CS188
Scalable Internet Services

John Rothfels, 10/15/20

(adapted from Ivan Chub guest lecture, 11/8/19)

Motivation

🕵️‍♀️ What is an API?

🤔 How do we build them?

Motivation

🕵️‍♀️ What is an API?

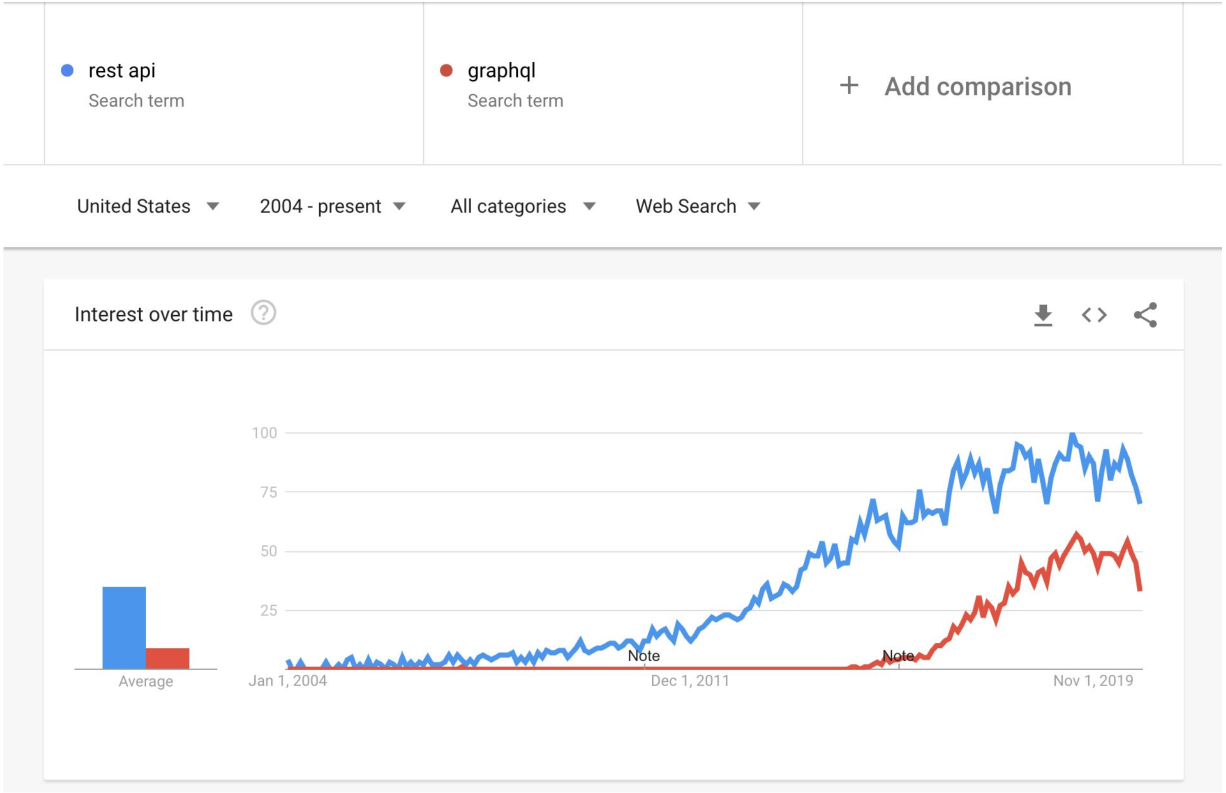
… Abstraction …

… Black Box ….

… Interface …

… Application …

Motivation



source

REST API

Representational **S**tate **T**ransfer: set of constraints/conventions for creating web services.

