



Webinar On



# Documenting GraphQL APIs: How is it different than REST?



Speaker

**Mark Wentowski**

API Documentation Specialist

Independent consultant

# Why learn GraphQL?

- Growing popularity
- Facilitate communication: frontend /backend
- Write interactive examples
- Enhance API developer experience

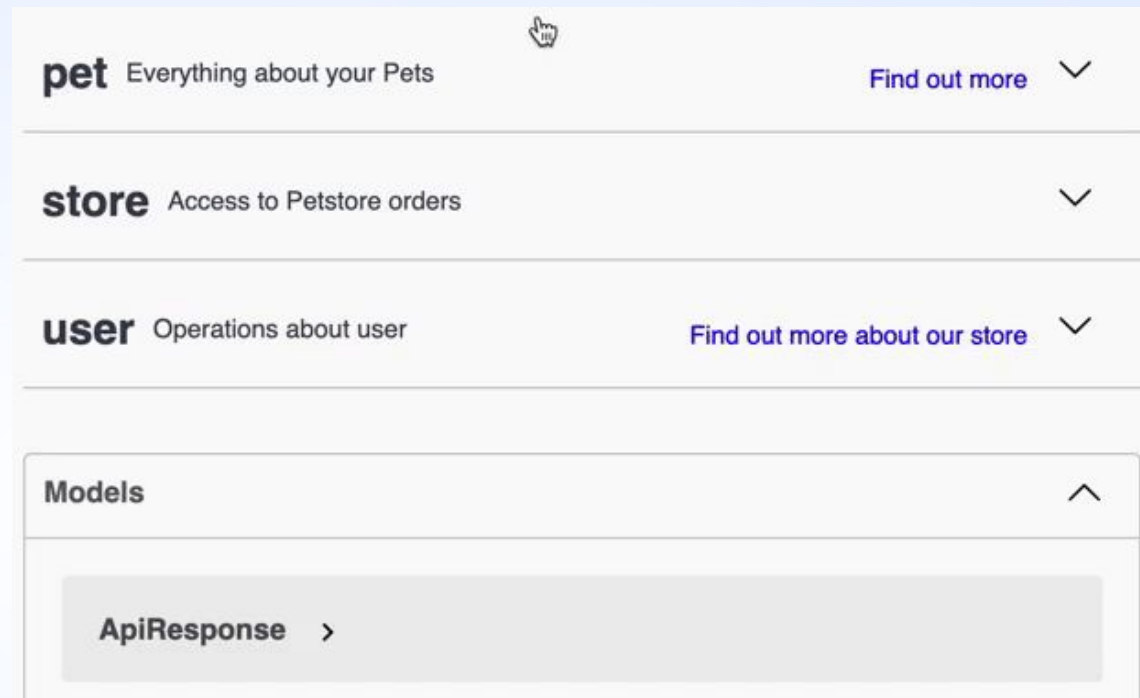
# GraphQL vs. REST

- They both approaches to designing APIs
- They differ significantly in how they structure data and how clients interact with the API
- Not mutually exclusive

# REST

- Architectural style for designing networked applications
- Manage information by using web addresses
- Strictly defined data structures

# REST - 'endpoint-based'



# REST - 'fixed and structured'

The screenshot shows a REST client interface for a POST request to the endpoint `/pet`. The request body is a JSON object representing a pet, with the following structure:

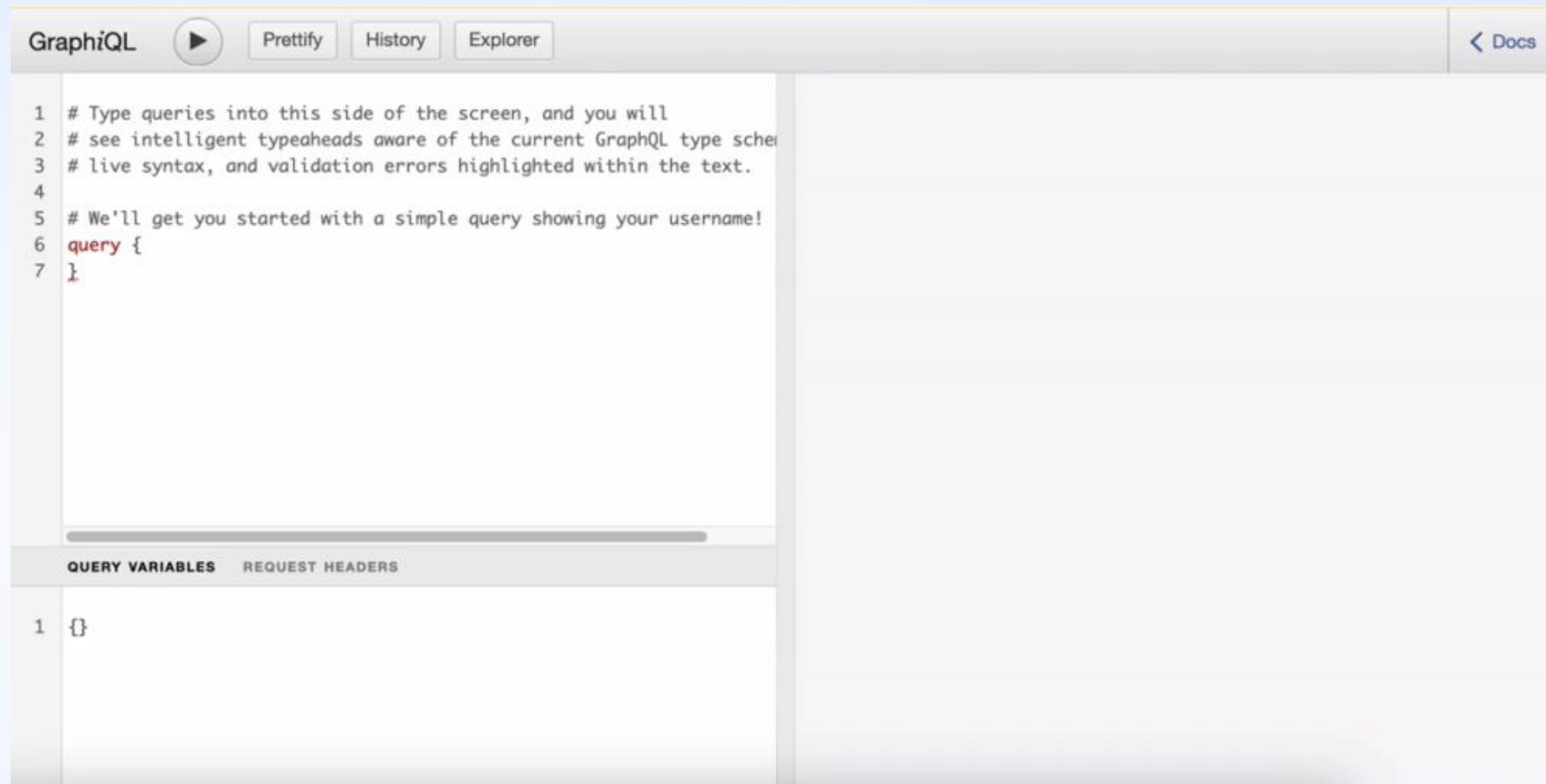
```
{
  "id": 1,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "doggie",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}
```

The interface includes a 'Parameters' section with a 'Cancel' button, a 'body \* required' section with a 'Pet object that needs to be added to the store' description, and a 'Parameter content type' dropdown menu set to 'application/json'. At the bottom, there are 'Execute' and 'Clear' buttons.

# GraphQL

- GraphQL is a query language
- Schema-based approach to requesting data

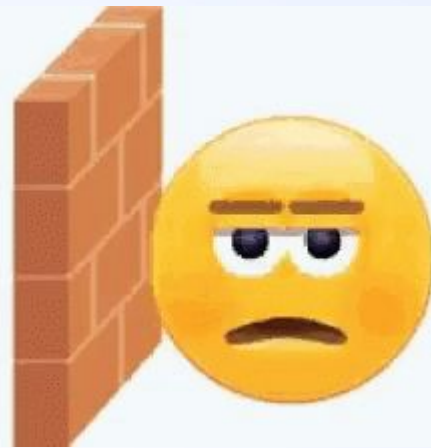
# GraphiQL - 'flexible'





# Underfetching / Overfetching

1. *Why can't I get all the data I need in one request?*



2. *Why do I have to get back all this data when I only need a subset?*

# Fake Blog API

GET	/api/posts	Get all blog posts	▼
POST	/api/posts	Create a new blog post	▼
GET	/api/posts/{postId}/comments	Get a specific blog post's comments	▼
PUT	/api/posts/{postId}	Update a blog post	▼
DELETE	/api/posts/{postId}	Delete a blog post	▼
POST	/api/users	Create a new user	▼
GET	/api/users/{userId}/posts	Get a user's posts	▼
GET	/api/comments	Get all comments	▼
GET	/api/comments/{commentId}	Get a specific comment	▼
PUT	/api/comments/{commentId}	Update a comment	▼
DELETE	/api/comments/{commentId}	Delete a comment	▼

# REST Example: Underfetching

I want details for a specific blog post and its comments.

