Designing GraphQL Schemas

Scott Walkinshaw
Lessons learned from creating and evolving production schemas at Shopify.
Rules
Rules
Guidelines
We believe these design guidelines work in most cases. They may not all work for you.
Create collection

Title

e.g. Summer collection, Under $100, Staff picks

Description

Collection type

- Manual
  Add your chosen products to this collection one by one. Learn more about manual collections.

- Automated
  Existing and future products that match the conditions you set will automatically be added to this collection. Learn more about automated collections.

CONDITIONS

Products must match: ○ all conditions ○ any condition

Product tag is equal to

Add another condition
Create collection

Title
- e.g. Summer collection, Under $100, Staff picks

Description

Collection type
- Manual
  Add your chosen products to this collection one by one. Learn more about manual collections.
- Automated
  Existing and future products that match the conditions you set will automatically be added to this collection. Learn more about automated collections.

CONDITIONS
- Products must match:
  - all conditions
  - any condition

Product tag is equal to

Add another condition
Create collection

Title

Example: Summer collection, Under $100, Staff picks

Description

Collection type

- Manual
  Add your chosen products to this collection one by one. Learn more about manual collections.

- Automated
  Existing and future products that match the conditions you set will automatically be added to this collection. Learn more about automated collections.

Conditions

Products must match: all conditions or any condition

- Product tag is equal to

Add another condition
Create collection

Title

e.g. Summer collection, Under $100, Staff picks

Description

Collection type

- Manual
  Add your chosen products to this collection one by one. Learn more about manual collections.

- Automated
  Existing and future products that match the conditions you set will automatically be added to this collection. Learn more about automated collections.

CONDITIONS

Products must match: ○ all conditions ○ any condition

Product tag is equal to

Add another condition
Create collection

Title

- e.g. Summer collection, Under $100, Staff picks

Description

Collection type

- Manual
  Add your chosen products to this collection one by one. Learn more about manual collections.

- Automated
  Existing and future products that match the conditions you set will automatically be added to this collection. Learn more about automated collections.

CONDITIONS

- Products must match:
  - all conditions
  - any condition

Product tag is equal to

Add another condition
interface Collection {
  title: String
}

type AutomaticCollection implements Collection {
  title: String
  rules: [AutomaticCollectionRule]
}

type ManualCollection implements Collection {
  title: String
}
interface Collection {
    title: String
}

type AutomaticCollection implements Collection {
    title: String
    rules: [AutomaticCollectionRule]
}

type ManualCollection implements Collection {
    title: String
}
interface Collection {
    title: String
}

type AutomaticCollection implements Collection {
    title: String
    rules: [AutomaticCollectionRule]
}

type ManualCollection implements Collection {
    title: String
}

type CollectionRule {...}
Guideline #1

Never expose implementation details in your API design.
type Collection {
  title: String
  rules: [CollectionRule]
  memberships: [CollectionMembership]
}

type Product {
  title: String
  memberships: [CollectionMembership]
}

type CollectionMembership {
  collectionId: ID!
  productId: ID!
}
type Collection {
    title: String
    rules: [CollectionRule]
    memberships: [CollectionMembership]
}

type Product {
    title: String
    memberships: [CollectionMembership]
}

type CollectionMembership {
    collectionId: ID!
    productId: ID!
}
type Collection {
  title: String
  rules: [CollectionRule]
  memberships: [CollectionMembership]
}

type Product {
  title: String
  memberships: [CollectionMembership]
}

type CollectionMembership {
  collectionId: ID!
  productId: ID!
}
type Collection {
  title: String
  rules: [CollectionRule]
  memberships: [CollectionMembership]
}

type Product {
  title: String
  memberships: [CollectionMembership]
}

type CollectionMembership {
  collectionId: ID!
  productId: ID!
}

type Collection {
  title: String
  rules: [CollectionRule]
  products: [Product]
}

type Product {
  title: String
  collections: [Collection]
}
type Collection {
    title: String
    rules: [CollectionRule]
    products: [Product]
}

type Product {
    title: String
    collections: [Collection]
}
Guideline #2

It's easier to add schema elements than to remove them.
type Collection {
  bodyHtml: String
  id: ID!
  imageId: ID
  products: [Product]
  rules: [CollectionRule]
  rulesApplyDisjunctively: Boolean
  title: String
}
type Collection {
    bodyHtml: String
    id: ID!
    imageId: ID
    products: [Product]
    rules: [CollectionRule]
    rulesApplyDisjunctively: Boolean
    title: String
}
type Collection {
  bodyHtml: String
  id: ID!
  imageId: ID
  products: [Product]
  rules: [CollectionRule]
  rulesApplyDisjunctively: Boolean
  title: String
}

type CollectionRuleSet {
  rules: [CollectionRule]
  applyDisjunctively: Boolean!
}

