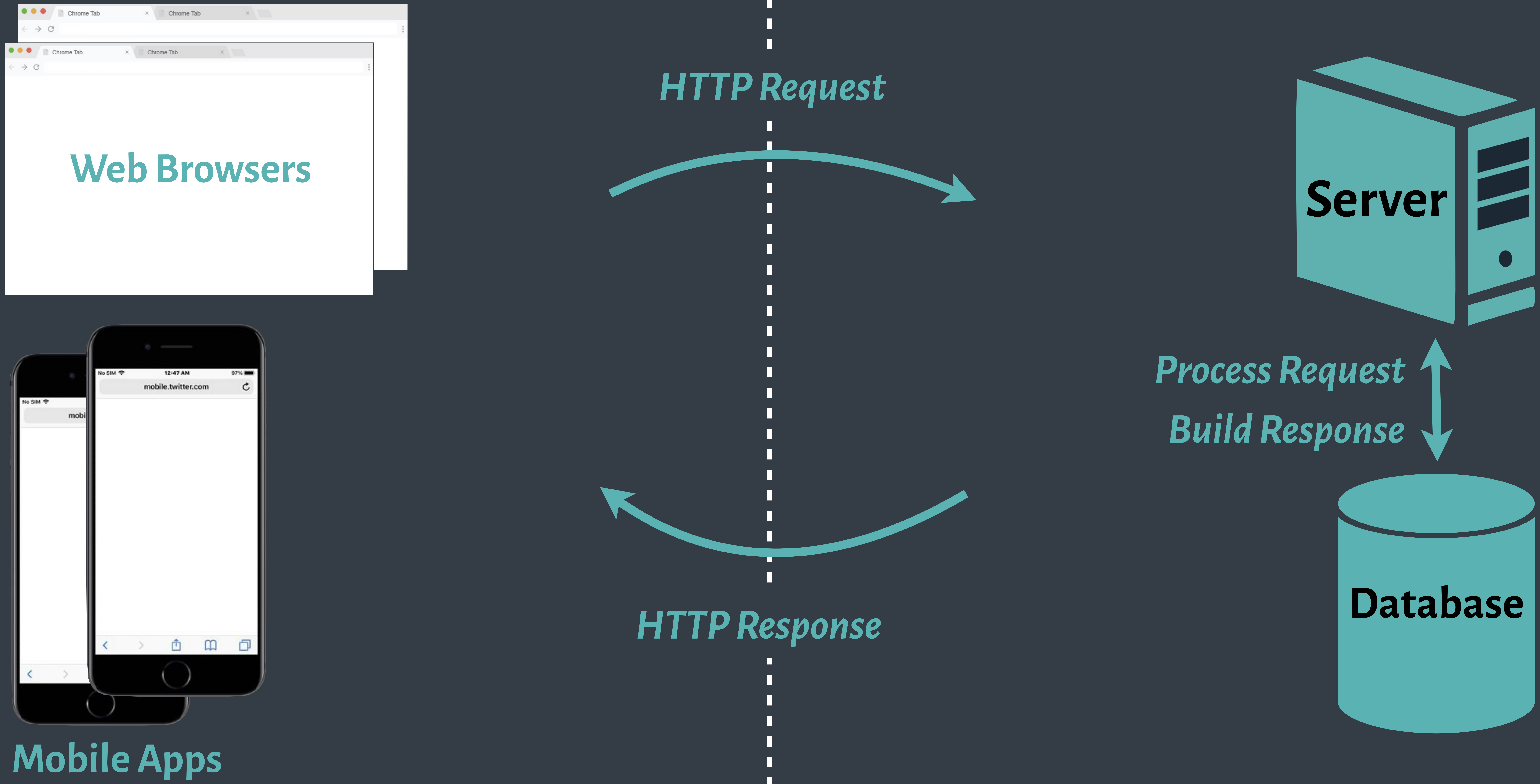
6.1040: Software Studio

Intro to Backend Design

Arvind Satyanarayan & Daniel Jackson

Client Side

Server Side



Web Browsers

Mobile Apps

HTTP Request

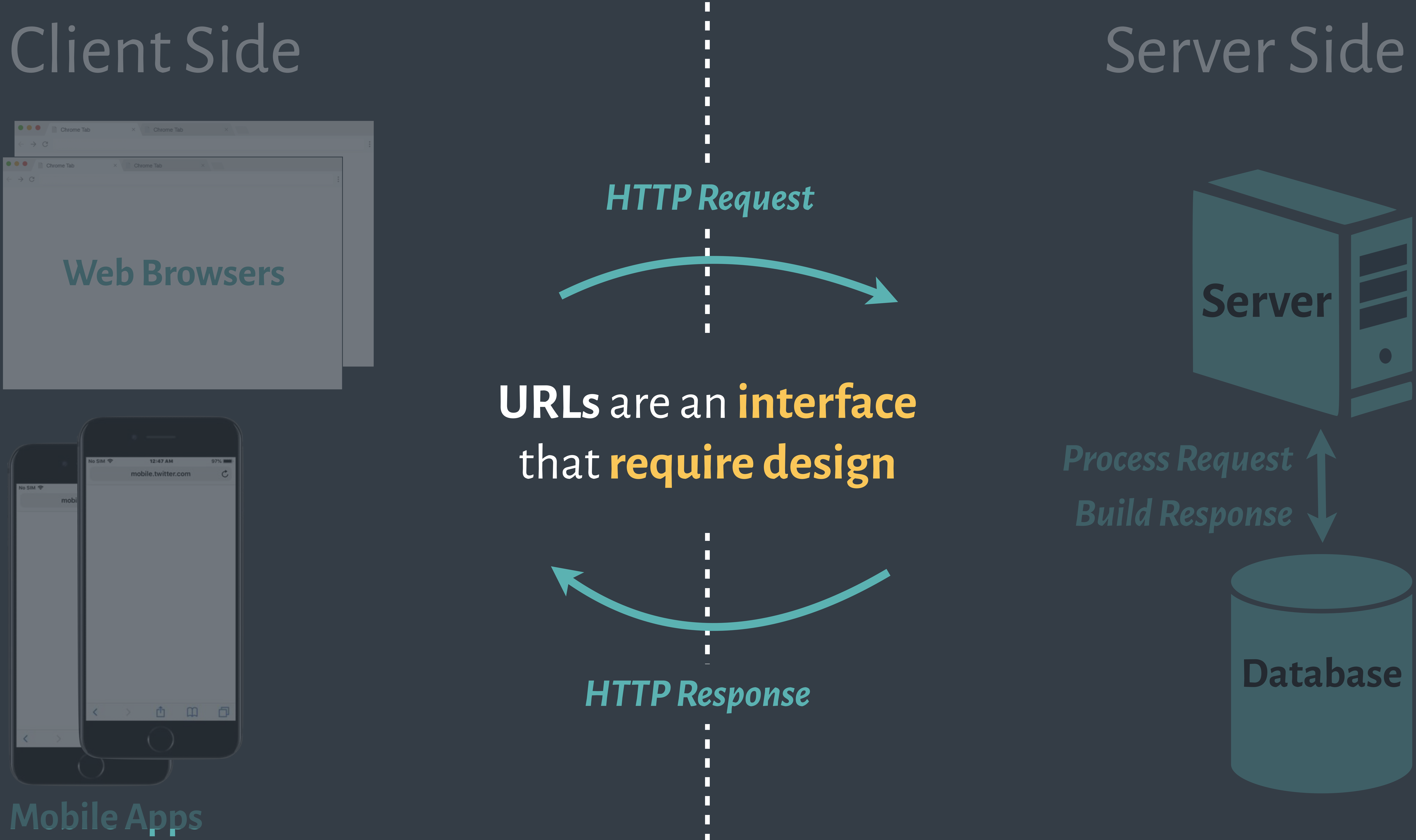
HTTP Response

Server

Process Request

Build Response

Database



Client Side

Server Side

Web Browsers

Mobile Apps

HTTP Request

HTTP Response

Server

Process Request

Build Response

Database

URLs are an **interface**
that **require design**

ANATOMY OF A URL

Protocol Host Path

http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya#topic-footer-buttons

Query Parameters Fragment

Representational Transfer

RESTful Design

State

RESTful Design