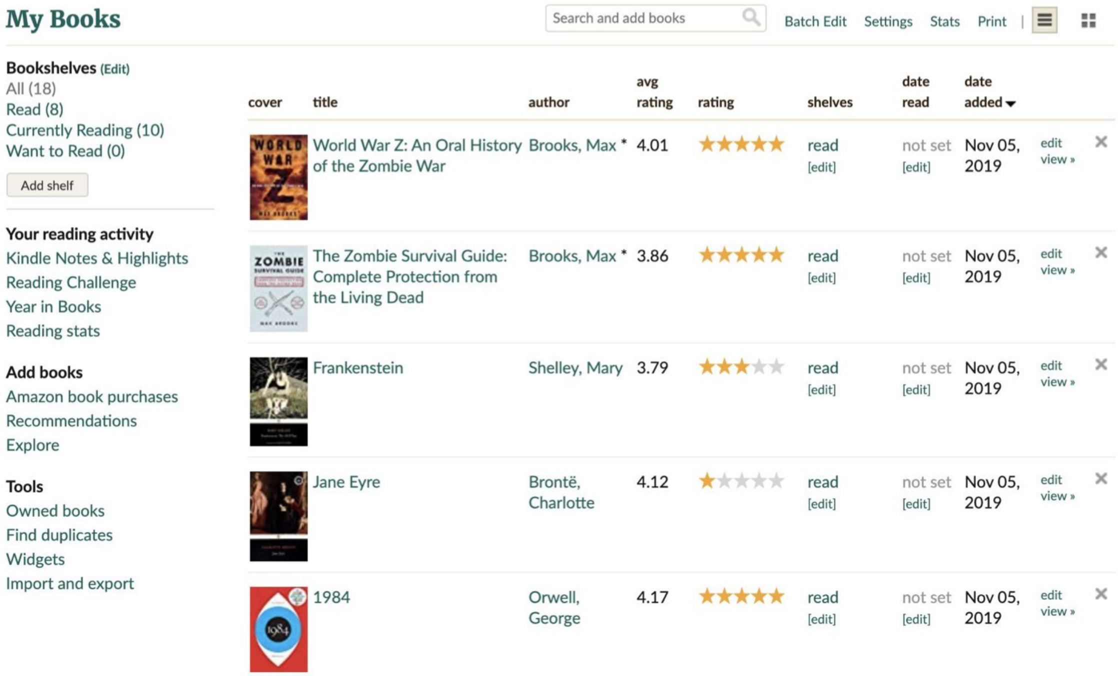
Allows clients to manipulate textual representations of resources using a uniform and predefined set of stateless operations:

- GET, HEAD, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS, TRACE
- (in reality: GET, POST)

ℹ️ Services may or may not conform to the REST architecture.

REST API

Let’s design a REST API for an online bookstore:



