apollographql.com
@apollographql

GraphQL + Apollo: Complete data management for modern apps

Sashko Stubailo, @stubailo
Open Source Lead, Apollo

React从入门到精通

掌握当下最热门的前端利器

王沛
eBay 资深技术专家

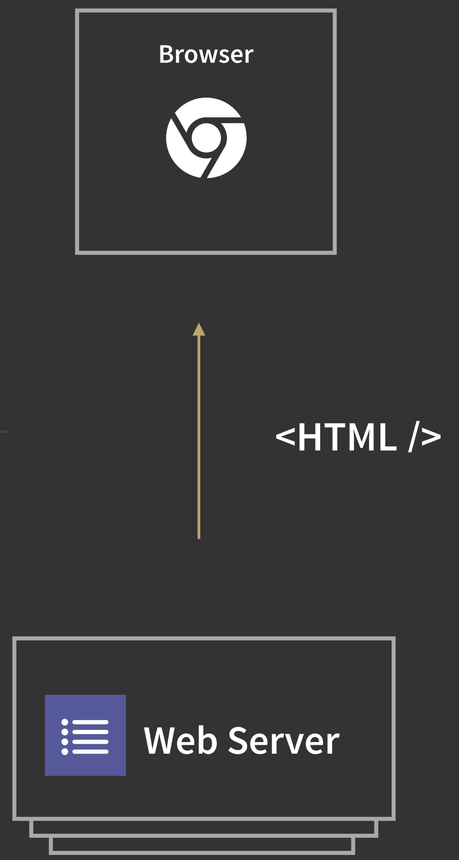
你将获得

1. 全面学习 React 常用技术栈
2. 深入理解 React 设计模式
3. 常见场景下的编程实战指南
4. 掌握用 React 开发大型项目的能力

立即扫码，免费试看

Let's start with a story about app development

A BRIEF HISTORY OF WEB APPS

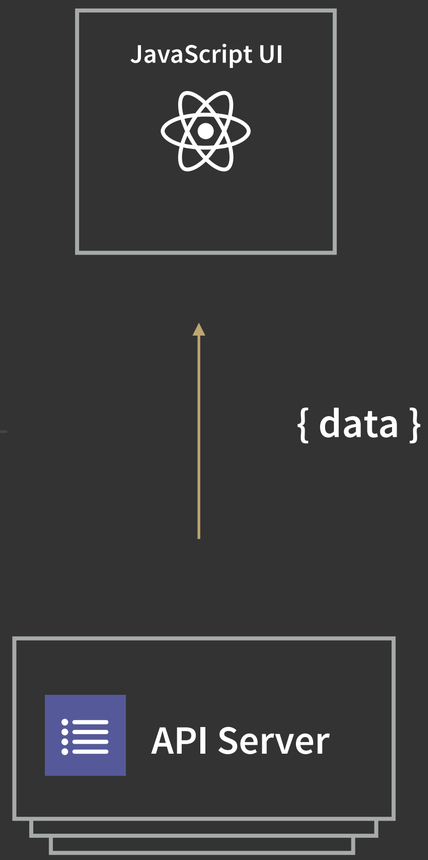


CLIENT

SERVER

Sending HTML from a web server to a browser

A BRIEF HISTORY OF WEB APPS

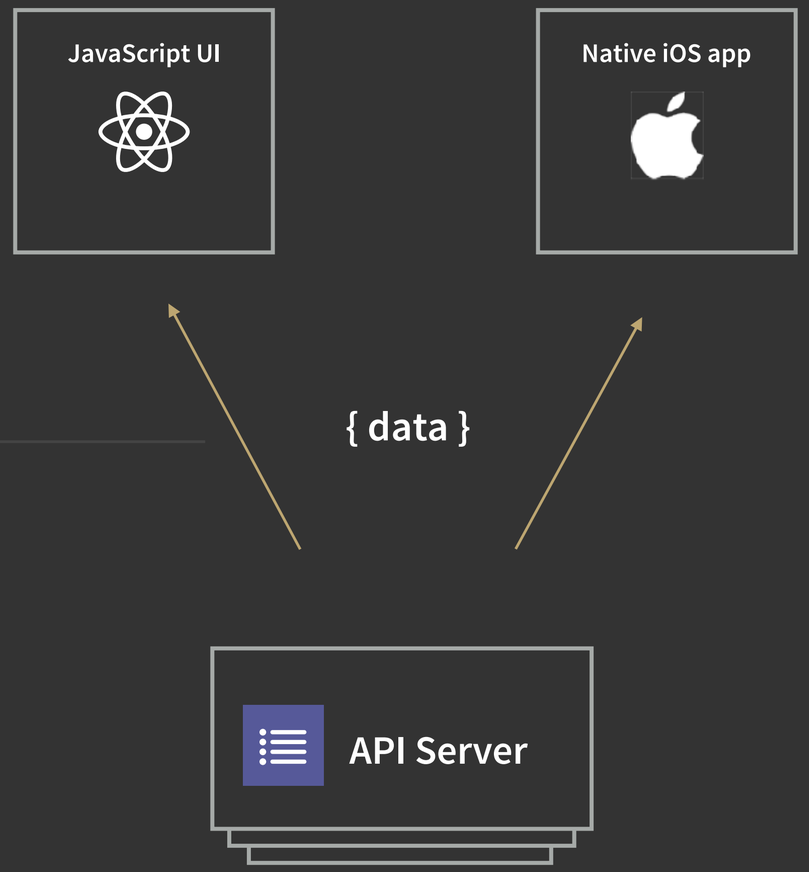


CLIENT

SERVER

Sending data from an API server to a single-page app, for better UI

A BRIEF HISTORY OF APPS

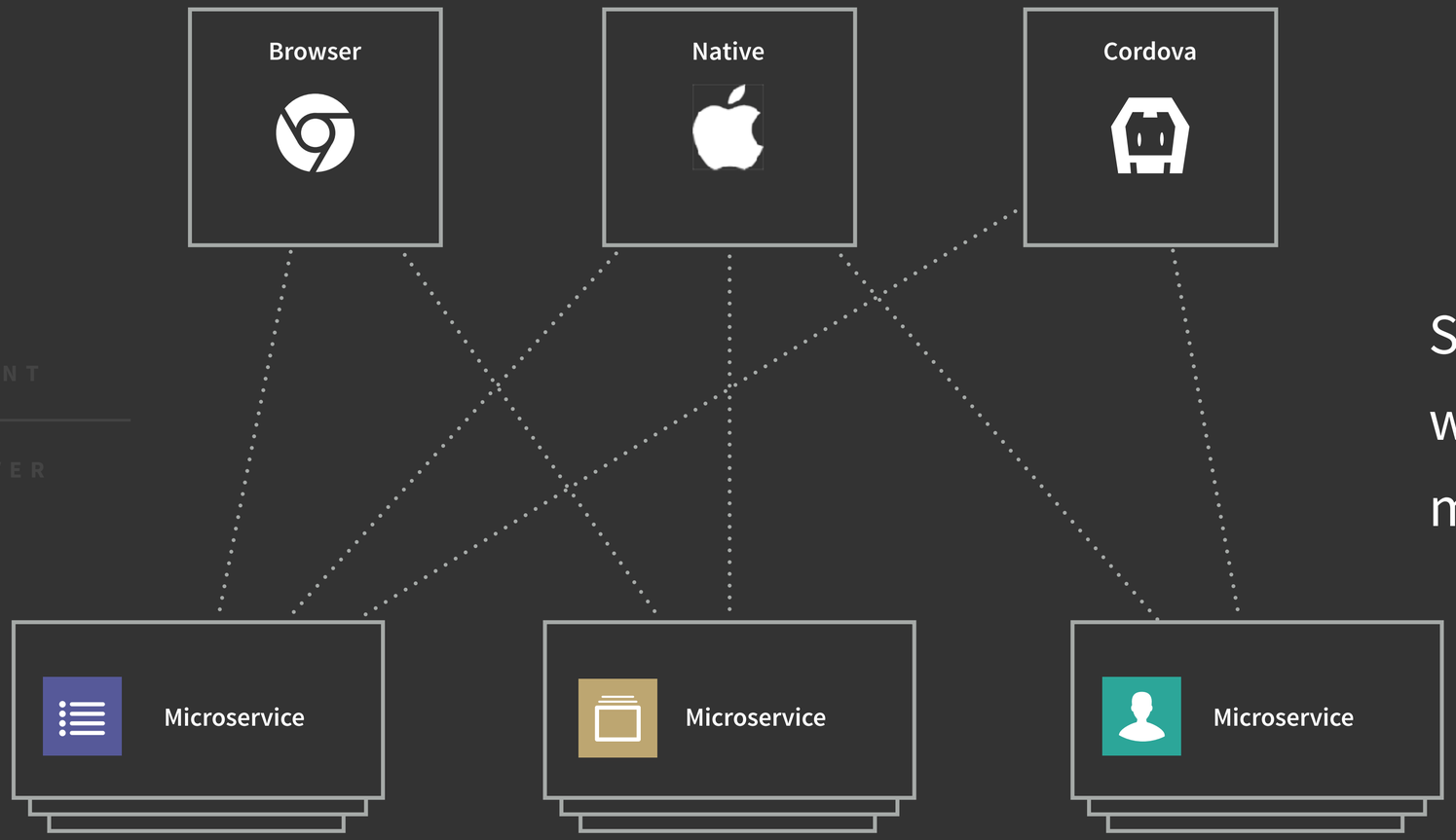


CLIENT

SERVER

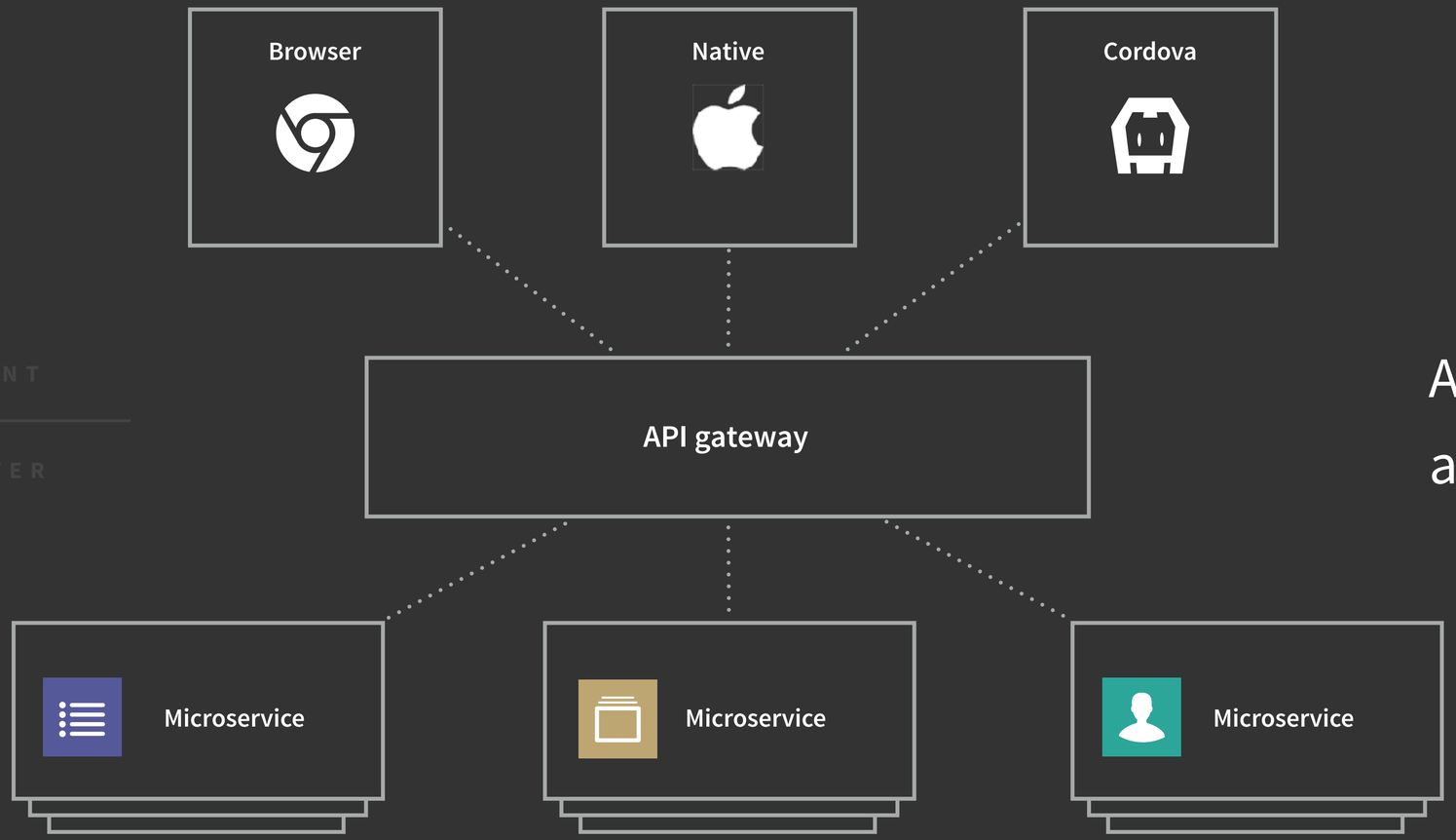
Sending data from an API server to multiple clients in different environments

