

GraphQL

where Frontend Meets Freedom

Pascal Sun

UWA NLP-TLP Group

The University of Western Australia

December 5, 2024

Table of Contents

1 Web Communication

2 RESTful to GraphQL

3 Hasura

4 GraphQL in React

5 Demo

6 Q&A

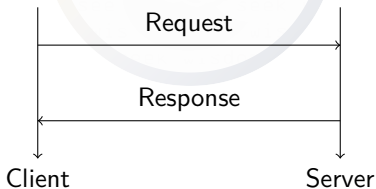


Web Communication

Traditional HTTP: The Beginning

How Web Communication Started

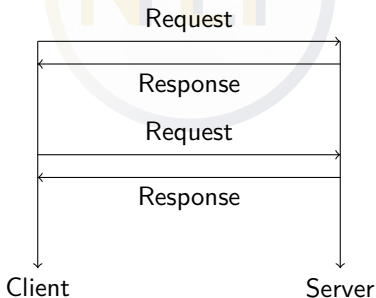
- Client sends request
- Server processes request
- Server sends response
- Connection closes
- New request = New connection



HTTP Limitations

The Problems We Faced

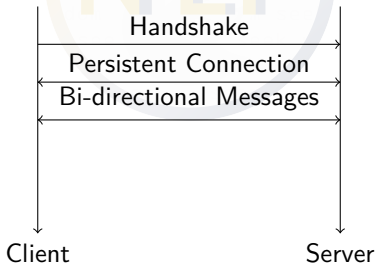
- Polling needed for updates
- High server load
- Bandwidth waste
- Delayed updates
- Multiple connections



Enter WebSocket (Proposed in 2008, standardized in 2011)

The Real-time Revolution

- Persistent connection
- Bi-directional communication
- Real-time updates
- Lower latency
- Less overhead



WebSocket Use Cases

Perfect for

- Live chat applications
- Real-time dashboards
- Gaming
- Live sports updates
- Collaborative tools

Not ideal for

- Simple CRUD operations
- File uploads
- One-time data fetches

AI Model Protocol: MCP

Model Context Protocol (2024.11.25)¹

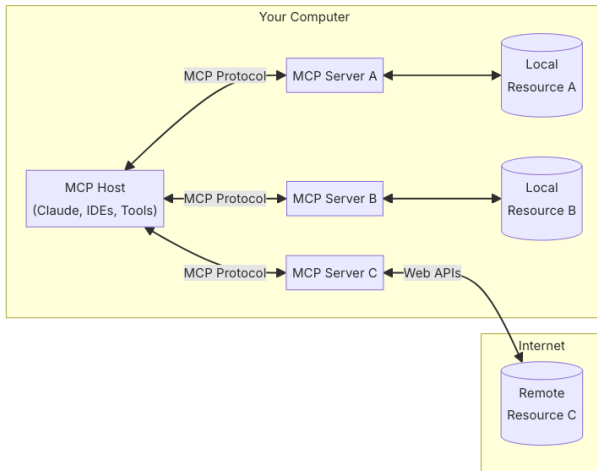


Figure: General overview for MCP released by Claude

Protocol Comparison

Feature	HTTP/2	WebSocket	MCP
Connection	Persistent	Persistent	Persistent
Bi-directional	No	Yes	Yes
Flow Control	Yes	Limited	Advanced
Use Case	RESTful	Real-time	AI Context

Best Practices

Use HTTP when

- Simple request/response
- RESTful architecture

Use WebSocket when

- Real-time updates
- Bi-directional comm
- Low latency required

Consider MCP when

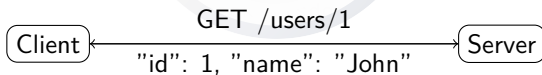
- It is coming for AI Agents

RESTful to GraphQL

What is RESTful API?