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

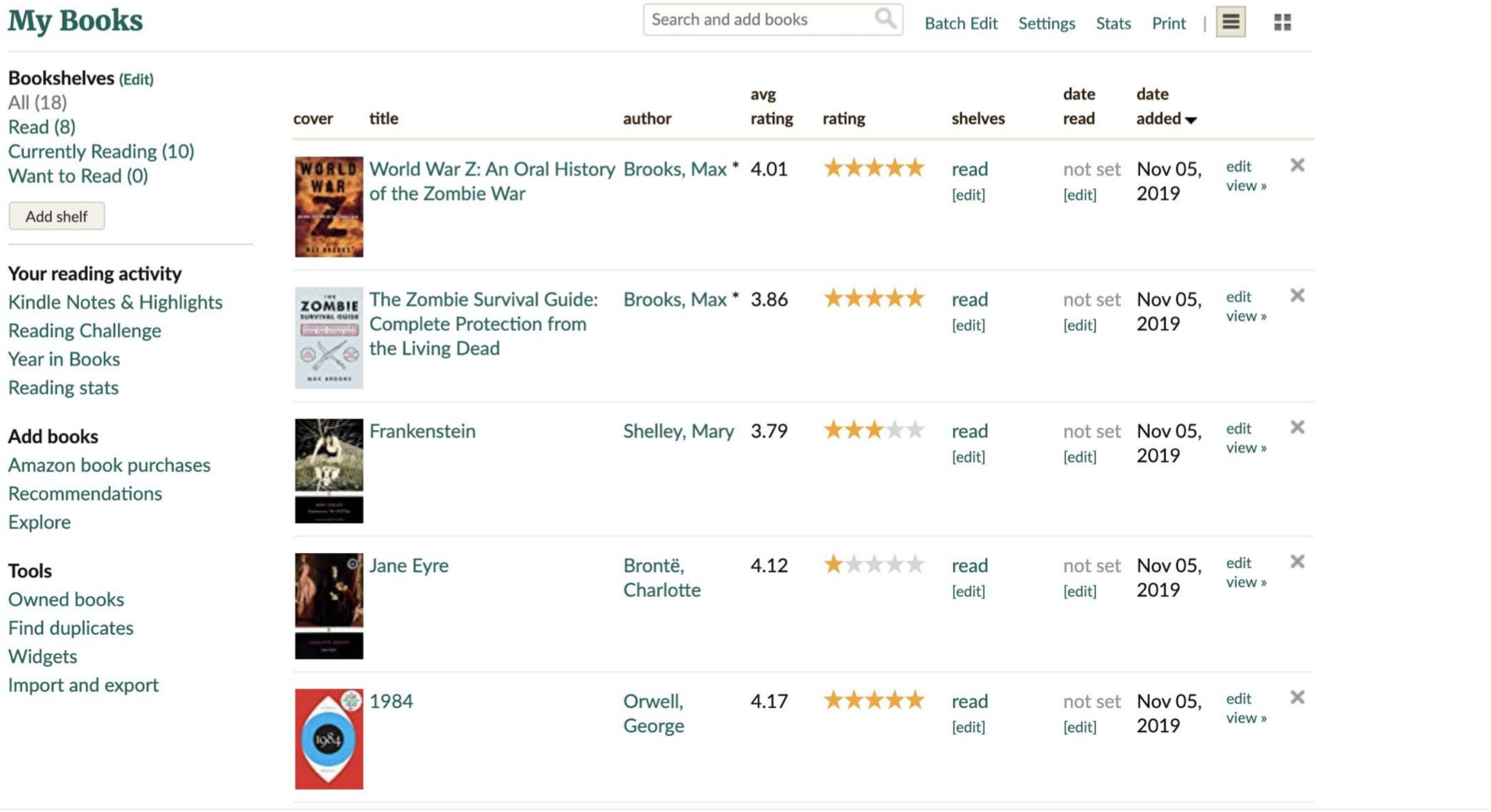
Widgets

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 »	✕

“Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”

- Linus Torvalds



My Books

Bookshelves (Edit)

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

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

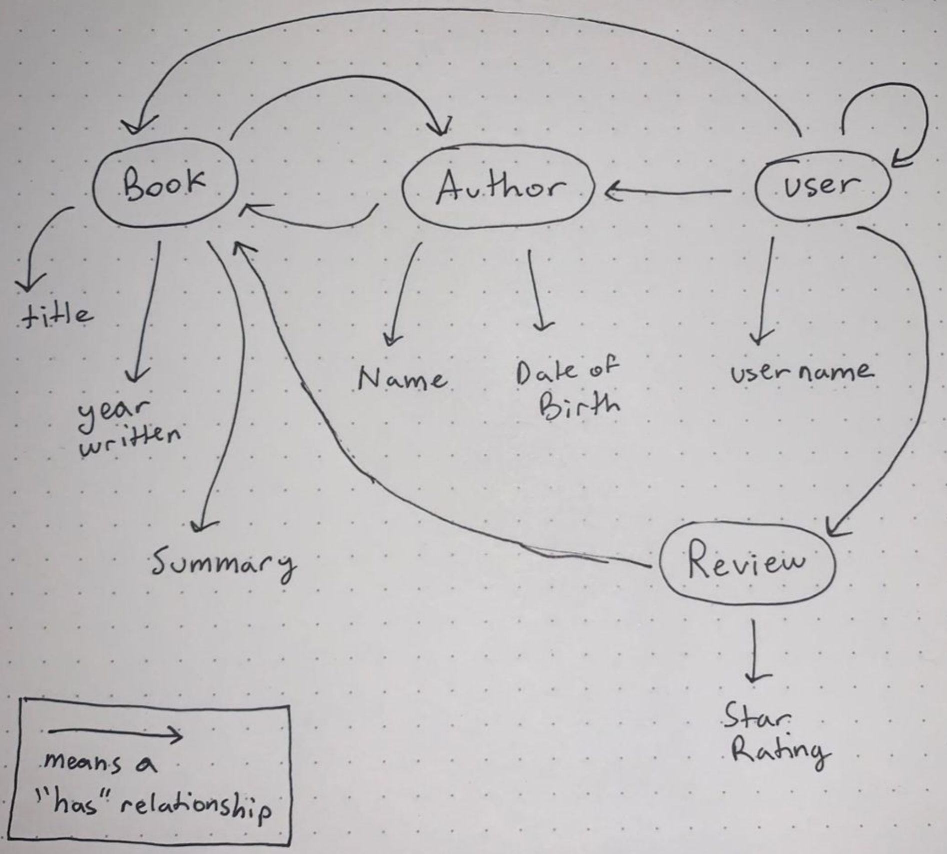
- Books
- Name
- Date of Birth

Users

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

Reviews

- Book
- Stars


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

REST API

- `GET /books`
- `GET /authors`
- `GET /users`

REST API

- `GET /books`
- `GET /authors`
- `GET /users`
- `GET /books/the+phantom+tollbooth`
- `GET /authors/mary+shelley`
- `GET /users/John+Rothfels`

