GraphQL

graphql.org: “A query language for your API”

What is an API?

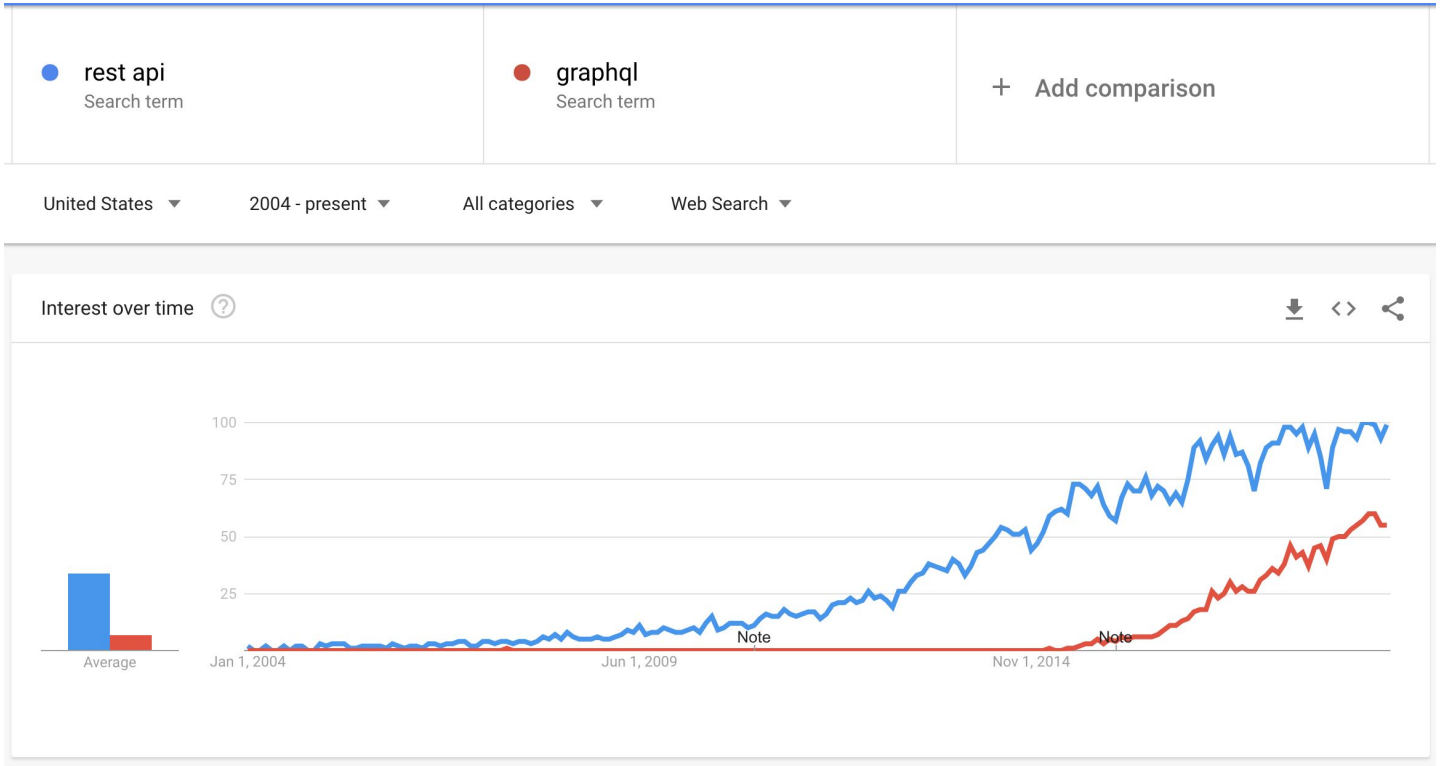
… Abstraction …

… Black Box ….

