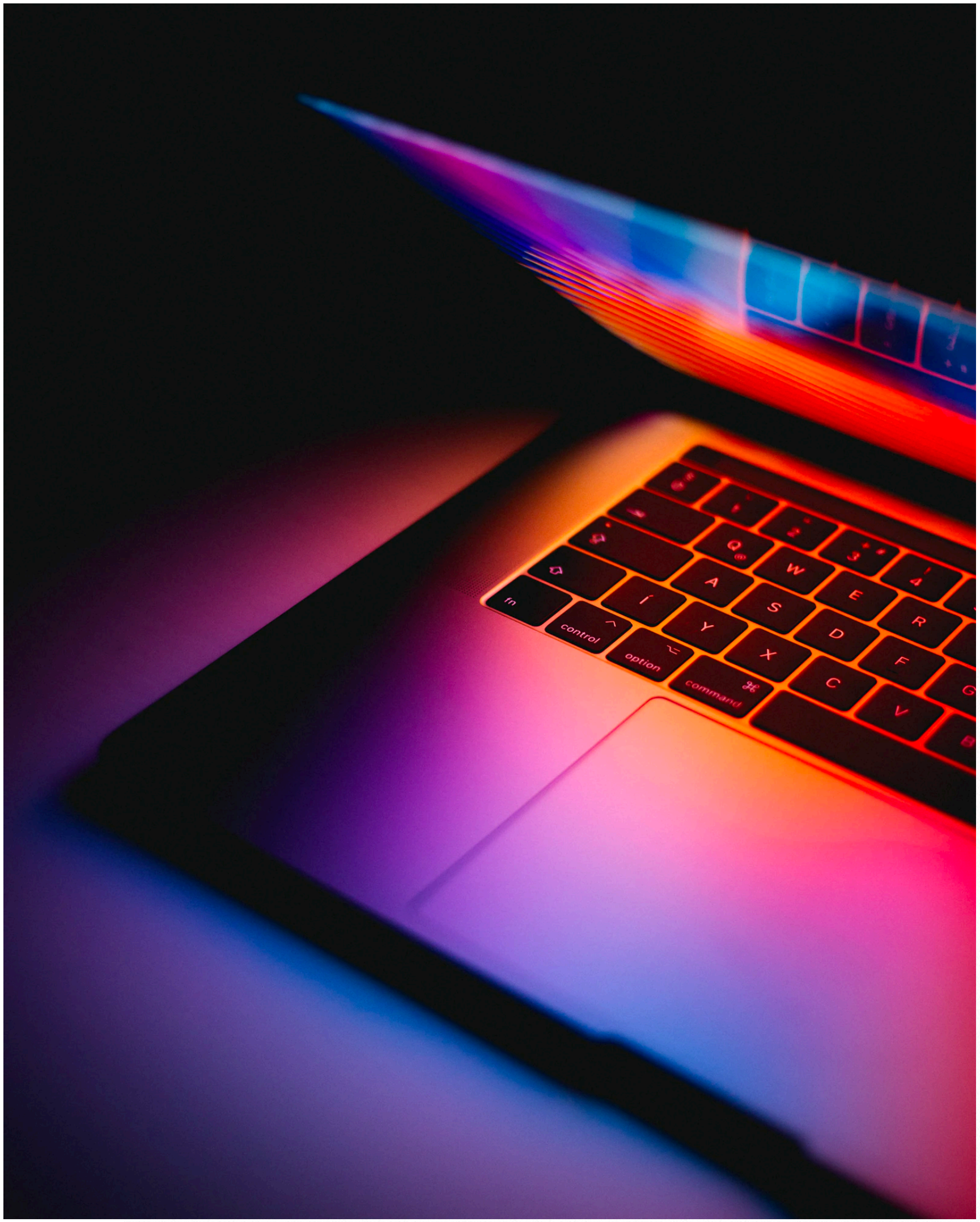
PRODUCTION READY GRAPHQL

GraphQL Schema Design Cheat Sheet



Introduction

Thank you for purchasing Production Ready GraphQL.

This is a useful schema cheat sheet you can refer to when thinking about a new schema design, and even implement it as a linter to keep your schema consistent. It is highly opinionated based on what I've seen work best over the years. Feel free to modify or add to these sheets with your own rules.

I hope this helps you design great APIs that clients will love using!

- Marc-Andre

1. General

General rules to keep in mind when adding new functionalities to a GraphQL API.

1.1 YAGNI

Keep your API minimal. Expose new fields and use cases only when needed.

1.2 Completeness

Minimal yet complete. Can clients achieve their use cases completely?

1.3 Domain Knowledge

Do you deeply understand this use case? Are there domain experts that could help you design this in a better way?