Guideline #3

Group closely-related fields together into their own type.
List-type fields should almost always non-null lists with non-null elements.
List-type fields should almost always non-null lists with non-null elements.
Like lists, boolean fields should almost always non-null.
Like lists, boolean fields should almost always non-null.
type Collection {
  bodyHtml: String
  id: ID!
  imageId: ID
  products: [Product]
  ruleSet: CollectionRuleSet
  title: String
}

type CollectionRuleSet {
  rules: [CollectionRule]
  applyDisjunctively: Boolean!
}

type CollectionRule {
  rules: [CollectionRule!]
  applyDisjunctively: Boolean!
}
type Collection {
  bodyHtml: String
  id: ID!
  imageId: ID
  products: [Product!]!
  ruleSet: CollectionRuleSet
  title: String
}
type Collection {
  bodyHtml: String
  id: ID!
  imageId: ID
  products: [Product!]!
  ruleSet: CollectionRuleSet
  title: String
}

type ProductConnection {
  after: String
  before: String
  first: Int
  last: Int
  pageInfo: PageInfo!
}

type PageInfo {
  hasNextPage: Boolean!
  hasPreviousPage: Boolean!
}

type ProductEdge {
  cursor: String!
  node: Product!
}
Guideline #4

Always check whether list fields should be paginated or not.
type Collection {
  bodyHtml: String
  id: ID!
  imageUrl: ID
  products(
    after: String
    before: String
    first: Int
    last: Int
  ): ProductConnection!
  ruleSet: CollectionRuleSet
  title: String
}
type Collection {
    bodyHtml: String
    id: ID!
    imageId: ID
    products(
        after: String
        before: String
        first: Int
        last: Int
    ): ProductConnection!
    ruleSet: CollectionRuleSet
    title: String
}

type Image {
    id: ID!
}

Guideline #5

Use object references instead of ID fields.
type Collection {
  bodyHtml: String
  id: ID!
  image: Image
  products(
    after: String
    before: String
    first: Int
    last: Int
  ): ProductConnection!
  ruleSet: CollectionRuleSet
  title: String
}
type Collection {
    bodyHtml: String
    id: ID!
    image: Image
    products(
        after: String
        before: String
        first: Int
        last: Int
    ): ProductConnection!
    ruleSet: CollectionRuleSet
    title: String
}

type Collection {
    description: String
    id: ID!
    image: Image
    products(
        after: String
        before: String
        first: Int
        last: Int
    ): ProductConnection!
    ruleSet: CollectionRuleSet
    title: String
}
Guideline #6

Choose field names based on what makes sense, not based on the implementation or what the field is called in legacy APIs.
type Collection {
  description: String
  id: ID!
  image: Image
  products(
    after: String
    before: String
    first: Int
    last: Int
  ): ProductConnection!
  ruleSet: CollectionRuleSet
  title: String
}

scalar HTML
Guideline #7

Use custom scalar types when you're exposing something with specific semantic value.
```typescript
// CollectionRuleSet

type CollectionRuleSet {
  rules: [CollectionRule!]!
  applyDisjunctively: Boolean!
}

// CollectionRule

type CollectionRule {
  field: String!
  relation: String!
  value: String!
}
```
type CollectionRuleSet {
  rules: [CollectionRule!]!
  applyDisjunctively: Boolean!
}

type CollectionRule {
  field: String! product_title
  relation: String! contains
  value: String! shirt
}
Collection type

- Manual
  Add your chosen products to this collection one by one. Learn more about manual collections.

- Automated
  Existing and future products that match the conditions you set will automatically be added to this collection. Learn more about automated collections.

CONDITIONS

Products must match:  
- all conditions
- any condition

- Product tag
  is equal to

- Add another condition
type CollectionRuleSet {
    rules: [CollectionRule]!
    applyDisjunctively: Boolean!
}

type CollectionRule {
    field: String!
    relation: CollectionRuleRelation!
    value: String!
}

type CollectionRuleField {
    PRICE
    TAG
    TITLE
    TYPE
    VENDOR
}

type CollectionRuleRelation {
    CONTAINS
    EQUALS
    ENDS_WITH
    GREATER_THAN
    LESS_THAN
    STARTS_WITH
}
Guideline #8

Use enums for fields which can only take a specific set of values.
Business Logic
const productsQuery = gql`
  collection(id: 1) {
    products(first: 10) {
      edges {
        node {
          id
        }
      }
    }
  }
`;

products = await client.query(productsQuery)
products.find(product => product.id == product.id)
type Collection {
  # ...
  hasProduct(id: ID!): Boolean!
}

Guideline #9

The API should provide business logic, not just data. Complex calculations should be done on the server, in one place, not on the client, in many places.
Guideline #10

Provide the raw data too, even when there's business logic around it.
Mutations
type MutationRoot {
    collectionUpdate(
        id: ID!
        input: CollectionInput!
    ): Collection
}
Logical Actions

- collectionCreate
- collectionDelete
- collectionUpdate
- collectionAddProducts
- collectionRemoveProducts
- collectionReorderProducts
Guideline #11

Write separate mutations for separate logical actions on a resource.
type MutationRoot {
  collectionUpdate(
    id: ID!
    input: CollectionInput!
  ): Collection
}

type MutationRoot {
  collectionUpdate(
    id: ID!
    input: CollectionInput!
  ): CollectionUpdatePayload!
}

type CollectionUpdatePayload {
  collection: Collection
}

Guideline #12

Use a payload return type for each mutation.
type MutationRoot {
  collectionUpdate(
    id: ID!
    input: CollectionInput!
  ): CollectionUpdatePayload!
}

type CollectionUpdatePayload {
  collection: Collection
  userErrors: [UserError!]
}

type UserError {
  field: [String!]
  message: String!
}
Guideline #13

Mutations should provide user/business-level errors via a `userErrors` field on the mutation payload.

The top-level query errors entry is reserved for client and server-level errors.
Guideline #14

Most payload fields for a mutation should be nullable.
Thanks!

@swalkinshaw