… Interface …

… Application …

GraphQL vs. Rest API



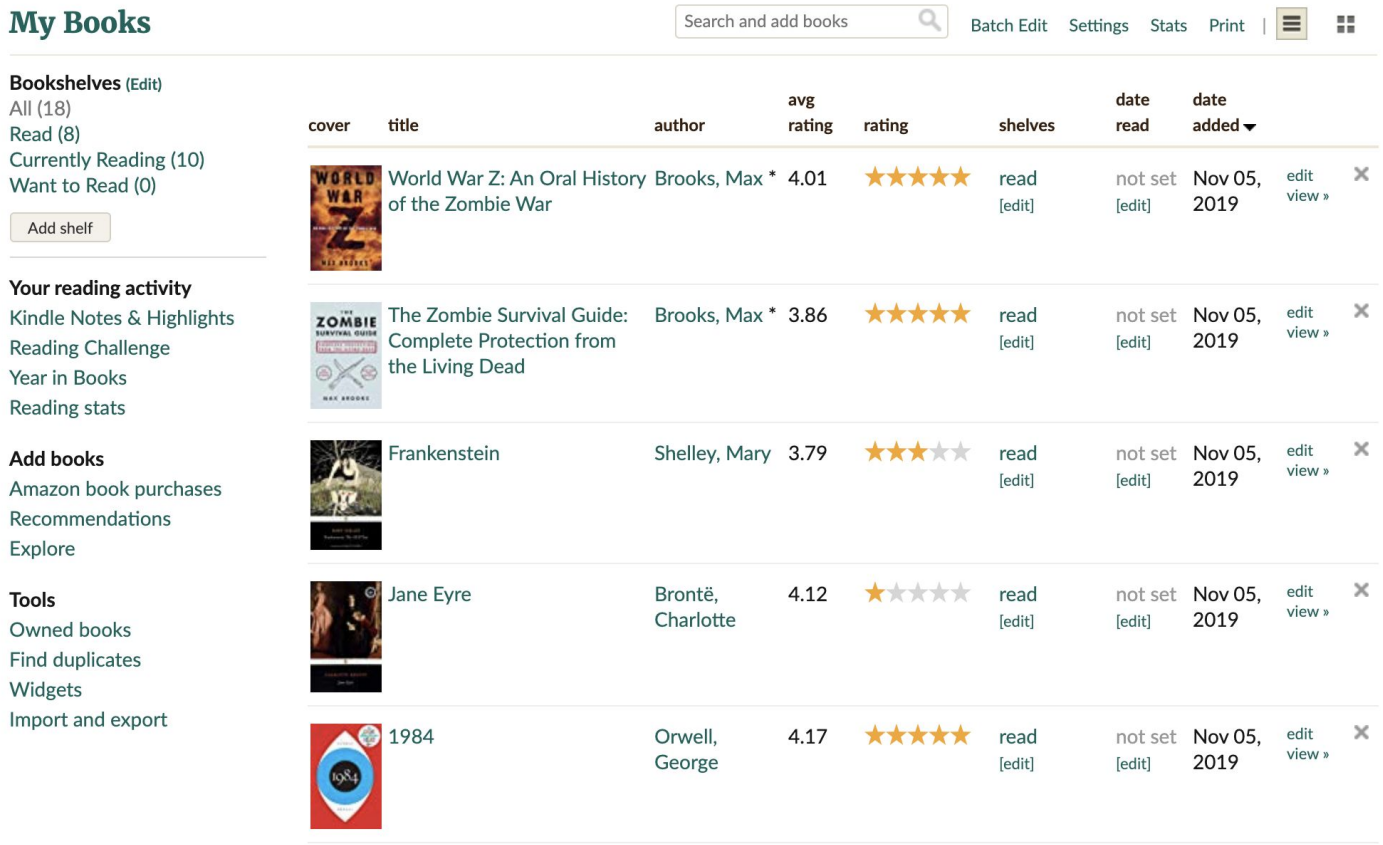
What is a REST API?

Representational State Transfer

GET, HEAD, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS and TRACE

(in reality: GET, POST)

Let’s design a REST API for an online Bookstore



"Bad programmers worry about the code. Good programmers worry about data structures and their relationships." – Linus Torvalds

My Books

Search and add books | Batch Edit | Settings | Stats | Print

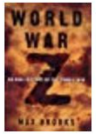
Bookshelves (Edit)
All (18)
Read (8)
Currently Reading (10)
Want to Read (0)

Add shelf

Your reading activity
Kindle Notes & Highlights
Reading Challenge
Year in Books
Reading stats

Add books
Amazon book purchases
Recommendations
Explore

Tools
Owned books
Find duplicates
Widgets
Import and export

cover	title	author	avg rating	rating	shelves	date read	date added	
	World War Z: An Oral History of the Zombie War	Brooks, Max *	4.01	★★★★★	read [edit]	not set [edit]	Nov 05, 2019	edit view »
	The Zombie Survival Guide: Complete Protection from the Living Dead	Brooks, Max *	3.86	★★★★★	read [edit]	not set [edit]	Nov 05, 2019	edit view »
	Frankenstein	Shelley, Mary	3.79	★★★☆☆	read [edit]	not set [edit]	Nov 05, 2019	edit view »
	Jane Eyre	Brontë, Charlotte	4.12	★☆☆☆☆	read [edit]	not set [edit]	Nov 05, 2019	edit view »
	1984	Orwell, George	4.17	★★★★★	read [edit]	not set [edit]	Nov 05, 2019	edit view »