A BRIEF HISTORY OF APPS



Splitting into microservices, waterfall loading and multiple APIs

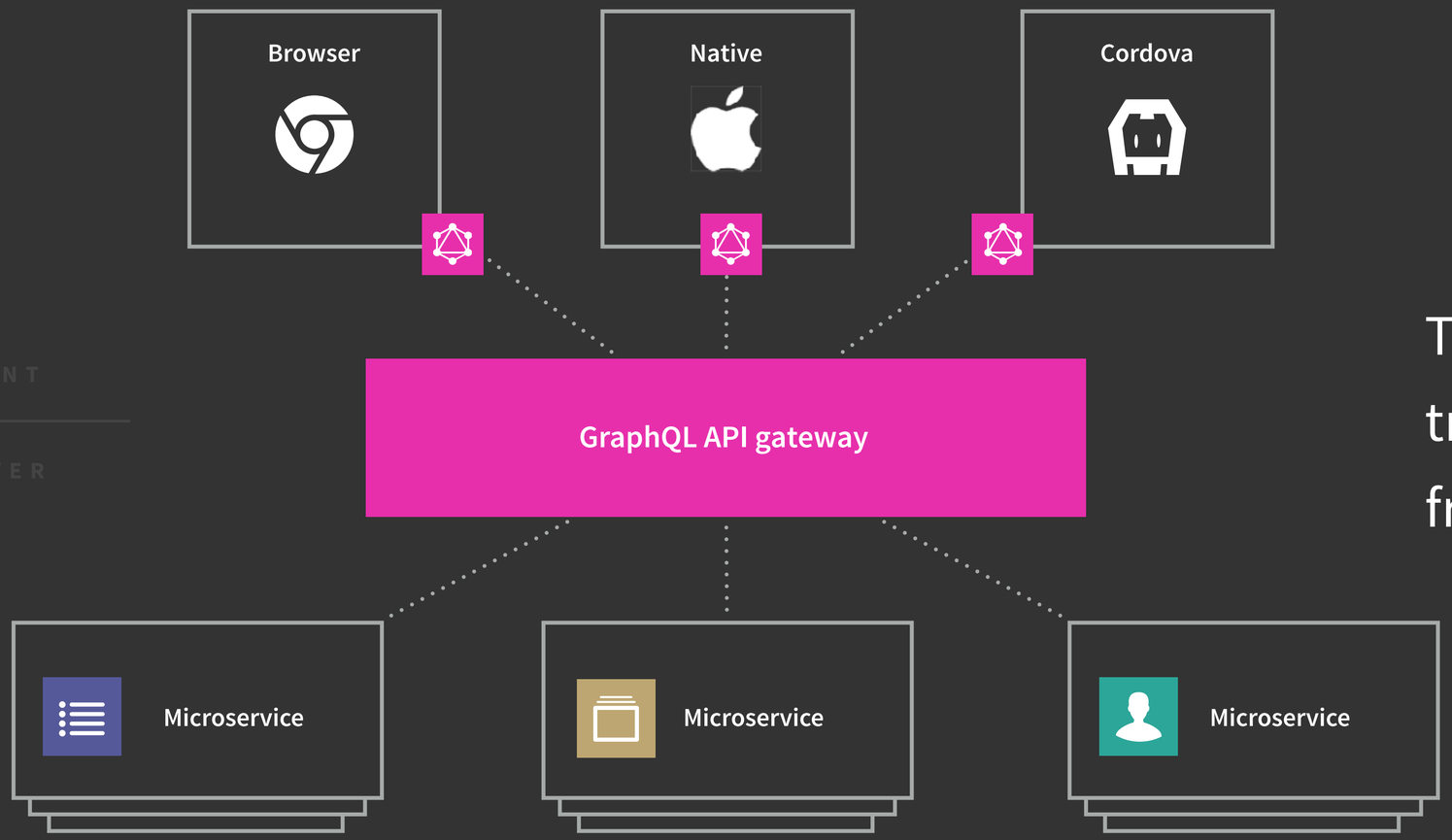
A BRIEF HISTORY OF APPS



Add an API gateway, now there's a development bottleneck

Backend team develops API		Frontend team develops API
API endpoints don’t meet UI needs	⟷	Too many API endpoints, one per UI feature
Takes too long to build the API for a new feature	⟷	Possible performance or security issues to ship faster

THE FUTURE OF APPS



The solution: GraphQL as the translation layer between frontend and backend

What’s GraphQL?

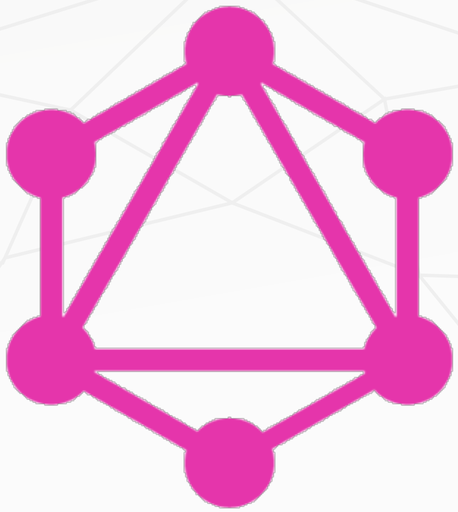
It’s an API specification that is quickly replacing REST for how people communicate between their backends and frontends.

```
type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post]
}

type Post {
  id: Int!
  title: String
  author: Author
  votes: Int
}

type Query {
  posts: [Post]
  author(id: ID!): Author
}
```

1. Define a schema for your data

```
1  query PostsForAuthor {
2    author(id: "1") {
3      firstName
4      posts {
5        title
6        vot
7      } votes
8    }
9  }
10
11
```

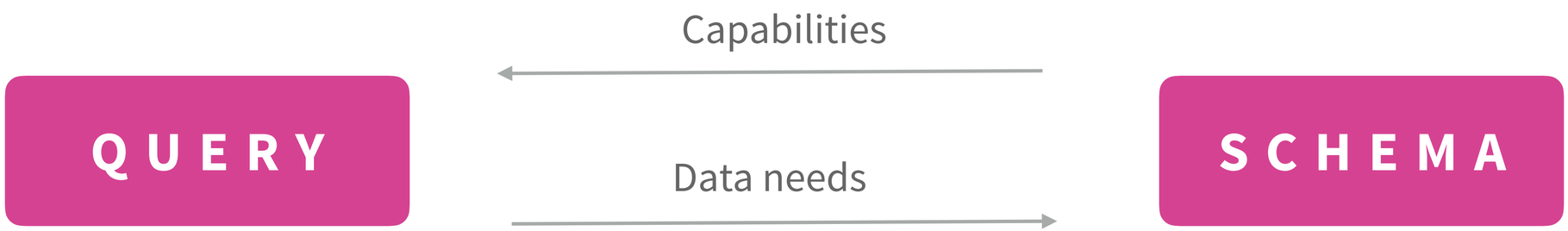
2. Write a query to get the data you need



Let clients specify their own **data**

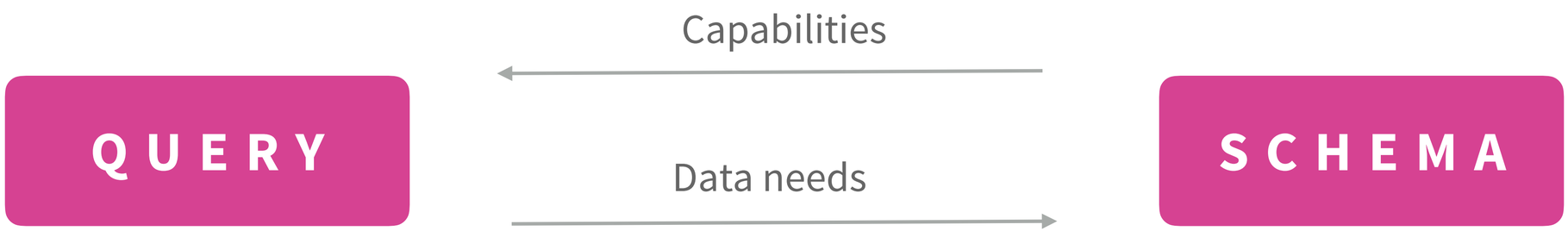
needs against the **capabilities**

exposed by the server




```
query {
  posts {
    title
  }
}
```

```
type Post {
  id: Int!
  title: String
  author: Author
  votes: Int
}
```



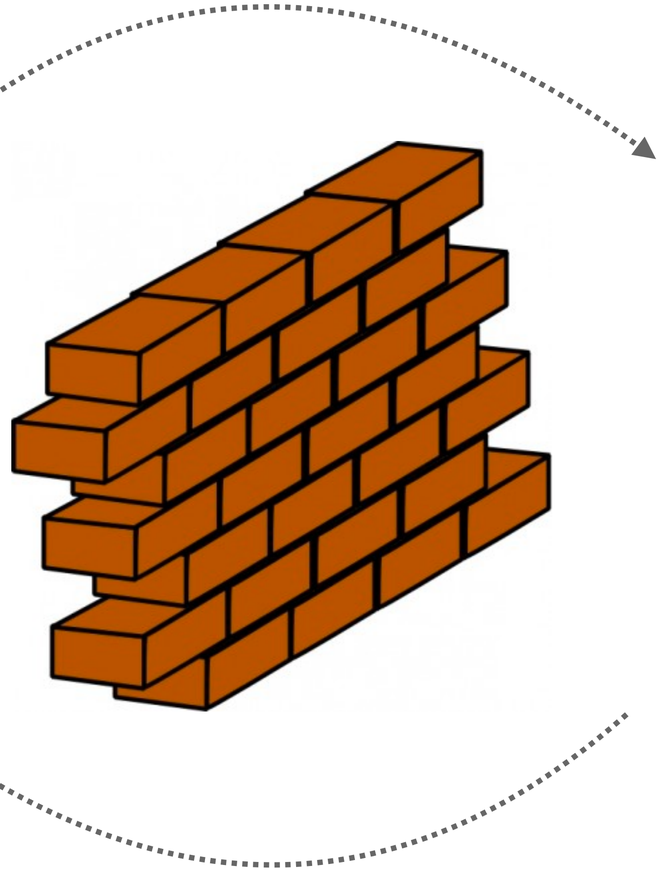

```
query {
  posts {
    title
    author {
      firstName
    }
  }
}
```

```
type Post {
  id: Int!
  title: String
  author: Author
  votes: Int
}

type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post]
}
```

