as someone willing to integrate with the API.

The "Admin API" section does highlight what's possible to achieve using the API, but it seems like there are missing docs on how to achieve certain use cases using GraphQL. There is a good list of tutorials, however, which is great.

On every object type's reference, fields seem grouped by "Connections" or "Fields", which seems odd. This seems again related to GraphQL internal concepts and not necessarily helpful for integrators.

On the plus side, certain types like Customer have a samples section, which has an embedded GraphiQL with example queries. This is absolutely great to let potential clients quickly understand how to query a certain type and the use cases it solves.

Rate Limiting

Shopify uses a complexity approach for rate-limiting similar to what we covered in the book. In the book, we covered GitHub's technique for letting integrators estimate the cost of queries using a special `rateLimit` field on the query root. Shopify uses a slightly different approach for this, which is great. Instead of a field on the root, this information is sent through the `extensions` key in the response:

```
"extensions": {
  "cost": {
    "requestedQueryCost": 12,
    "actualQueryCost": 12,
    "throttleStatus": {
      "maximumAvailable": 1000,
      "currentlyAvailable": 988,
      "restoreRate": 50
    }
  }
}
```

This approach is great because this information is transmitted outside of the schema itself. The downside is that it can't use the schema itself for documentation.

Plural Fields, Specific Finder fields

Most of Shopify's finder fields offer both a singular and plural version which is great for static queries. Not only that, but they don't only offer `node` and `nodes` fields. Most nodes can be fetched through typed fields.

```
type QueryRoot {
  customer(id: ID!): Customer

  customers(
    first: Int,
    after: String,
    last: Int,
    before: String,
    reverse: Boolean = false,
    sortKey: CustomerSortKeys = ID,
    query: String
  ): CustomerConnection!
}
```

While most fields offer plural versions, notice that some don't. For example, there is an `app` field, but no apps fields. It's hard to consistently offer both. Maybe this is not a use case that was needed yet, or maybe it's something that was forgotten. This kind of consistency can make a great API, this is possibly something to be added to your linters and/or tests!

Shared Input Types

```
input ProductVariantInput {
  # Specifies the product variant to update or create a
  # new variant if absent.
  id: ID

  # ...
}
```

Something I noticed about Shopify's schema is that they reuse input types across different mutations. For example, the input type ProductVariantInput is used for both the `productVariantCreate` and the `productVariantUpdate` mutations.

As we covered in the book, I'm not convinced about this approach. While it has advantages, like the ability for clients to share code generated structures for similar inputs, it makes the schema a bit less strong. In this example, a description had to be added to the id field to explain it is only to be used when updating a variant. This has to be a runtime check and we lose a bit of the beauty of the GraphQL schema.

Asynchronous Mutations

Another clever idea from Shopify is to use asynchronous mutations. Some mutations may be too slow to respond to in a synchronous manner. While typical HTTP endpoints would return a 202 ACCEPTED status code, Shopify's GraphQL API returns a mutation payload with a `job` field.

```
mutation discountAutomaticBulkDelete {
  discountAutomaticBulkDelete {
    job {
      id
    }
  }
}
```

Using that job ID, clients can then poll a job field for updates. The cool thing is that a Job object as a `query` field which exposes the query root type. This means that clients may refetch anything they want from the graph when the job has completed!

```
query PollJob {
  job(id: “abc123”) {
    done
    query {
       ...
    }
  }
```

Batch Mutations

As much as specific, finer-grained mutations are usually a better client experience, sometimes, coarser-grained, batch style mutations are needed to achieve specific use cases. Shopify addresses this on certain mutations with input types that different lists of IDs to add, remove and or update. For example on this `PriceRuleCustomerSelectionInput` input type.

```
input PriceRuleCustomerSelectionInput {
  customerIdsToAdd: [ID!]
  customerIdsToRemove: [ID!]
}
```

Make Impossible State Impossibles

Batch mutations are great, but sometimes they make an API more confusing to use. In the same example, we see a schema that is quite hard to reason about for a client. This input type allows clients to select customers for which a `PriceRule` applies. As you can see al inputs are optional. What happens if `forAllCustomers` is set to true, but `customerIdsToAdd` contains a list of customer ids? The savedSearchIds field contains customers that match a certain search query. What if we pass those but also include all customers? This is a good example of an API that allows for impossible states.

```
input PriceRuleCustomerSelectionInput {
  customerIdsToAdd: [ID!]
  customerIdsToRemove: [ID!]
  forAllCustomers: Boolean
  savedSearchIds: [ID!]
}
```

Input unions could potentially improve this API down the line. Another idea would be to split the massive `priceRuleUpdate` mutation into more usable mutations.