1.4 No Implementation Details

Is my API decoupled from internal, implementation details? (ex: generated from a database schema, coupled to another API, uses internal naming, etc)

1.5 Client First

Does my API answer a clear client use case? Avoid building APIs without a need first: Have a first client or use dogfooding.

2. Naming

Rules to keep in mind when naming schema members.

2.1 Specific Names

Is the name specific enough? What are the chances this could conflict with another concept down the line?

```
# Bad
type User {
  login: String!
}

# Good
type InstagramUser implements Actor {
  login: String
  followerCount: Int!
}
```

2.2 Clear Intent

Does the need name reveal an intent to users? Is it easy to understand what it represents/does?

```
# Bad: Is this the definition of an alert, or an active alert?
type Alert {
  # ...
}

# Good
type AlertDefinition {
  conditions: [Condition!]!
}

type AlertNotification {
  triggeredAt: DateTime!
}
```

2.3 Consistent Language

Is the naming consistent with other fields/types in the schema? Avoid synonyms, and be consistent in entity names.

```
type Mutation {
  deleteUser(input: DeleteUserInput!): DeleteUserPayload

  # Bad
  destroyItem(input: DestroyItemInput!): DestroyItemPayload

  # Good
  deleteItem(input: DeleteItemInput!): DeleteItemPayload
}
```

2.4 Use “camelCase” for fields and inputs

```
type User {
  # Bad
  archived_posts: [Post!]!

  # Good
  archivedPosts: [Post!]!
}
```

2.5 Use “PascalCase” for types

```
# Bad
type lineItem {
  price: Int!
}

# Good
type LineItem {
  price: Int!
}
```

2.6 Use “SCREAMING_SNAKE_CASE” for enum values

```
# Bad
enum OrderState {
  paid
  pending
  inProgress
}

# Good
enum OrderState {
  PAID
  PENDING
  IN_PROGRESS
}
```

2.7 Avoid Abbreviations

```
type User {
  # Bad
  mgr: User!

  # Good
  manager: User!
}
```

3. Fields and Mutations

Rules to keep in mind when designing fields and mutations.

3.1 Use good default values

Use default values to make an API more predictable and communicate intent

```
type Query {
  # Bad
  products(first: Int!, orderBy: ProductsOrdering)

  # Good
  products(first: Int!, orderBy: ProductsOrdering = CREATED_AT)
}
```

3.2 Avoid implicit and clever side effects

Avoid surprise side effects in fields and mutations. The name should convey what it does at runtime.

3.3 Prefer `verbEntity` to `entityVerb`

Prefer verb prefixes to entity prefixes. Use explicit namespaces, or even better organization directives to achieve the same results.

```
type Mutation {
  # Bad
  productCreate(input: ProductCreateInput!):
    ProductCreatePayload

  # Good
  createProduct(input: CreateProductInput!):
    CreateProductPayload

  # Good
  createProduct(input: CreateProductInput!):
    CreateProductPayload @tags(names: ["Product"])
}
```

3.4 Do one thing well

Prefer specific solutions rather than overly clever, generic fields

```
type Query {
  # Bad
  posts(first: Int!, isDeleted: Boolean)

  # OK
  posts(first: Int!, isDeleted: Boolean = False)

  # Better
  posts: [Post!]!
  deletedPosts: [Post!]!
}
```

3.5 Think beyond CRUD

Use action or behavior specific fields and mutations to help clients use your API

```
# Bad
mutation ArchivePost {
  updatePost(input: { ... , archived: true }) {
     ...
  }
}

# Good
mutation ArchivePost {
  archivePost(input: { postID: “abc” }) {
     ...
  }
}
```

3.6 Design beyond data

Add fields that represent domain logic beyond simple data fields

```
# Bad
query ShouldShowPostonProfile {
  post(id: “1”) {
    # Client is guessing that if the post is not `archived`
    # and is `featured`, it should show this post on a user profile.
    archived
    featured
  }
}

# Better
query ShouldShowPostonProfile {
  post(id: “1”) {
    # Specific field for what clients were looking for.
    # Avoids brittle client side logic.
    shouldFeatureOnProfile
```

3.7 Use Input & Payload types for mutations

Use specific object types for mutation inputs and outputs

```
type Mutation {
  # Bad
  createProduct(name: String!):
    Product

  # Good
  createProduct(input: CreateProductInput!):
    CreateProductPayload
}

input CreateProductInput {
  name: String!
}

type CreateProductPayload {
  product: Product!
}
```

3.8 Prefer a single input argument on mutations

A single input argument is a Relay convention and allows clients to use a single variable.

```
mutation UpdatePost {
  # Bad
  updatePost(title: “Schema Design”) {
    ...
  }

  # Good
  updatePost(input: { title: “Schema Design” }) {
    ...
  }
}
```