REST API

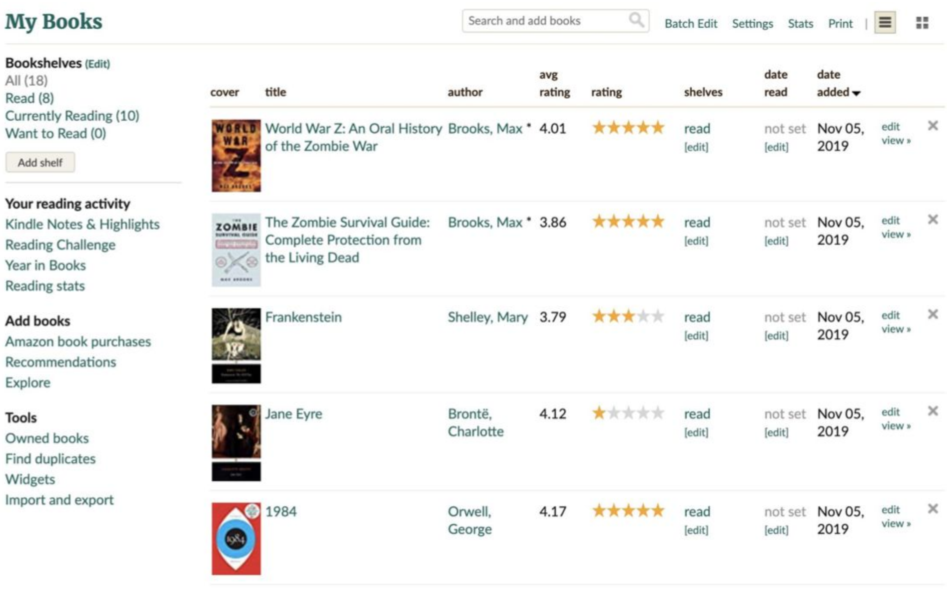
- `GET /books`
- `GET /authors`
- `GET /users`
- `GET /books/:book`
- `GET /authors/:author`
- `GET /users/:user`

REST API

- GET /books
- GET /authors
- GET /users
- GET /books/:book
- GET /authors/:author
- GET /users/:user
- POST /books/add

```
{ "Title": "CS188", "Author": "John Rothfels" }
```

How would you use this API?



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
- 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>
```

REST API

Following REST API design, if we build a client rendered application:

- It will have to make many HTTP requests to our backend
- Our backend needs to define a new routes/handlers for each resource
- Some requests will have to wait for others to complete first, so we know what to request
  - E.g. “get user” comes before “get books” because we have to know which books the user likes

⚠️ **As we scale / increase complexity, this can quickly lead to poor performance in our client rendered application!**

⁉️ What can we do to workaround this?

```
GET /landingpagedata/Ivan+Chub

{

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

}
```

REST API

We can create a special resource to serve a particular view with data!

Pros:

- Simple

Cons:

- Verbose
- Overfetching
- Underfetching
- Coupled to presentation layer

GraphQL

“A query language for your API”

What is GraphQL?

Convention on top of HTTP where you:

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

GraphQL: data types

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

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

GraphQL: data types

You may define custom types that contain fields.

```
type Person {

    name: String!