REST (Representational State Transfer)

- Architectural style for distributed systems
- Based on HTTP protocol
- Uses standard HTTP methods (GET, POST, PUT, DELETE)
- Resources identified by URLs
- Stateless communication



RESTful API Example

Typical REST Endpoints

```
1 GET    /api/users           // Get all users
2 GET    /api/users/1        // Get user with ID 1
3 GET    /api/users/1/posts  // Get posts of user 1
4 POST   /api/users          // Create new user
5 PUT    /api/users/1        // Update user 1
6 DELETE /api/users/1        // Delete user 1
7
```

Problems with REST

Over-fetching

- Receiving more data than needed
- Wastes bandwidth
- Increases response time

Under-fetching

- Need multiple requests
- N+1 problem
- Increased latency

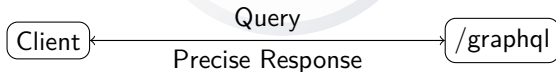
Other Issues

- Multiple endpoints for related data
- Versioning challenges
- Documentation complexity
- Limited client control
- Frontend and Backend developers fight

Enter GraphQL

What is GraphQL?

- Query language for APIs
- Single endpoint architecture
- Developed by Facebook (2012)
- Released publicly in 2015
- Precise data fetching



GraphQL vs REST: Example

REST (Multiple Requests)

```
1 GET /api/users/1
2 GET /api/users/1/posts
3 GET /api/users/1/followers
4
```

GraphQL (Single Request)

```
1 query {
2   user(id: 1) {
3     name
4     posts {
5       title
6     }
7     followers {
8       name
9     }
10  }
11 }
12
```

Key Benefits of GraphQL

For Developers

- Strong typing system
- Better developer experience
- Self-documenting
- Easier versioning

For Applications

- Reduced network requests
- Precise data fetching
- Real-time capabilities
- Better performance

GraphQL Operations

Three Types of Operations

- **Query:** Fetch data (READ)
- **Mutation:** Modify data (CREATE, UPDATE, DELETE)
- **Subscription:** Real-time updates => Supported by WebSocket

```
1 # Query Example
2 query {
3   user(id: 1) {
4     name
5     email
6   }
7 }
8 # Mutation Example
9 mutation {
10   createUser(name: "John", email: "john@example.com") {
11     id
12     name
13   }
14 }
15
```

When to Choose GraphQL

Ideal Use Cases

- Complex data relationships
- Mobile applications
- Microservices architecture
- Real-time features

Consider REST for

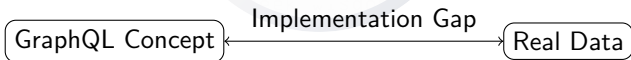
- Simple CRUD operations
- File uploads
- Simple data structures
- Caching requirements
- Things can not be easily done by GraphQL



GraphQL Implementation Challenges

The Gap Between Concept and Reality

- GraphQL is fantastic in theory
- But how do we connect it to our data?
- Who implements the resolvers?
- How to handle optimization?
- Performance concerns



Early GraphQL Implementation Challenges

RESTful Advantages

- Mature ecosystem
- Many battle-tested packages
- Clear best practices
- Easy to implement

GraphQL Challenges

- Limited community support
- Complex implementation
- Performance optimization
- Learning curve

Python GraphQL Challenges

Technical Limitations

- Limited async support in Python
- N+1 query performance issues
- Complex data loader implementation
- Resource-intensive resolvers

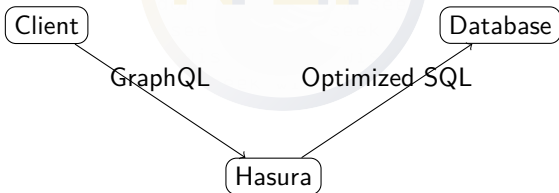
The Database Distance Problem

- GraphQL queries should be close to DB
- Multiple layers add latency
- Complex query optimization needed

Enter Hasura

What is Hasura?