**Request #1:**

GET

`/api/posts/{postId}` Get a specific blog post

**Request #2:**

GET

`/api/posts/{postId}/comments` Get a specific blog post's comments

# GraphQL - Fetch with one request

## Request

```
1 query {  
2   post(id: 1) {  
3     id  
4     title  
5     content  
6     createdAt  
7     comments {  
8       id  
9       text  
10      createdAt  
11    }  
12  }  
13 }  
14
```



## Response

```
"data": {  
  "post": {  
    "id": 1,  
    "title": "Sample Blog Post",  
    "content": "This is a sample blog post content.",  
    "createdAt": "2023-07-25T12:34:56Z",  
    "comments": [  
      {  
        "id": 101,  
        "text": "Great post!",  
        "createdAt": "2023-07-25T14:00:00Z"  
      },  
      {  
        "id": 102,  
        "text": "Thanks for sharing!",  
        "createdAt": "2023-07-25T15:30:00Z"  
      }  
    ]  
  }  
}
```

# REST Example: Overfetching

I want specific user details.

**Request:**

GET

/api/users/{userId} Get user details

**Response:**

Example Value | Schema

```
{
  "id": 0,
  "name": "string",
  "email": "string",
  "age": 0,
  "bio": "string",
  "website": "string"
}
```

# GraphQL - Fetch subset of data

## Request

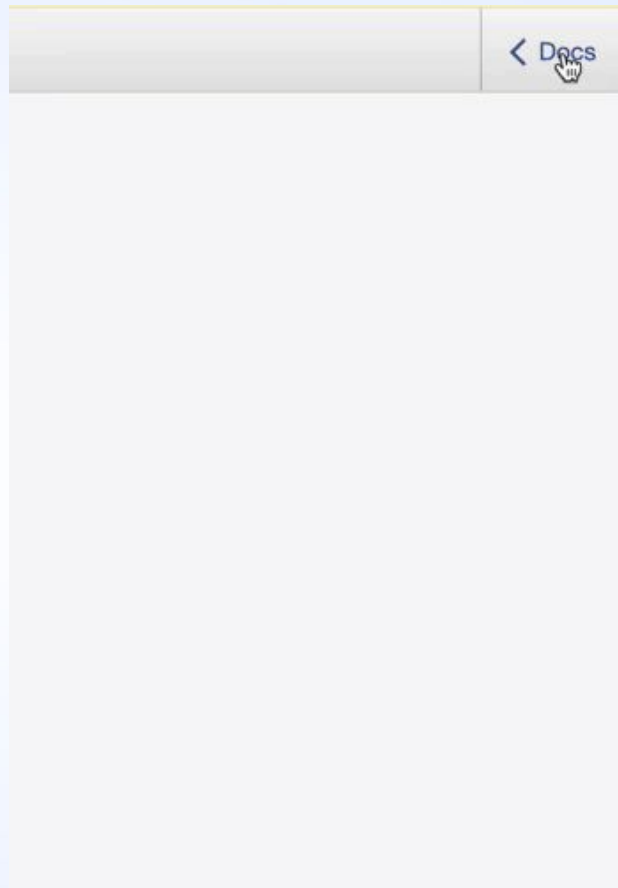
```
1 query {  
2   user(id: 1) {  
3     email  
4     bio  
5   }  
6 }  
7
```



## Response

```
{  
  "data": {  
    "user": {  
      "email": "john.doe@example.com",  
      "bio": "Passionate about blogging and technology."  
    }  
  }  
}
```

# Documenting GraphQL APIs - Field descriptions



```
1 type Book {  
2   id: ID!  
3   title: String!  
4   author: String!  
5 }  
6  
7 type Query {  
8   books: [Book!]!  
9 }  
10
```

## Code comments

Type	Field	Description
Book	id	The unique identifier of the book.
Book	title	The title of the book.
Book	author	The author of the book.
Query	books	

## Google sheets + scripts

```
1 {  
2   "Book": {  
3     "id": "The unique identifier of the book.",  
4     "title": "The title of the book.",  
5     "author": "  
6   },  
7   "Query": {  
8     "books": "Get a list of all books in the library."  
9   }  
10 }
```