WITH REST

Think carefully about API needs

User Interface



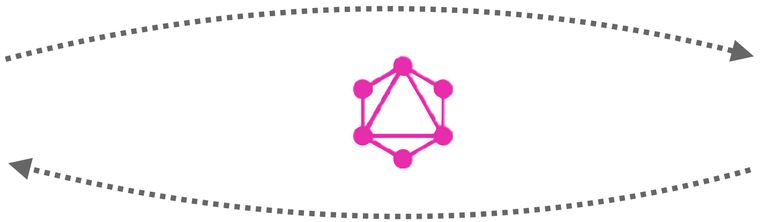
Backend

Hardcode special endpoints

WITH GRAPHQL

User Interface

Ask for data

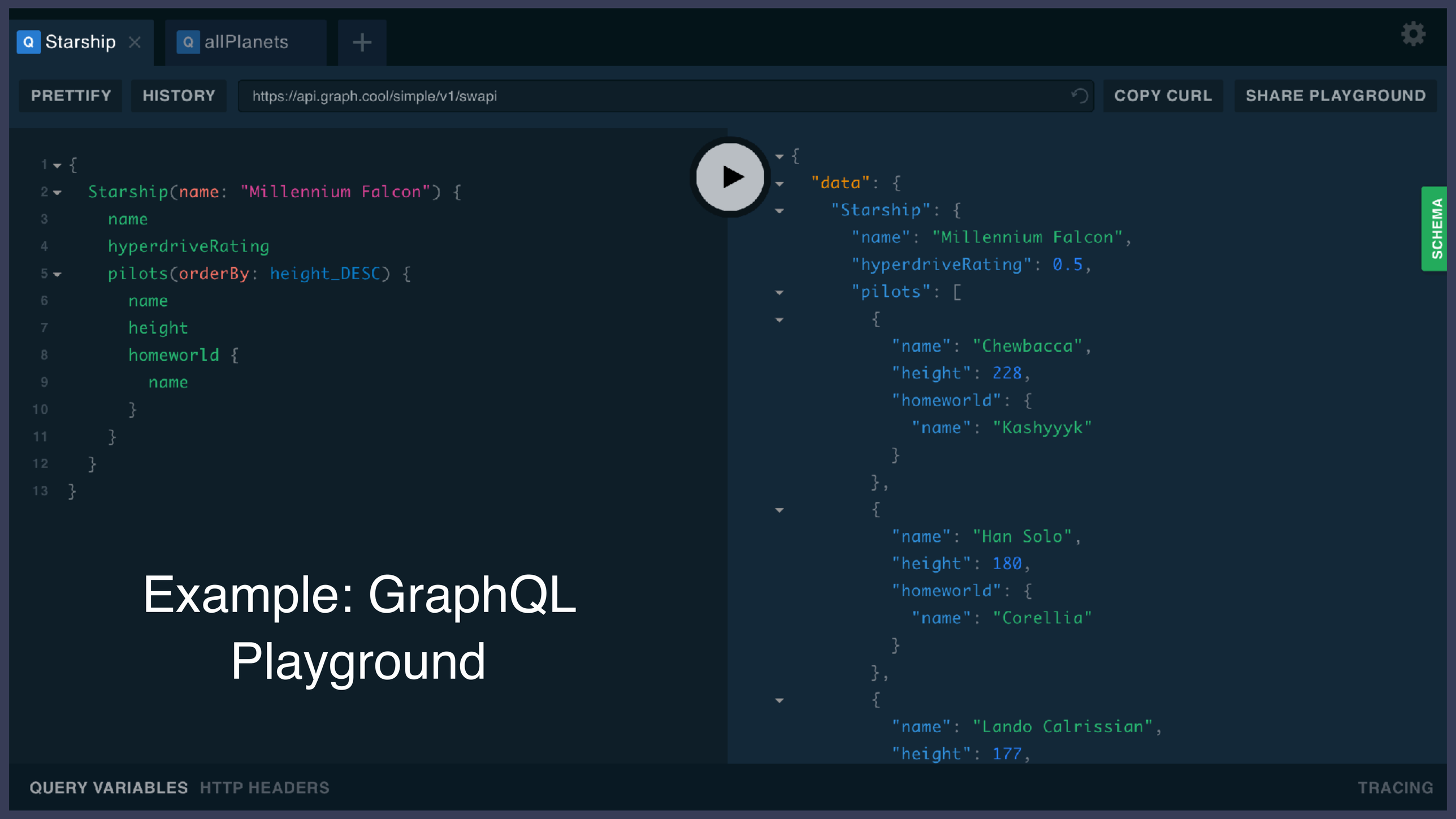


Get the data

Backend

A better API experience

GraphQL has been designed
from the start with
developer experience in mind



Example: GraphQL Playground

GraphQL tooling that extends across the entire stack

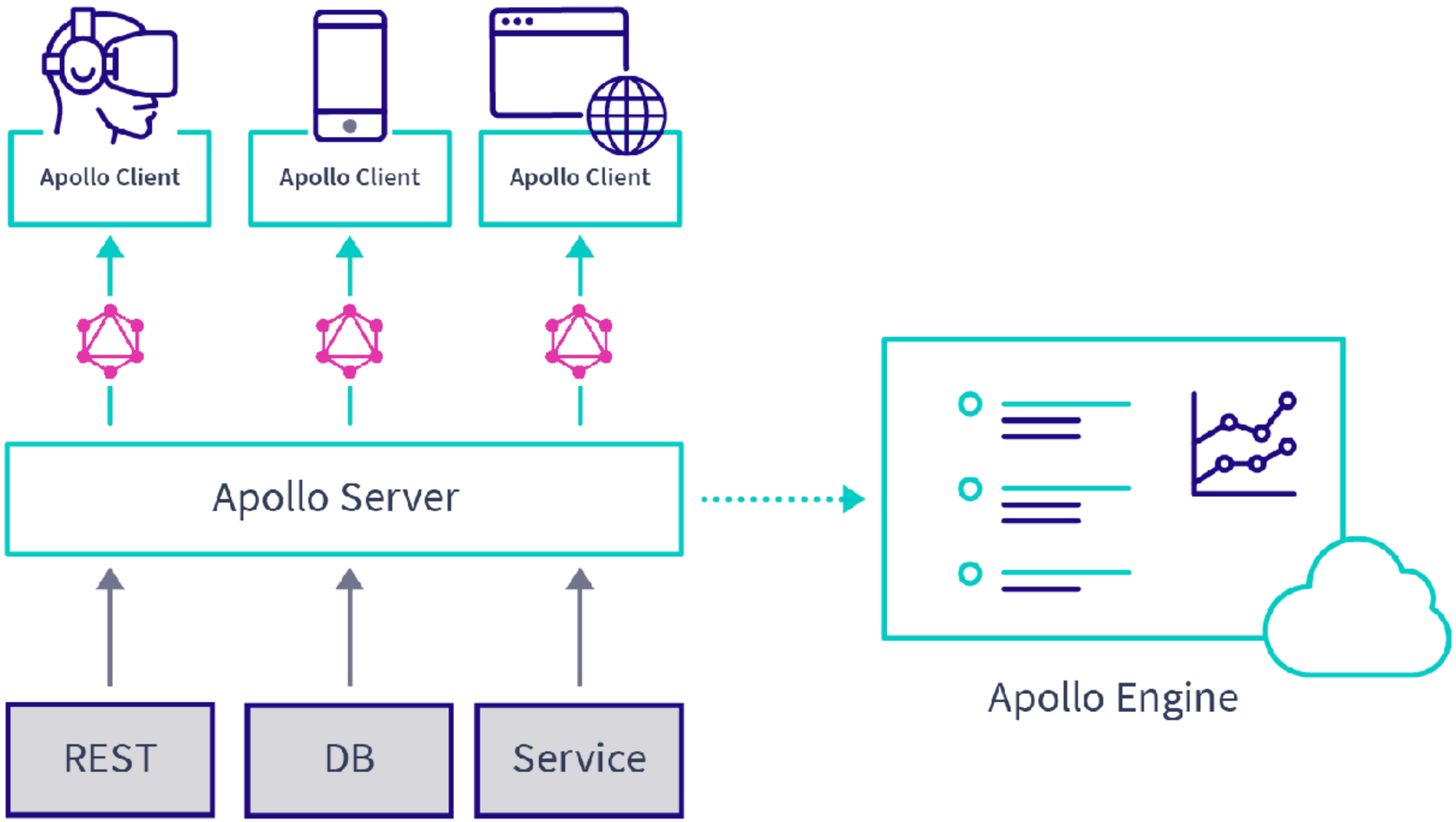
What's Apollo?

Apollo is GraphQL tooling that extends across the whole application stack.

Apollo Server for translating your backends into GraphQL

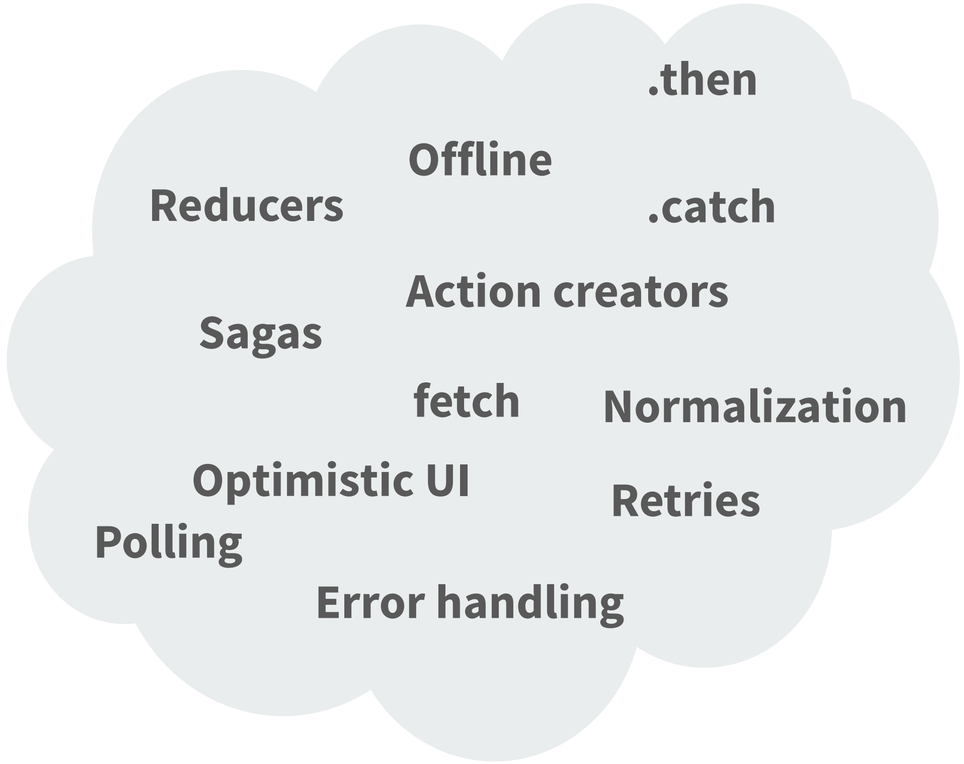
Apollo Client to connect GraphQL data to your UI

Apollo Engine to get insights about your app and connect to other services



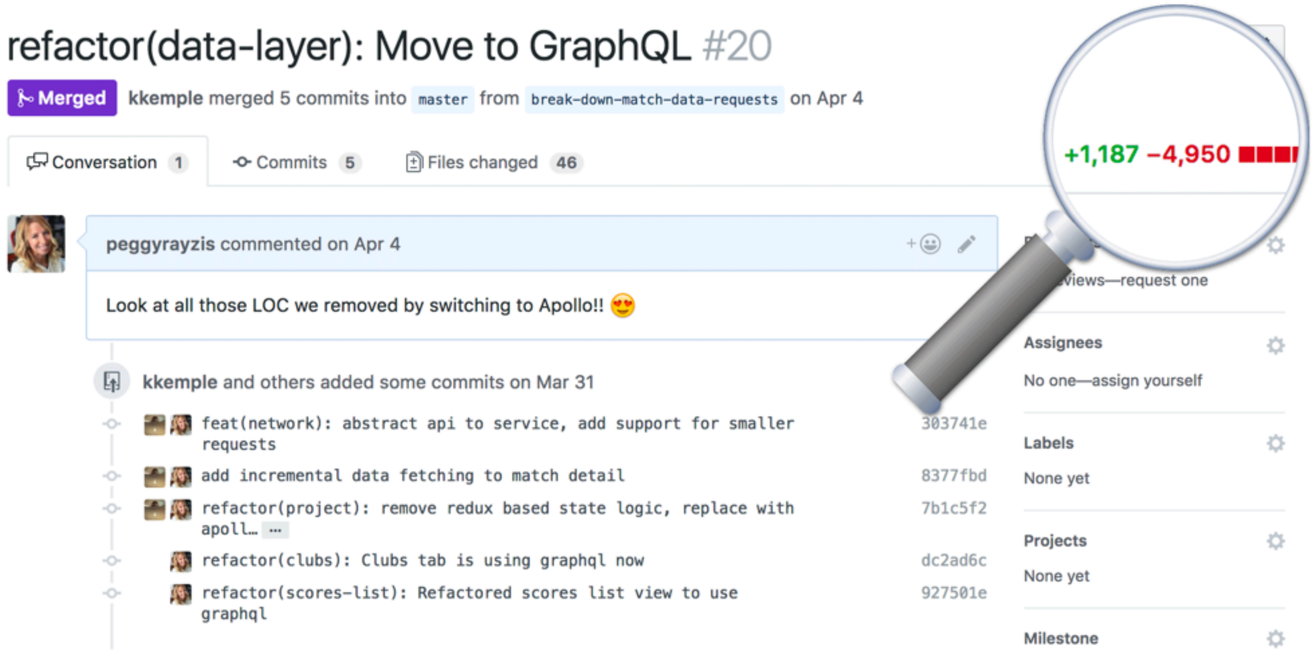
Apollo makes writing product code dramatically simpler.

Instead of manipulating data on the client, you get it in just the shape that you need right away.




```
const GET_DOGS = gql`
  {
    dogs {
      id
      breed
      displayImage
    }
  }
`;

const Dogs = () => (
  <Query query={GET_DOGS} pollInterval={1000}>
    {({ loading, data, error, fetchMore }) => {
      if (loading) return null;
      if (error) return 'Error!';

      return data.dogs.map(dog =>
        <Dog key={dog.id} { ...dog} />
      );
    }}
  </Query>
);
```

Replace complicated data management code!



Mark Johnson Follow
Web designer, developer, and teacher. Working at the cross-section of learning and technology. Co-Founder, CTO of Pathwright. Launcher of side projects.
Mar 9 · 4 min read

How GraphQL Replaces Redux

"What?!", you say. "GraphQL is a server side query language. Redux is a client-side state management library. How could one replace the other?" Good question.

Hold onto your butts, because I'm about to answer it.

Switching to React

First, a little back-story. Back in 2016, the front-end team at Pathwright began switching our client-side code from a Backbone & Marionette stack to React. The declarative model for UI made much more sense than the MVC patterns we'd been dealing with.

It was a breath of fresh air and still is to a large degree.

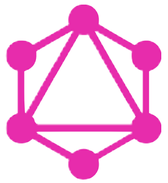
Reducing our Redux code with React Apollo

Switching to a declarative approach at Major League Soccer

Top highlight

I'm a firm believer that the best code is no code. More code often leads to more bugs and more time spent maintaining it. At Major League Soccer, we're a very small team, so we take this principle to heart. We try to optimize where we can, either through maximizing code reuse or lightening our maintenance burden.

