Supercharged Data Fetching to Power GraphQL

hyper

THAT®

CONFERENCE

THANK YOU, THAT CONFERENCE PARTNERS!

CUNA MUTUAL GROUP

CORE BTS

Unspecified SOFTWARE CO

Progress

nvisia connect. build. enable.

ionic

GrapeCity

symplr

Google Cloud

TREK

Algorand

twilio

Supercharged Data Fetching to Power your Graph

- Start with a poorly structured, poorly performing Graph
- Get Spicy 🌶️ w/ some live code (May the demo gods be with us 🙏)
- Make the Graph "Not So Bad"™

hyper

Takeaways

- Learn how to structure types in a GraphQL project
- Learn how to write resolvers that are:
  - Efficient
  - Easy to reason about
  - Easy to test
  - Easy to maintain
  - Reusable
- Learn how dataloader makes everything better, no 🧢

hyper

Me

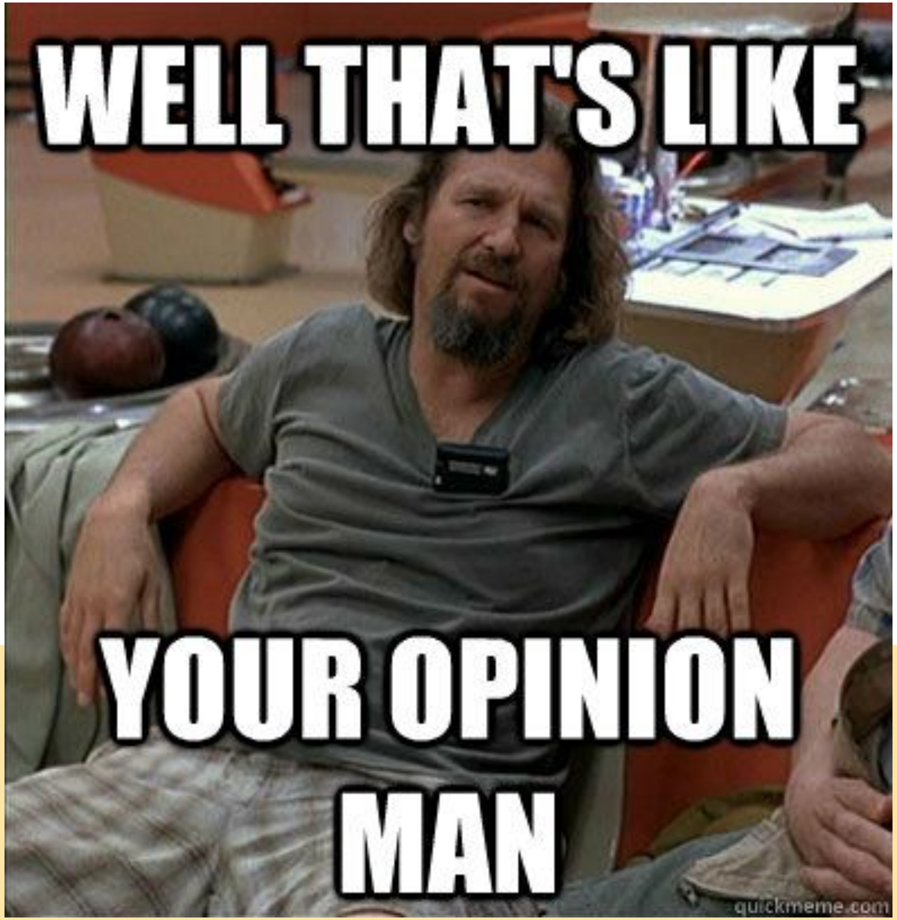
Tyler Hall

Principal Software Engineer @ hyper

- Enterprise GraphQL
    - ~25 micro-graphs combined using schema-stitching/federation
- Lots of projects leveraging GraphQL
- Jiu-Jitsu, Outdoors

Tech I like: hyper, GraphQL, Deno, Svelte, Arweave

hyper



hyper

Why GraphQL

- Fetch the data you need
- In the shape you need it
- No client-side overfetching/data manipulation
- No server-side overfetching (impl specific)
- Code to interfaces (schema)
- OOTB DTOs (your Types!)