}
```

In GraphQL, this defines a new type called **`Person`**, with a single field called **`name`**. That field is of type **`String!`**

GraphQL: data types

Within your custom types, fields are either:

- scalar (**`Int`**, **`Boolean`**, …)
- a custom **`type`** you define
- a list/sequence/array of either of the above, represented by **`[]`**

We can represent non-nullability with an exclamation point **`!`**

- `Int`
- `Int!`
- `[Int]`
- `[Int!]`
- `[Int!]!`

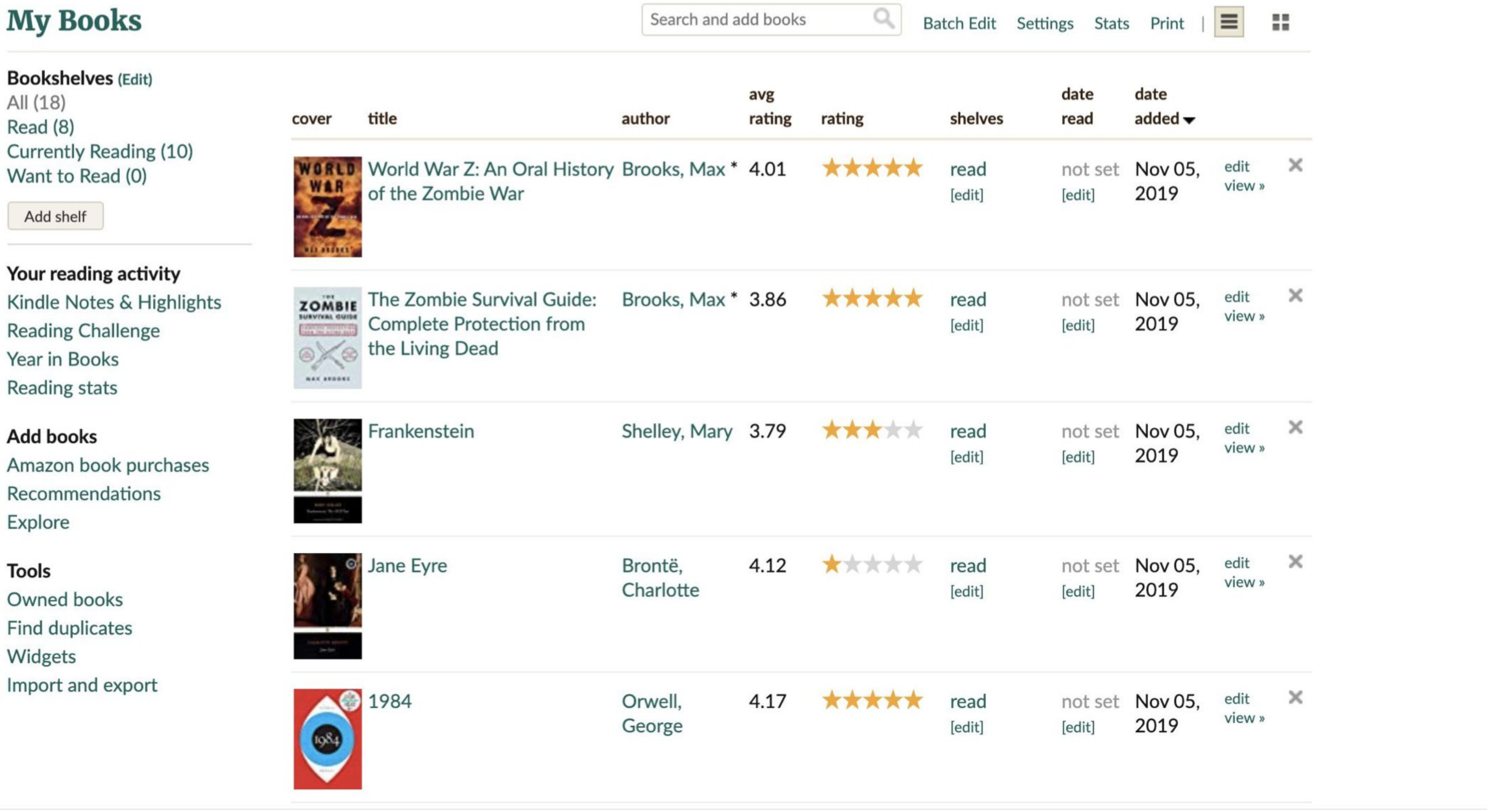
GraphQL: data types

You can also define custom **enum**s, 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!

    marketingDescription: String
    previousGeneration: Car

    ... add your own ...

}
```

How would we redesign the bookstore API with GraphQL?