Benefits of Apollo + GraphQL: Improved developer productivity



Common data loading patterns built in

Pagination

Prefetching

Server-side rendering

Prefetching

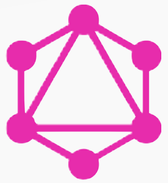
Error handling

Optimistic UI

Offline persistence

Deduplication

Retries



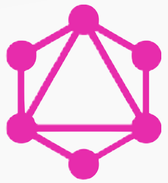
Static query analysis inside your code

```
30        currentUser: {
31          query: gql`
32            query getUserData ($id: String!) {
33              invalidQuery(id: $id) {
34                emails {
35                  address
36                  verified
37                }
38                randomString
39              }
40            }
41          `,
42          variables: {
43            id: ownProps.userId,
44          },
45        },
```

Without running the code, find:

- Typos in fields
- Wrong arguments
- Deprecated fields
- Identify field usage

apollostack/eslint-plugin-graphql



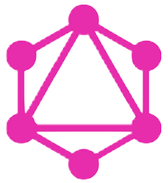
Typed code generation: Swift, Java, TS, Flow

```
query HeroName($episode: Episode) {
  hero(episode: $episode) {
    ...DescribeHero
  }
}

fragment DescribeHero on Character {
  name
  appearsIn
}
```

The above query combined with schema information outputs the type definitions on the right.

apollostack/apollo-codegen

```
// The episodes in the Star Wars trilogy
export type Episode =
  "NEWHOPE" | // Star Wars Episode IV: A New Hope, released in 1977.
  "EMPIRE" | // Star Wars Episode V: The Empire Strikes Back, released in 1980.
  "JEDI"; // Star Wars Episode VI: Return of the Jedi, released in 1983.


export interface HeroNameQueryVariables {
  episode: Episode | null;
}

export interface HeroNameQuery {
  hero: DescribeHeroFragment;
}

export interface DescribeHeroFragment {
  name: string;
  appearsIn: Array< Episode | null >;
}
```



Detailed GraphQL execution tracing

```
query FeedSimple {
  feed(limit: 0, type: NEW) {
    postedBy {
      login
    }
    repository {
      name
      owner {
        login
      }
    }
  }
}
```

Resolvers	Timing	TTL
⌄	1.18s	
⌄ feed:[Entry]	11.6ms	1 min
› feed.0	291ms	
› feed.1	290ms	
› feed.2	276ms	
⌄ feed.3		
⌄ repository:Repository	1.16s	4 min
› owner:User	157µs	4 min
name:String!	15.6µs	
› postedBy:User	274ms	4 min
› feed.4	289ms	

engine-prod

364.5 Requests per minute **318ms** p95 service time **0%** Error percentage

ON
Daily Reports

PERFORMANCE ERRORS SCHEMA ALERTS

TYPE

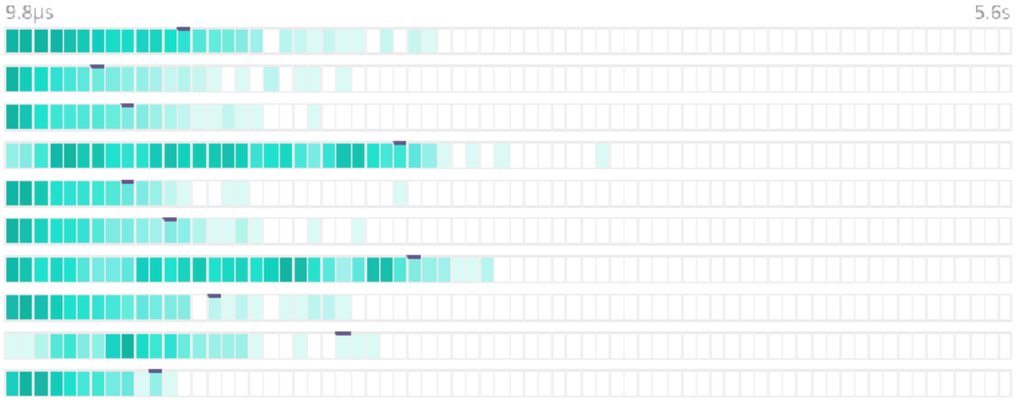
Request Rate •••
Range: 0.017rpm - 523.8rpm

Field Service Time Distribution
Range: 9.8µs - 5.6s
Percentile Marker
p99

UsageStats

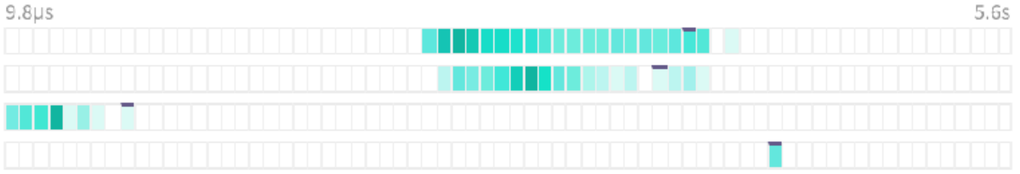
uncachedLatencyHistogram: DurationHistogram!	523.8rpm
queryById: QueryStats	196rpm
totalLatencyHistogram: DurationHistogram!	63.7rpm
queries: [QueryStats!]!	57.7rpm
cacheTTLHistogram: DurationHistogram!	51.9rpm
requestsWithErrorsCount: Long!	37.8rpm
uniqueFieldsCount: Int!	25.9rpm
cacheLatencyHistogram: DurationHistogram!	21.2rpm
pathErrorStats: [PathErrorStats!]!	4.3rpm
__typename: String!	3.1rpm

Load more fields (1 hidden)



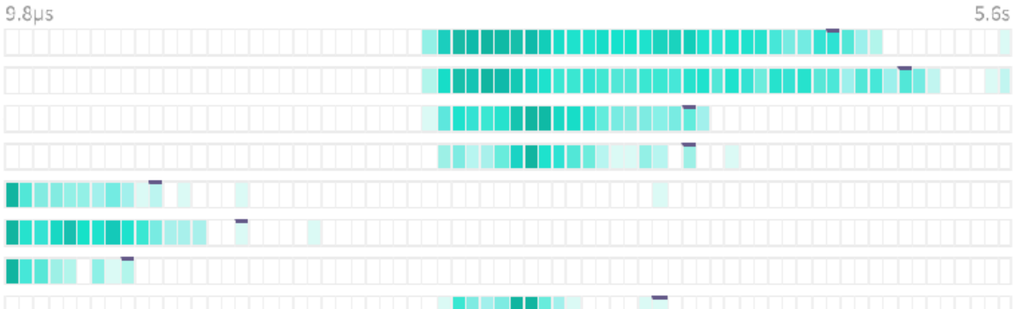
QueryRoot

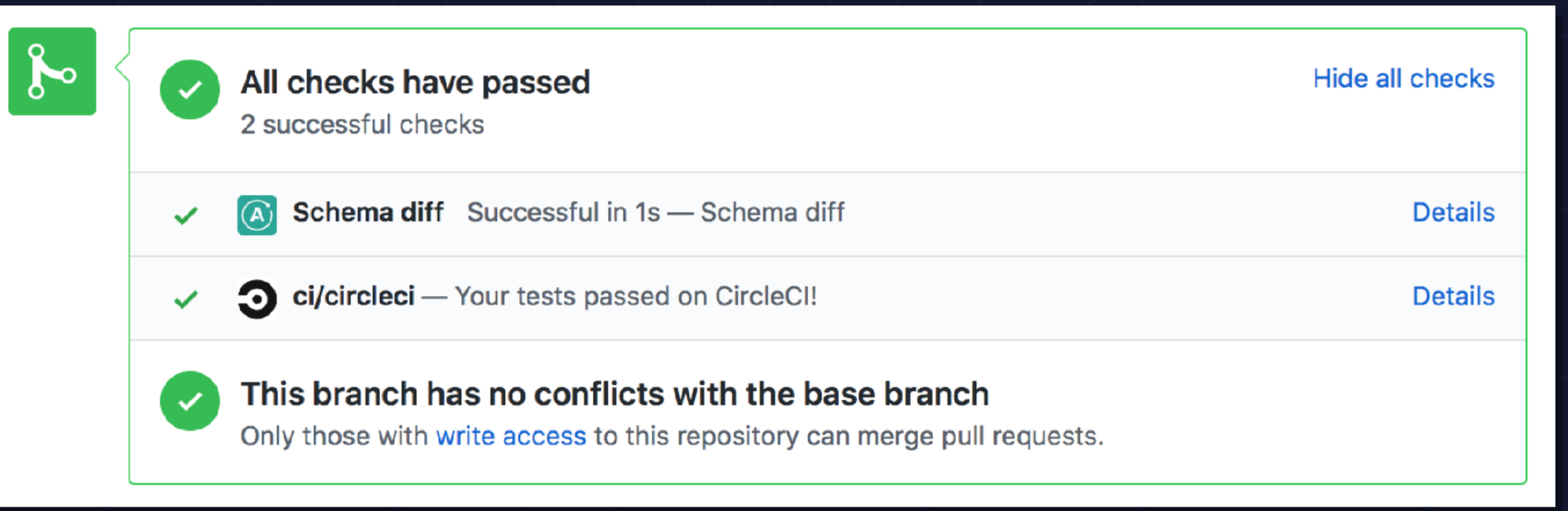
service: Service	350rpm
account: Account	13rpm
me: Identity	0.483rpm
allUsers: [User!]	0.017rpm

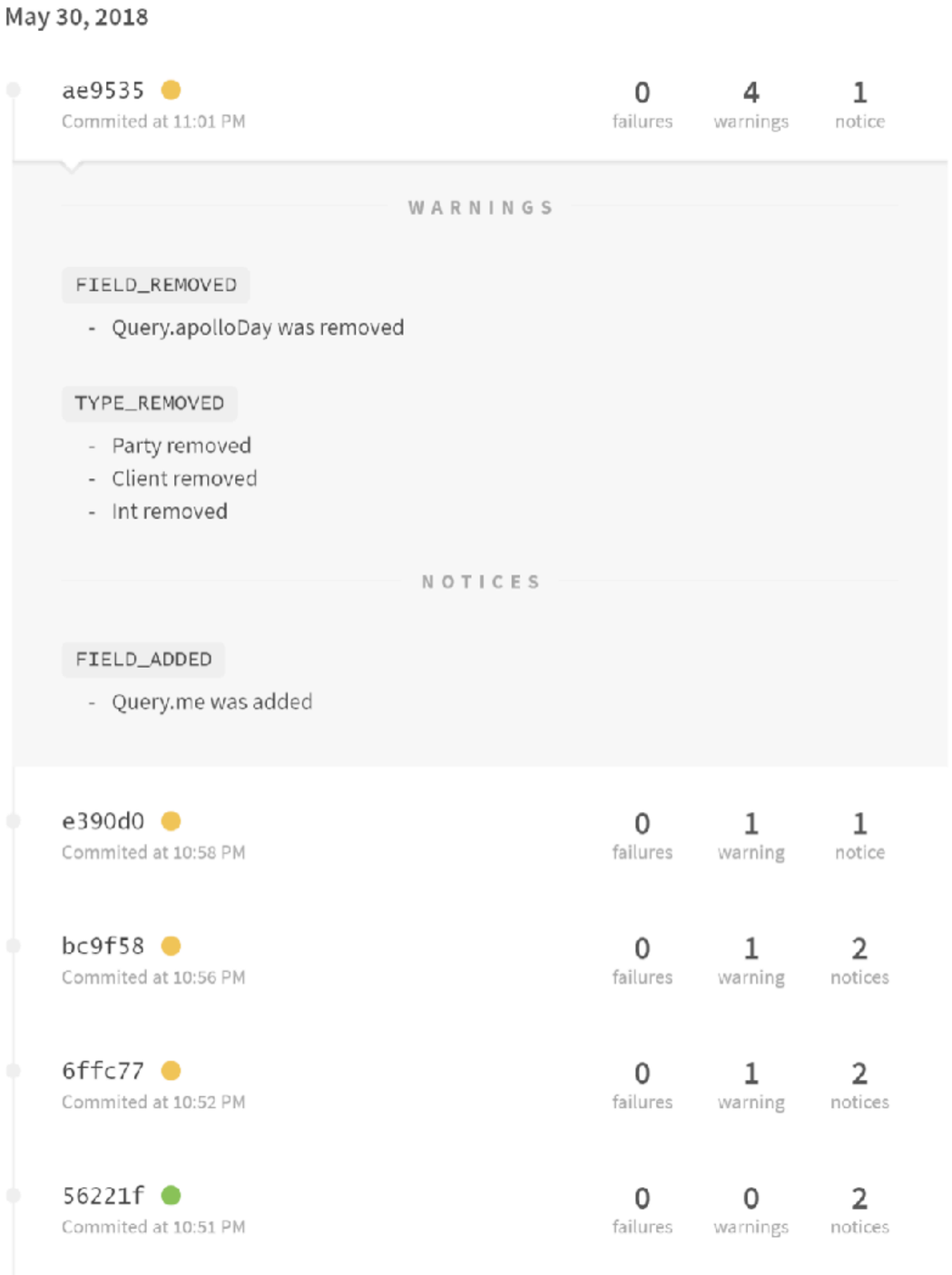


Service

report: ServiceReport	231.7rpm
trends: ServiceTrends	81.9rpm
traceIDsPerBucket: PerBucketTraceIDs!	28.9rpm
trace: Trace	8.2rpm
apiKeys: [ApiKey!]	7.2rpm
id: ID!	2.9rpm
scheduledSummaries: [ScheduledSummary!]!	1.9rpm




All checks have passed
2 successful checks
Hide all checks
Schema diff Successful in 1s — Schema diff
Details
ci/circleci — Your tests passed on CircleCI!
Details
This branch has no conflicts with the base branch
Only those with write access to this repository can merge pull requests.



Schema version tracking

- Understand how your schema is changing over time
- Compare schema changes against actual production data
- Easily review updates
- Available now in Apollo Engine


GitHub
The New York Times
AMERICAN EXPRESS
CNN
EXPRESS
intuit
airbnb
OpenTable
ticketmaster
yelp
twitch
coursera
Product Hunt
Pinterest
shopify
Expo
credit karma
heroku
UDACITY
@WalmartLabs

Adding GraphQL to your stack

GraphQL + Apollo can be added incrementally.

To get the main benefits for product development velocity we talked about above, you don’t need to rethink your architecture.

You can get a demo working in just a few hours, and something running in production in days.