- GraphQL engine that sits directly on your database
- Instant GraphQL API from your schema
- Written in Haskell for performance
- Built-in authorization and caching



Why Hasura Shines

Performance Benefits

- Direct DB connection
- Optimized SQL generation
- Built-in caching
- Real-time subscriptions

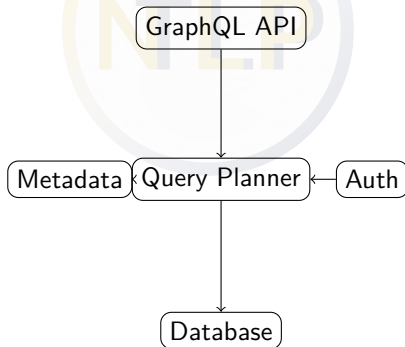
Developer Experience

- Zero GraphQL boilerplate
- Automatic CRUD
- Built-in authorization
- Auto-documentation

Hasura Architecture

Key Components

- Metadata storage
- Permission system
- Query planner
- Event trigger system



Hasura in Action

From Schema to API

```
1 -- Your database schema
2 CREATE TABLE users (
3     id SERIAL PRIMARY KEY,
4     name TEXT,
5     email TEXT
6 );
7
8 -- Hasura automatically generates:
9 type Query {
10     users: [User!]!
11     user_by_pk(id: Int!): User
12 }
13 type User {
14     id: Int!
15     name: String
16     email: String
17 }
18
```

Hasura Features

Core Features

- Auto CRUD
- Real-time subs
- Relations

Security

- Row-level security
- Column permissions
- JWT auth

Advanced

- Remote schemas
- Actions
- Events

Downloaded from <http://ajphaphysocpharm.sagepub.com> at 11:01 11 November 2014

GraphQL REST [Follow List](#) [Security](#)

GraphQL Endpoint

POST <https://graphql.galaxia.com/v1/graphql> ☐ Relay API

Request Headers

ENABLE	KEY	VALUE	
☒	content-type	application/json	X
☒	x-hausio-admin-secret	-----	X
	Enter Key	Enter Value	

Explorer GraphQL ☐ POST ☐ History ☐ Explorer ☐ Cache ☐ Code Exporter ☐ REST ☐ Derive action ☐ Analysis

- + where:
- annotation_id
- annotations
- copies
- comp_pdf
- id
- page_num
- pdf_id
- vls_key
- + core_figures.aggregate
- + core_formulas
- + core_formula.aggregate
- + core_project
- + core_tabs
- + core_tabs.aggregate
- ocr_result_id
- + ocrresult_results
- + ocrresult_results.aggregate
- vls_key
- id
- layout_annotations
- path
- layout_annotations_draft
- path
- name
- ocr_annotations
- path
- ocr_annotations_draft
- project_id
- semantic_annotations
- semantic_annotations_draft
- status
- updated_at

```

1 query MyQuery {
2   core_pdf(id: "1A", first: 10, distinct_on: created_at) {
3     id
4     core_figures {
5       id
6     }
7     file_key
8     created_at
9     ocr_annotations
10    name
11    layout_annotations_draft
12    layout_annotations {
13      }
14  }
15 }
        
```

```

{
  "data": {
    "core_pdf": [
      {
        "id": "15P",
        "core_figures": [
          {
            "id": "1487"
          },
          {
            "id": "1485"
          },
          {
            "id": "1489"
          }
        ],
        "file_key": "ADG_Register as a Generator in the NEM.pdf",
        "created_at": "2024-10-13T18:15:12.298895+00:00",
        "ocr_annotations": [
          {
            "x": 0.2367672121997145,
            "y": 0.845190213700376,
            "id": "9f583d94-dbb4-44ec-99ba-fc0b34acc73f",
            "text": "Register as a Generator in the NEM",
            "type": "rectangle",
            "label": "Title",
            "width": 0.4831542961998373,
            "height": 0.4022793214736085,
            "pageNum": 1
          },
          {
            "x": 0.8278193557879332,
            "y": 0.978984949612358,
            "id": "ff4db3d-e2cf-4e6d-9f5c-34dc7bf9af30",
            "text": "https://www.com.au/energy-systems/electricity/national-electricity-market-new/participants-in-the-market/register-as-a-generator-in-the-nem",
            "type": "rectangle",
            "label": "Footnote",
            "width": 0.87134684875461,
            "height": 0.8189863657983742,
            "pageNum": 1
          }
        ]
      }
    ]
  }
}
                
```