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

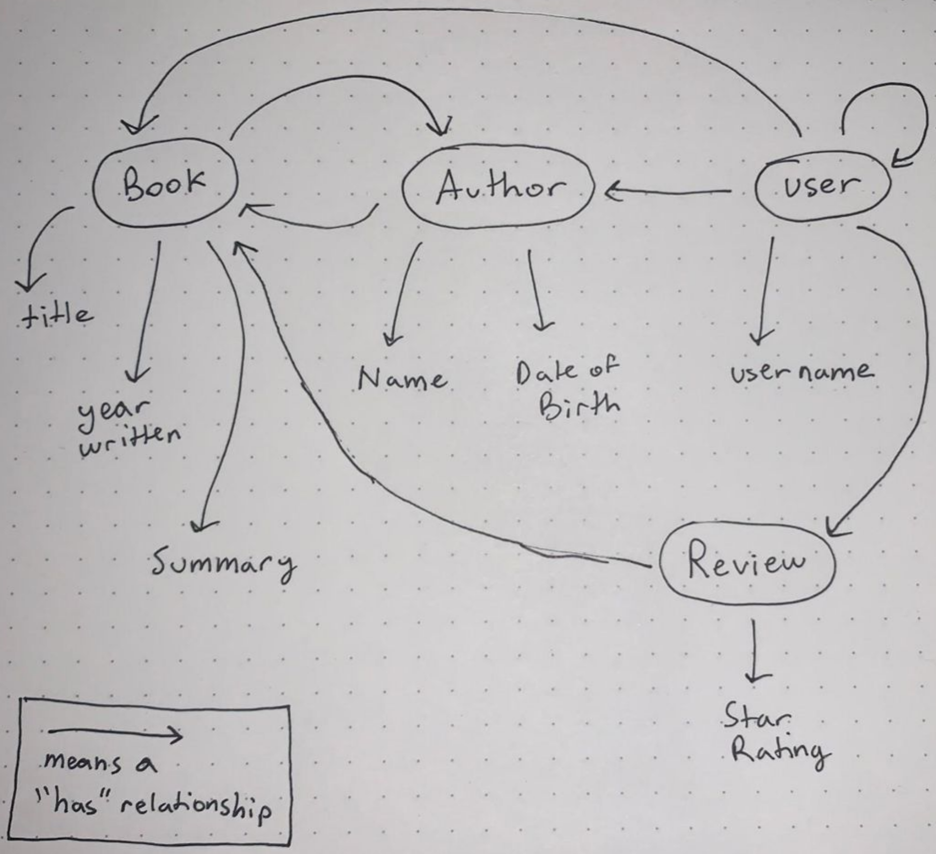
Add books

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
username
Review
Star Rating
means a "has" relationship

```
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!
}
```

GraphQL API

We can define the types of our data model. How do we define an API?

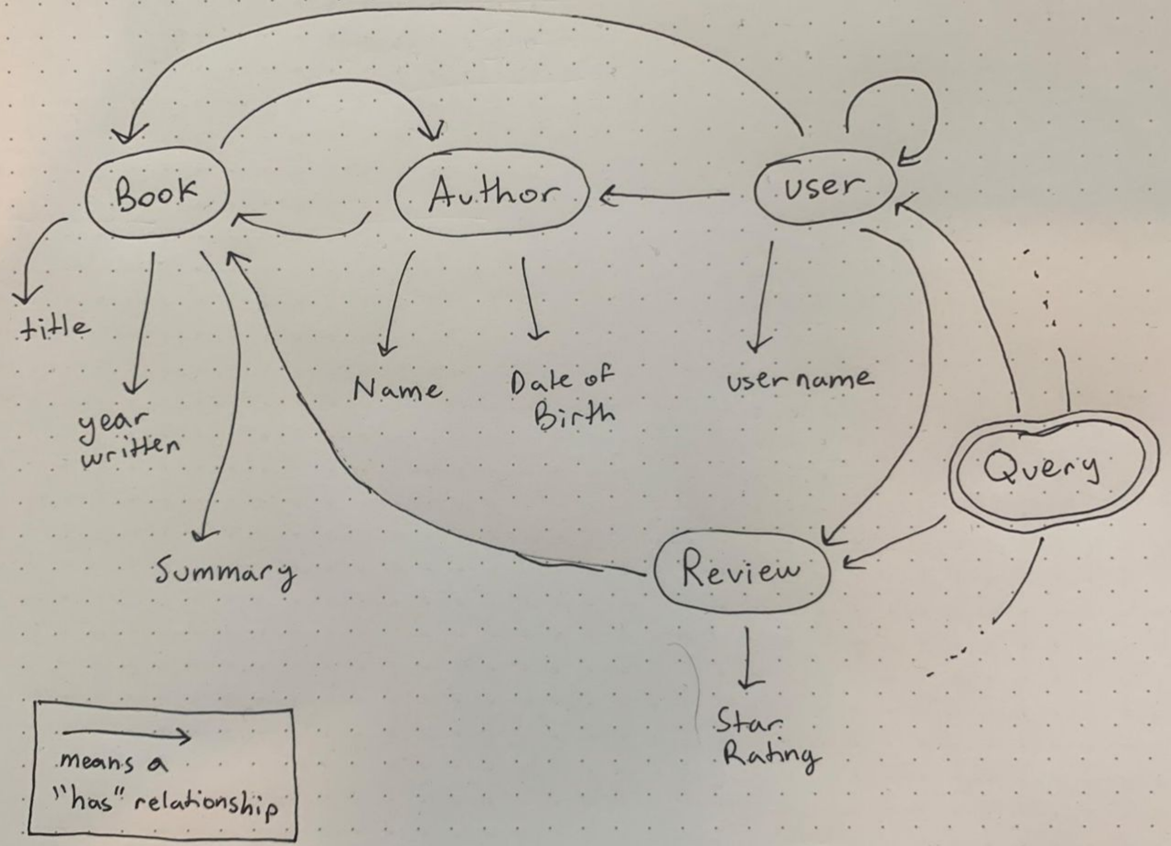
Declare a special type called **`Query`** (and another called **`Mutation`**). Your API is the combination of the two!