3.9 Don’t hide input coupling

Group related and coupled input fields under input types

```
# Bad
input CreateProductInput {
  name: String!
  priceCents: Int
  priceCurrency: Int
}

# Good
input CreateProductInput {
  name: String!
  price: ProductPriceInput
}

input ProductPriceInput {
  priceCents: Int!
  priceCurrency: Int!
}
```

3.10 Don’t reuse input and payload types

Avoid reusing input and payload types across multiple mutations

```
# Bad
input ProductInput {
  # Only required when used in update mutations
  id: ID
  name: String
}

# Good
input CreateProductInput { ... }
input UpdateProductInput { ... }
```

3.11 Group related fields under object types

Common prefixes can often point towards the need for a new object type

```
# Bad
type Customer {
  cardLastFour: Int!
  cardExpiryMonth: Int!
  cardExpiryYear: Int!
  cardName: String!
}

# Good
type CreditCard {
  lastFour: Int!
  expiry: CreditCardExpiry!
  name: String!
}

type CreditCardExpiry {
  month: Int!
  year: Int!
}
```

3.12 Use Non-Null With Care

Use Non-Null mostly on scalar fields and use them with care on associations.

```
type User {
  # Prefer nullable associations, especially if backed by external
  # services.
  profilePicture: ProfilePicture

  # Scalars are usually safer
  name: String!
```

3.13 Prefer required arguments

Prefer a strong schema rather than runtime checks. Use more specific fields if needed.

```
type Query {
  # Bad
  userByNameOrID(id: ID, name: String): User

  # Good
  userByName(name: String!): User
  userByID(id: ID!): User
}
```

3.14 Use custom scalars to convey semantic

Custom scalars may help clients handle scalars in a better way.

```
type Query {
  # OK
  description: String

  # Better
  description: Markdown
}
```

3.15 Use custom scalars in inputs for stricter validation

```
input CreateUser {
  email: EmailAddress
}

# A valid email address according to RFC5322
scalar EmailAddress
```

3.16 Offer plural versions fields

Plural fields allow clients to use variables and static queries to query your API.

```
type Query {
  # Bad: Only offering a singular version
  userByName(name: String!): User

  # Good: Plural + singular if needed
  usersByName(names: [String]!): UserConnection
  userByName(name: String!): User
}
```

3.17 Offer “batch” mutations

Design transactions as batch mutations rather than letting clients use multiple mutation fields.

```
mutation {
  # Bad: Only offering highly specific mutation
  # increases complexity on client
  addProducts(productIDs: [ ... ]) {  ...  }
  removeProducts(productIds: [ ... ]) {  ...  }

  # Good: Coarse grained version to answer client use case.
  # Specific versions as well if needed
  updateCart(input: { productToAdd: [ ... ], productsToRemove: [ ... ] }) {  ... }
}
```

4. Enums

Rules to keep in mind related to Enums and Enum values.

4.1 Prefer enums over strings

Use enums rather than strings when defining a fixed set of values

```
# Bad
type Order {
  status: String!
}

# Good
enum OrderStatus {
  PENDING
  PAID
}

type Order {
  status: OrderStatus!
}
```

4.2 Consider a default value instead of null

Use enums rather than strings when defining a fixed set of values

```
enum OrderStatus {
  PENDING
  PAID
  # NO_STATUS means we could not compute a status for this order.
  NO_STATUS
}
```

4.3 Avoid redundant suffixes

Avoid `Type`, `Kind` style suffixes for enum type names

```
# Bad
enum OrderStatusType {
   ...
}
enum OrderStatusEnum {
   ...
}

# Good
enum OrderStatus {
  PENDING
  PAID
}
```

5. Abstract Types

Rules to keep in mind related to interface and union types

5.1 Use interfaces when types share a common contract

Use unions rather than interfaces when types share no common contract

```
interface Closeable {
  isClosed: Boolean!
}

type Issue implements Closeable {
  isClosed: Boolean!
}


type PullRequest implements Closeable {
  isClosed: Boolean!
}
```

5.2 Use Unions when there is no common contract

Use unions rather than interfaces when types share no common contract

```
type Query {
  search(query: String!): [SearchResult]
}

union SearchResult = Image | User | Article
```

5.3 Respect the “Liskov Substitution Principle” when adding interface implementations

It also makes adding interface implementations less of a dangerous change.