hyper

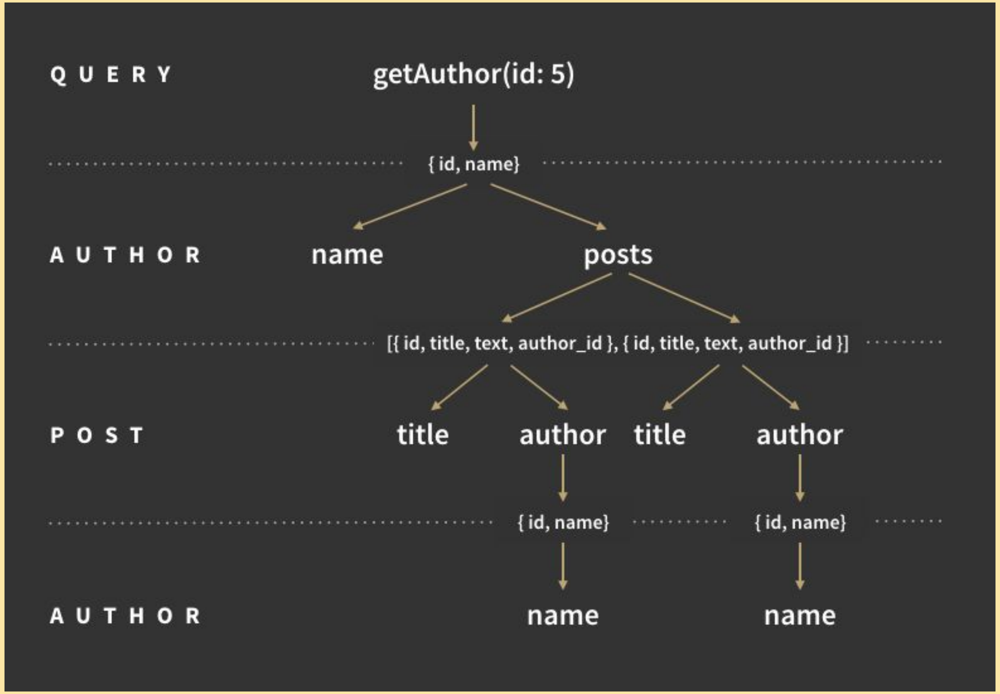
Resolvers Rules

- Resolvers are executed breadth-firstly
  - Siblings are executed in parallel
  - A Child is executed only after its parent resolves
- If the field is a type, then execution continues to the field's resolvers requested on that type
- If the field is a scalar (or returns null), execution completes.

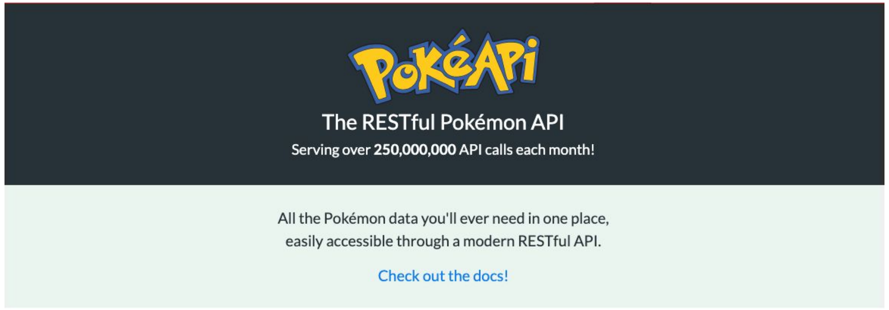
Every field, on every type, in your Graph, has a resolver.

hyper

```
query {
 getAuthor (id: 5) {
  name
  posts {
   title
   author {
    name
   }
  }
 }
}
```



hyper



The RESTful Pokémon API

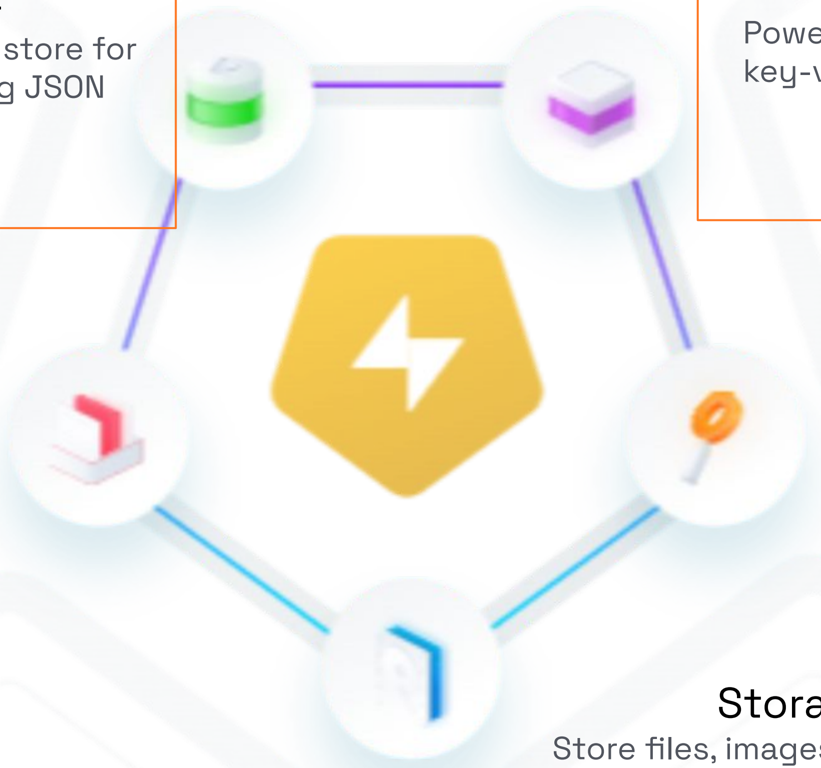
Serving over **250,000,000** API calls each month!

