GraphQL

where Frontend Meets Freedom

Pascal Sun

UWA NLP-TLP Group

The University of Western Australia

December 5, 2024

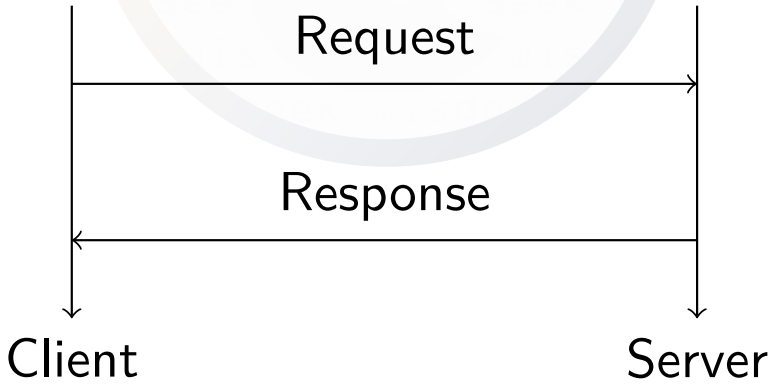
Table of Contents

Web Communication

Traditional HTTP: The Beginning

How Web Communication Started

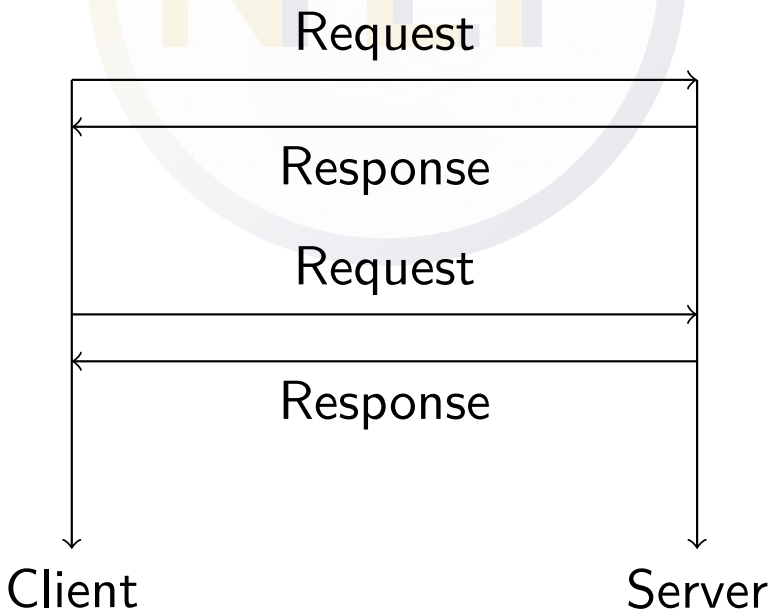
- Client sends request
- Server processes request
- Server sends response
- Connection closes
- New request = New connection



HTTP Limitations

The Problems We Faced

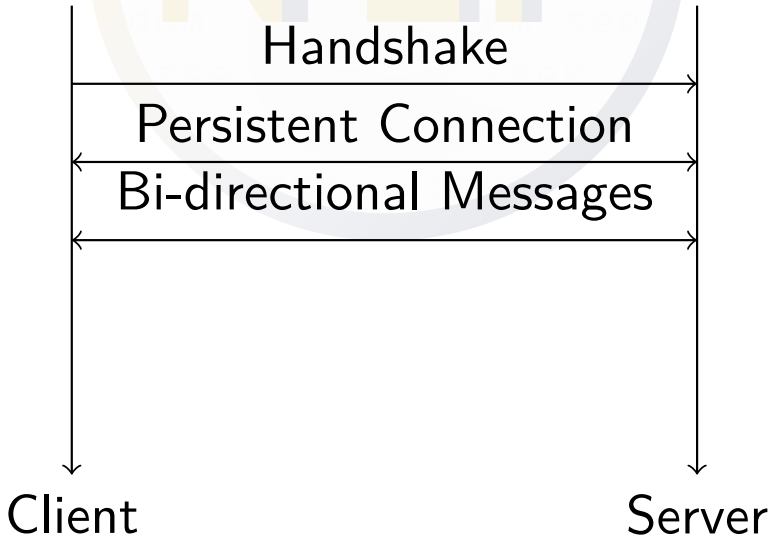
- Polling needed for updates
- High server load
- Bandwidth waste
- Delayed updates
- Multiple connections