"Applying verbs to nouns"

GET /profs/arvind

GET /profs/arvind

Noun aka Resource

(U**R**L)

URL paths identify a *representation* of a resource

Profile page: `/profs/arvind.html`
Profile picture: `/profs/arvind.jpg`
Data structure: `/profs/arvind.json`

GET /profs/arvind

Use hierarchies to imply *structure*

collections

`/profs`
`/profs/reviews`
`/profs/arvind/reviews`
`/profs/arvind/reviews?n=5`

instances

`/profs/arvind`
`/profs/arvind/reviews/275`

GET /profs/arvind

Noun aka Resource
(U**R**L)

GET /profs/arvind

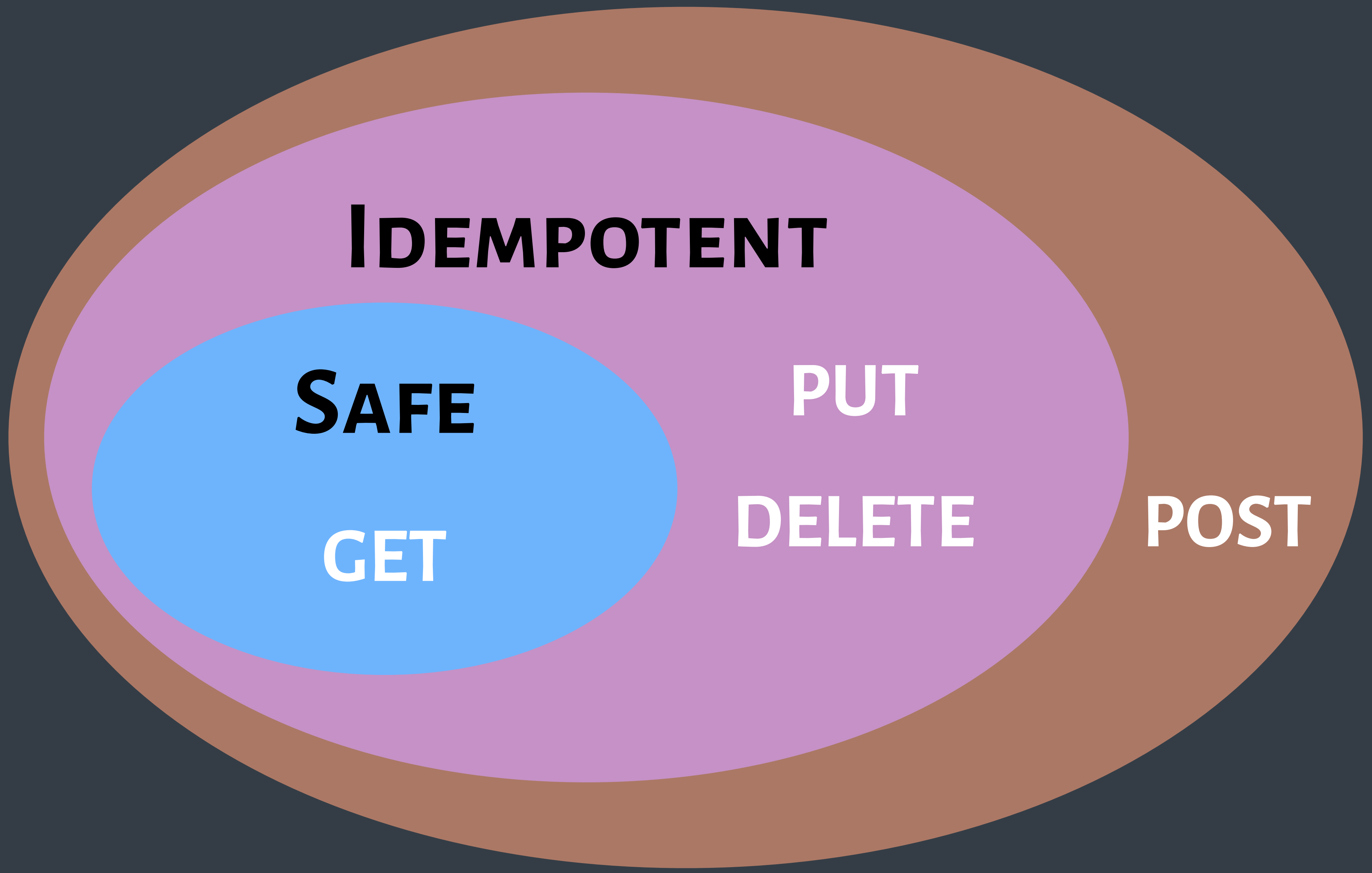
Verb	Noun aka Resource
(HTTP Method)	(U**R**L)

GET /profs/arvind

HTTP methods imply *different actions* on the *same resource*.