Books

- Title
- Author
- Year Written
- Summary

Authors

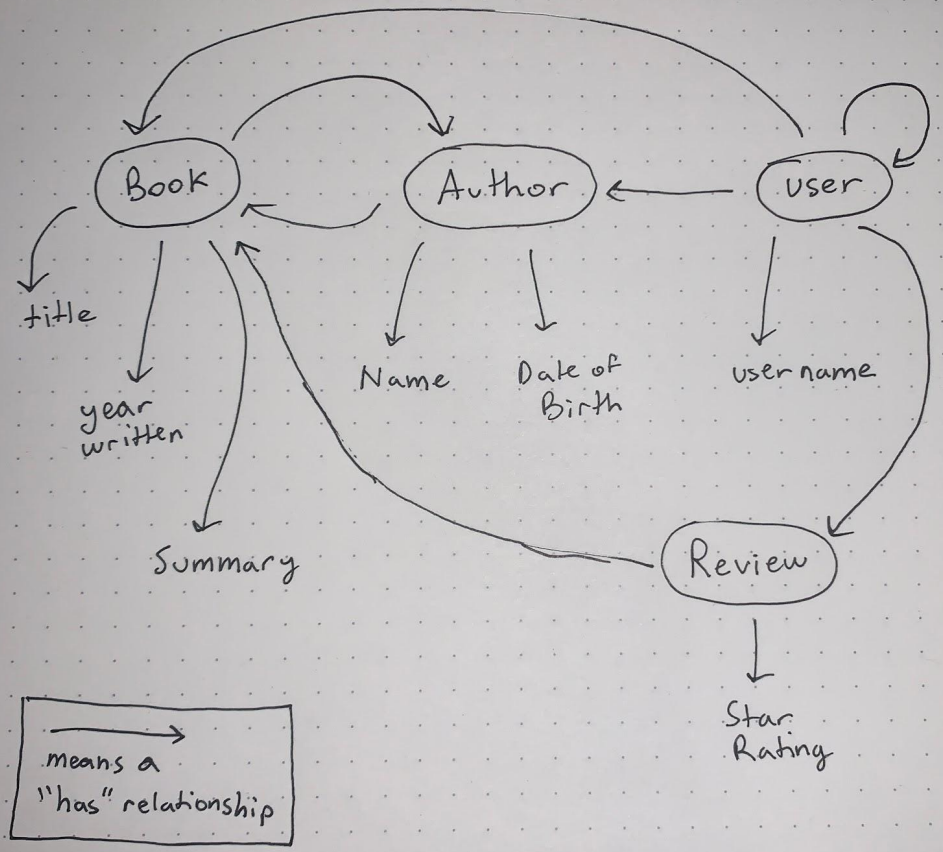
- Books
- Name
- Date of Birth

Users

- User Name
- Read Books
- Friends
- Favorite Author
- Reviews

Reviews

- Book
- Stars


Book
Author
user
title
year written
Summary
Name
Date of Birth
user name
Review
Star Rating
means a "has" relationship

Rest APIs

```
GET my-books-site.com/books

GET my-books-site.com/authors
```

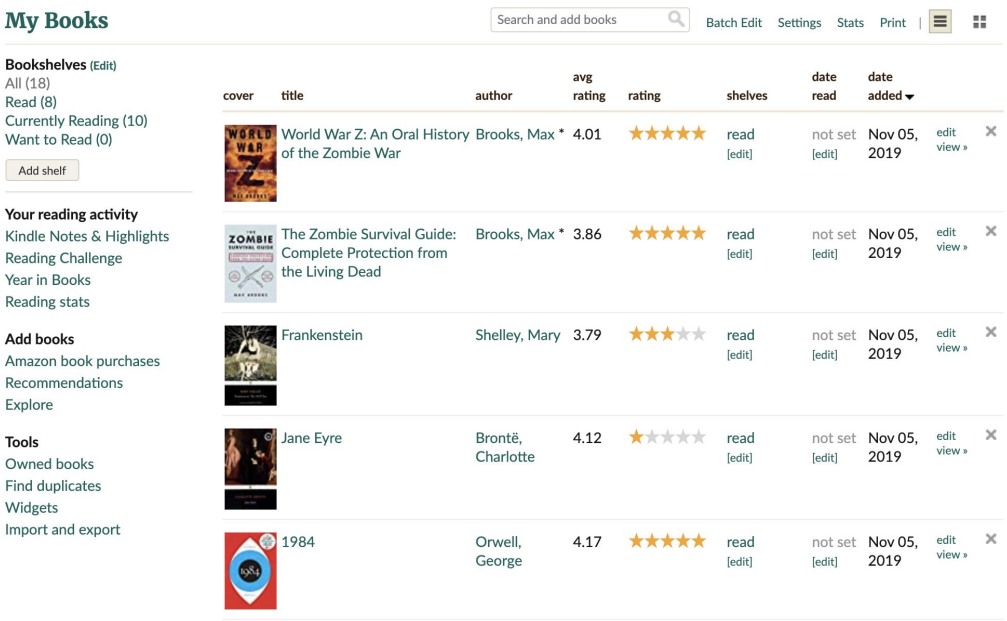
Rest APIs

```
GET my-books-site.com/books

GET my-books-site.com/authors

GET my-books-site.com/books/the+phantom+toolbooth

GET my-books-site.com/authors/mary+shelley

GET my-books-site.com/users/Ivan+Chub
```

Rest APIs cont.

```
POST my-books-site.com/books/add
{
    “Title”: “Cracking the Coding Interview”
    “Author”: “Gayle Laakmann McDowell” or “AuthorID”: 12
}
```

How would you use this API?