ℹ️ We use a special keyword called **`schema`** to declare an API. A **`schema`** (and it's Query/Mutation/types) are declared in a GraphQL schema file.

- `schema.graphql` in your starter project

GraphQL API

```
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
username
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
    }
  }
}
```

```
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
    }
  ]
}
```

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

```
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
    }
  }
}
```

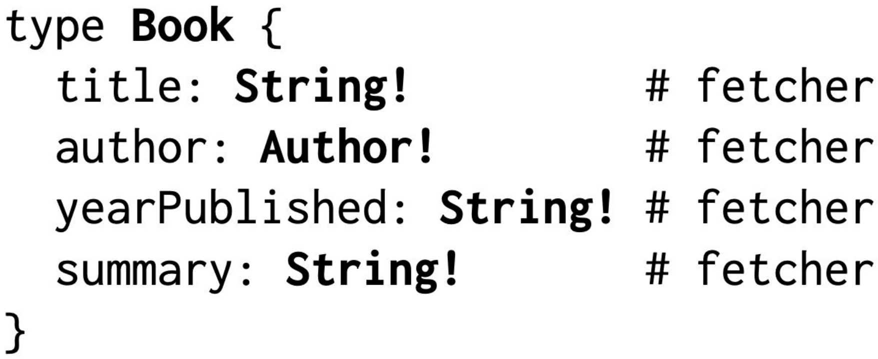
```
query HomePageQuery($userName: String!) {
  user(userName: $userName) {
    userName
    favoriteAuthor
    booksRead {
      title
      author
      yearPublished
      summary
    }
    friends {
      userName
    }
    reviews {
      book {
        title
      }
      stars
    }
  }
}
```

GraphQL API

How does it work?

A GraphQL schema defines an **interface**. Your server must implement that interface!

Every **`Type -> Field`** pair in your schema has a **fetcher**.