Create	`POST /profs/arvind/reviews`
Read	`GET /profs/arvind/reviews`
Update	`PUT /profs/arvind/reviews/4`
Delete	`DELETE /profs/arvind/reviews/5`

And help us think about *data safety*



IDEMPOTENT

SAFE

GET

PUT

DELETE

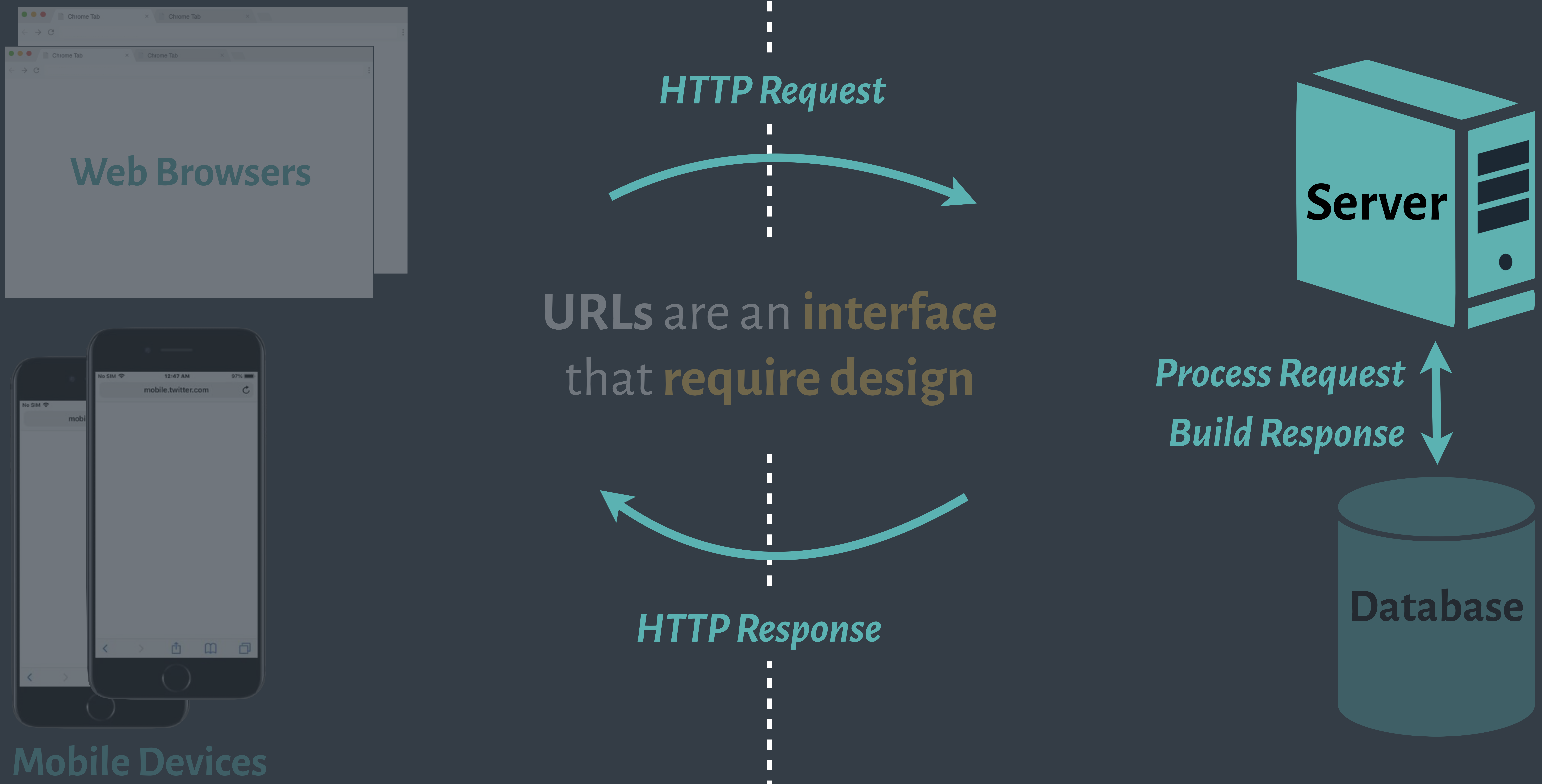
POST

Client Side
Server Side
Chrome Tab
Chrome Tab
Chrome Tab
Chrome Tab
Web Browsers
No SIM
12:47 AM
97%
mobile.twitter.com
mobi
Mobile Devices
HTTP Request
HTTP Response
Server
Process Request
Build Response
Database

URLs are an **interface** that **require design**

Client Side

Server Side



URLs are an **interface** that **require design**

Routing

(server.js)

```
let express = require("express");
let app = express();

let profs = require("./profs.js");
let classes = require("./classes.js");

app.use("/profs", profs);
app.use("/classes", profs);

app.listen(3000);

console.log("Listening on port 3000...");
```

(profs.js)

```
let express = require("express");
let router = router.Router();

router.get("/:name", function(req, res) {
  res.send("Hello " + req.params.name);
});

module.exports = router;
```

- ✓ **Ease of programming**: don't have to manually parse URLs.
- ✓ **Separation of concerns**: separate functions for different verbs (actions) on the same resource.
- ✓ **More modularity:** can split actions across files using a route prefix.
- ✓ **Safety and security:** undefined routes are rejected (404 error).

Templating

(server.js)

```
let express = require("express");
let app = express();
let mustache = require("express-handlebars");

app.engine("html", mustache());
app.set("view engine", "html");
app.set("views", "./views");

app.get("/say", function(req, res) {
  res.render("say", {
    message: req.query.msg
  });
});

app.listen(3000);
```

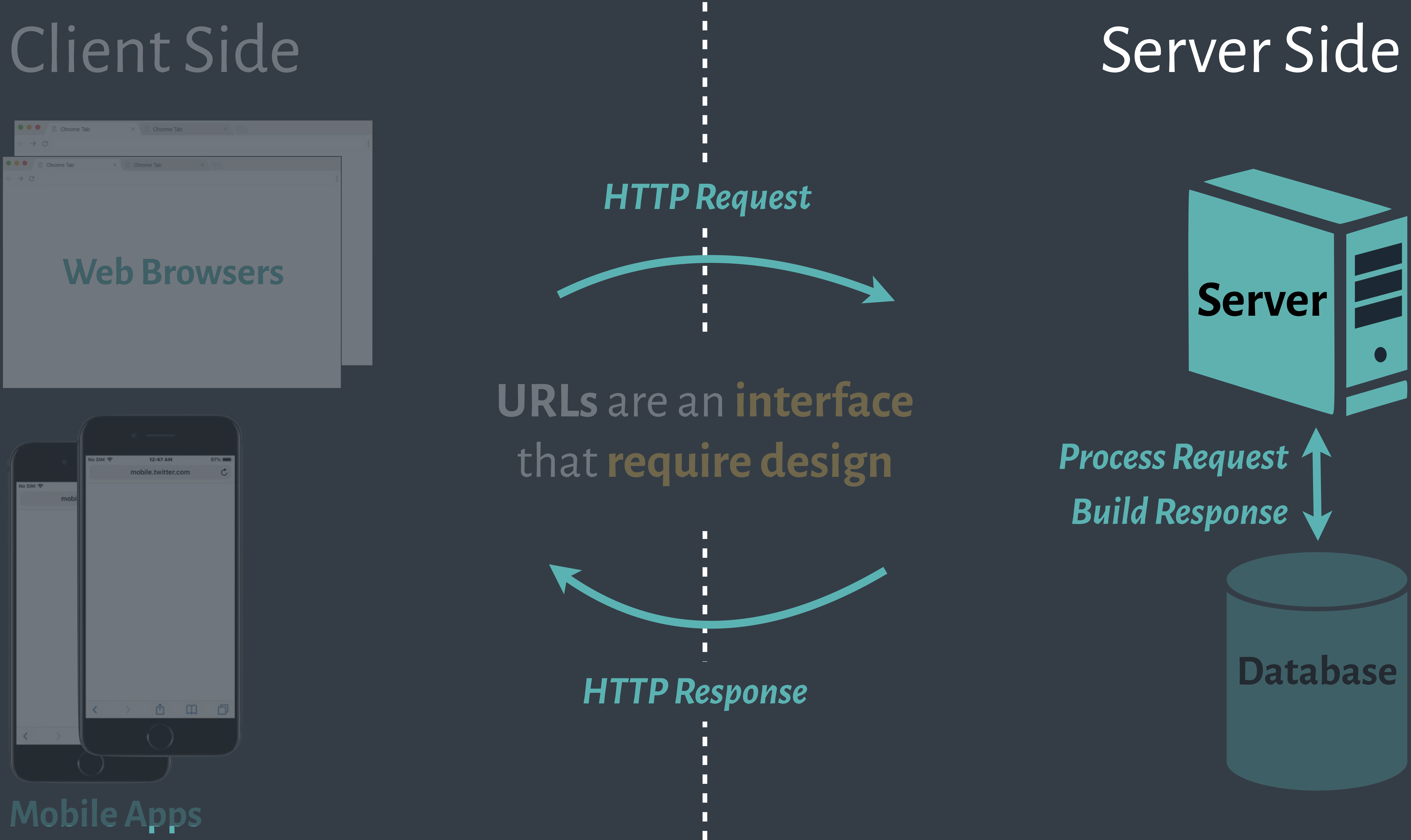
(say.html)

```
<html>
  <body>
    Alright, I'll say it: {{message}}
  </body>
</html>
```