Books

- Title
- Author
- Year Written
- Summary
- Picture URL

Authors

- Books
- Name
- Date of Birth

Users

- User Name
- Read Books
- Friends
- Favorite Author

Reviews

- Book
- Stars

1. GET /users/Ivan+Chub

   { likedBookIds: [...], userName: “Ivan Chub”, friends: [<empty>] reviewIds: [...] }

2. GET /books/<id 1>

   { title: “Frankenstein” ... }

3. GET /books/<id 2>
4. …
5. GET /books/<id N>

6. GET /reviews/<review 1>
7. GET /reviews/<review 2>
8. …
9. GET /reviews/<review N>

11, 12, 13 … : GET /images/<book image N>

GET /landingpagedata/Ivan+Chub

{

likedBooks: [{..}, {..}, {..}], **userName**: “Ivan Chub”, **friends**: [<empty>] **reviews**: [{**book**: “frankenstein”, **stars**: 2} …]

}

Pros

- Simple

Cons

- Verbose
- Overfetching
- Underfetching
- Coupled to presentation layer

Ok, Ivan, we know all that …

Can you tell us about GraphQL already???

GraphQL

“A query language for your API”

What is GraphQL?

1. Formally describe your data, and relationships between them.
2. Query for precisely what you want, nothing more, and nothing less.
3. The shape of the response is exactly what you expect, checked against your schema

Let’s talk about the data

GraphQL comes with a set of default scalar types out of the box:

- Int: A signed 32-bit integer.
- Float: A signed double-precision floating-point value.
- String: A UTF-8 character sequence.
- Boolean: `true` or `false`.

```
type Person {
  name: String!
}
```

This defines a new type called ‘Person’

With a single field called ‘name’

That is of type ‘String!’

```
Int

Int!

[Int]

[Int!]

[Int!]!
```

You can also define custom enums, which are types that can be one of N values.

```
enum Color {
  red
  green
  blue
  pink
}
```

```
type Car {
  wheelCount: Int!
  tirePressures: [Int!]!
  color: String!
  weight: Float!
  passedEmissions: Boolean!

  requiredOctane: Int
  marketingDescription: String

  previousGeneration: Car

  … add your own …
}
```

My Books

Search and add books | Batch Edit | Settings | Stats | Print

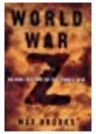
Bookshelves (Edit)
All (18)
Read (8)
Currently Reading (10)
Want to Read (0)

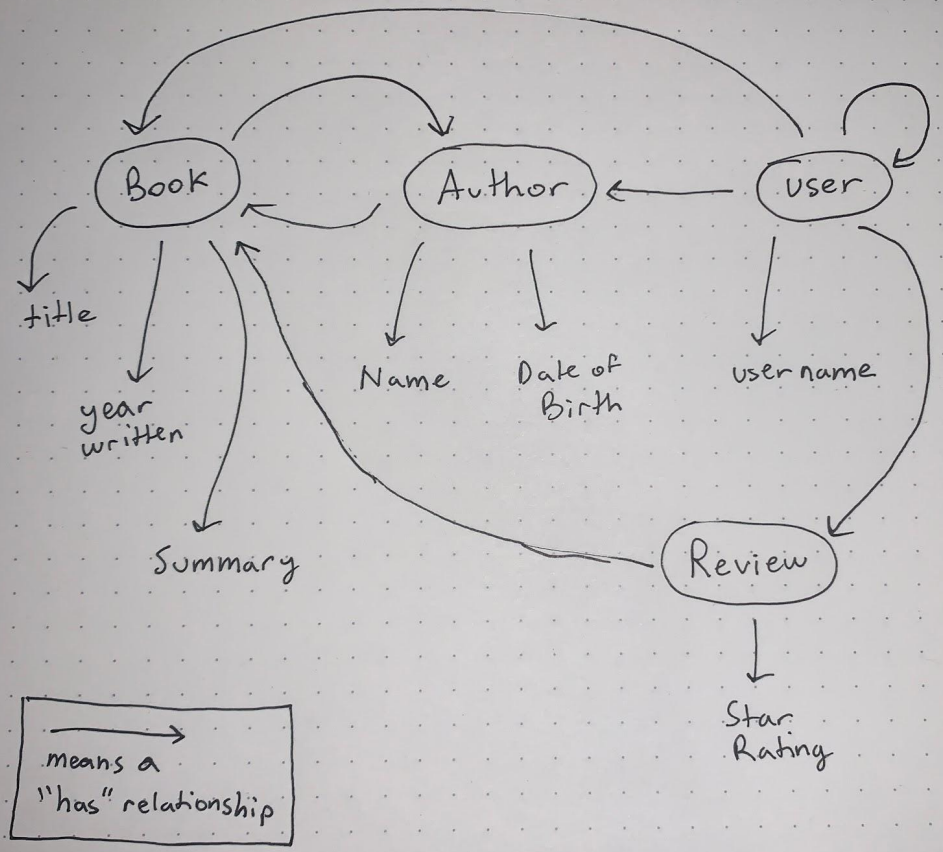
Add shelf

Your reading activity
Kindle Notes & Highlights
Reading Challenge
Year in Books
Reading stats

