Migrating from REST to GraphQL

THIS DOT LABS

Hi! I’m Dustin!

- Dustin Goodman
- Engineer Manager
- This Dot Labs
- @dustinsgoodman
- Web Consultant specializing in web technologies

THIS DOT

We love our clients!

Meta Google node JS twilio

T Mobile ROBLOX Cloudinary Deloitte.

WIKIMEDIA FOUNDATION PlayStation Microsoft bill.com

AGENDA

- Why migrate?
- Designing your Schema
- Strategies & Considerations for Migration
- Migration Demo

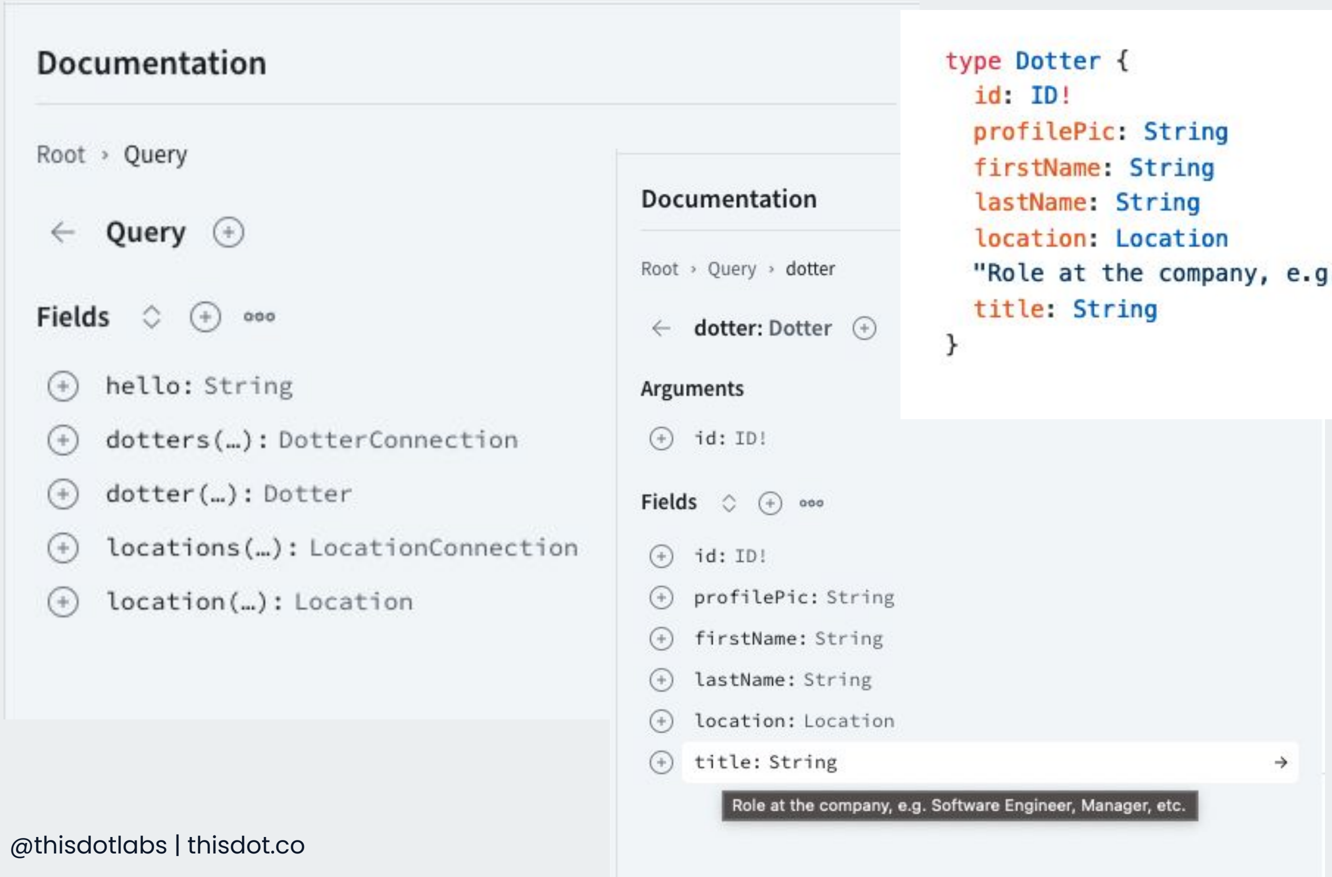
Why migrate?