Bulk Operations

The last thing we'll look at is Shopify's bulk operations. Bulk operations allow clients to run asynchronous, longer running queries. In the book, we talked a bit about the tension between data-intensive APIs vs more business oriented APIs. Bulk operations allow Shopify to offer both use cases.

```
mutation {
  bulkOperationRunQuery(
   query: """
    {
      products {
        edges {
          node {
            id
            title
          }
        }
      }
    }
    """
  ) {
    bulkOperation {
      id
      status
    }
  }
}
```

How meta is this? Clients can use the GraphQL API to enqueue a GraphQL query to be run asynchronously. Notice how the products field is not actually paginated by the client here. This is because Shopify's bulk operation system will auto paginate and return the entire result in a JSON file. This is really great: It avoids performance issues on Shopify's side and rate limit issues on the client's side. Something to consider for public GraphQL API providers wanting to provide more data-oriented use cases.

WordPress GraphQL API

Wordpress is another very active public GraphQL API. Because of how common word press is, it seemed like a great choice for a schema review and analysis. Let's see what they got!

Reference: https://docs.wpgraphql.com/

Documentation

I was very pleased by wp-graphql's documentation. The sidebar for the docs doesn't split things up around GraphQL concept but around domain concepts like "Posts", "Pages", "Users", etc.

Every subpage for domain concepts starts with example queries, which is also a really great idea. I was able to quickly grasp what was possible with the API. This is truly a great example of GraphQL documentation. Even arguments are explained further with more example queries.

Under every domain concept, we also find mutations that can be used against that concept. This is a great example that shows that naming our mutations `postCreate` or `createPost` doesn't matter as much as how we document these things, and how our tools can categorize our concepts. Kudos to wp-graphql for this great documentation site.

Schema Design

The pageBy field

It’s possible to fetch pages using wp-graphql’s API by using a pageBy field.

```
query GET_PAGE_BY_URI {
  pageBy(pageId: 1) {
    title
  }
}
```

The problem is that this field may take three different inputs:

```
query {
  pageBy(
    pageId: 1,
    id: 1,
    uri: “ ... ”,
  ) {
    title
  }
}
```

As we saw, this field design makes the API a bit awkward. Are all these fields optional? Are all of them required? It’s hard to know from the schema. The intent here is to provide three different ways to fetch a page. Why not offer three different fields? I think this would be a potential improvement to the API.

```
query {
  pageByPageId(pageId: 1) { .. }
  pageById(id: “abc”) { .. }
  pageByURI(uri: “ ... ”) { .. }
}
```

The where clause

We talked a bit about the danger of overly generic fields in the book. There is an interesting example of such a field in the API. Posts can be filtered using an argument called `whereArgs`

```
query GET_POSTS($first: Int, $dateQuery: DateQueryInput) {
  posts(first: $first, where: {dateQuery: $dateQuery}) {
    edges {
      node {
        id
        title
        date
      }
    }
  }
}
```

The where argument has quite a list of possible input values. They map to a word press concept behind the scenes which might make it a bit better. In either case, it seems like this API could be split into more specific fields. I'd also be worried about possible performance issues of such a generic, SQL-like search filter. However, it may be an actual use case in which case this field could be well designed.

Recursive data structure

GraphQL can't express recursive data structures very well since clients must select leaf fields. This sometimes leads schema designers to use Strings or other custom scalars like JSON to "exit" the GraphQL schema and return a potentially recursive data structure. Comments have this structure in wp-graphql's API:

```
type Comment {
  children: CommentToCommentConnection!
}
```

This is a great design choice, since clients are unlikely to want to load an infinite nesting of comments anyways, and can always fetch more if needed.