All the Pokémon data you'll ever need in one place,
easily accessible through a modern RESTful API.

Check out the docs!

hyper

hyper's 5 backend services



Data

Document data store for storing/querying JSON documents.

Cache

Powerful, in-memory key-value store.

Queue

A persistent queue, great for event driven, serverless, or asynchronous workloads

Search

Powerful full-text search index

Storage

Store files, images, videos, text...

Live Code Time - May The Demo Gods Bless Us

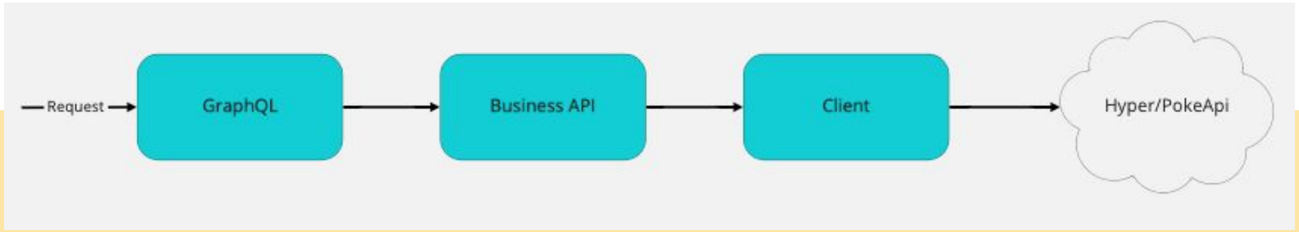
Praise Lord Helix

🙏🙏🙏

hyper

Context (3rd resolver argument)

Great for dependency injection!



hyper

The Default Resolver

```
const resolvers = {
 SomeTypeWithNoResolversDefined: {
  // graphql.js adds all of these for you
  id: ({ id }) => id,
  name: ({ name }) => name,
  age: ({ age }) => age
  ...
 }
}
```

Every field, on every type, in your Graph, has a resolver.

hyper

This does not scale.

- Overfetching on the server!
- Strain on datasources!
- Valid operations that cannot be resolved!
- Esoteric
- Gives rise to bad patterns

We're losing all of the benefits of GraphQL! 😱

hyper

In REST

- Verbs
- Resources (represented as URL segments)

GET /pokemon/pikachu -> Give me Pokemon ‘pikachu’

REST Mindset for GraphQL

- Query
- Mutation
- Subscription

Query pokemon(name: ‘Pikachu’) -> Get Pokemon “Pikachu”
Query trainer(name: Ash) -> Get Trainer “Ash”

This is an incorrect, or at the very least, an incomplete understanding

hyper

Graph Mindset for GraphQL

Query -> Trainer

Query -> Pokemon -> Trainer

Query -> Pokemon -> Move -> Pokemon -> Trainer

Query -> Trainer -> Pokemon -> Move -> Pokemon -> Trainer

...

Cyclically "re-entering" a type, from a new spot in the graph

Do we eliminate cyclical references?

NO!

How can we leverage GraphQL?

Remember: GraphQL will **ALWAYS** call a field’s resolver, if the field is in the operation. Before, it was calling the “default” resolver.

So let’s define the resolver and GraphQL will call it!

What We Will Do

- Won’t depend on default resolvers
- Each field will have its own resolver
- Each field will fetch its own data
- If a field resolves to another type, it will return the identifier for the type being resolved
    - That identifier is the parent that is passed to the child resolvers.