GraphQL is Declarative

- Write API as types and operations
- Use the scalar types to define complex types
- Can create custom scalar types
- Can use enums out of the box
- Data is *resolved* using functions

```
const gql = require('graphql-tag');

const CharacterTypeDef = gql`
  type Character {
    id: ID!
    avatar: URL
    currentLocation: Location
    gender: Gender
    name: String!
    origin: Location
    species: String!
    status: CharacterStatus
    type: String!
  }

  type CharacterConnection {
    nodes: [Character]
    pageInfo: PageInfo
  }

  enum CharacterStatus {
    ALIVE
    DEAD
    UNKNOWN
  }

  enum Gender {
    GENDERLESS
    FEMALE
    MALE
  }

  type Query {
    characters(pagination: PaginationInput): CharacterConnection
    character(id: ID!): Character
  }
`;

module.exports = CharacterTypeDef;
```

GraphQL is Self-Documenting



Documentation

Root › Query

← Query

Fields

```
hello: String
dotters(…): DotterConnection
dotter(…): Dotter
locations(…): LocationConnection
location(…): Location
```

Documentation

Root › Query › dotter

← dotter: Dotter

Arguments

```
id: ID!
```

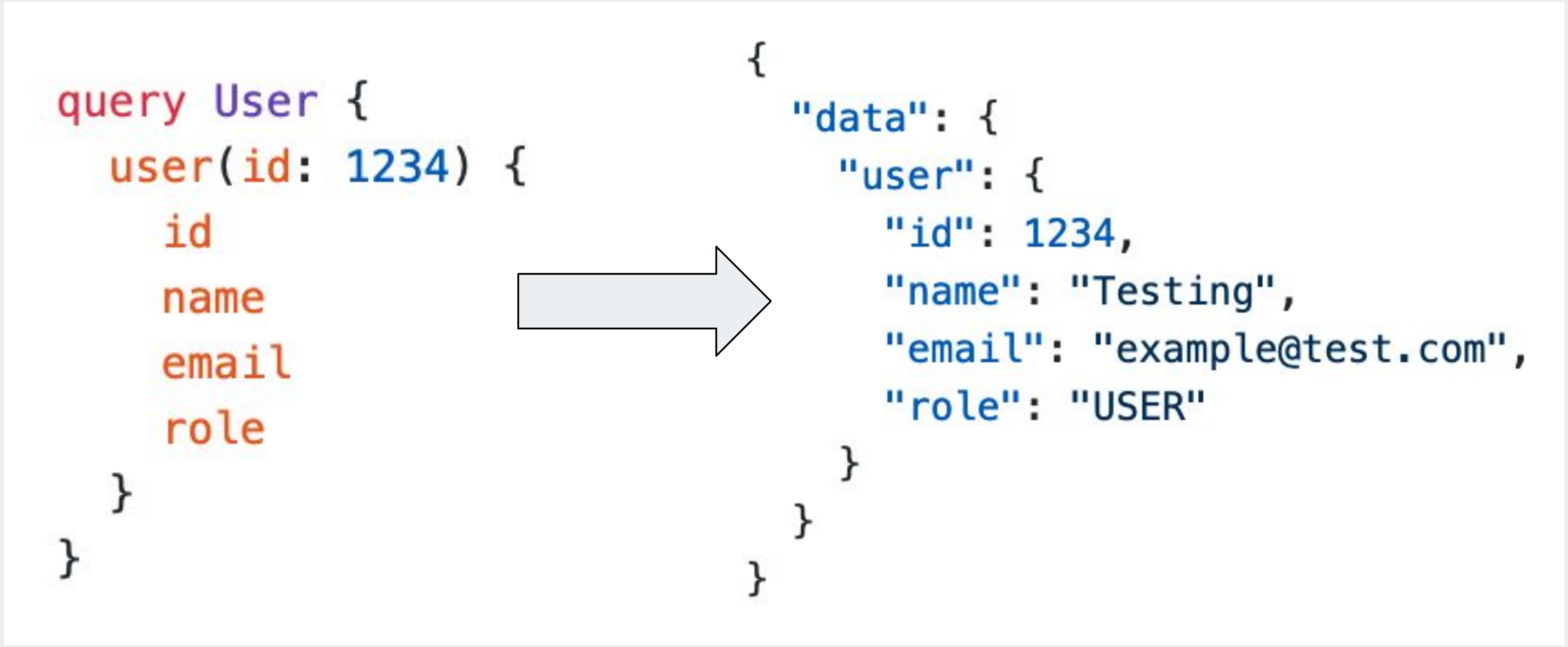
Fields

```
id: ID!
profilePic: String
firstName: String
lastName: String
location: Location
title: String
```