Connections

The API has an interesting naming convention around paginated connections. It uses the owner type name and the connection node type to build a name like `CommentToCommentConnection`. This is really interesting. While this name doesn't necessarily convey a lot of domain meaning, it has a very nice side effect of being unique and specific to a certain relationship. This ends up being quite a nice design

```graphql
type Query {
  comments: RootQueryToCommentConnection!
}
```

Mutation Design

Mutations are tricky to design. I like how the API classifies mutations under domain-specific sections and that mutation titles are how to achieve certain use cases rather than the name of the field. For example, under "Add Tag to Post", we have this example of the `updatePost` mutation:

```graphql
mutation ADD_TAG_TO_POST($input: UpdatePostInput!) {
  updatePost(input: $input) {
    post {
      id
      title
      tags {
        nodes {
          id
          name
        }
      }
    }
  }
}
```

Here are the variables that go with this mutation example:

```
{
  "input": {
    "clientMutationId": "UpdatePost",
    "id": "cG9zdDoxMzY0",
    "tags": {
      "append": true,
      "nodes": [
        {
          "name": "New Tag"
        }
      ]
    }
  }
}
```

The `updatePost` mutation takes a tags input object which allows us to modifies the tags on a post. To allow us to append tags rather than setting all tags as a unit, the API supports an optional `append` argument. To me, this makes the API a bit hard to use. A separate `addTagsToPost` mutation would've been nicer to use by clients, and the `updatePost` mutation does not support removing tags anyways. I think this is a good example of why providing specific fields to achieve clear use cases is often better than overloading generic fields!

GitLab GraphQL API

GitLab has a great public GraphQL API with some really interesting design choices. It was a perfect pick to learn more about schema design

Reference: https://docs.gitlab.com/ee/api/graphql/

Documentation

GitLab's documentation is great. Their getting started guide helps users understand how to use their platform wonderfully. It would have been nice to see more documentation on **what** is possible to do with the API. Example use cases or workflows, for example, would be really useful.

I like how the query root is described on the landing page of the API, which helps integrators getting started with main queries. It also describes the main domain concepts a user might interact with using the API.

Naming

It looks like the naming of mutations could use a bit more consistency. The `verbEntity` naming pattern is applied to some mutations, but not on others:

```
type Mutation {
  updateEpic(input: UpdateEpicInput!): UpdateEpicPayload

  epicAddIssue(input: EpicAddIssueInput!): EpicAddIssuePayload
  epicSetSubscription(input: EpicSetSubscriptionInput!):
EpicSetSubscriptionPayload
}
```

As we saw in the book, symmetry is also important in APIs. One mutation uses the wording delete while others use destroy:

```
type Mutation {
  designManagementDelete(input: DesignManagementDeleteInput!):
DesignManagementDeletePayload
  destroyNote(input: DestroyNoteInput!): DestroyNotePayload
}
```

In general, there seems to be a lack of descriptions of fields and mutations. Sometimes we may feel like fields are self-explanatory, but that's not always the case to external users. Remember that not all integrators will know the domain concepts, as well as your team, does.

Implementation Concerns

I noticed something interesting in a mutation:

```
“””
Updates a DiffNote on an image (a Note where the position.positionType is
"image"). If the body of the Note contains only quick actions, the Note
will be destroyed during the update, and no Note will be returned
“””
updateImageDiffNote(input: UpdateImageDiffNoteInput!):
UpdateImageDiffNotePayload
```

Something seems odd about the fact position.positionType needs to be documented this way. Is it possible it's missing a type here? There's also a clever side effect here that needs to be documented. This may be totally fine if users are well used to this pattern in the UI.

Either way it's possible the design could be improved by being more specific and maybe introducing a new mutation for quick actions specifically.

Fine-Grained Mutations

I like the design of mergeRequest mutations in GitLab's API. They use very specific and optimized mutations for certain use cases.

```
type Mutation {
  mergeRequestSetAssignees(input: MergeRequestSetAssigneesInput!):
MergeRequestSetAssigneesPayload
  mergeRequestSetLabels(input: MergeRequestSetLabelsInput!):
MergeRequestSetLabelsPayload
  mergeRequestSetLocked(input: MergeRequestSetLockedInput!):
MergeRequestSetLockedPayload
  mergeRequestSetMilestone(input: MergeRequestSetMilestoneInput!):
MergeRequestSetMilestonePayload
}
```

Notice the choice of accepting a list of IDs on each of these mutations. These let us write static queries and batch adds rather than using multiple mutation fields in an operation.