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]

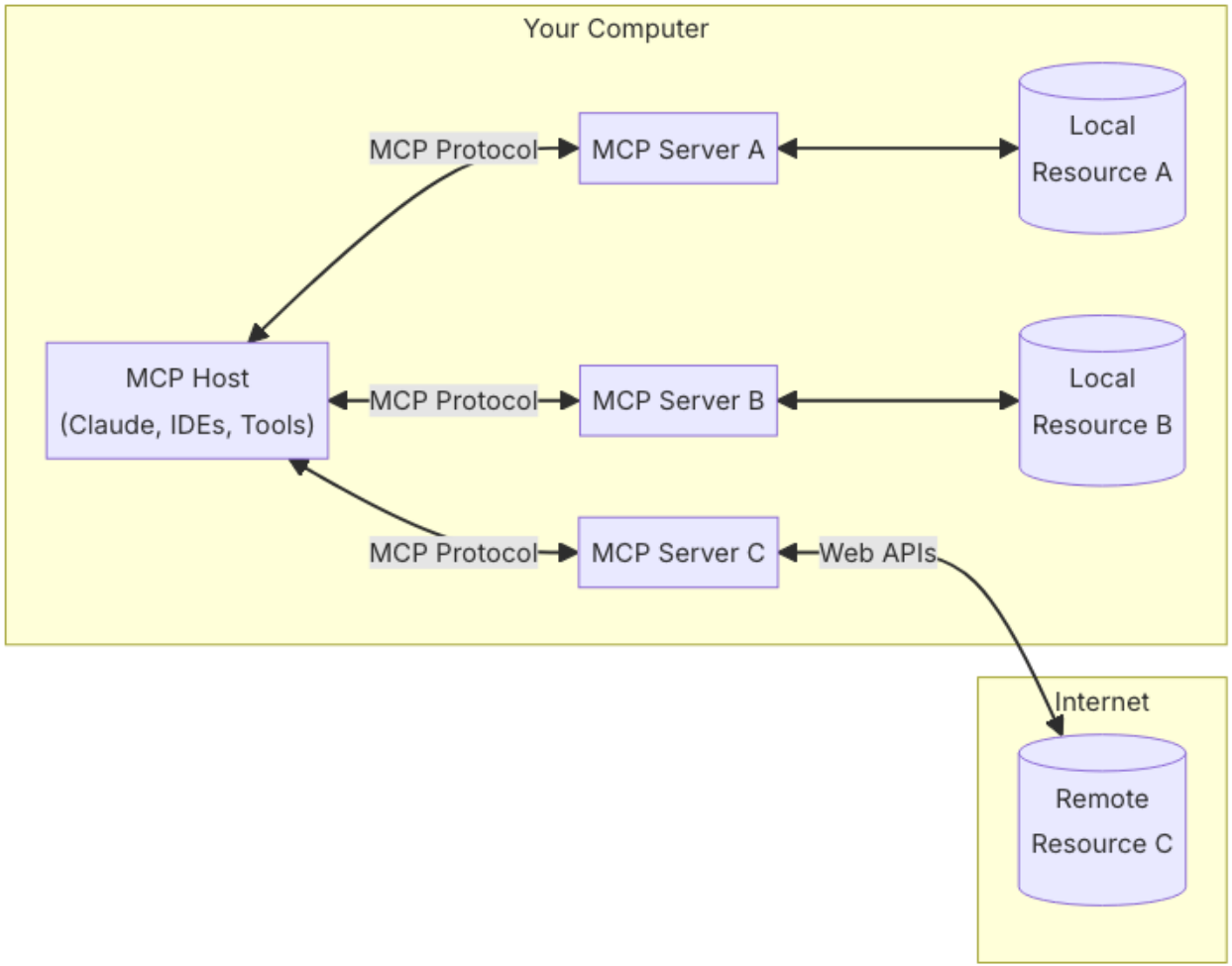


Figure: General overview for MCP released by Claude

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

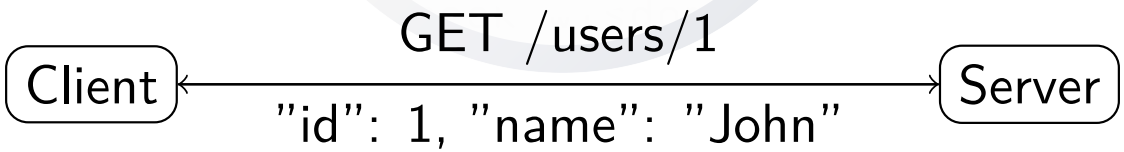
- It is coming for AI Agents

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

```
GET    /api/users            // Get all users
GET    /api/users/1          // Get user with ID 1
GET    /api/users/1/posts    // Get posts of user 1
POST   /api/users            // Create new user
PUT    /api/users/1          // Update user 1
DELETE /api/users/1          // Delete user 1

```

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

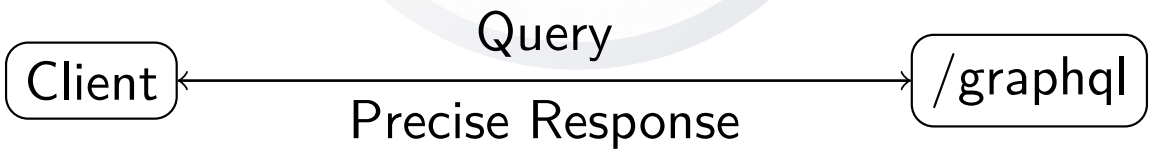
Other Issues

- 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)

```
GET /api/users/1
GET /api/users/1/posts
GET /api/users/1/followers

```

GraphQL (Single Request)

```
query {
  user(id: 1) {
    name
    posts {
      title
    }
    followers {
      name
    }
  }
}

```

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

```
# Query Example
query {
  user(id: 1) {
    name
    email
  }
}
# Mutation Example
mutation {
  createUser(name: "John", email: "john@example.com") {
    id
    name
  }
}

```

When to Choose GraphQL

Ideal Use Cases

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

Consider REST for

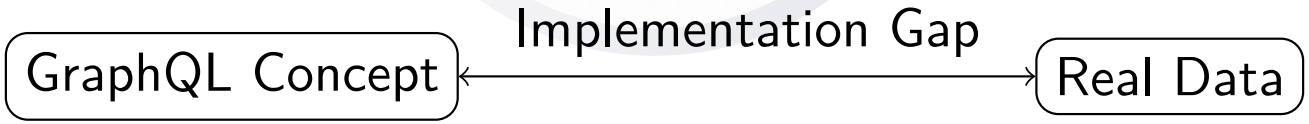
- Simple CRUD operations
- File uploads
- Simple data structures
- Caching requirements
- Things can not be easily done by GraphQL