## JSON / YML + scripts



# Documenting GraphQL APIs - Conceptual docs

- Knowledge bases
- Help authoring tools
- Static site generators (Markdown / git)

# Single endpoint

<b>Example</b>	GraphQL endpoint: <ul style="list-style-type: none"><li>● <a href="https://fakeblogapi.com/graphql">https://fakeblogapi.com/graphql</a></li></ul>
<b>Documentation Strategy</b>	<ul style="list-style-type: none"><li>● Provide a code example demonstrating how to send queries to the GraphQL endpoint using libraries.</li></ul>

# Query language focus

<b>Example</b>	"GraphQL is the query language used by the Blog API to allow clients to fetch data from the server."
<b>Documentation Strategy</b>	<ul style="list-style-type: none"><li>● Introduce GraphQL and key features</li><li>● Compare GraphQL with traditional RESTful APIs</li></ul>

# Schema documentation

<b>Example</b>	<pre>1 type Post { 2   id: ID! 3   title: String! 4   content: String! 5   createdAt: String! 6   author: User! 7   comments: [Comment!]! 8 }</pre>
<b>Documentation Strategy</b>	<ul style="list-style-type: none"><li>● Document types, fields and relationships.</li></ul>

# GraphiQL support

<b>Example</b>	"The Blog API supports GraphiQL, an interactive IDE for exploring and testing GraphQL queries."
<b>Documentation Strategy</b>	<ul style="list-style-type: none"><li>● Introduce GraphiQL</li><li>● Access instructions</li><li>● Testing examples</li></ul>

# Query Variables

<b>Example</b>	<pre>1 query GetUserDetails(\$userId: ID!) { 2   user(id: \$userId) { 3     id 4     name 5     email 6   } 7 }</pre>
<b>Documentation Strategy</b>	<ul style="list-style-type: none"><li>• Describe role of query variables</li><li>• Demonstrate how to use query variables</li></ul>

# Introspection queries

<b>Example</b>	<pre>1 query IntrospectionQuery { 2   __schema { 3     types { 4       name 5     } 6   } 7 }</pre>
<b>Documentation Strategy</b>	<ul style="list-style-type: none"><li>● Explain introspection in GraphQL</li><li>● Provide examples that developers can execute to explore the API's schema.</li></ul>

# Security

<b>Example</b>	<pre>1 query GetUserDetails(\$userId: ID!) { 2   user(id: \$userId) { 3     id 4     name 5     email 6     password 7   } 8 }</pre>
<b>Documentation Strategy</b>	<ul style="list-style-type: none"><li>● Explain that sensitive information should not be requested.</li><li>● Provide best practices for ensuring confidential information is not exposed.</li></ul>



# Error handling

<b>Example</b>	<pre>1 { 2   <u>"errors":</u> [ 3     { 4       "message": "Invalid input: Name cannot be empty.", 5     } 6   ] 7 }</pre>
<b>Documentation Strategy</b>	<ul style="list-style-type: none"><li>● Explain the structure of error responses</li><li>● Document common error scenarios and how to handle them</li></ul>

# Tutorials

<b>Example</b>	Describe the scenario where a user wants to create a new blog post through the API.
<b>Documentation Strategy</b>	Provide real-world examples and step-by-step guides, developers can understand how to interact with the API in practical scenarios.

# Example tutorial

## Create a new blog post

Explain the purpose of the mutation and its expected input fields (title, content, and authorId).

1. **Mutation Query:** Provide the mutation query with placeholders for the required variables.
2. **Query Variables:** Explain the purpose of each query variable (\$title, \$content, and \$authorId) and their expected data types.
3. **Execution:** Show how to execute the mutation with actual values for the query variables.
4. **Response:** Explain the structure of the response and how to interpret the returned data (in this case, the id, title, and createdAt fields of the newly created post).

## Sample request

```
1 mutation CreateNewPost($title: String!, $content: String!, $authorId: ID!) {  
2   createPost(title: $title, content: $content, authorId: $authorId) {  
3     id  
4     title  
5     createdAt  
6   }  
7 }
```

## Sample response

```
{  
  "data": {  
    "createPost": {  
      "id": "123",  
      "title": "New Blog Post",  
      "createdAt": "2023-07-25T12:34:56Z"  
    }  
  }  
}
```

Questions ?

# Thank You!