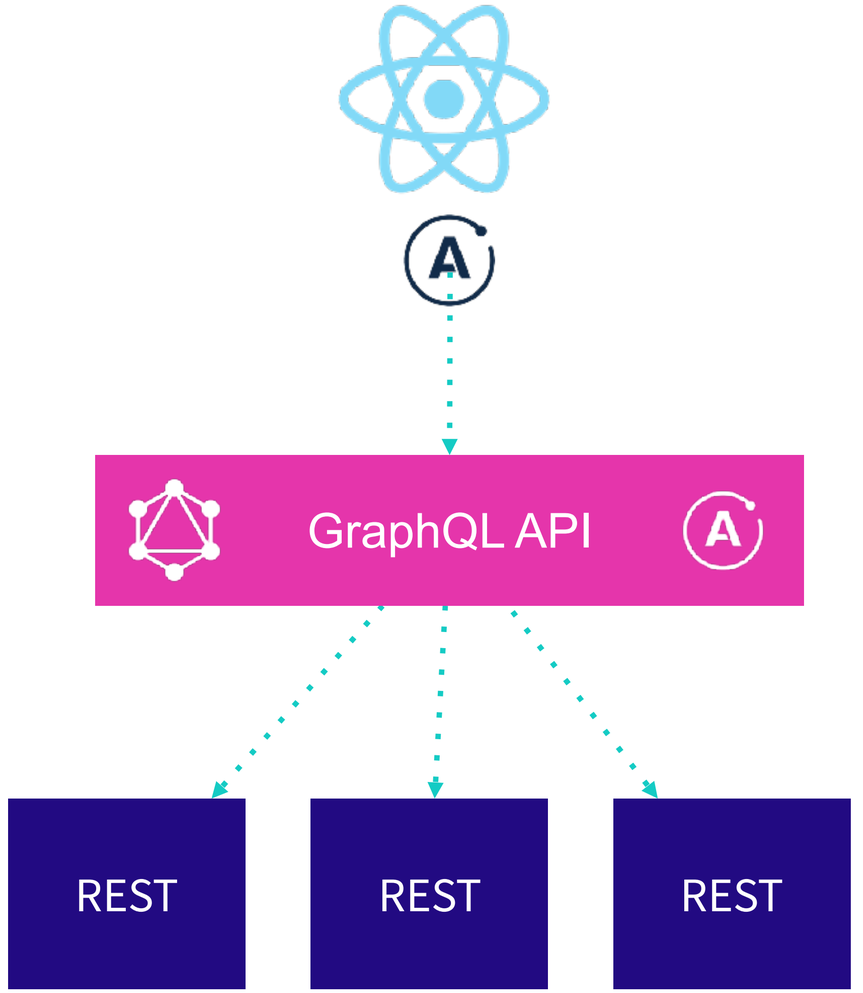
Put GraphQL over REST.

Put a GraphQL layer in between your existing REST backends and your React app.

Get all of the benefits of a GraphQL architecture without modifying your backends.

From the point of view of the REST API, the same calls will be made as before.

But now, the frontend doesn't have to format, filter, and manage the data anymore.



Same feature, better result, less work

A case study with sandwiches.

With REST

```
GET /api/sandwich/bread
GET /api/sandwich/meat
GET /api/sandwich/toppings
```

Filter down data to what you want

Do waterfall requests for related data and combine everything together yourself

With GraphQL

```
query {
  sandwich {
    bread { baguette }
    meat { turkey }
    toppings { avocado, tomato }
  }
}
```

- Same REST API calls
- No roundtrips from the client
- No manual data reformatting and management
- Only get the fields you need - reduce bandwidth usage and increase performance

Applause from you, Oleg Ilyenko, and 44 others

sachee
software engineer @medium. pokemon gym leader. space witch. potato compatible.
Mar 21 · 4 min read

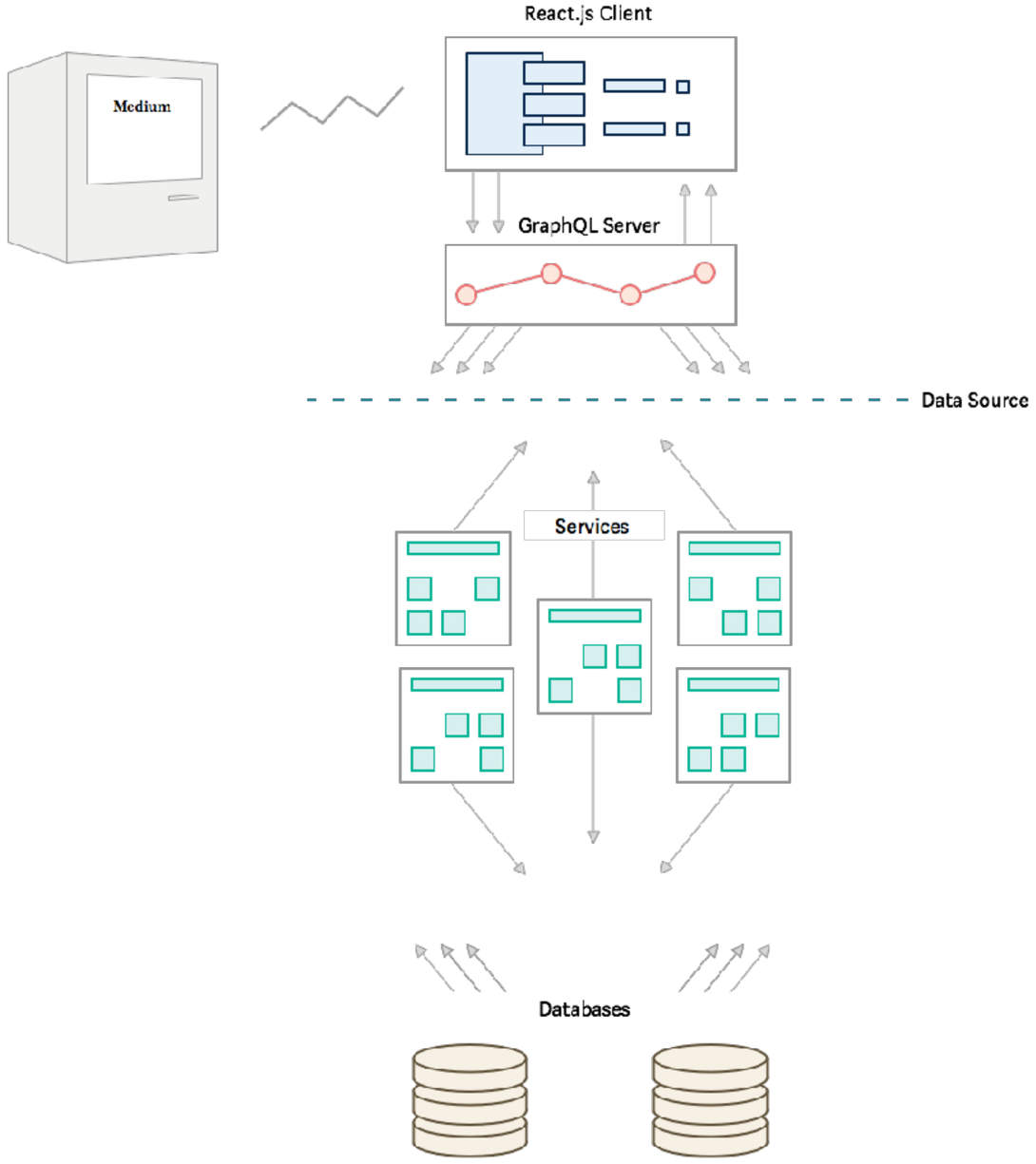
2 Fast 2 Furious: migrating Medium's codebase without slowing down

Five years ago, Medium was built using the latest tools and frameworks by people who had experience with those tools. It's time that we update these tools and frameworks.

However, migrating an entire system to new tools and frameworks isn't an easy task. And doing that while not impacting feature development? That's even harder.

So, how *would* you migrate off of your existing system, without hindering feature development, but also incrementally gain the benefits of the new system along the way?

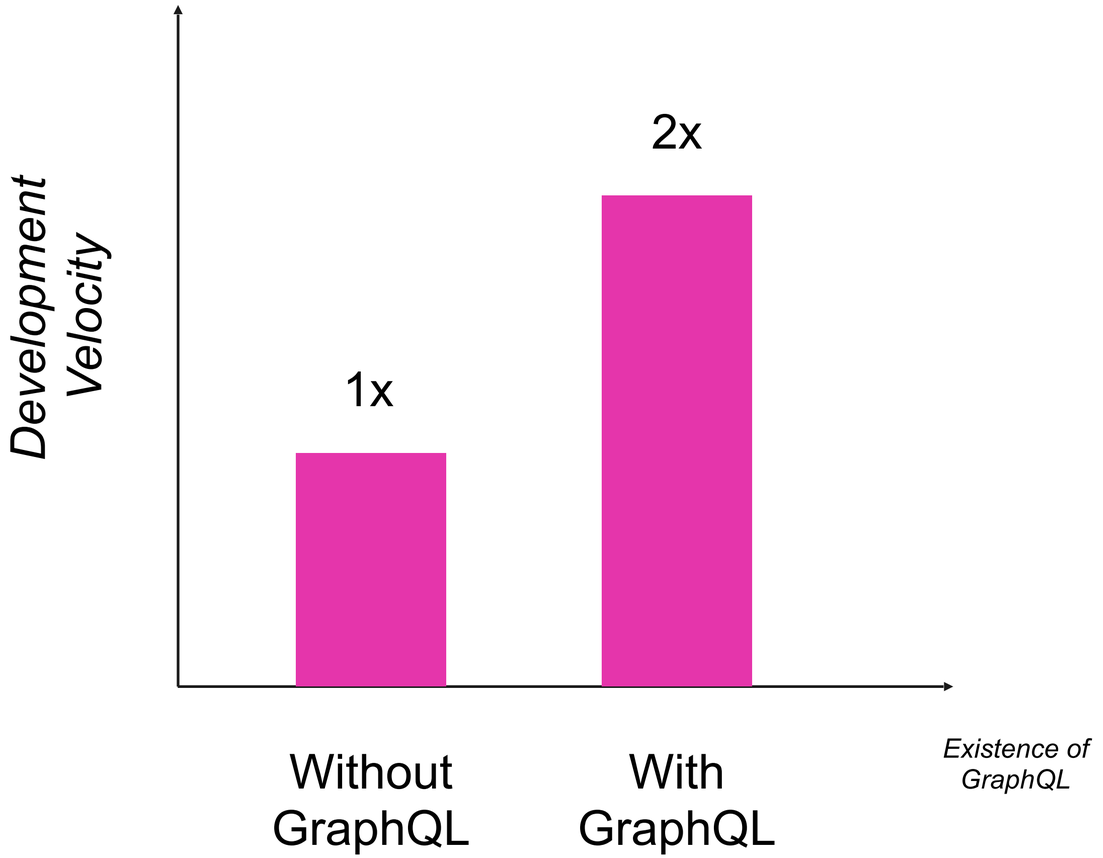
We began by testing out different frameworks and technologies and figuring out which ones deliver on the dimensions we care about.



GraphQL is changing the game.

We talk to companies using GraphQL in production every day, and some teams tell us they can ship features fully **twice as fast** as before.

That more than makes up for the effort of getting that new API up and running.



OK, so where do I start?

Answer: With the schema.

Describe the API you wish you had.

We've found that teams are *most* successful when the GraphQL API is designed primarily by the frontend developers that will use it.

Upcoming events

Justin Timberlake
@jtimberlake

Man of the Woods tour
April 29, 2018

Man of the Woods tour
June 14, 2018

Man of the Woods tour
July 30, 2018

Beyoncé
@beyonce

Coming up with part of our schema

Upcoming events

Justin Timberlake
@jtimberlake

Man of the Woods tour
April 29, 2018

Man of the Woods tour
June 14, 2018

Man of the Woods tour
July 30, 2018

Beyoncé
@beyonce

```
query UpcomingEvents {
  coolArtists {
    name
    twitterUrl
    events {
      image
      name
      startDateTime
    }
  }
}
```

```
const typeDefs = gql`
  type Query {
    coolArtists: [Artist]
  }

  type Artist {
    id: ID
    name: String
    image: String
    twitterUrl: String
    events: [Event]
  }

  type Event {
    name: String
    image: String
    startDateTime: String
  }
`;
```