Add books
Amazon book purchases
Recommendations
Explore

Tools
Owned books
Find duplicates
Widgets
Import and export

cover	title	author	avg rating	rating	shelves	date read	date added	
	World War Z: An Oral History of the Zombie War	Brooks, Max *	4.01	★★★★★	read [edit]	not set [edit]	Nov 05, 2019	edit view »
	The Zombie Survival Guide: Complete Protection from the Living Dead	Brooks, Max *	3.86	★★★★★	read [edit]	not set [edit]	Nov 05, 2019	edit view »
	Frankenstein	Shelley, Mary	3.79	★★★☆☆	read [edit]	not set [edit]	Nov 05, 2019	edit view »
	Jane Eyre	Brontë, Charlotte	4.12	★☆☆☆☆	read [edit]	not set [edit]	Nov 05, 2019	edit view »
	1984	Orwell, George	4.17	★★★★★	read [edit]	not set [edit]	Nov 05, 2019	edit view »


Book
Author
user
title
year written
Summary
Name
Date of Birth
user name
Review
Star Rating
means a "has" relationship

Formal Description Cont.

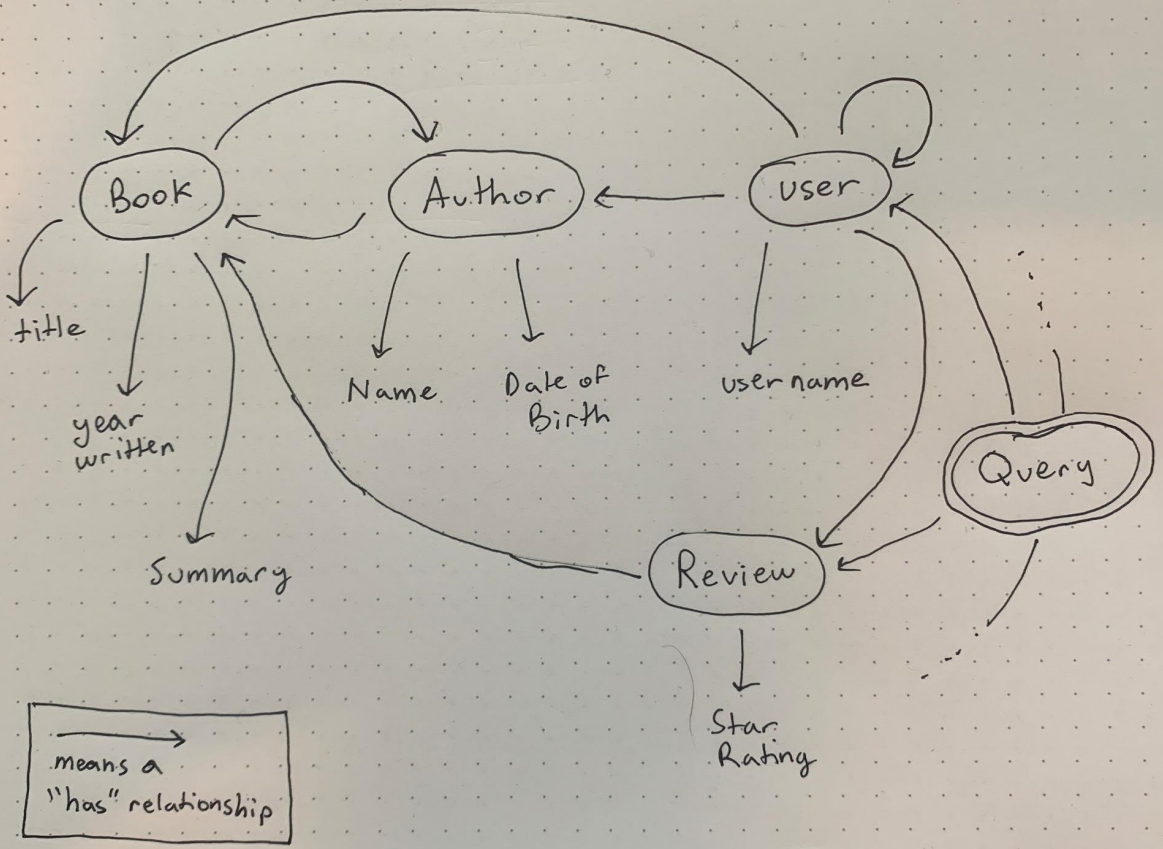
```
type Book {
  title: String!
  author: Author!
  yearPublished: String!
  summary: String!
}

type Author {
  name: String!
  booksPublished: [Book!]!
  dateOfBirth: String
}
```

```
type User {
  userName: String!
  booksRead: [Book!]!
  friends: [User!]!
  favoriteAuthor: Author
  reviews: [Review!]!
}

type Review {
  book: Book!
  stars: Int!
}
```

Formal Description Cont.

```
GET my-books-site.com/books

GET my-books-site.com/authors

…
```

```
schema {
  query: Query
  mutation: Mutation
}

type Query {
  books: [Book!]!
  authors: [Author!]!
  users: [User!]!

  book(bookName: String!): Book
  author(authorName: String!): Author
  user(userName: String!): User
}
```