Role at the company, e.g. Software Engineer, Manager, etc.

```graphql
type Dotter {
  id: ID!
  profilePic: String
  firstName: String
  lastName: String
  location: Location
  "Role at the company, e.g. Software Engineer, Manager, etc."
  title: String
}
```

GraphQL is easy to consume

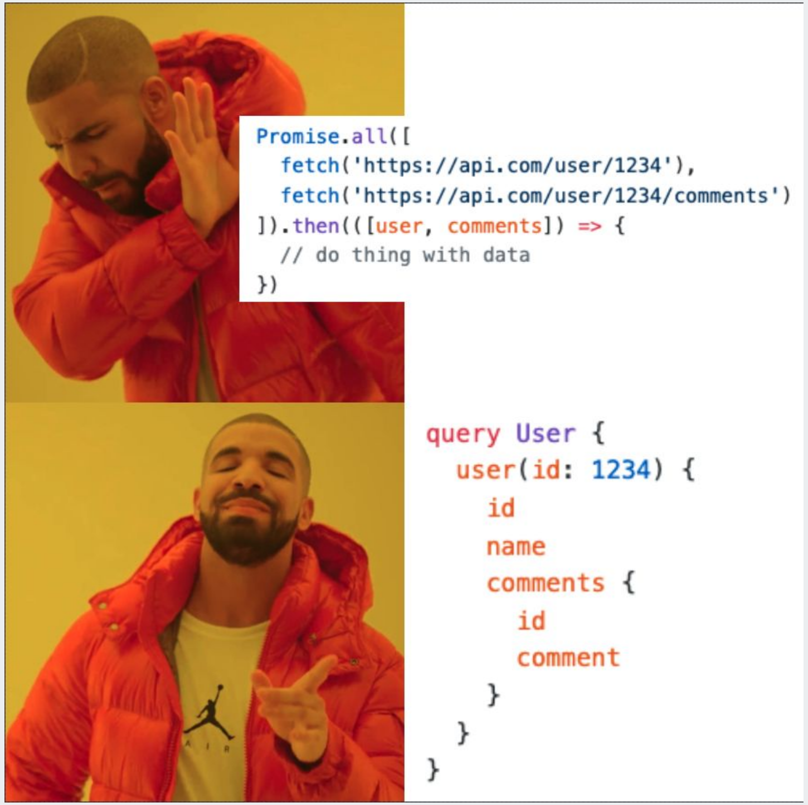


```
query User {
  user(id: 1234) {
    id
    name
    email
    role
  }
}
```

```
{
  "data": {
    "user": {
      "id": 1234,
      "name": "Testing",
      "email": "example@test.com",
      "role": "USER"
    }
  }
}
```

GraphQL v REST: Under-Fetching

- No more under-fetch
- Get exactly what you need the first time
- No Promise.all