Apollo Server 2.0
apollographql.com/docs

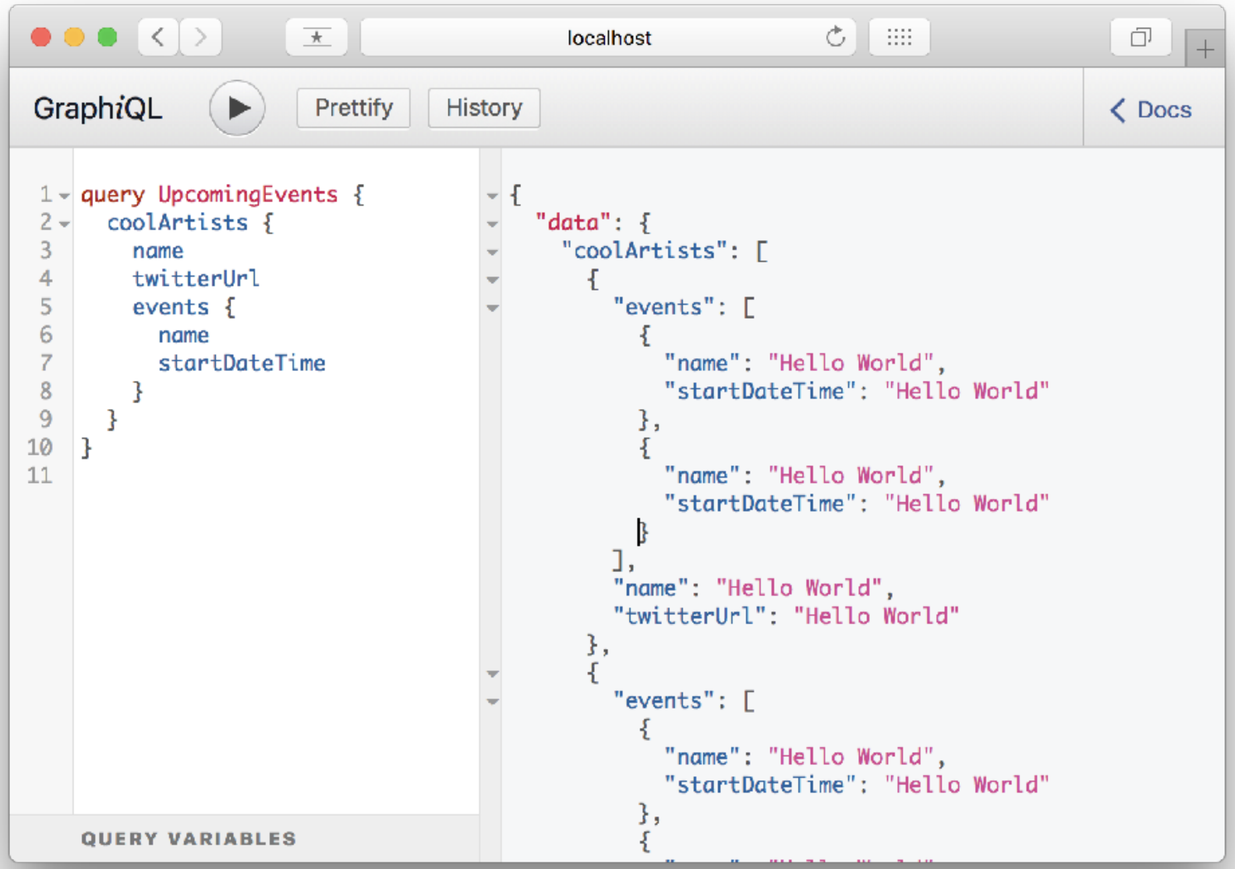
Apollo Server makes it easy to create a production-ready GraphQL schema using the schema language.

```
const server = new ApolloServer({
  typeDefs,
  resolvers
});

server.listen()
  .then(() => {
    console.log('Server ready! 🚀')
  })
```

```
// Construct a schema using the schema language
const typeDefs = gql`
  type Query {
    coolArtists: [Artist]
  }

  type Artist {
    id: ID
    name: String
    image: String
    twitterUrl: String
    events: [Event]
  }

  type Event {
    name: String
    image: String
    startDateTime: String
  }
`;
```

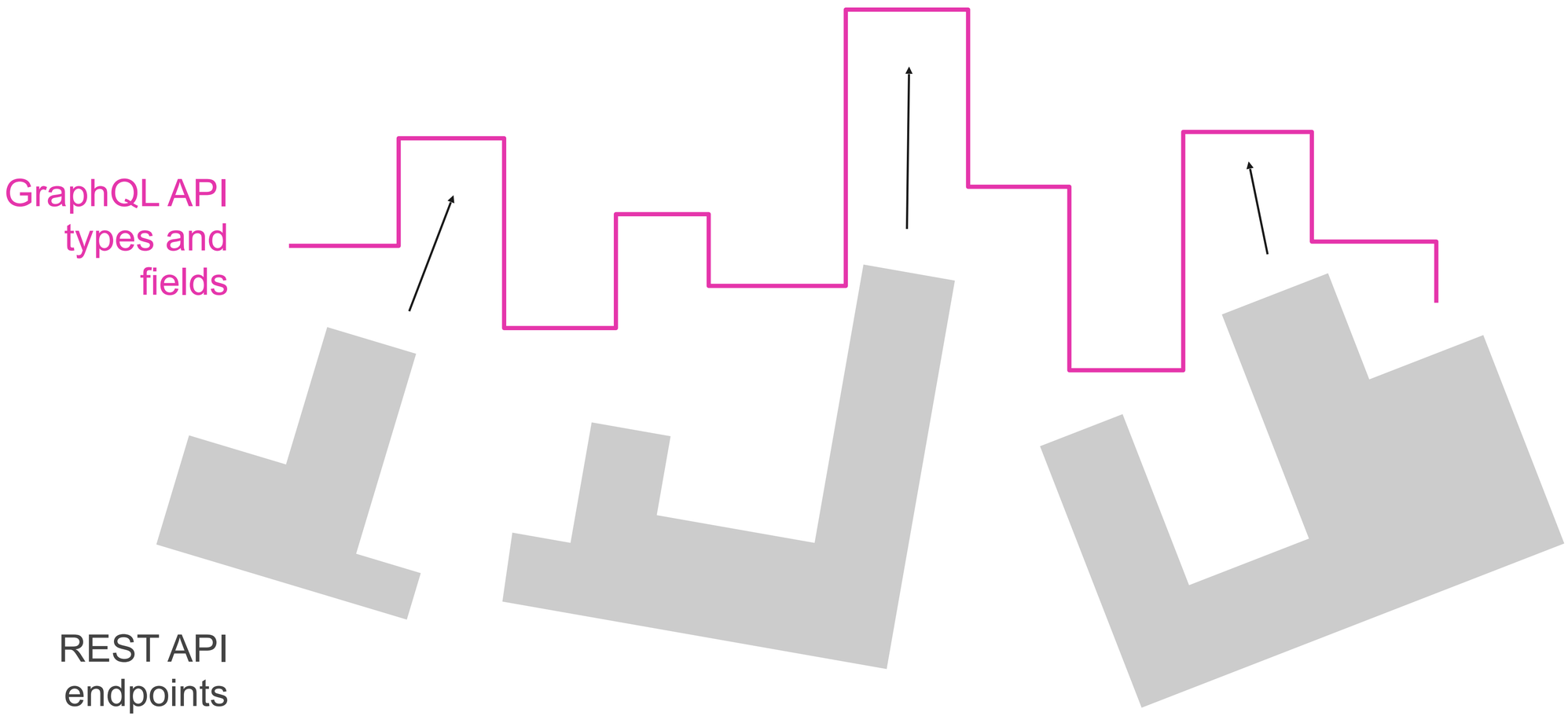
Apollo Server mocking

```
// Construct a schema using the schema language
const typeDefs = gql`
  type Query {
    coolArtists: [Artist]
  }

  type Artist {···
  }

  type Event {···
  }
`;

// Convert it to a GraphQL.js schema object
const schema = makeExecutableSchema({
  typeDefs
});

// Apply automatic mocks (can also be customized)
addMockFunctionsToSchema({ schema });
```



Now, fill in the data.

It’s just like filling in the blanks.



Every GraphQL schema field is a function.

You can think of a GraphQL schema as a web of tiny endpoints, and a query is a way of calling a bunch of them in one request.

The function that implements a field is called a *resolver*, and it can do basically anything.

Resolvers give GraphQL the flexibility of being a general-purpose API layer.

```
type Query {
  coolArtists: [Artist]
}
```

```
const resolvers = {
  Query: {
    coolArtists: () => {
      return [{
        name: "Justin Timberlake",
        id: "K8vZ91754g7"
      }, {
        name: "Beyoncé",
        id: "K8vZ9175rX7"
      }, {
        name: "Kansas",
        id: "K8vZ9171C-f"
      }];
    }
  },
```

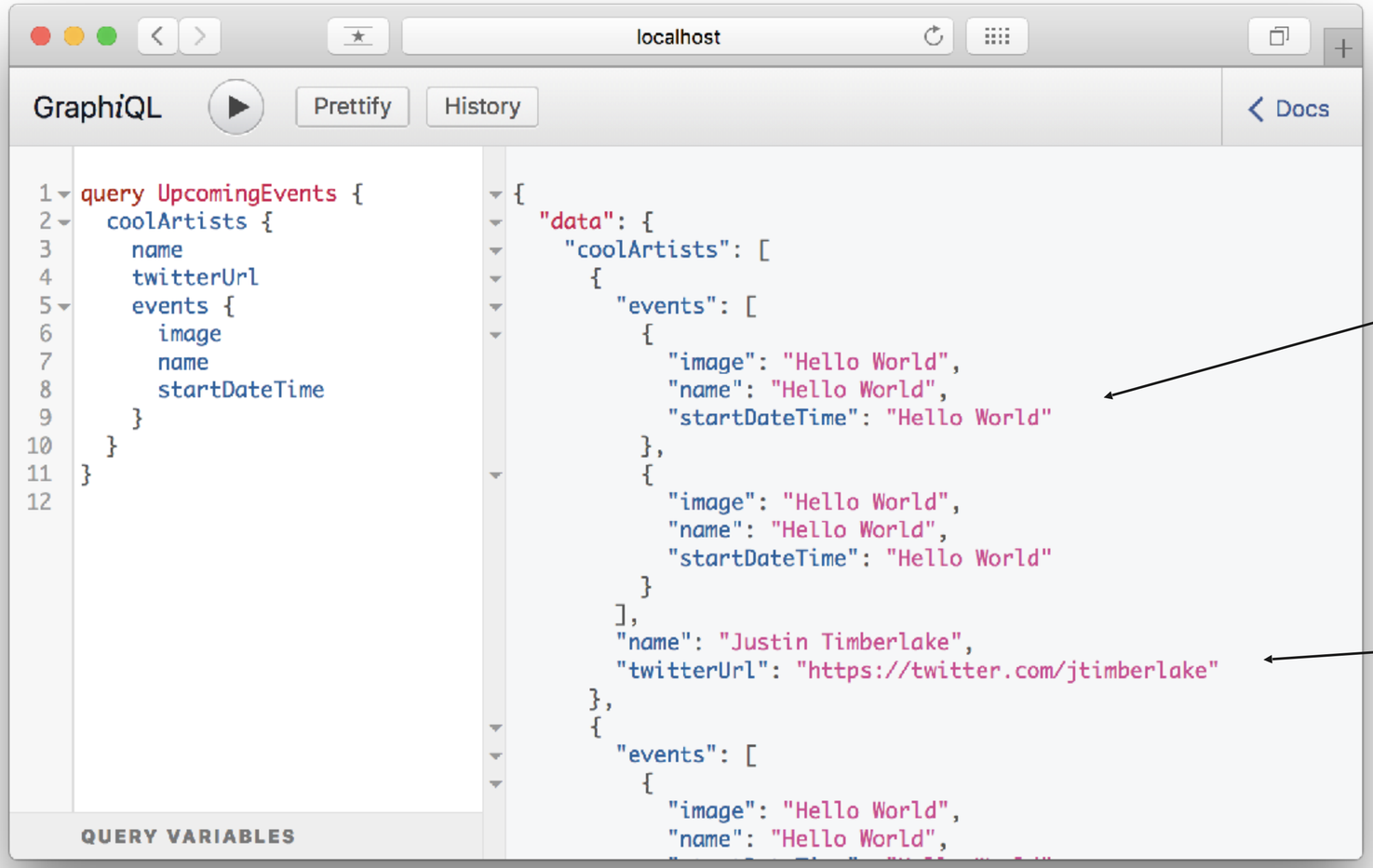
Your GraphQL resolvers can call your REST backend.

We can take the REST data fetches from the client, and move them into the server.

This organizes all our REST API calls in one place, removing data management complexity from the client.

```
const resolvers = {
  Query: {
    coolArtists: (root, args, context) => {
      return Promise.all(
        coolArtists.map(({ name, id }) => {
          return fetch(
            `${base}/attractions/${id}.json?apikey=${
              context.secrets.TM_API_KEY
            }`
          )
            .then(res => res.json())
            .then(data => {
              return Object.assign({ name, id }, data);
            });
        })
      );
    }
  },
```

Partially mocked data



Fake “Hello World” fields from the mock.

Real artist information from the API.