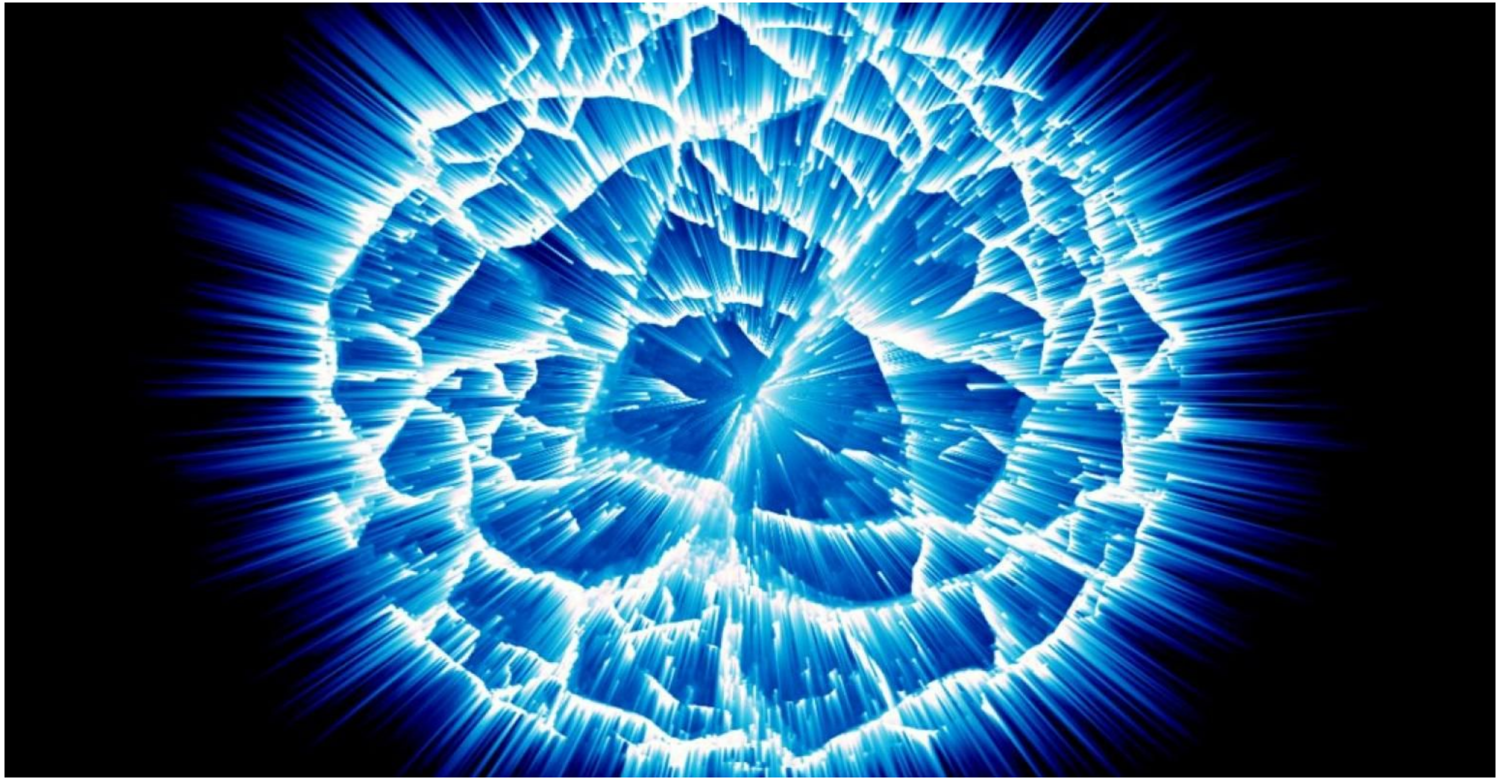
What do we have

- Each resolver is:
  - Easy to debug
  - Easy to test
  - Easy to maintain
  - Easy to reason about
  - A explicit separation of the presentation model

BUT…

hyper

Datasources go Brrrrrrrrr



Dataloader

- Maintained by GraphQL team
- Can be used with any datasource
- Dedupes requests, and batches them to “loader” function
- Caches the results in memory (memoization)
  - Cache be anything that implements the Map api

hyper

Dataloader

- Provide a “batch” function that:
  - Receives an array
  - Returns an array
    - Must be same length and order as received
  - Call with .load()
  - Prime cache with .prime()

- Dedupes and batches load calls into batch function:
  - load(1), load(2), load(1), load(3) -> batch([1, 2, 3])
- Caches results
  - load(1) -> returns previous result of load(1)

What We’ll Do

- Wrap clients with dataloader
- Busienss API calls into dataloaders instead of clients

hyper

How do we load our data

- By _id (Trainer, Pokemon)
- By parent (Trainer -> Pokemon)
- By type (Trainer)
- By move name (Move -> Pokemon)
- By move name (Move fields ie. accuracy, power, pp)

hyper

Can we make it better?

Prime across dataloaders!

hyper

Can we make it better?

Dataloaders pull from cache!

hyper

What do we have

- Efficient data fetching
  - No server over-fetching
- Now Business logic can also be “stingy” with fetching data
  - Dataloaders dedupe!
  - Easier to test business logic
- Clean boundaries
- Extensibility

hyper

Things to keep in mind

- On mutations, clear dataloader caches before resolving
    - Can be done with middleware or composition
- You'll most likely want dataloaders scoped to a request
- Implement Map API and give to dataloader as cacheMap to have custom caching functionality.

hyper

Thank You

- https://github.com/TillaTheHun0/supercharged-data-fetching-talk
- https://hyper.io

We are available for development/consulting. Let us help you build something awesome!