- ✓ **Ease of programming**: No ugly string concatenation. Template engine (e.g., mustache) provides language constructs (e.g., iteration).
- ✓ **Separation of concerns**: presentation of content from semantics of action.
- ✓ **More modularity:** template "partials" can be reused across views.
- ✓ **Safety and security:** automatically escaping code to prevent injection/ XSS attacks.

Client Side

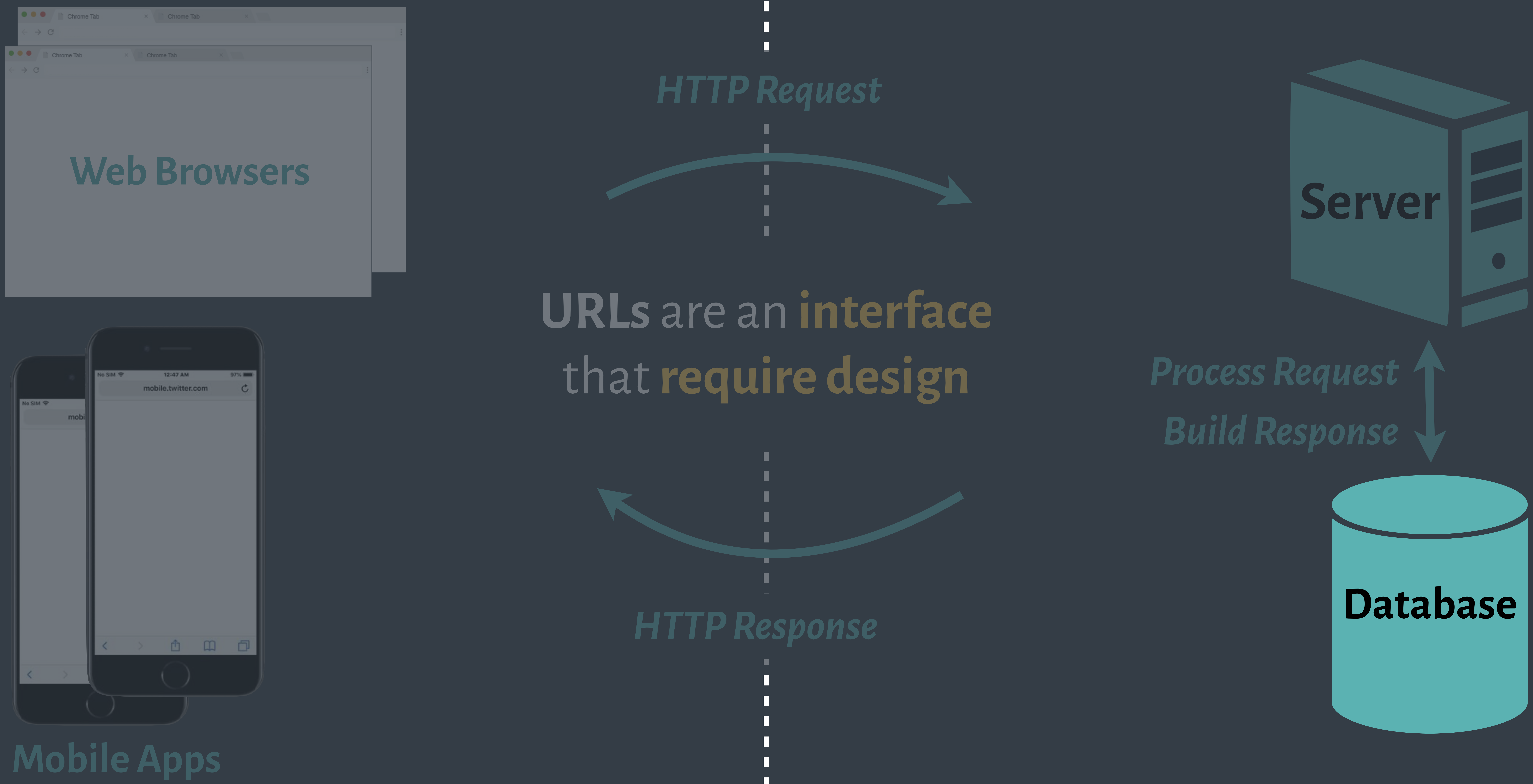
Server Side



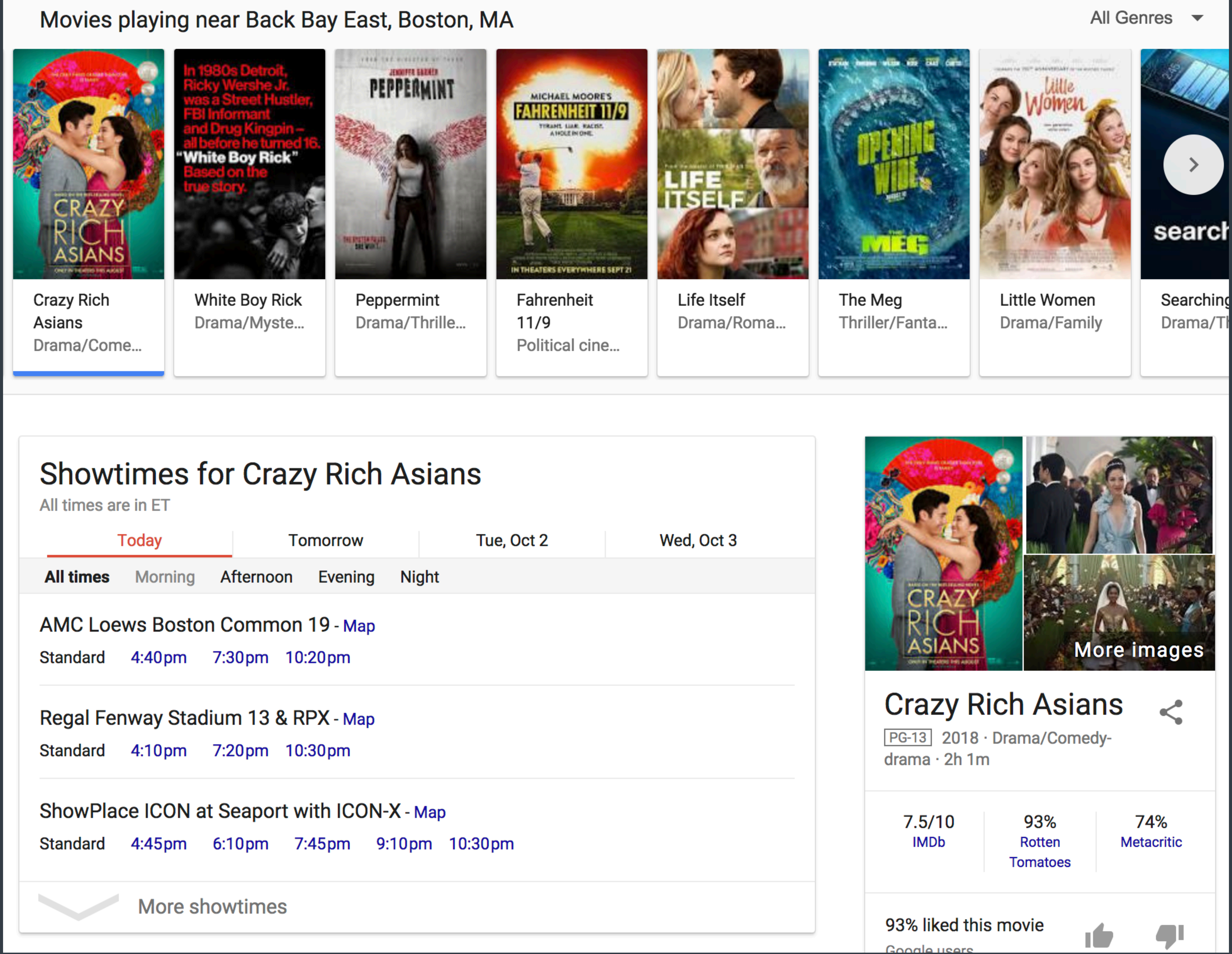
URLs are an **interface** that **require design**

Client Side

Server Side



URLs are an **interface** that **require design**



Movies playing near Back Bay East, Boston, MA

All Genres ▾

Crazy Rich Asians	Drama/Come...
White Boy Rick	Drama/Myste...
Peppermint	Drama/Thrille...
Fahrenheit 11/9	Political cine...