Book
Author
user
title
year written
Summary
Name
Date of Birth
user name
Query
Review
Star Rating
means a "has" relationship

Arguments

```
type User {
  userName: String!
  booksRead: [Book!]!
  friends: [User!]!
  favoriteAuthor: Author
  reviews: [Review!]!

  review(bookName: String!): Review
}
```

```
GET my-books-site.com/books/the+phantom+toolbooth
```

Querying

```
type Book {
  title: String!
  author: Author!
  yearPublished: String!
  summary: String!
}

type Author {
  name: String!
  booksPublished: [Book!]!
  dateOfBirth: String
}
```

```
type User {
  userName: String!
  booksRead: [Book!]!
  friends: [User!]!
  favoriteAuthor: Author
  reviews: [Review!]!
}

type Review {
  book: Book!
  stars: Int!
}
```

```
query HomePageQuery {
  books {
    title
  }
}
```

What does this return? JSON 😍

```
[
  {
    "title": "Frankenstein"
  },
  {
    "title": "Bob the Builders Excellent Adventure"
  },
  ...
]
```

Querying

```
type Book {
  title: String!
  author: Author!
  yearPublished: String!
  summary: String!
}

type Author {
  name: String!
  booksPublished: [Book!]!
  dateOfBirth: String
}
```

```
type User {
  userName: String!
  booksRead: [Book!]!
  friends: [User!]!
  favoriteAuthor: Author
  reviews: [Review!]!
}

type Review {
  book: Book!
  stars: Int!
}
```

```
query HomePageQuery {
  user(userName: "Ivan Chub") {
    userName
    favoriteAuthor
    booksRead {
      title
      author
      yearPublished
      summary
    }
    reviews {
      book {
        title
      }
      stars
    }
  }
}
```

What does this return?

JSON 😍

```
{
  "userName": "Ivan Chub",
  "favoriteAuthor": null,
  "booksRead": [
    {
      "title": "Frankenstein",
      ...
    }
    ...
  ],
  ...
  "reviews": [
    {
      "book": {
        title: "Frankenstein",
      },
      "stars": 5
    }
  ]
}
```

Dynasty.com queries (and GraphIQL)

LobbyPageQueries

```
query test {
  self {
    id
    name
  }
}

query test {
  search(prefix:"ivan" podId: 0)
{
    displayName
    path
  }
}
```

```
mostRecentClockEvent {
      id
      clockInTime
      clockOutTime
      user {
        id
      }
    }
```

Ensures the correct things return

```
query test {
  self {
    id
    name
  }
  user(userId:13123123) {
    id
  }
}
```

https://dynasty.com/leasing/api

Fragments

```
query HomePageQuery($userName: String!) {
  user(userName: $userName) {
    userName
    favoriteAuthor
    booksRead {
      title
      author
      yearPublished
      summary
    }
    friends {
      userName
    }
    reviews {
      book {
        title
      }
      stars
    }
  }
}
```

```
query BooksPageQuery($userName: String!) {
  user(userName: $userName) {
    booksRead {
      title
      author
      yearPublished
      summary
    }
  }
}
```

Fragments cont.

```
query HomePageQuery {
  user(userName: “Ivan Chub”) {
    userName
    favoriteAuthor
    ...books
    friends {
      userName
    }
    reviews {
      book {
        title
      }
      stars
    }
  }
}
```

```
query BooksPageQuery {
  user(userName: “Ivan Chub”) {
    ...books
  }
}
```

```
fragment books on User {
  booksRead {
    title
    author
    yearPublished
    summary
  }
}
```

Variables

```
query HomePageQuery {
  user(userName: "Ivan Chub") {
    userName
    favoriteAuthor
    booksRead {
      title
      author
      yearPublished
      summary
    }
    friends {
      userName
    }
    reviews {
      book {
        title
      }
      stars
    }
  }
}
```

```
query HomePageQuery($userName: String!) {
  user(userName: $userName) {
    userName
    favoriteAuthor
    booksRead {
      title
      author
      yearPublished
      summary
    }
    friends {
      userName
    }
    reviews {
      book {
        title
      }
      stars
    }
  }
}
```

Variables

Request Payload view source

```
{operationName: "LookAtStation", variables: {stationId: 5},…}
  operationName: "LookAtStation"
  query: "mutation LookAtStation($stationId: Int) {
    setLookingAt(stationId: $stationId) {
      id
      ...LookingAtFragment
      __typename
    }
  }

  fragment LookingAtFragment on User {
    lookingAt {
      id
      name
      __typename
    }
    lookingAtUpdated
    desiresToLookAtAutoStation
    __typename
  }
  "
  variables: {stationId: 5}
    stationId: 5
```