RESPONSE TIME: 1.0 ms RESPONSE SIZE: 237387B bytes

Figure: GraphQL example in Hasura

Hasura Metadata Management

What is Metadata?

- Configuration of your GraphQL API
- Tracked tables and relationships
- Permissions and roles
- Actions and remote schemas
- Event triggers

Hasura Metadata Management

Export Metadata

```
1 # Using Hasura CLI
2 hasura metadata export
3
4 # Generates files in metadata/ directory:
5 metadata/
6   |- actions.yaml
7   |- allow_list.yaml
8   |- databases/
9   |- rest_endpoints.yaml
10  |- version.yaml
11
```

Hasura Metadata Management

Apply Metadata

```
1 # Apply metadata from files
2 hasura metadata apply
3
4 # Reset and apply fresh
5 hasura metadata reload
6
```

Metadata Best Practices

Version Control

- Commit metadata to git
- Document changes
- Use CI/CD pipelines
- Test before applying

Common Issues

- Inconsistent states
- Missing dependencies
- Permission conflicts
- Relationship errors

Key Points

- Always backup before applying
- Use environment variables
- Maintain metadata per environment
- Regular health checks

GraphQL in React

GraphQL Client Libraries

Popular Options

- Apollo Client
- URQL
- React Query + GraphQL Request
- SWR + GraphQL Request



Apollo Client Features

Advantages

- Rich feature set
- Robust caching
- Dev tools
- Large community
- Full GraphQL support

Considerations

- Bundle size
- Learning curve
- Configuration complexity
- Memory usage

Apollo Client Example

Basic Setup

```
1 // _app.tsx
2 import { ApolloClient, ApolloProvider } from '@apollo/client';
3
4 const client = new ApolloClient({
5   uri: 'your-graphql-endpoint',
6   cache: new InMemoryCache(),
7 });
8
9 function MyApp({ Component, pageProps }) {
10   return (
11     <ApolloProvider client={client}>
12       <Component {...pageProps} />
13     </ApolloProvider>
14   );
15 }
16
```

Using Apollo in Components

```
1 import { gql, useQuery } from '@apollo/client';
2
3 const GET_USERS = gql`
4   query GetUsers {
5     users {
6       id
7       name
8       email
9     }
10  }
11 `;
12 function UserList() {
13   const { loading, error, data } = useQuery(GET_USERS);
14   if (loading) return 'Loading...';
15   if (error) return `Error: ${error.message}`;
16   return (
17     <ul>
18       {data.users.map(user => (
19         <li key={user.id}>{user.name}</li>
20       ))}
21     </ul>
22   );
23 }
```

Advanced GraphQL Query

Variables & Fragments

```
1 # Variables
2 query GetUser($id: ID!) {
3   user(id: $id) {
4     ...UserFields
5   }
6 }
7
8 # Fragment
9 fragment UserFields on User {
10   id
11   name
12   email
13   posts {
14     title
15   }
16 }
17
```



Full Stack GraphQL implementation

- 1 Define the data model
- 2 Database migration with Django
- 3 Track the created tables and relations in Hasura
- 4 Setup Apollo Client in Next.js
- 5 WebSocket/HTTP query with GraphQL



Homework

- 1 Follow what have been done in the demo, implement it based on your project



Q&A

Questions?

Feel free to ask anything