Life Itself	Drama/Roma...
The Meg	Thriller/Fanta...
Little Women	Drama/Family
Searching	Drama/Th

Showtimes for Crazy Rich Asians

All times are in ET

Today | Tomorrow | Tue, Oct 2 | Wed, Oct 3

All times Morning Afternoon Evening Night

AMC Loews Boston Common 19 - Map

Standard 4:40pm 7:30pm 10:20pm

Regal Fenway Stadium 13 & RPX - Map

Standard 4:10pm 7:20pm 10:30pm

ShowPlace ICON at Seaport with ICON-X - Map

Standard 4:45pm 6:10pm 7:45pm 9:10pm 10:30pm

More showtimes

More images

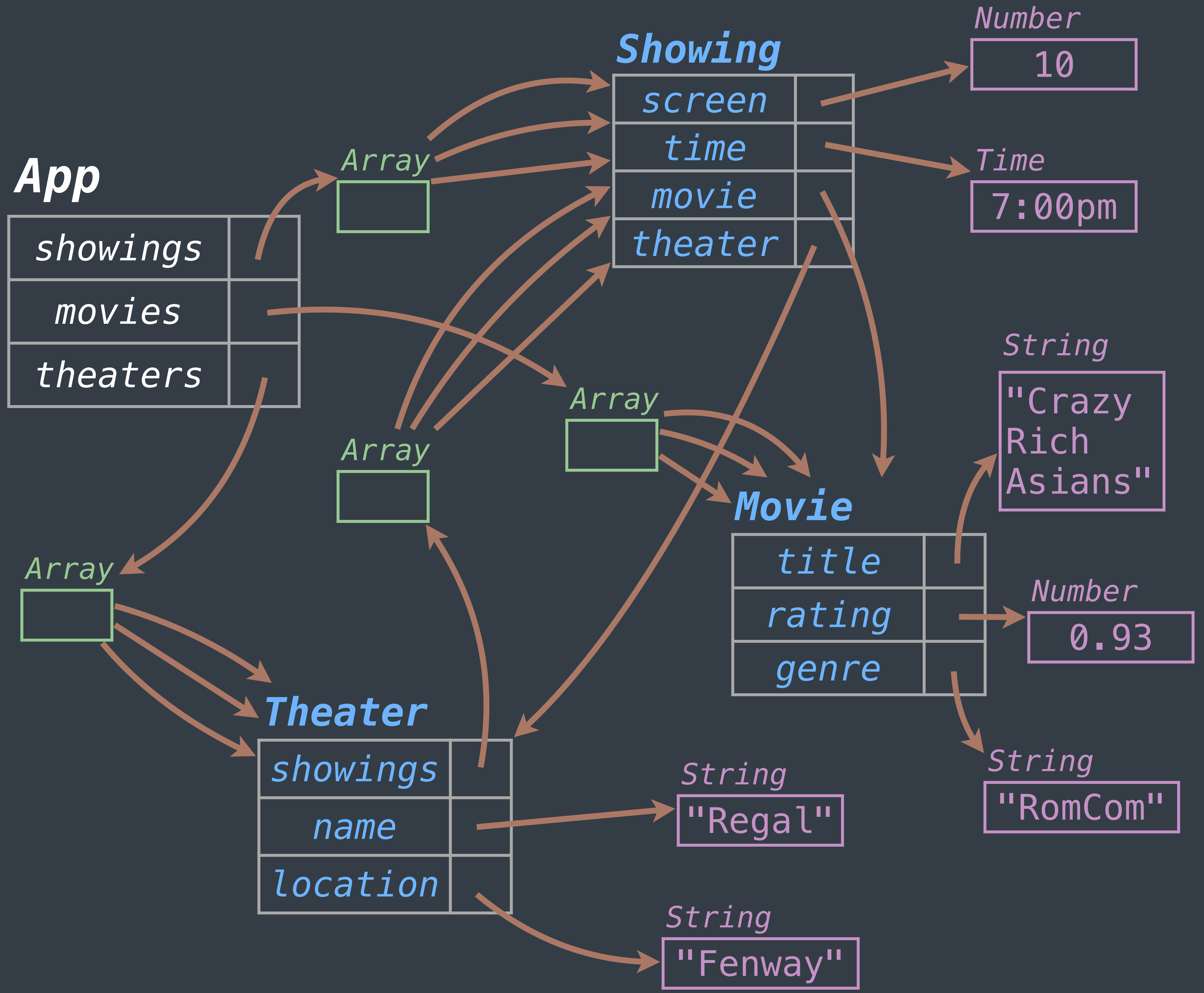
Crazy Rich Asians

PG-13 2018 · Drama/Comedy-drama · 2h 1m

7.5/10	93%	74%
IMDb	Rotten Tomatoes	Metacritic

93% liked this movie

Google users



Object Model

Application root references collections of class instances that describe primitive data.