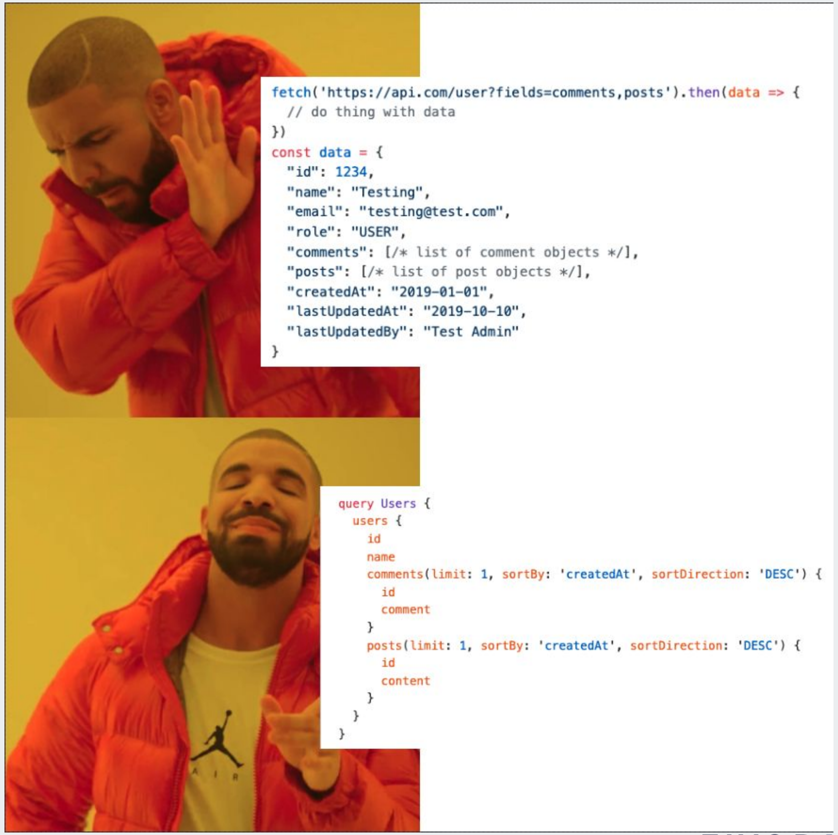


```
Promise.all([
  fetch('https://api.com/user/1234'),
  fetch('https://api.com/user/1234/comments')
]).then(([user, comments]) => {
  // do thing with data
})
```

```
query User {
  user(id: 1234) {
    id
    name
    comments {
      id
      comment
    }
  }
}
```

GraphQL v REST: Over-Fetching

- No more over-fetch
- Specify only the data you want
- Leave the rest of the data behind on the serve
- Get rid of those poorly documented query params



```
fetch('https://api.com/user?fields=comments,posts').then(data => {
  // do thing with data
})
const data = {
  "id": 1234,
  "name": "Testing",
  "email": "testing@test.com",
  "role": "USER",
  "comments": [/* list of comment objects */],
  "posts": [/* list of post objects */],
  "createdAt": "2019-01-01",
  "lastUpdatedAt": "2019-10-10",
  "lastUpdatedBy": "Test Admin"
}
```

```
query Users {
  users {
    id
    name
    comments(limit: 1, sortBy: 'createdAt', sortDirection: 'DESC') {
      id
      comment
    }
    posts(limit: 1, sortBy: 'createdAt', sortDirection: 'DESC') {
      id
      content
    }
  }
}
```