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

GraphQL API

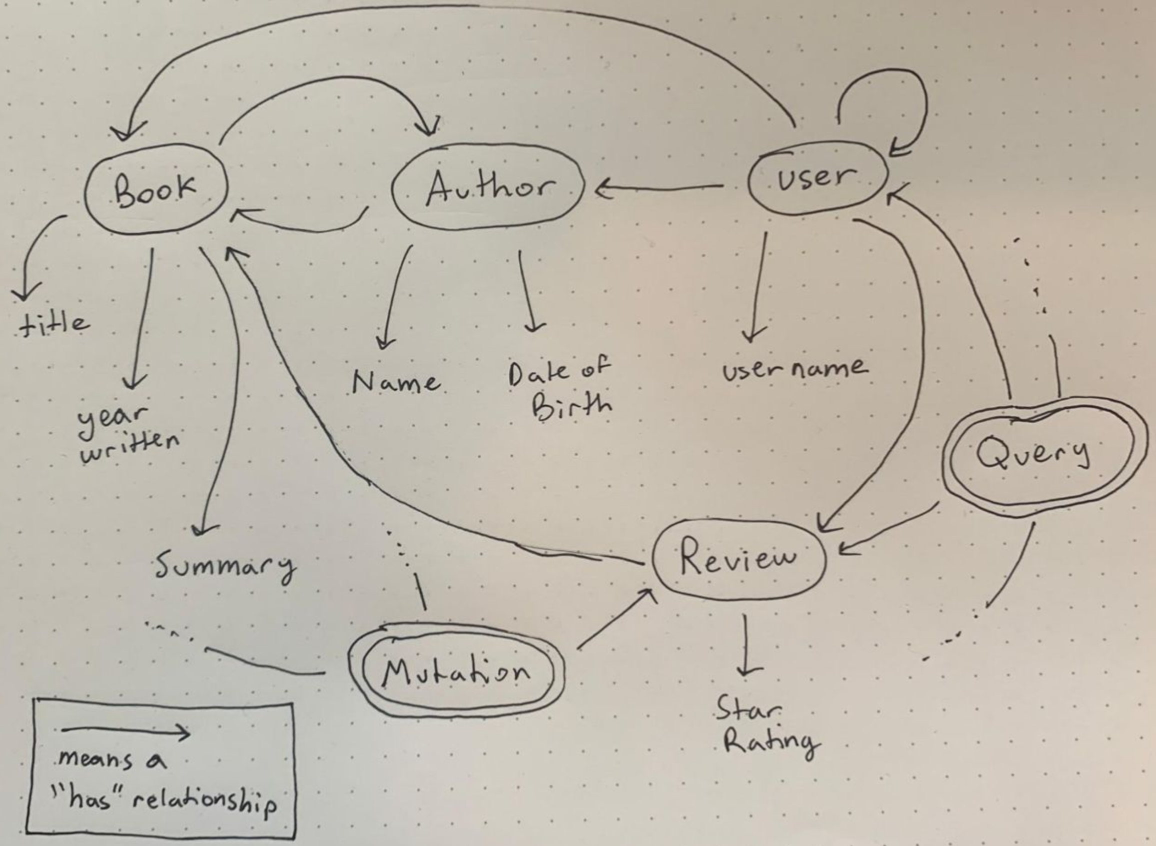
A **fetcher** is a function that returns:

- null
- a scalar (number, boolean, String)
- an object (JSON)
- a list of any of the above
- a **`Promise`** of any of the above (e.g. your fetcher must read from the database, network)

ℹ️ When you return an object, that object implicitly has a fetcher on it for every field. That fetcher just reads the field.

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
      }
    }
  }
}
```

GraphQL API

Recap: what is GraphQL?

- **Schema** (definition of types + query type + mutations)
- **Fetchers** (implementation of schema on the backend)
  - Typically you'll use a backend library to supply the graph. You just write functions (fetchers) to comply with your schema.
- **Queries** (you write these yourself)
  - You can use a frontend library to do the querying. It can help with server-side rendering, caching, MVC.

ℹ️ In this class, we'll use Apollo libraries on both the client and the server

GraphQL API: pros 😍

- Everything goes through one POST endpoint, meaning your API definition is **not verbose**. You define the nodes and edges of your data, GraphQL does the heavy lifting
- Changing frontend requirements no longer results in changes to the back end. You just change your query. This means **no more tight coupling**.
- **No more underfetching or overfetching**, you ask for precisely what you need in exactly one request.
- Your data is **STRICTLY defined**!
  - Your API has a built-in type system! We can code-generate TypeScript interfaces from your API.
  - Strict typing benefit cannot be overstated!!
- Plays SUPER nicely with frontend libraries like React
- Has a story for server->client communication (subscriptions)

GraphQL API: cons 😢

- Querying complexity increased over traditional REST
- Caching is more difficult
  - Normal REST endpoints use native HTTP caching
  - GraphQL requests are all POST, which don’t cache
- Error handling is a little funky
  - Some failed requests are supposed to return status code 200, with a message about the failure.

GraphQL API: cons

Does this seem 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?
  - Yes, if you’re not careful
- Security: how do you restrict what data different kinds of users can access?
  - Any can check context (e.g. current user) and make assertion. Your client should not request data the current user can’t access.
- Rate limiting: how does it work?

Lab tomorrow

- You will start building features with your project group.
  - Make sure your group is listed on this spreadsheet.
  - Make sure your group has a repository, and your teammates can commit to it. Email the TAs if you are having trouble.
- If you don't have a project group yet, **email me right away**.
- While working on your project:
  - Submit bugs/issues on Piazza. Help other students where you can.
  - Use bug filing instructions (and troubleshooting guide). Will post/pin to Piazza.
  - Pair with your project teammates! Look at other people's code, the course website for examples.
- I will check on your project status through GitHub commits, not during lab!
  - ⚠️ If I don't see you committing code to your repository, it can impact your grade at the end of the quarter!