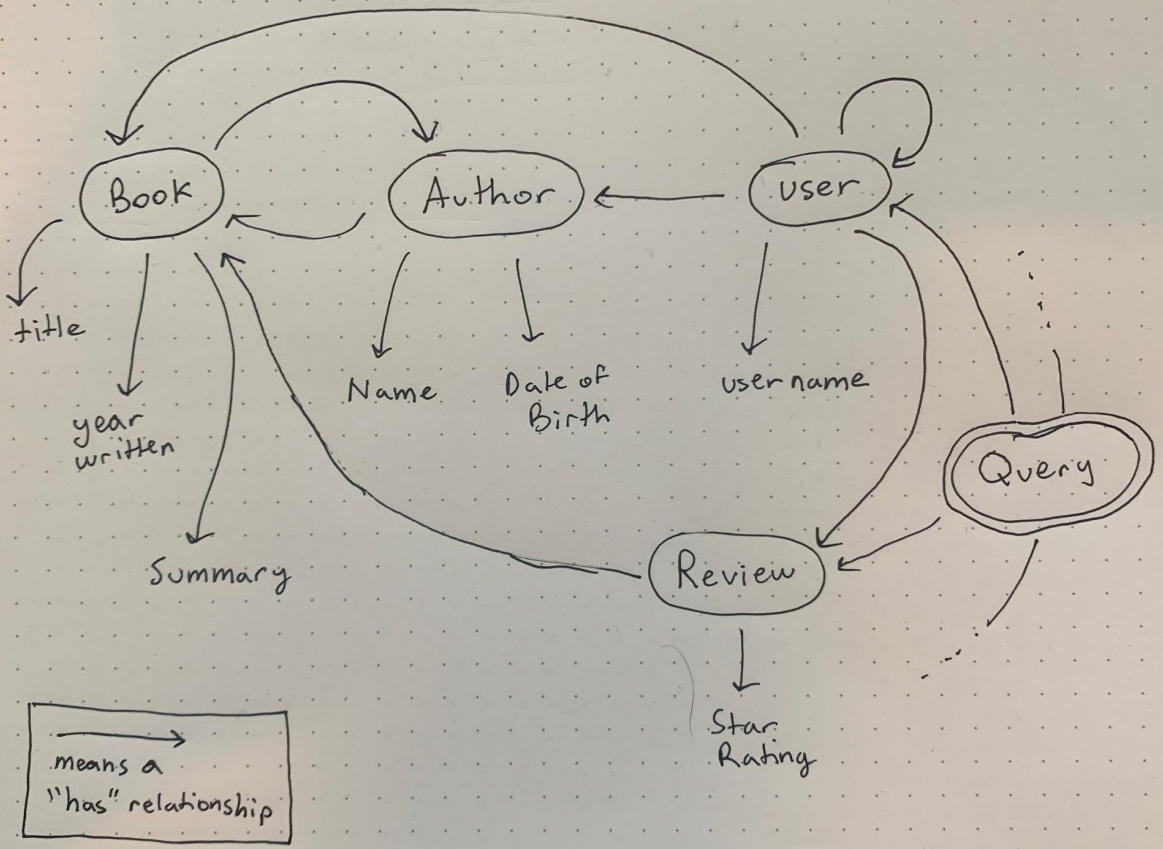
But… how does it work?

every **Type -> Field** pair has a fetcher

```
type Book {
  title: String!          # fetcher
  author: Author!         # fetcher
  yearPublished: String! # fetcher
  summary: String!        # fetcher
}
```

```java
class Book { // java
  public long id;
  public long authorId;
  public String title;
}

// graphql can do this one automagically
String fetchTitleFromBook(Book book) {
  return book.title;
}

// these ones often require some thought
Author fetchAuthorFromBook(Book book) {
  return AuthorDB.getAuthor(book.authorId);
}

Book fetchBookByName(Query root, String name) {
  return BookDB.findBook(name);
}
```

```graphql
type Book { #graphql
  title: String!
  author: Author!
  yearPublished: String!
  summary: String!
}
```


Book
Author
user
title
year written
Summary
Name
Date of Birth
user name
Query
Review
Star Rating
means a "has" relationship

How does GraphQL match your query to fetchers?