GraphQL v REST: Documented Fields

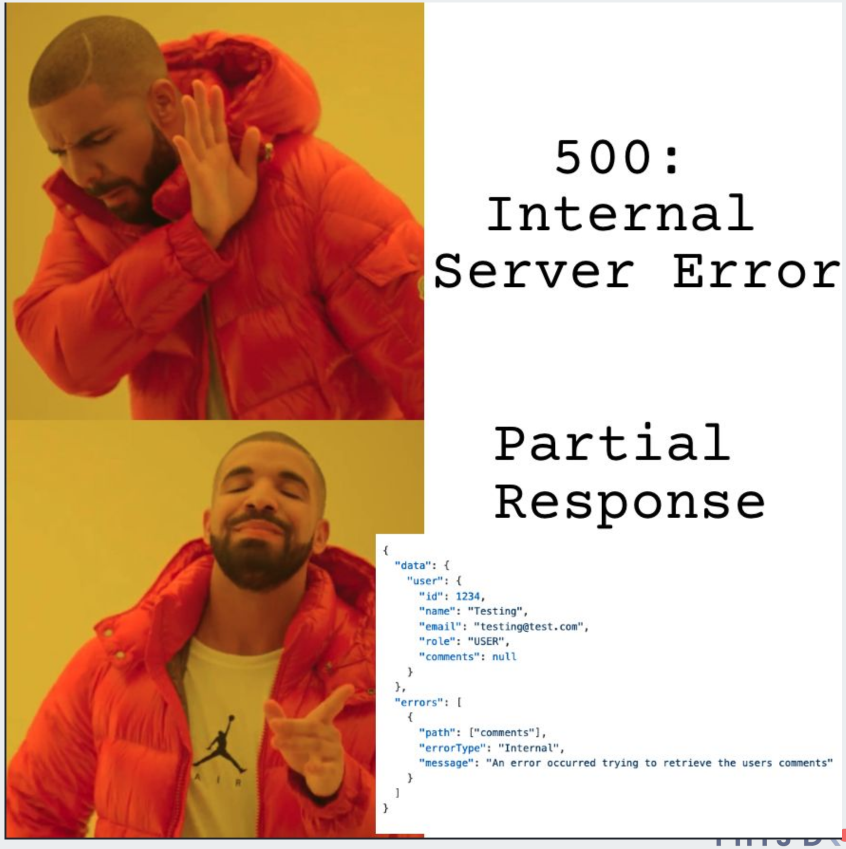
- With REST, you can use Swagger, Apiary, etc. to document your API
- With GraphQL, it's the **default**!
- Mutating data has explicit input types with server validation to tell you what works

data payload shape?
typed inputs

```
input CreateDotterInput {
  firstName: String!
  lastName: String!
  title: String!
  profilePic: String!
  location: CreateLocationInput!
}
```

GraphQL v REST: Error Handling

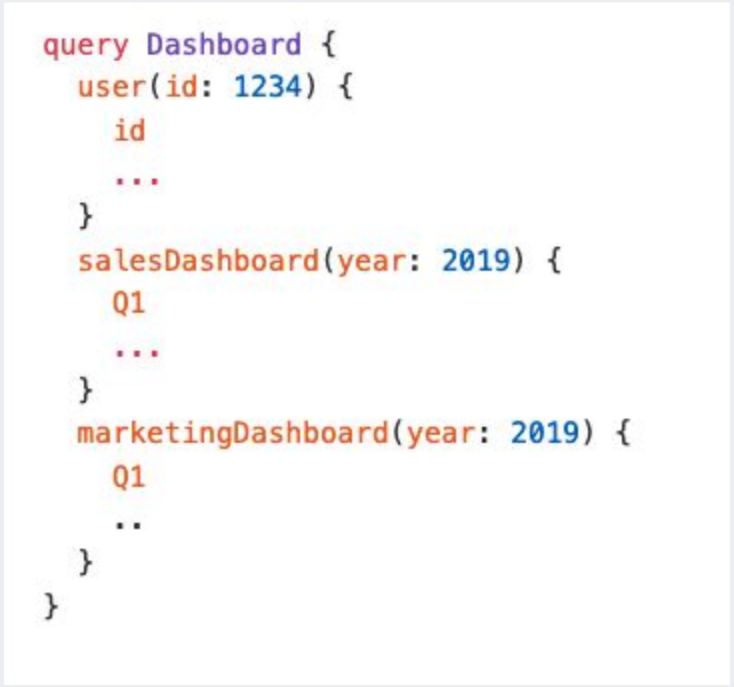
- REST is an all-or-nothing experience with responses
- GraphQL gives you everything it can AND the errors that arose



```
{
  "data": {
    "user": {
      "id": 1234,
      "name": "Testing",
      "email": "testing@test.com",
      "role": "USER",
      "comments": null
    }
  },
  "errors": [
    {
      "path": ["comments"],
      "errorType": "Internal",
      "message": "An error occurred trying to retrieve the users comments"
    }
  ]
}
```