5.4 Use “interface mutations”

Mutations may be part of an interface contract. Consider exposing mutations that can apply to any member of an interface.

```
type Mutation {
  closeCloseable(input: CloseCloseableInput!):
    CloseCloseablePayload
}
```

5.5 Avoid interfaces only to share data, share behaviors instead

Interfaces used when types don’t share behavior are unlikely to evolve well.

```
“””
Not recommended: Use composition / inheritance within definition code
instead of using interfaces in the schema
“””
interface HasTitleAndDescription {
  title: String!
  description: String!
}
```

6. Errors

Rules to keep in mind related to errors

6.1 Use GraphQL Errors for developer-facing errors.

Use enums rather than strings when defining a fixed set of values

```
{
  "errors": [
    {
      "message": "Service Unavailable",
      "locations": [
        {
          "line": 2,
          "column": 10,
        }
      ],
      "path": ["user", "login"],
    }
  ]
}
```

6.2 Use error extensions to encode additional information

Use enums rather than strings when defining a fixed set of values

```
{
  "errors" ⇒ [
    {
      "message": "Service Unavailable",
      "extensions": {
        "code": “SERVICE_UNAVAILABLE”
      },
      "locations": [
        {
          "line": 2,
          "column": 10,
        }
      ],
      "path": ["user", "login"],
    }
  ]
```

6.3 Use “errors in schema” for user-facing errors

Use the schema instead of top-level errors for errors that are meant to be displayed or used within the application.

```
type CreateProductPayload {
  product: Product

  # userError on mutation payloads
  userErrors: [UserError]!
}

# Or use union result types
union CreateProductResult = ProductPayload | ProductAlreadyExist
```

6.4 Error types should implement a common interface

This lets clients handle new error types more gracefully.

```
type CreateProductPayload {
  product: Product
  userErrors: [UserError]!
}

interface UserError {
  # Path to the input field in error
  field: [String!]
  message: String!
}
```

6.5 Add useful fields to concrete error types

Concrete types for errors can include data for clients to handle specific scenarios.

```
interface UserError {
  field: [String!]
  message: String!
}

type ProductTakenError {
  field: [String!]
  message: String!
  existingProduct: Product!
}
```

7. Lists & Pagination

Best practices for list types and pagination

7.1 Always consider paginating lists

Avoid plain List types unless 100% confident that it won't grow large.

7.2 Implement the connection specification

Unless your use case requires jumping to specific pages, prefer the connection specification

```
type Query {
  # Good https://facebook.github.io/relay/graphql/connections.htm
  products(first: Int!, after: Int!): ProductsConnection

  # Bad
  products(page: Int!, offset: Int!): [Product]
}
```

7.3 Use edge types to encode data on the relationship itself

Use edge types to add fields about a relationship and avoid polluting types with context-dependent fields.

```
type TeamMemberEdge {
  isTeamAdmin: Boolean!
  node: User!
}
```

7.4 Use `Connection` and `Edge` suffixes

For convention, suffix connections and edges with their according suffixes

```
type TeamMembersConnection { ... }
type TeamMemberEdge { ... }
```

7.5 Create new connection types for different relationships

Don’t use common connection types even if the nodes are of the same type.

```
# Bad: A generic users connection reused everywhere
type UsersConnection {
  edges: [UserEdge]!
}

# Good: Relation specific connections
type TeamMemberConnection {
  edges: [TeamMemberEdge]!
}

type FriendsConection {
  edges: [FriendshipEdge]!
}
```

7.6 Don’t reuse edge types between connections

```
# Bad: Reusing edge types in different connections
type UsersConnection {
  edges: [UserEdge]!
}

type TeamMembersConnection {
  edges: [UserEdge]!
}

# Good
type UsersConnection {
  edges: [UserEdge]!
}

type TeamMembersConnection {
  edges: [TeamMemberEdge]!
}
```

7.7 Use `total_count` fields with care

`total_count` fields can often become performance problems depending on how resolvers are implemented.

7.8 Offer a `nodes` shortcut

Avoid the verbosity of `edges` by allowing clients to query the nodes directly.

```
query {
  posts(first: 10) {
    nodes {
      title
    }
  }
}
```

8. Identification

Best practices for list types and pagination

8.1 Use Global IDs

Use global and unique IDs, and use the ID scalar

```
type Query {
  # Good
  product(id: ID!): Product

  # Bad
  product(type: String!, databaseId: Int!): Product
}
```

8.2 Opaque IDs

Use opaque IDs to discourage “hacking ids”.

```
{
  “goodId”: “cHJvZHVjdDox”,
  “badId”: “product:1”
}
```

8.3 Support `node` and `nodes` fields

Support both `node` and `nodes` fields for Relay support and refetching of individual nodes

```
type Query {
  node(id: ID!): Node
  nodes(ids: [ID]!): [Node]
}
```

8.4 Implement `Node`, but not on all types

Not all object types have to implement the `Node` interface, for example, types that don't exist without a parent object.

```
# Product may be fetched globally
type Product implements Node {
   ...
}

# ProductPricingDetails does not make sense without its parent product object
type ProductPricingDetails {
   ...
}
```

8.5 Type-specific fetchers

Feel free to add specific node fetch fields rather than only `node` and `nodes` fields.

```
type Query {
   # Good
   product(id: ID!): Product
}
```

8.6 Encode enough information in IDs for global fetching

Prefer encoding more than not enough for evolution. This might mean including the parent object id, the shard key, etc.