Global IDs

GitLab uses Global IDs for nodes, but does not expose node or nodes fields on the query root. This may make things a bit harder for Relay clients.

```
type Query {
  project(fullPath: ID!): Project
}

type Project {
  id: ID!
  fullPath: ID!
}
```

The ID scalar is used in two different ways across the API which may cause problems down the line. It also makes it a bit confusing for clients to know what kind of ID they're dealing with. For example, the fullPath field returns a full path to a project, like this:

```
"gitlab-org/gitlab-foss"
```

Using paths like these as global IDs is not a bad idea at all. It is globally unique and while not opaque, it does act as a good ID. In other places, it looks like GitLab uses Rails GIDs:

```
{
  "id": "gid://gitlab/PersonalSnippet/1955940"
}
```

The team possibly should've looked into using another scalar for project paths instead of reusing ID. As far as opaque IDs, I generally advise keeping all global IDs opaque but this may work well for others.

As you can see, these IDs reveal implementation details like the model name and probably the database ID which might encourage users to build their own IDs. This makes things much harder to change down the line.

Learning from Deprecations

One of the best ways to learn is from mistakes. I've taken a look at some GraphQL deprecations and see if a better design could have avoided them.

GitHub

```
type AssignedEvent {
  user: User! @deprecated(reason: "Assignees can now be mannequins. Use 
the assignee field instead. Removal on 2020-01-01 UTC.")
  assignee: Assignee!
}
```

Our first example is from GitHub. The AssignedEvent is a GitHub event in which someone gets assigned to an Issue for example. Initially, to represent who was assigned to such an issue, a field `user` was added, with type User. Quickly the team realized a User was not the only type that was possible in this context. An organization or bot could be assigned for example.

Instead, a field `assignee` with type `Assignee`, a union type was added. What we can learn from this is that API evolution can be greatly helped by using context-specific type and avoiding too much reuse of our basic types. We could even push this further by introducing a specific `Assignment` type, which would allow us to encode data on the assignment itself, outside of the event.

```
type Assignement {
  on: DateTime!
  assignee: Assignee!
}
```

And here's another one from the GitHub API related to naming.

```
type EnterpriseBillingInfo {
  availableSeats: Int! @deprecated(reason: “availableSeats will be
replaced with totalAvailableLicenses to provide more clarity on the value
being returned Use EnterpriseBillingInfo.totalAvailableLicenses instead.
Removal on 2020-01-01 UTC.”)
  totalAvailableLicenses: Int!
}
```

More specific naming would have helped here. It's possible that seats may be jargon for licenses, but outside that context made that field hard to reason about for integrators. `totalAvailableLicenses` clearly conveys what the field returns which makes it a great improvement.

Note the great description message in that example. It clearly answers why that field is getting deprecated, and what should be used instead.

Shopify

```
type Query {
  automaticDiscount(id: ID!): DiscountAutomatic @deprecated(reason: “Use
automaticDiscountNode instead”)
  automaticDiscountNode(id: ID!): DiscoundAutomaticNode
}
```

Another interesting one comes from Shopify’s API. When looking quickly it seems like the two fields are quite similar. When we look closer, we realize that the DiscountAutomatic return type is a Union type of possible discount types.

Looking a the new field, it returns an object type with extra information:

```
type DiscountAutomaticNode implements Node & HasEvents {
  automaticDiscount: DiscountAutomatic!
  events: EventConnection!
  id: ID!
}
```

A few things to note here. First, types that we can fetch through global IDs should probably always implement the Node interface. But that’s not all, union types are harder to evolve since we can’t expand them to contain shared information, in this case, that events field. It may be a good idea to consider wrapping unions types within an object type in case metadata needs to be added to the union itself.

Let's look at another one coming from Shopify's API, this time on the mutation side of things.

```
type Mutation {
  collectionPublish( … ): CollectionPublishPayload @deprecated(reason:
"Use publishablePublish instead")

  publishablePublish( … ): PublishablePublishPayload
}
```

It's generally good practice to offer mutations that can operate on our abstract types like interfaces. Without knowing more, I wonder if `collectionPublish` still could have remained in the schema while favoring `publishablePublish` going forward.

When dealing with interfaces, ask yourself if you should offer mutations that implement a certain contract for an interface rather than one for each concrete type.

One thing that makes those mutations a bit hard to use for clients is that it's hard to see on which concrete type it can apply. For example, we can only assume this works on all types implementing Publishable, but it's not absolutely clear from the schema.

One possible way to address this is by using a custom schema directive:

```
input PublishablePublishInput {
  id: ID! @possibleTypes(types: ["Product", "Collection"])
  publishable: PublishableInput!
}
```