GraphQL v REST: Request Batching

- Batch support out-of-the-box
- Fetch from multiple disparate sources with a single operation
- Restriction: 1 operation type



```
query Dashboard {
  user(id: 1234) {
    id
    ...
  }
  salesDashboard(year: 2019) {
    Q1
    ...
  }
  marketingDashboard(year: 2019) {
    Q1
    ..
  }
}
```

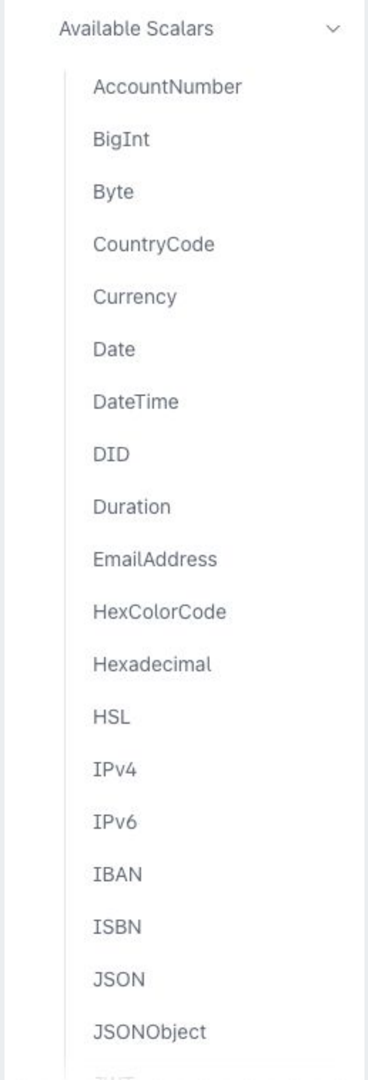
Designing your Schema

Selecting Schema Rules

- Multiple specifications for schema design exist
- Choose rules or a spec that your team can use consistently
- Example Rules
    - Pagination Style - page or cursor based?
    - Allow or Disallow Foreign Keys
    - Field definition rules: order, convention, custom scalars