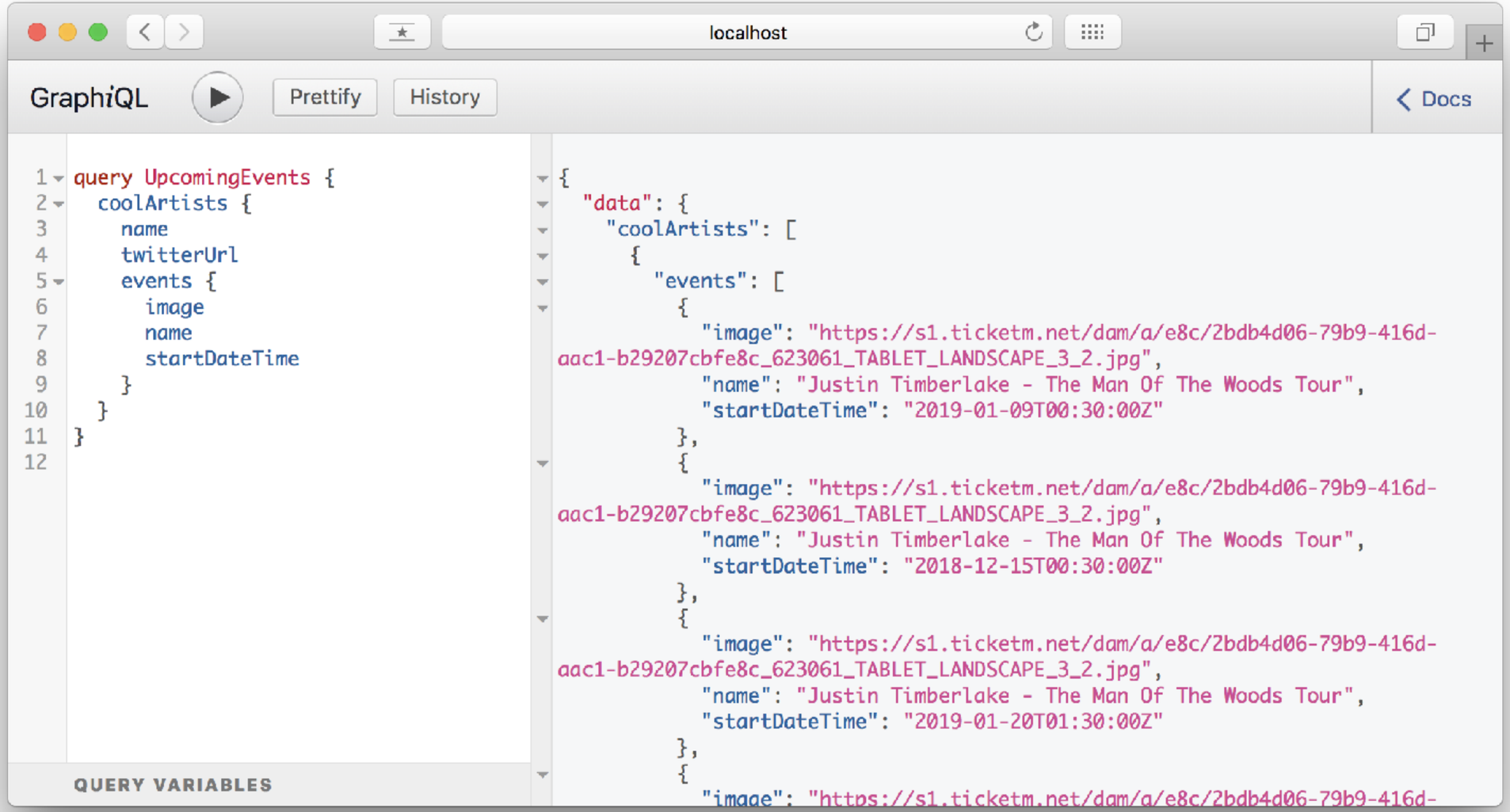
Let’s finish it up.

Let’s take a partially implemented schema, and add a resolver to call one more endpoint.

Now, with a single GraphQL query, we can retrieve data that used to take two endpoint fetches in a single request.

```
Artist: {
  events: (artist, args, context) => {
    return fetch(
      `${base}/events.json?size=10&apikey=${
        context.secrets.TM_API_KEY
      }&attractionId=${artist.id}`
    )
      .then(res => res.json())
      .then(data => {
        // Sometimes, there are no upcoming events
        return (data && data._embedded && data._embedded.events) || [];
      });
  },
```

Fully API-driven data



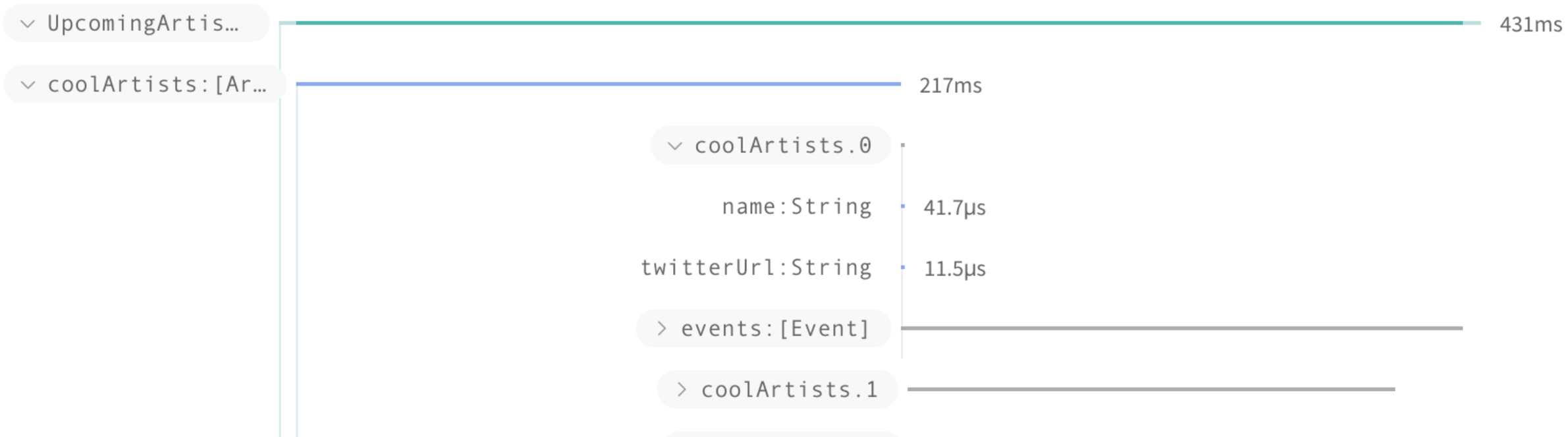
Visualizing execution with Apollo Tracing + Engine

Trace 5ab3d150.749b96a0630449b4
Sampled at 3/22/2018 8:52am

1 m
Operation TTL
PUBLIC

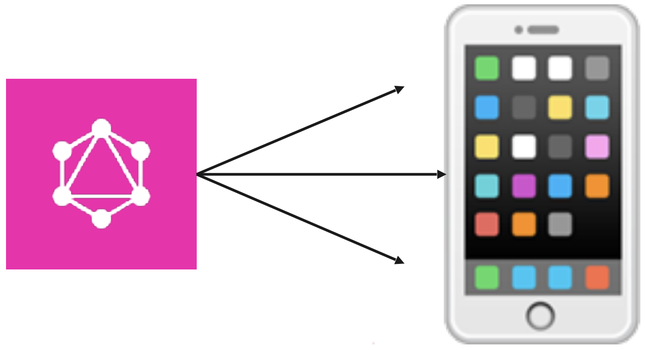
431.3 ms
Service time
0th Percentile

TRACE VARIABLES ERRORS CACHE

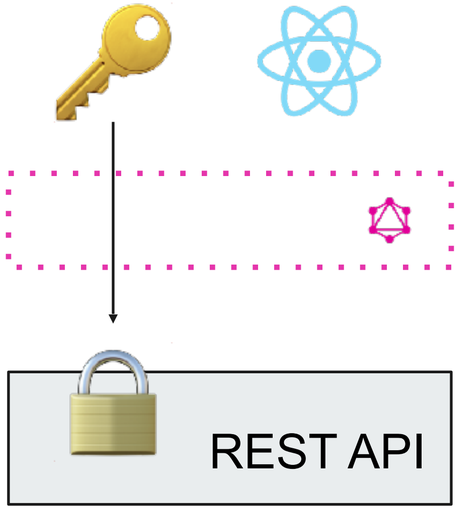


Tips when wrapping a REST API with GraphQL

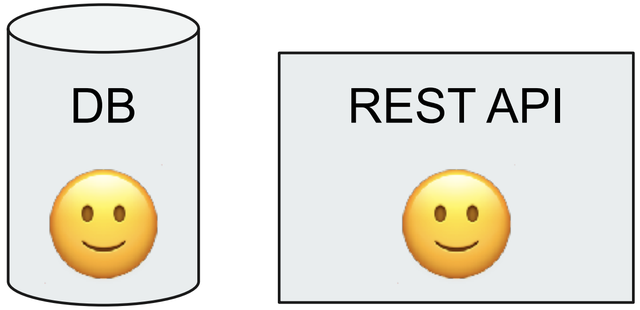
When you’re starting, build just enough data for one view of your app with one query.



If your REST API has permissions, just pass through your header from the GraphQL request to the underlying REST API.



The performance bar is that your new GraphQL API doesn’t make more REST calls than the equivalent view did in React.