- ✓ Quick to prototype.
- ✓ Easy to experiment with arbitrary data structures.
- ✗ Refactoring is difficult.
- ✗ No advanced querying: can only iterate over collections, follow references.

Relational Model (SQL)

Showings

id	*theater*	*screen*	*movie*	*time*
1	3	5	2	7:00pm
..	...	...	...	

Theaters

id	*name*	*location*
...	...	...
3	"Regal"	"Fenway"

Movies

id	*title*	*rating*	*genre*
...	...	...	
2	“Crazy Rich	“PG–13”	“RomCom”

Relations (aka tables) of attributes (aka columns) and tuples (aka rows).

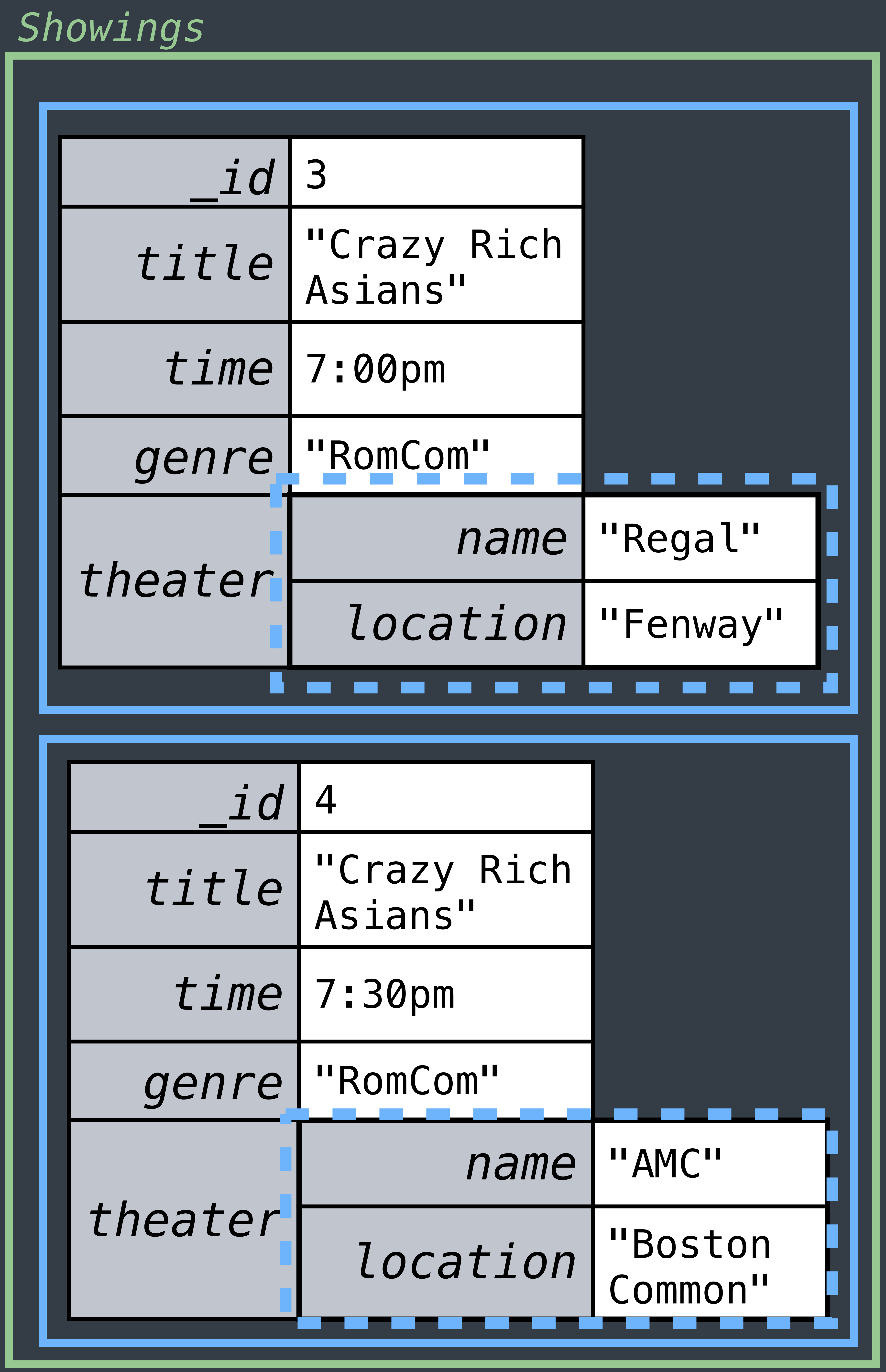
✓ Standardized query language (SQL) regardless of backend engine (MySQL, PostgreSQL, SQLite, …).

✓ Relational theory encourages better separation of concerns (called "normalization").

✓ Over 40 years of research into performance and robustness (indexing, transactions, integrity, …).

✗ (Until recently) did not offer JSON types.

✗ (Until recently) difficult to scale horizontally. Vertical scaling (i.e., make server more powerful) was the easiest option.



Showings

_id	3
title	"Crazy Rich Asians"
time	7:00pm
genre	"RomCom"
theater	name: "Regal" location: "Fenway"

_id	4
title	"Crazy Rich Asians"
time	7:30pm
genre	"RomCom"
theater	name: "AMC" location: "Boston Common"

NoSQL

"Not Only SQL"

Collections of *nested* documents (or graph structures).

- ✓ Quick to prototype (documents stored as JSON).
- ✓ Easy to experiment with arbitrary data structures.
- ✓ Pattern matching by document structure.
- ✓ *Horizontal* performance (i.e., many less-powerful servers, rather than a single very powerful one).
- ✗ No standardized query language.
- ✗ Embedded documents = easier to make poor design decisions.
- ✗ (Until recently) no references between collections: complexity of lookups occurs at the application level.