Custom Scalars Considerations

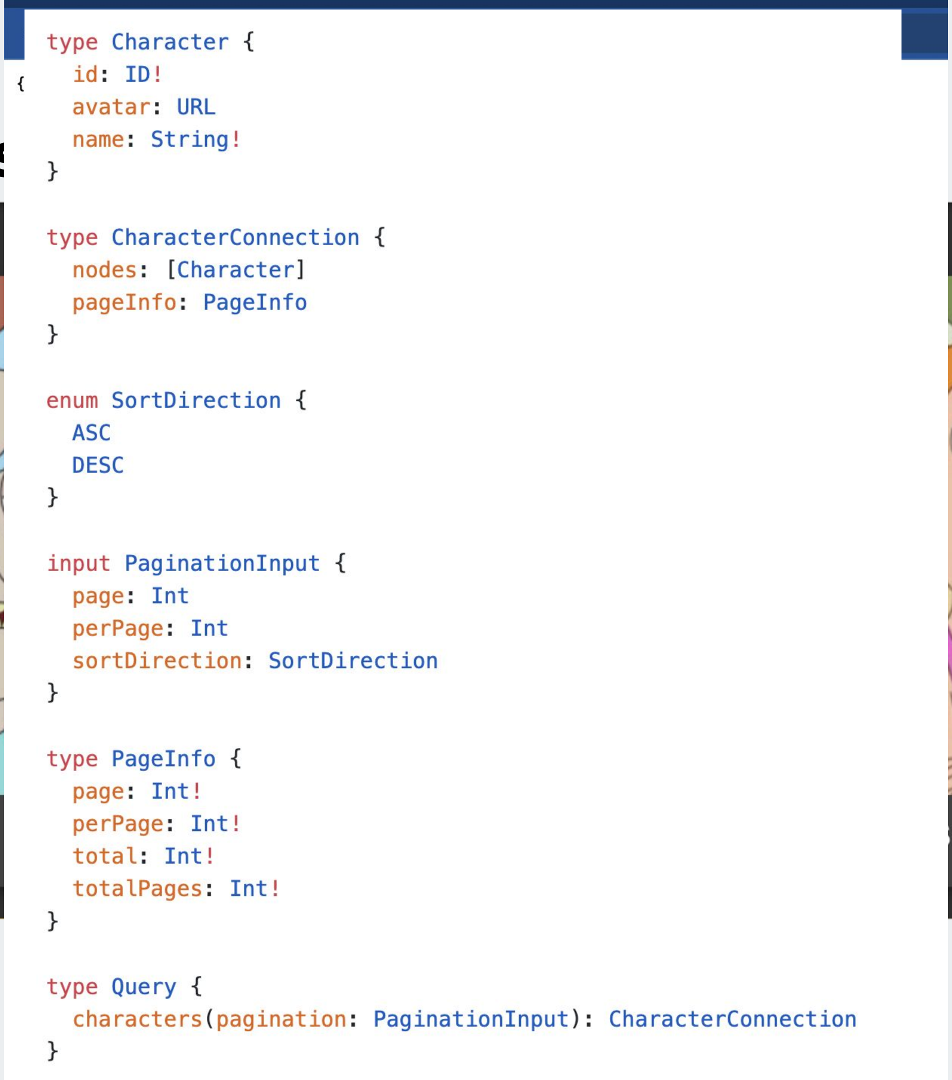
- Custom scalars are a great way to inform users more information about fields in your API
- Requires strict validation rules
- Consider using existing ones: https://www.the-guild.dev/graphql/scalars/docs



Available Scalars

- AccountNumber
- BigInt
- Byte
- CountryCode
- Currency
- Date
- DateTime
- DID
- Duration
- EmailAddress
- HexColorCode
- Hexadecimal
- HSL
- IPv4
- IPv6
- IBAN
- ISBN
- JSON
- JSONObject

Defining bas



```
type Character {
  id: ID!
  avatar: URL
  name: String!
}

type CharacterConnection {
  nodes: [Character]
  pageInfo: PageInfo
}

enum SortDirection {
  ASC
  DESC
}

input PaginationInput {
  page: Int
  perPage: Int
  sortDirection: SortDirection
}

type PageInfo {
  page: Int!
  perPage: Int!
  total: Int!
  totalPages: Int!
}

type Query {
  characters(pagination: PaginationInput): CharacterConnection
}
```

Strategies & Considerations for Migration

Leverage existing APIs to power the new API

- **Pros**:
  - Faster initial implementation
  - Keeps REST problems on the server
- **Cons**:
  - Modifications are harder to implement
  - Slower APIs

Migrate existing API logic to new resolvers

- **Pros**:
  - Finer control over implementation details
  - Easier to tune performance
  - Scales over time
- **Cons**:
  - Slower initial implementation

Server Considerations

- Monolith or microservices? Federation?
- Using a server implementation like Apollo or Relay?
- If leveraging existing REST endpoints and Apollo, RESTDataSources?
- General Considerations
    - Security
    - Query Complexity
    - Rate Limiting

Client Migrations

- Dependent on your frontend implementations
- Using a services architecture can simplify this migration
- Old patterns don’t necessarily translate to the new model
- Consider a first-class GraphQL client implementation

Migration Demo

Additional Notes

- Utilizing GraphQL codegen can provide your frontend types and queries
- Don’t just remap REST fields to GraphQL - consider how they’re used and converting into fields that handle the business logic
- **Use Dataloaders!** The GraphQL N+1 problem does exist but dataloaders eliminates it from the equation.
- Don’t forget to write validation and leverage custom scalars for advanced type validation at the API layer

Q&A

Thank you!