Hasura

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

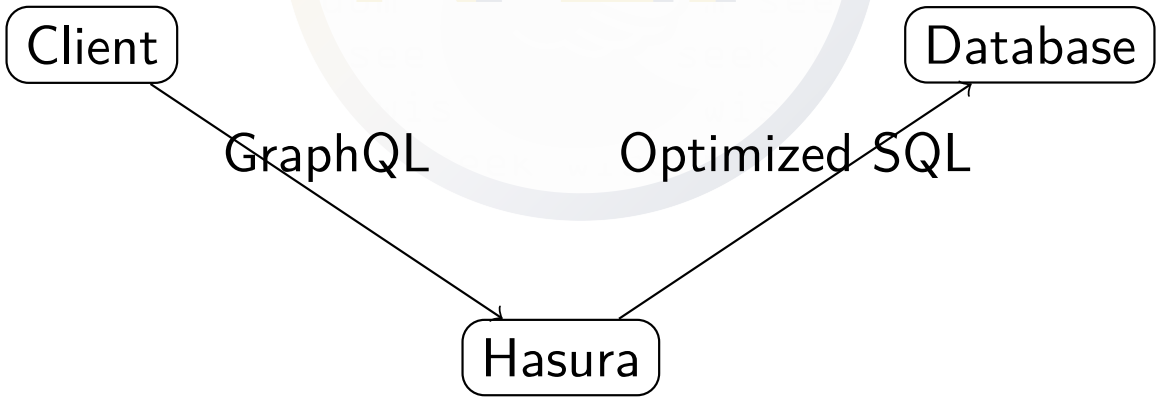
The Database Distance Problem

- 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

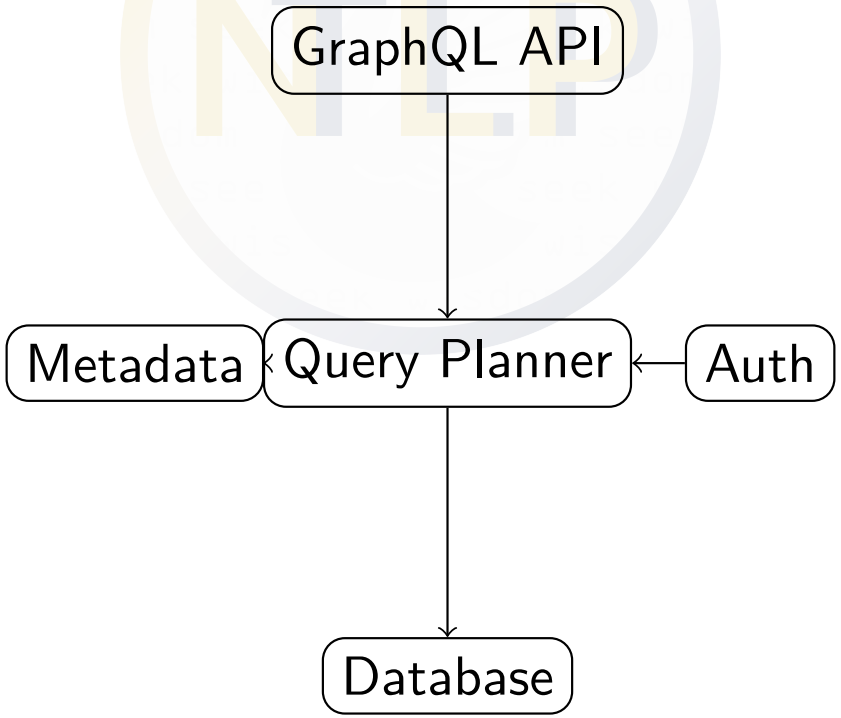
Developer Experience

- 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

```
-- Your database schema
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  name TEXT,
  email TEXT
);

-- Hasura automatically generates:
type Query {
  users: [User!]!
  user_by_pk(id: Int!): User
}
type User {
  id: Int!
  name: String
  email: String
}