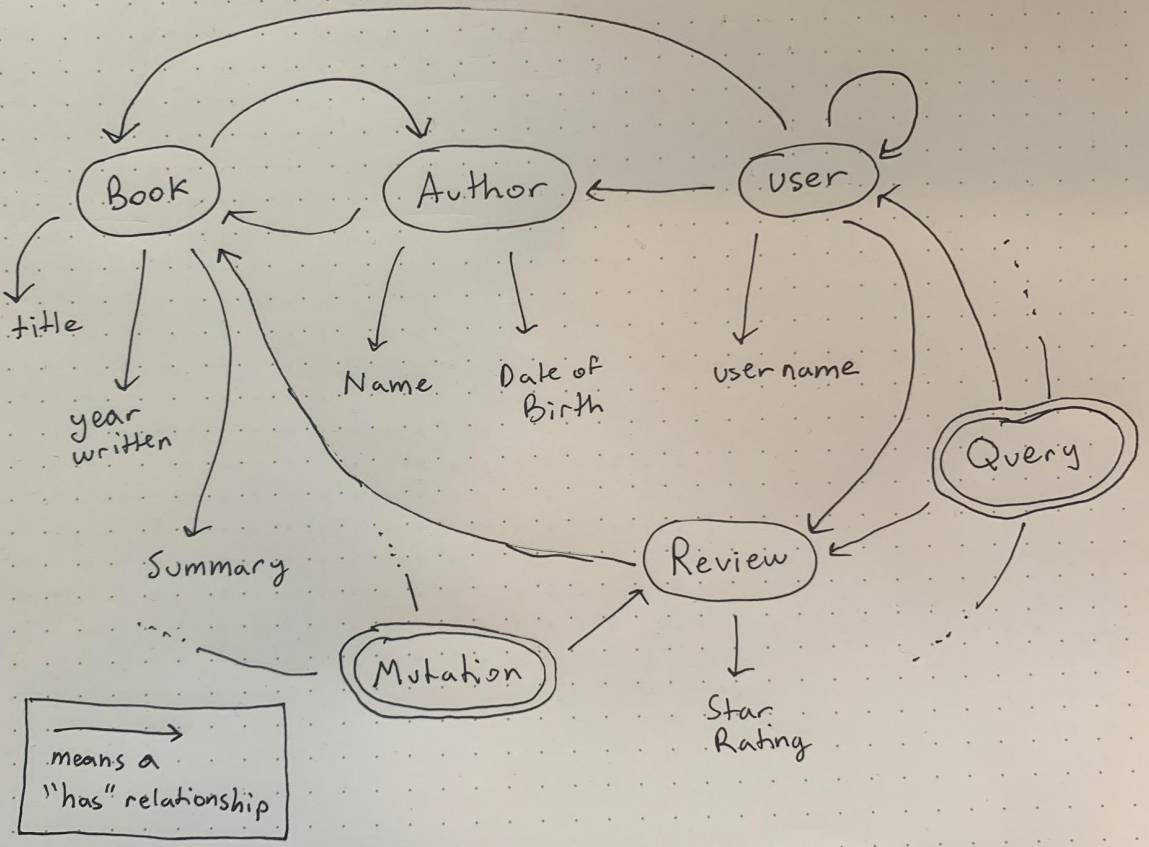
```
.type("LeasingInfo", typeWiring -> typeWiring
    .dataFetcher("propertyType", env -> ((LeasingInfo) env.getSource()).propertyType.name()))
.type("TaskAgent", typeWiring -> typeWiring
    .dataFetcher("onboardingRoles", taskAgentOnboardingRolesFetcher())
    .dataFetcher("weekHours", taskAgentWorkHoursFetcher())
    .dataFetcher("agent", taskAgentAgentFetcher()))
.type("BotThread", typeWiring -> typeWiring
    .dataFetcher("showCountdown", botThreadShowCountdownFetcher())
    .dataFetcher("action", botThreadActionFetcher()))
.type("Action", typeWiring -> typeWiring
    .dataFetcher("commandJsons", botActionCommandJsonsFetcher()))
.type("Thread", typeWiring -> typeWiring
    .dataFetcher("lastMessageInbound", threadLastMessageInboundFetcher())
```

Go to GraphQLApi.java

- Schema.idl
- Fetchers
- Type wiring
- Fragments

Mutations

```
schema {
  query: Query
  mutation: Mutation
}

type Mutation {
  …
  addBook(title: String!, authorName: String!): Book
  …
}
```


Book
Author
User
title
year written
Summary
Name
Date of Birth
user name
Query
Review
Mutation
Star Rating
means a "has" relationship

Mutation Example

```
mutation AddBookMutation {
  addBook(title: String!, authorName: String!) {
    title
    author {
      books {
        title
      }
    }
  }
}
```

Production Mutation Example

```
mutation test {
  setLookingAt(stationId: 3) {
    lookingAt {
      id
    }
    id
    name
    activeOperatorShift {
      id
    }
  }
}
```

This seems dangerous?

- Ever heard of SQL injection? Is there graphql injection?
  - No (because query arguments are passed separately from query)
- Can people just download your entire graph?
- clockInClockOutFetcher
- `checkOperator`
- `checkSuperOperator`
- `GraphQLApi rate limiter`

So what is GraphQL?

- Schema (definition of types + query type + mutations)
- Fetchers + type wiring
- Queries (you write these yourself)

- Frontend library to do the querying (not required)
- Backend library to supply the graph (would recommend you have this)

Wow!

- Everything goes through one POST endpoint, meaning your API definition is <u>not verbose</u>. You define the nodes and edges of your data, GraphQL does the heavy lifting
- Changing front end requirements no longer results in changes to the back end - you just change your query. This means <u>no more tight coupling</u>
- <u>No more underfetching or overfetching</u>, you ask for precisely what you need in exactly one request. 🏃💨
- You data is <u>STRICTLY defined</u>!
- Plays SUPER nicely with front end libraries, like React or Angular

Disadvantages

- Querying complexity
- Caching is difficult
  - Normal REST endpoints just use native HTTP caching
  - GraphQL requests are all POST, which don’t cache
- Error handling is a lil funky
  - Requests always return 200

Other Benefits

The strict typing benefit cannot be overstated

- Typescript
- Apollo codegen (types for objects, and query responses)
- types.gen
- LobbyPage.tsx

Thanks for listening!