MongoDB CRUD Operations

```
db.showings.insertOne({})
db.showings.insertMany([{}, {}, ...])
```

```
{
  "_id": ObjectId(),
  "title": "Crazy Rich Asians",
  "genre": "RomCom",
  "showtime": Date("2022-10-07 15:30"),
  "theater": {
    "name": "AMC",
    "location": "Boston Common"
  }
}
```

Documents are JSON-like structures ("BSON") that offers additional data types like `Date`, `RegExp`, or binary data.

Every document must have an `_id`, and it must be unique within the collection.

`_id` is generated automatically by MongoDB via `ObjectId` (you can override it, but you really shouldn't!).

MongoDB CRUD Operations

```
db.showings.insertOne({})
db.showings.insertMany([{}, {}, ...])

db.showings.findOne({})
db.showings.find({})

{"title": "Crazy Rich Asians"}

{
  "theater": {
    "name": "AMC"
  }
}

{
  "title": "Crazy Rich Asians",
  "theater.name": "AMC"
}
```

```
{"$or": [
  {"title": "Crazy Rich Asians"},
  {"theater.name": "AMC"}
]}

{"theater.name": {
  "$in": ["AMC", "Regal"]
}}

{"showtime": {
  "$gte": Date("2022-10-07")
  "$lte": Date("2022-10-10")
}}
```

MongoDB CRUD Operations

```
db.showings.insertOne({})
db.showings.insertMany([{}, {}, ...])

db.showings.findOne({})
db.showings.find({})

db.showings.updateOne({}, {"$set": ...})
db.showings.updateMany({}, {"$set": ...})
db.showings.replaceOne({}, {})

db.showings.deleteOne({})
db.showings.deleteMany({})
db.showings.drop()
```

Mongoose's API will handle these for you pretty transparently.

Multiple Collections vs. Embedded Documents

```
db.theaters.insertOne({
  "_id": 1, "name": "AMC", ...
})


db.movies.insertOne({
  "_id": 3,
  "title": "Crazy Rich Asians",
  ...
})


db.showings.insertOne({
 "_id": 5, "theater": 1, "movie": 3,
   "showtime": Date()
})
```

```
db.movies.insertOne({
  "_id": 3,
  "title": "Crazy Rich Asians",
  "showings": [
   {
    "theater": {"name": "AMC", ...},
    "showtime": Date()
   }
  ]
})
```

Multiple Collections vs. Embedded Documents

✓ More flexible querying (e.g., sorting results)

✘ Separate collections require more work: you have to manually join things together.

```
amc = db.theaters.find({"name": "AMC"})
amc_ids = amc.map(t => t._id)
movies = db.movies.find({
  "theater": "$in": amc_ids
})
```

✘ Limited to insertion order

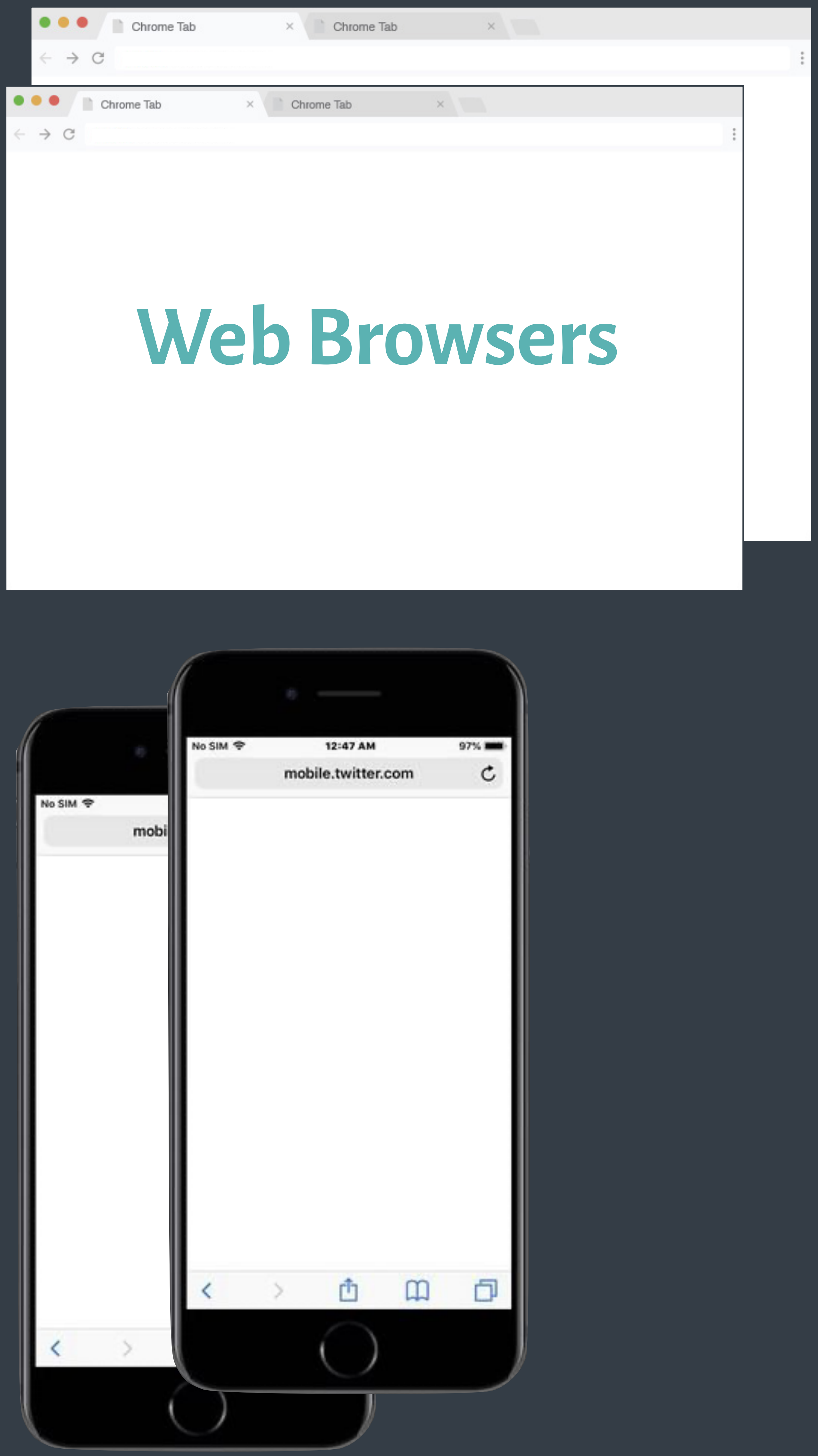
✘ Each document (including all embedded documents, arrays, etc) cannot be larger than 16MB.

```
{"theater.name": "AMC"}
```

1. How many embedded objects do you have? One? A few? Many?
2. Does the embedded document relate to any other collections?
3. How often will you need the embedded document *without* the parent, or vice versa?