Up next for Apollo

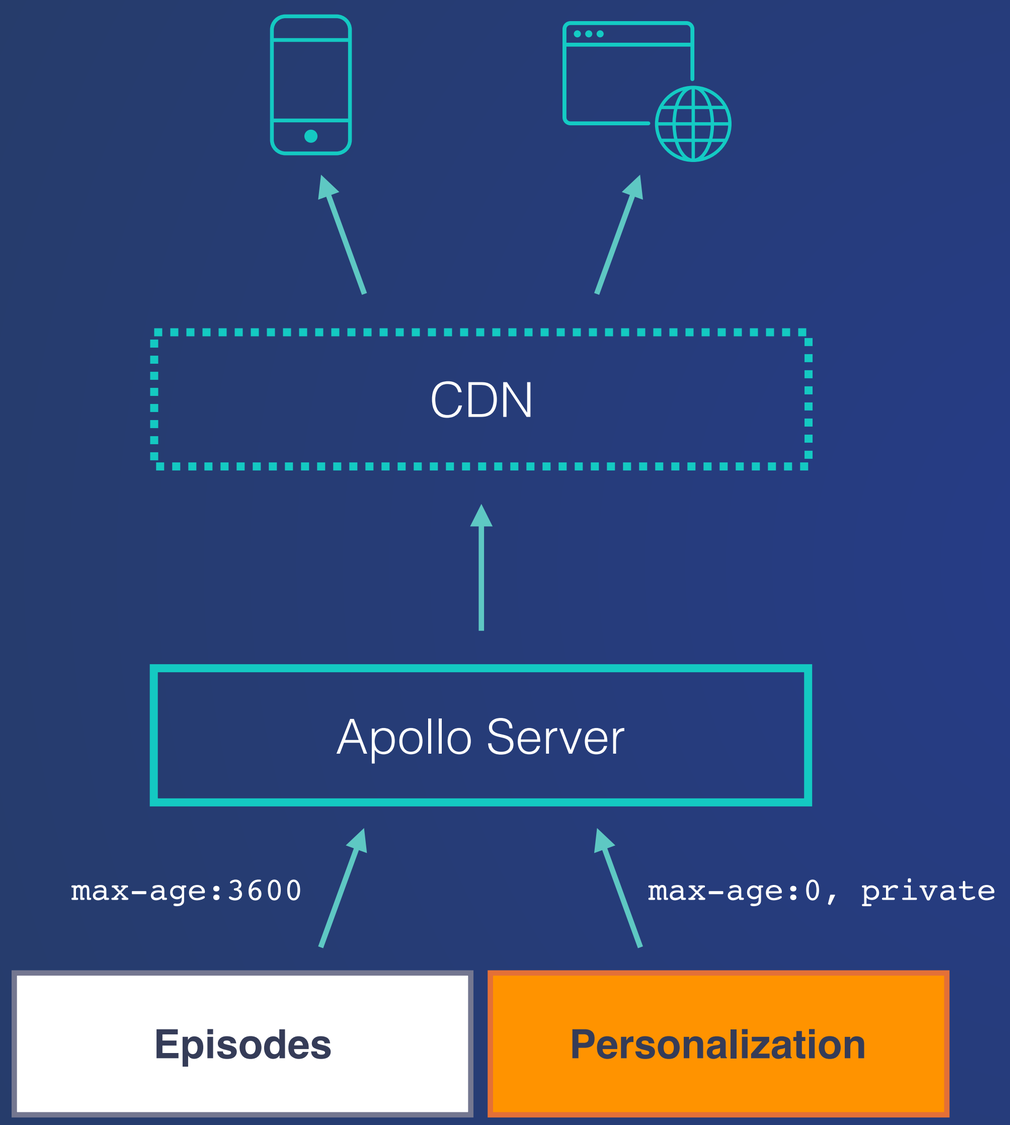
Data sources

A best pattern for fetching from REST endpoints

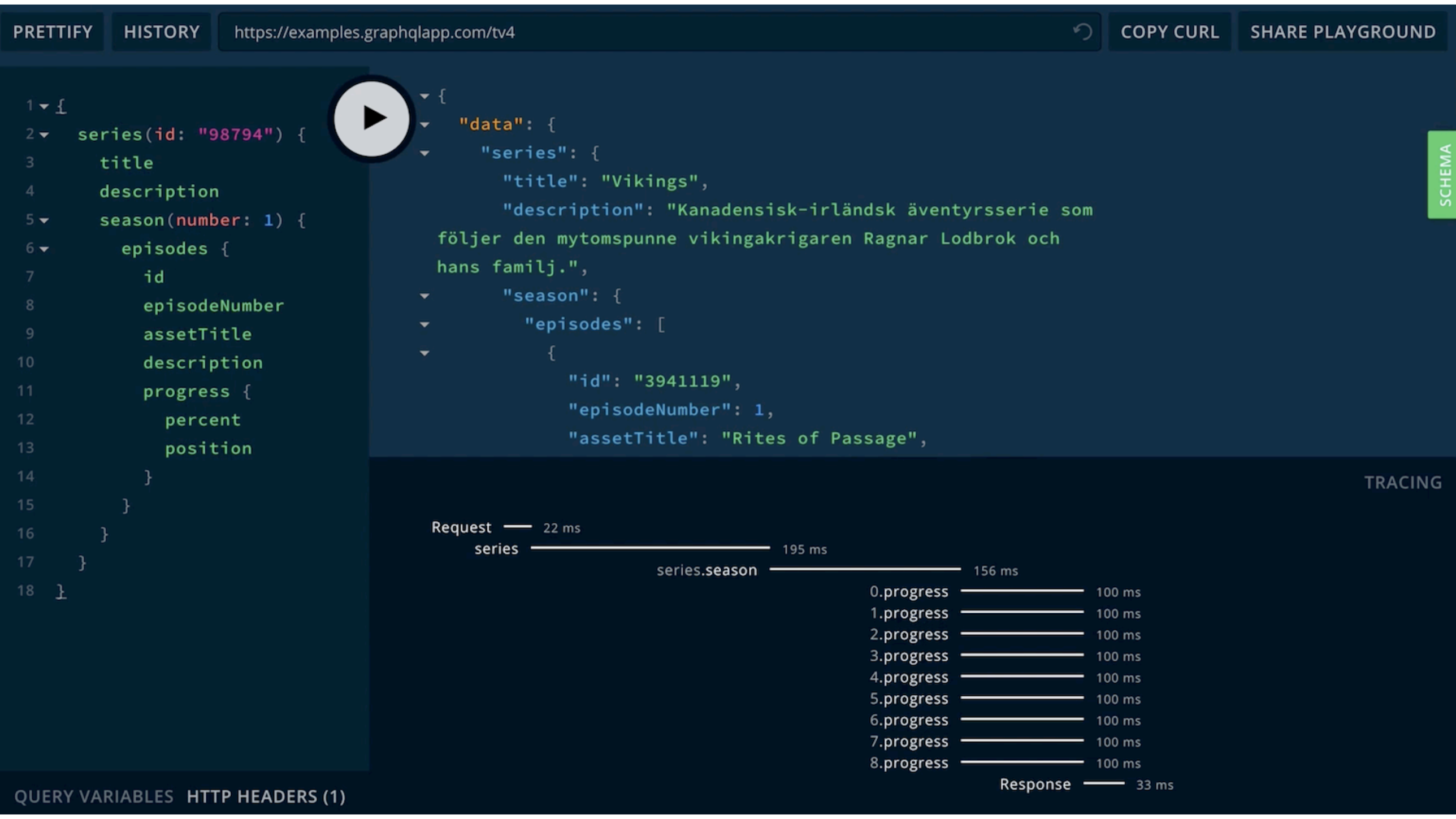
Data sources: Integrate REST endpoints with Apollo Server

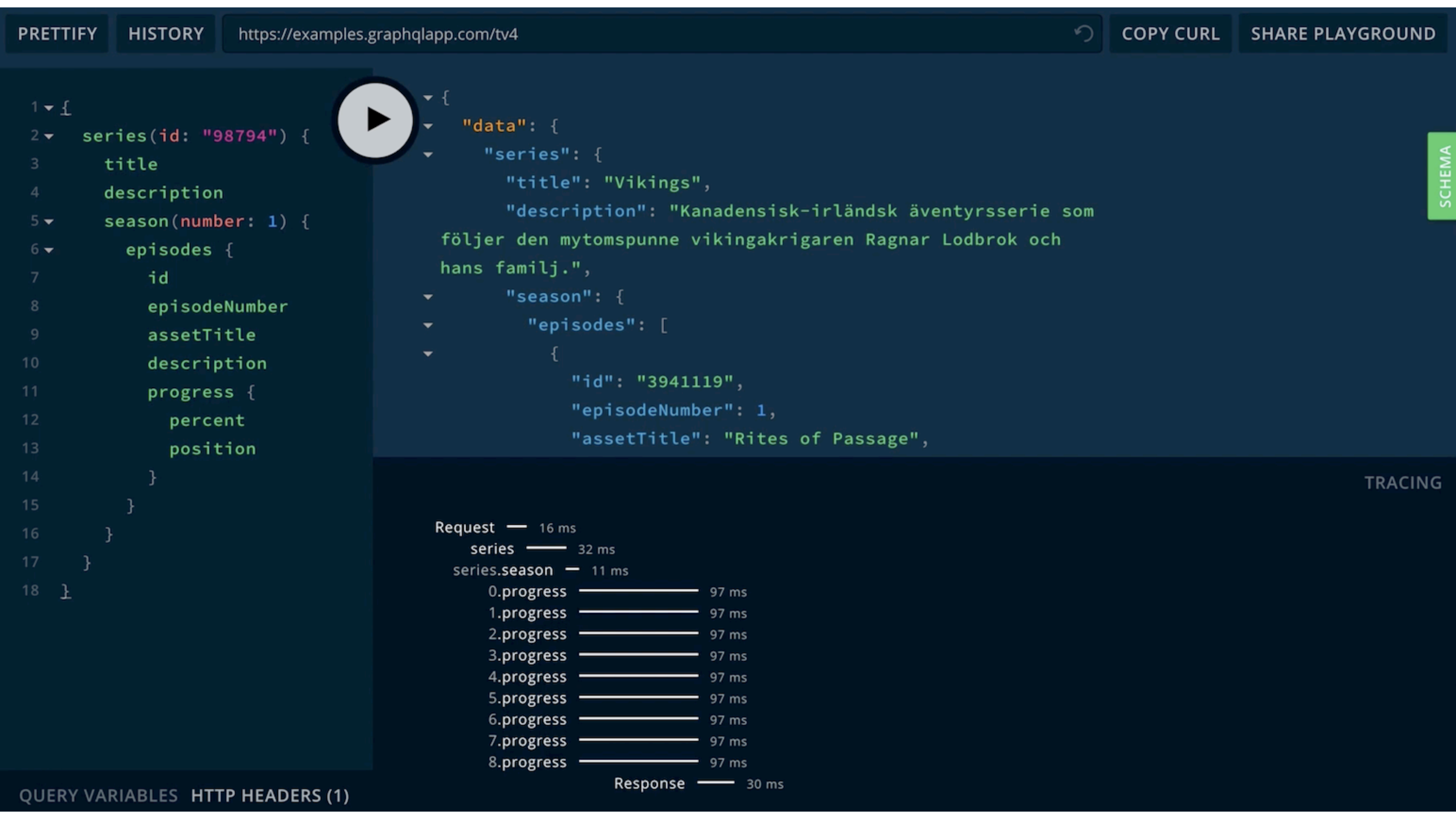
```
class MovieAPI extends RESTDataSource {
  baseURL = 'https://movieapi.com/';

  getProgram(id) {
    return this.get(`movies/${id}`);
  }

  async getMostViewedMovies() {
    const body = await this.get('movies', {
      per_page: 10,
      order_by: 'most_viewed',
    });
    return body.results;
  }
}
```

- Easiest way to encapsulate access to a REST API
- Interacts with a shared cache upon request
- Supports whole and partial query caching




```
{
  series(id: "98794") {
    title
    description
    seasons {
      seasonNumber
      episodes {
        episodeNumber
        assetTitle
        description
        progress {
          percent
          position
        }
      }
    }
  }
}
```

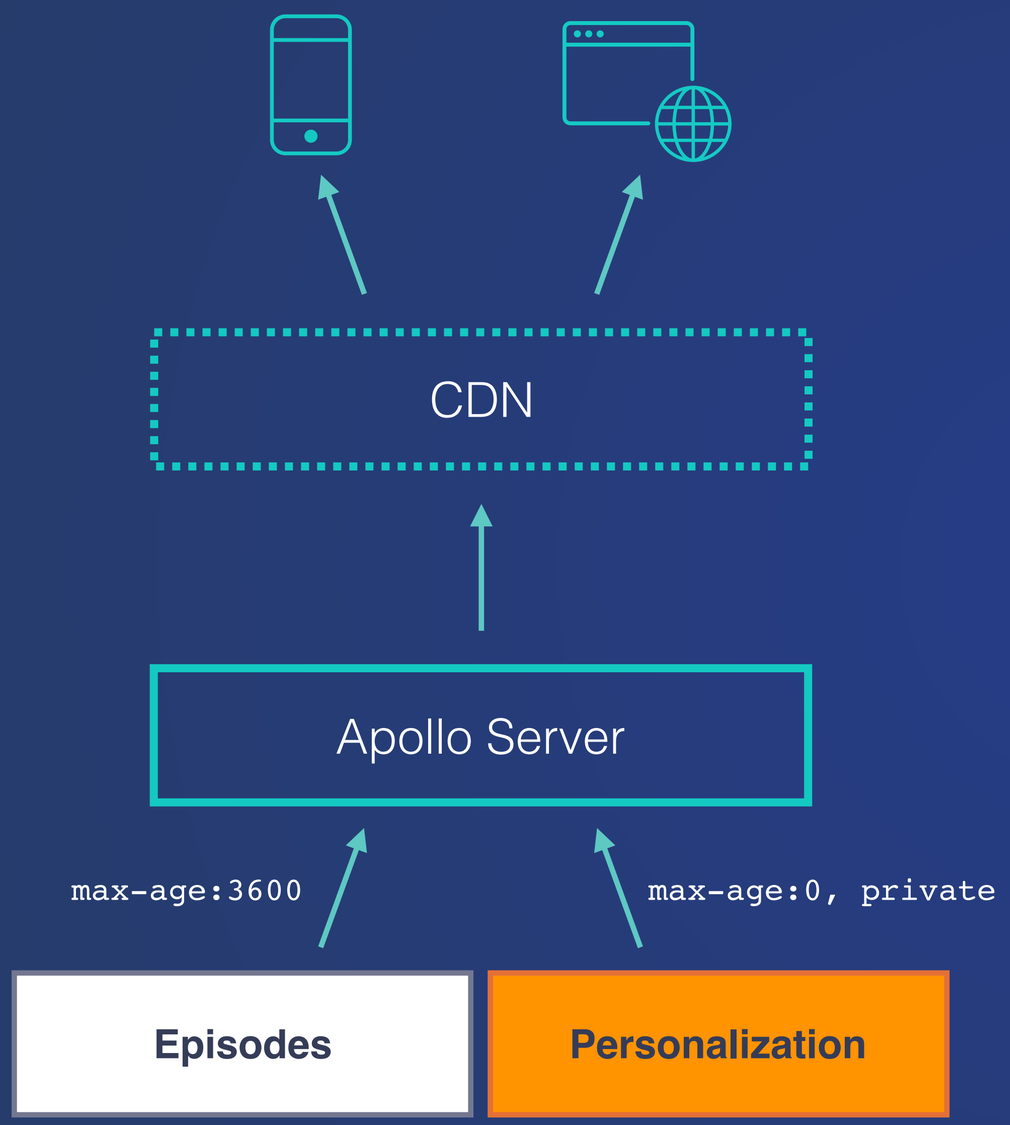

PRETTIFY
HISTORY
https://examples.graphqlapp.com/tv4
COPY CURL
SHARE PLAYGROUND
SCHEMA
{
series(id: "98794") {
title
description
season(number: 1) {
episodes {
id
episodeNumber
assetTitle
description
progress {
percent
position
}
}
}
}
}
{
"data": {
"series": {
"title": "Vikings",
"description": "Kanadensisk-irländsk äventyrsserie som följer den mytomspunne vikingakrigaren Ragnar Lodbrok och hans familj.",
"season": {
"episodes": [
{
"id": "3941119",
"episodeNumber": 1,
"assetTitle": "Rites of Passage",
TRACING
Request 22 ms
series 195 ms
series.season 156 ms
0.progress 100 ms
1.progress 100 ms
2.progress 100 ms
3.progress 100 ms
4.progress 100 ms
5.progress 100 ms
6.progress 100 ms
7.progress 100 ms
8.progress 100 ms
Response 33 ms
QUERY VARIABLES
HTTP HEADERS (1)


PRETTIFY
HISTORY
https://examples.graphqlapp.com/tv4
COPY CURL
SHARE PLAYGROUND
SCHEMA
{
series(id: "98794") {
title
description
season(number: 1) {
episodes {
id
episodeNumber
assetTitle
description
progress {
percent
position
}
}
}
}
}
{
"data": {
"series": {
"title": "Vikings",
"description": "Kanadensisk-irländsk äventyrsserie som följer den mytomspunne vikingakrigaren Ragnar Lodbrok och hans familj.",
"season": {
"episodes": [
{
"id": "3941119",
"episodeNumber": 1,
"assetTitle": "Rites of Passage",
TRACING
Request 16 ms
series 32 ms
series.season 11 ms
0.progress 97 ms
1.progress 97 ms
2.progress 97 ms
3.progress 97 ms
4.progress 97 ms
5.progress 97 ms
6.progress 97 ms
7.progress 97 ms
8.progress 97 ms
Response 30 ms
QUERY VARIABLES
HTTP HEADERS (1)


CLOUDFLARE®

+


APOLLO




```
{
  series(id: "98794") {
    title
    description
    seasons {
      seasonNumber
      episodes {
        episodeNumber
        assetTitle
        description
        progress {
          percent
          position
        }
      }
    }
  }
}
```

CLOUDFLARE®

Cloudflare Workers

apollographql.com/edge

极客邦企业培训与咨询

帮助企业与技术人成长

精品课程

Excellent Course

- ✔《互联网大规模分布式架构设计与实践》
- ✔《基于大数据的企业运营与精准营销》
- ✔《大数据和人工智能在金融领域的应用》
- ✔《区块链应用与开发技术高级培训》
- ✔《通往卓越管理的阶梯》

扫码关注官方微信服务号
了解更多课程详细信息

Geekbang
极客邦科技

apollographql.com
@apollographql

Thanks!
Get started at apollographql.com/docs

Sashko Stubailo, @stubailo
Open Source Lead, Apollo