```

Hasura Features

Core Features

- Auto CRUD
- Real-time subs
- Relations

Security

- Row-level security
- Column permissions
- JWT auth

Advanced

- Remote schemas
- Actions
- Events

Hasura Example

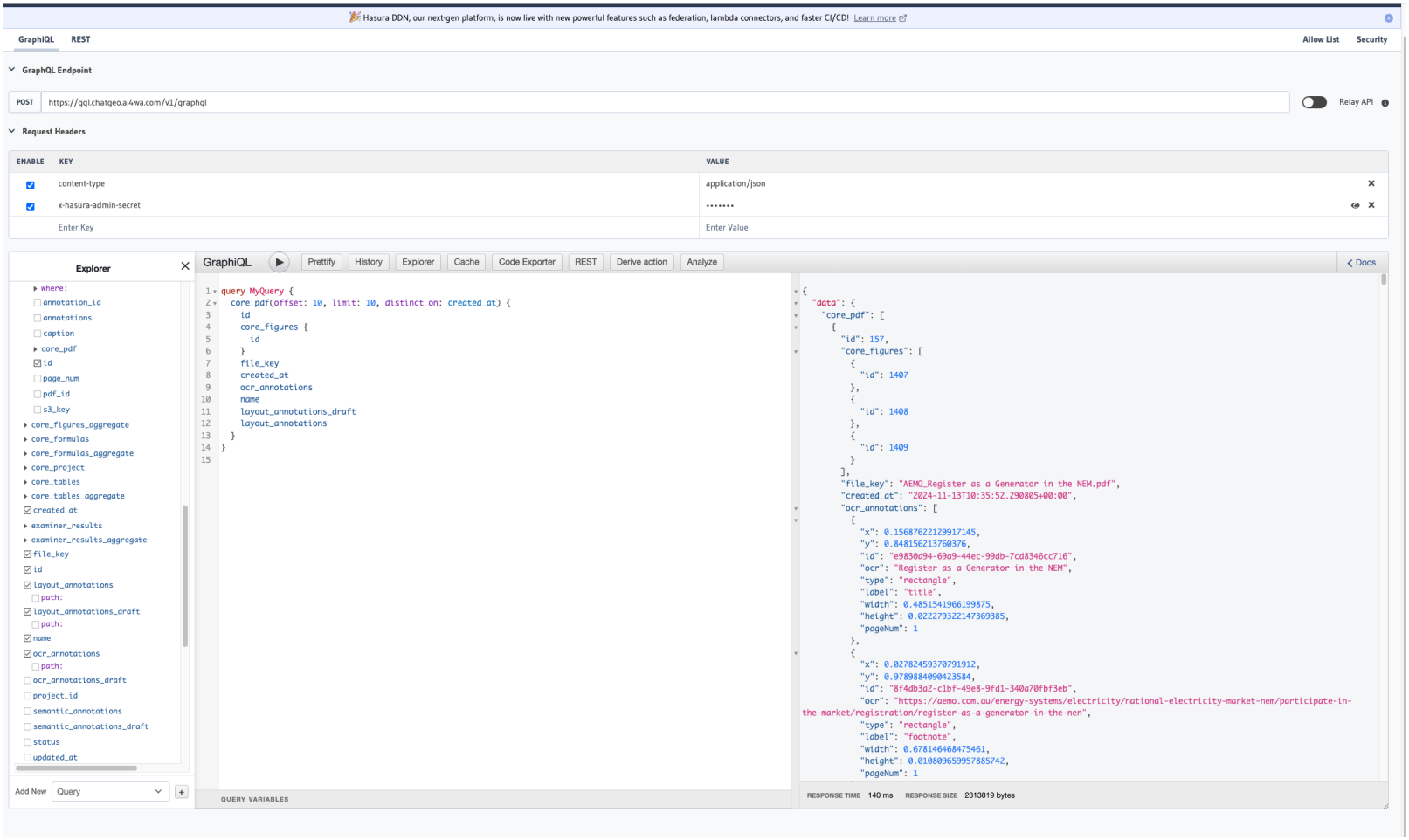


Figure: GraphQL example in Hasura

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

```
# Using Hasura CLI
hasura metadata export

# Generates files in metadata/ directory:
metadata/
   |- actions.yaml
   |- allow_list.yaml
   |- databases/
   |- rest_endpoints.yaml
   |- version.yaml

```

Hasura Metadata Management

Apply Metadata

```
# Apply metadata from files
hasura metadata apply

# Reset and apply fresh
hasura metadata reload

```

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

```
// _app.tsx
import { ApolloClient, ApolloProvider } from '@apollo/client';

const client = new ApolloClient({
  uri: 'your-graphql-endpoint',
  cache: new InMemoryCache(),
});

function MyApp({ Component, pageProps }) {
  return (
    <ApolloProvider client={client}>
      <Component {...pageProps} />
    </ApolloProvider>
  );
}

```

Using Apollo in Components

```
import { gql, useQuery } from '@apollo/client';

const GET_USERS = gql`
  query GetUsers {
    users {
      id
      name
      email
    }
  }
`;
function UserList() {
  const { loading, error, data } = useQuery(GET_USERS);
  if (loading) return 'Loading...';
  if (error) return `Error: ${error.message}`;
  return (
    <ul>
      {data.users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

Advanced GraphQL Query

Variables & Fragments

```
# Variables
query GetUser($id: ID!) {
  user(id: $id) {
    ...UserFields
  }
}

# Fragment
fragment UserFields on User {
  id
  name
  email
  posts {
    title
  }
}

```

Demo

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

Q&A

Homework

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

Q&A

Questions?

Feel free to ask anything