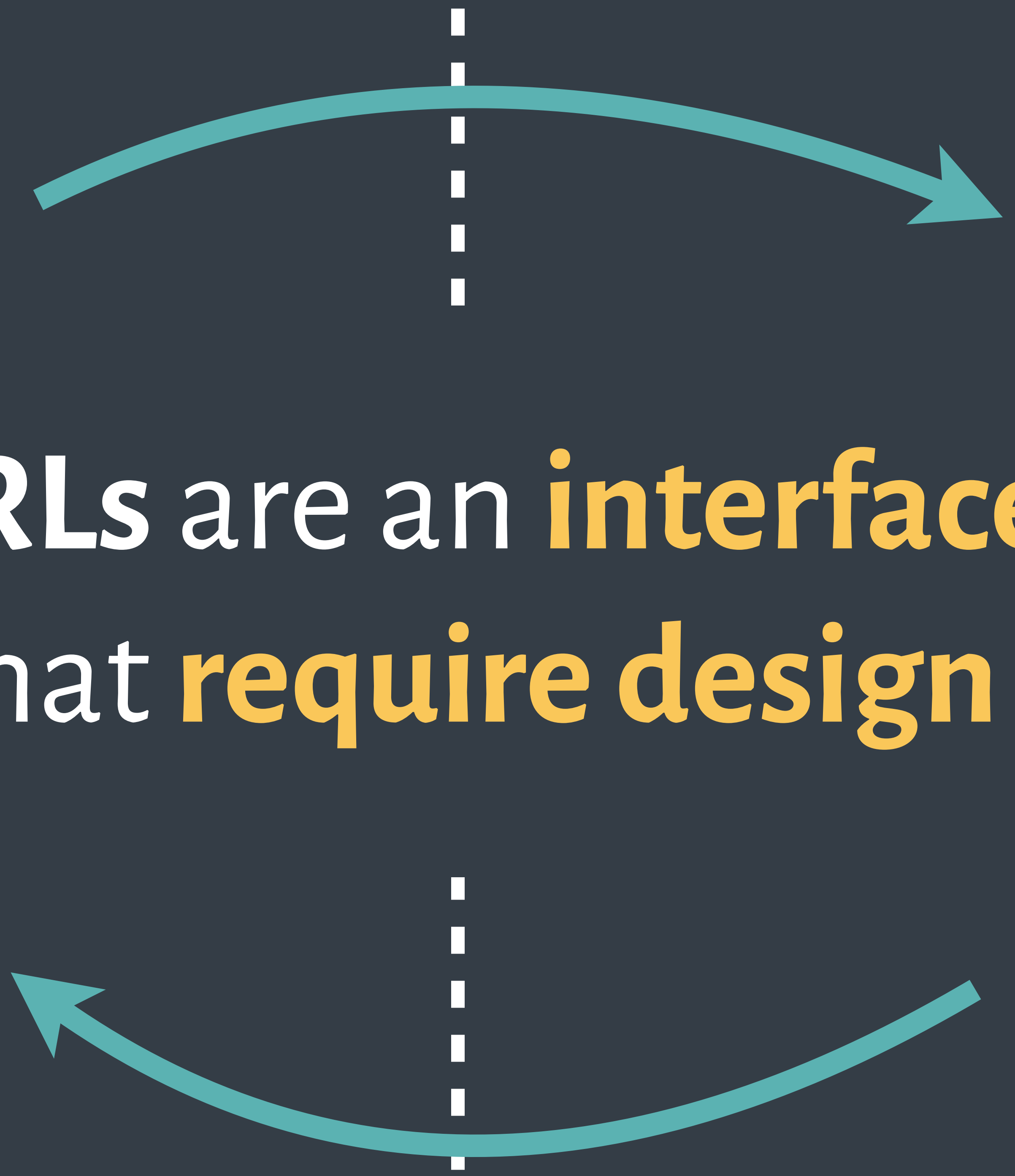
Client Side

Server Side



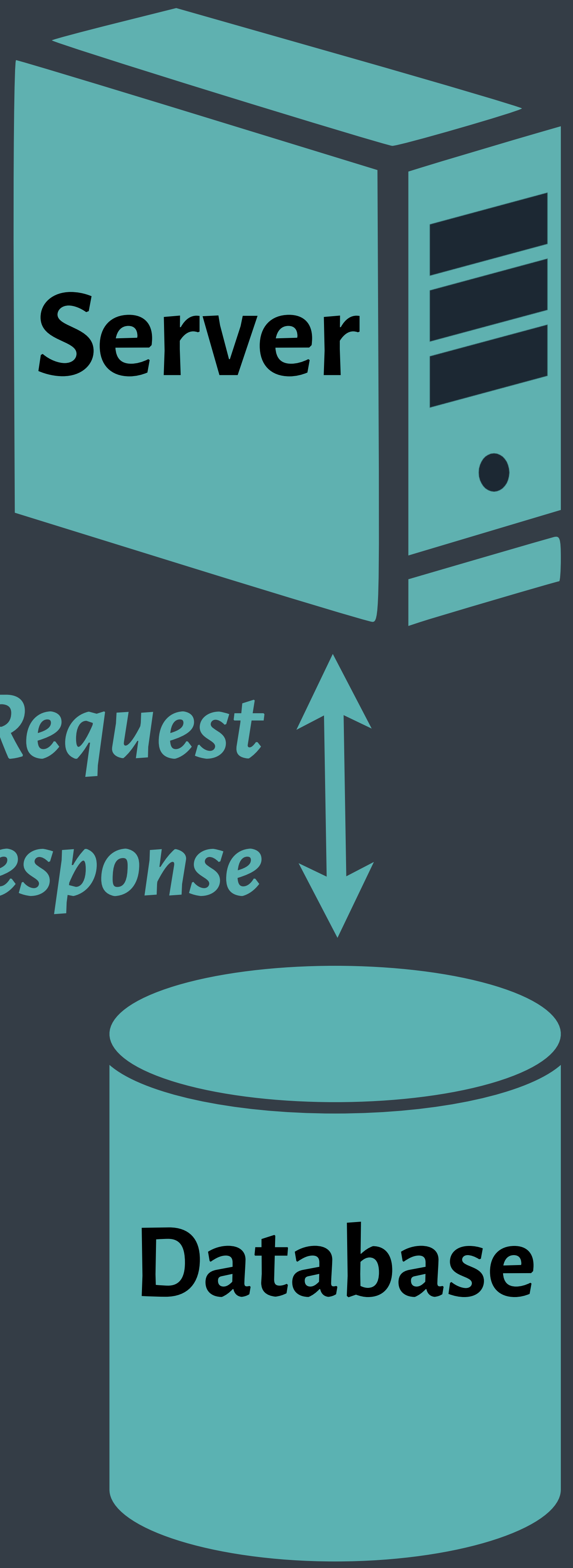
Mobile Apps

HTTP Request



URLs are an **interface**
that **require design**

HTTP Response



Fill Out Your MUD Cards

http://tiny.cc/61040-fa22-mud

Give us Feedback

http://tiny.cc/61040-fa22-feedback

RSVP to Reading Group

http://tiny.cc/61040-fa22-pizza