

# CS 4530: Fundamentals of Software Engineering

## Module 4: Web Applications

---

Adeel Bhutta, Rob Simmons, and Mitch Wand

Khoury College of Computer Sciences

© 2026 Released under the [CC BY-SA](#) license

# Learning Goals for this Lesson

---

At the end of this lesson, you should be able to

- Explain the role of “client” and “server” in the context of web application programming
- Explain the role of REST versus WebSocket communication
- Describe the fundamental differences between the three layers of the controller, service, and repository layers in a C-S-R architecture
- Understand how the C-S-R architecture works in the context of a basic Express application
- Be able to answer an interview question about “business logic,” “horizontal and vertical scaling,” or “microservices”

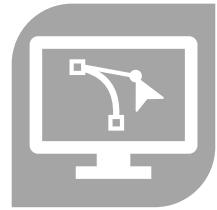
# So, software engineering must encompass:



PEOPLE



PROCESSES

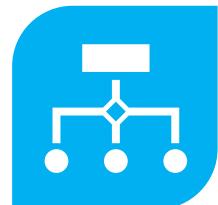


PROGRAMS

PLANNING



ORGANIZING



IMPLEMENTING



We're gonna be  
stuck over here for  
a bit.

# Web Applications are Distributed Systems

---

Distributed systems are hard!

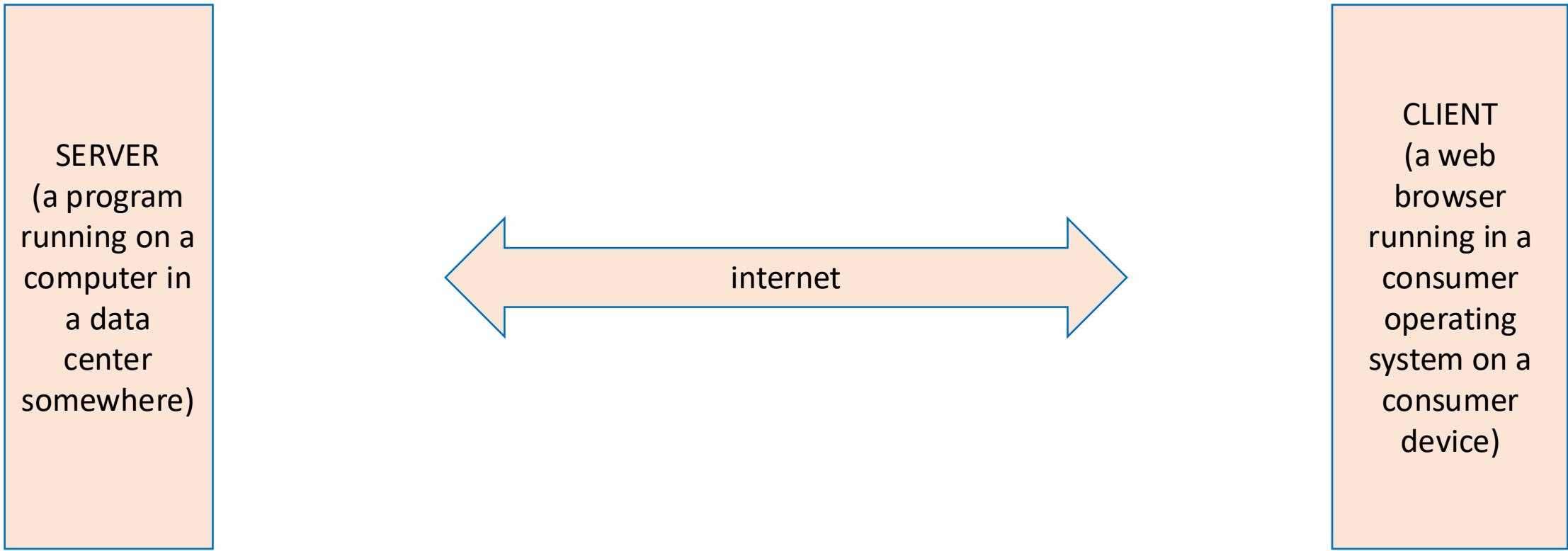
- Web applications are designed to only be *kinda* difficult-to-build distributed systems
- Most of this lecture is bad advice if you're Google, Netflix, or Amazon

Web applications are distributed systems *because*

1. You don't live in the cloud
2. Scalability: Netflix needs at *least* two computers

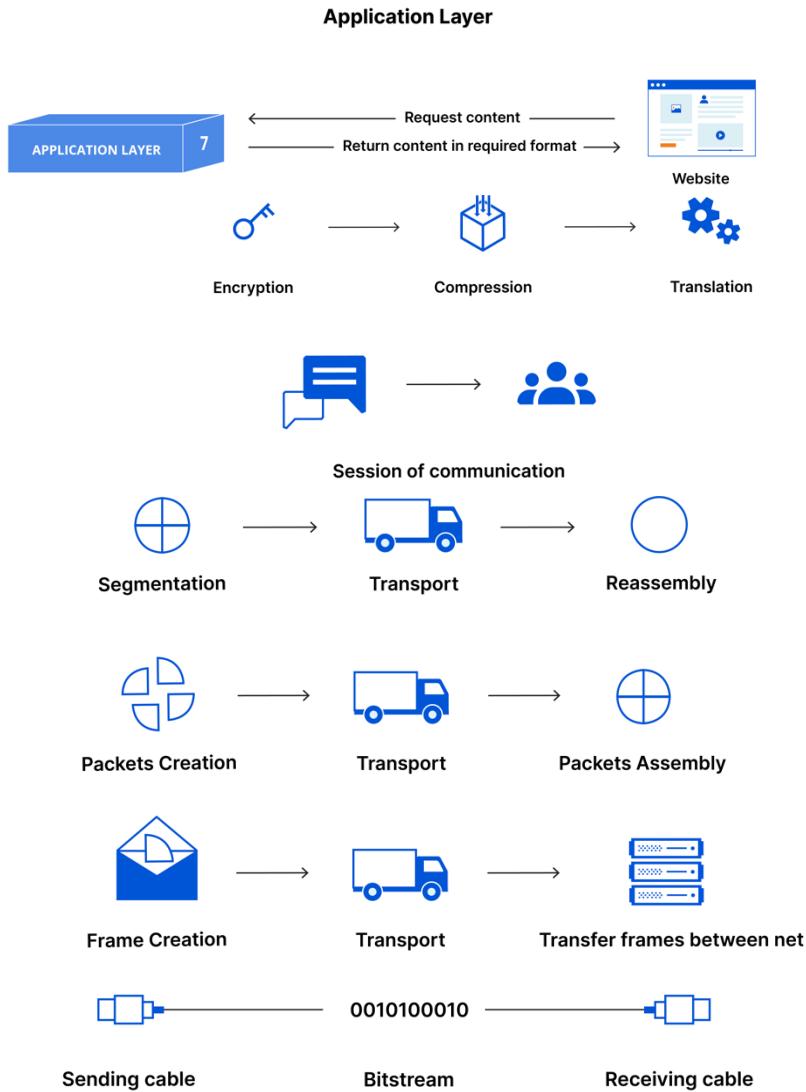
# An Insultingly Shallow Intro to Networking

---



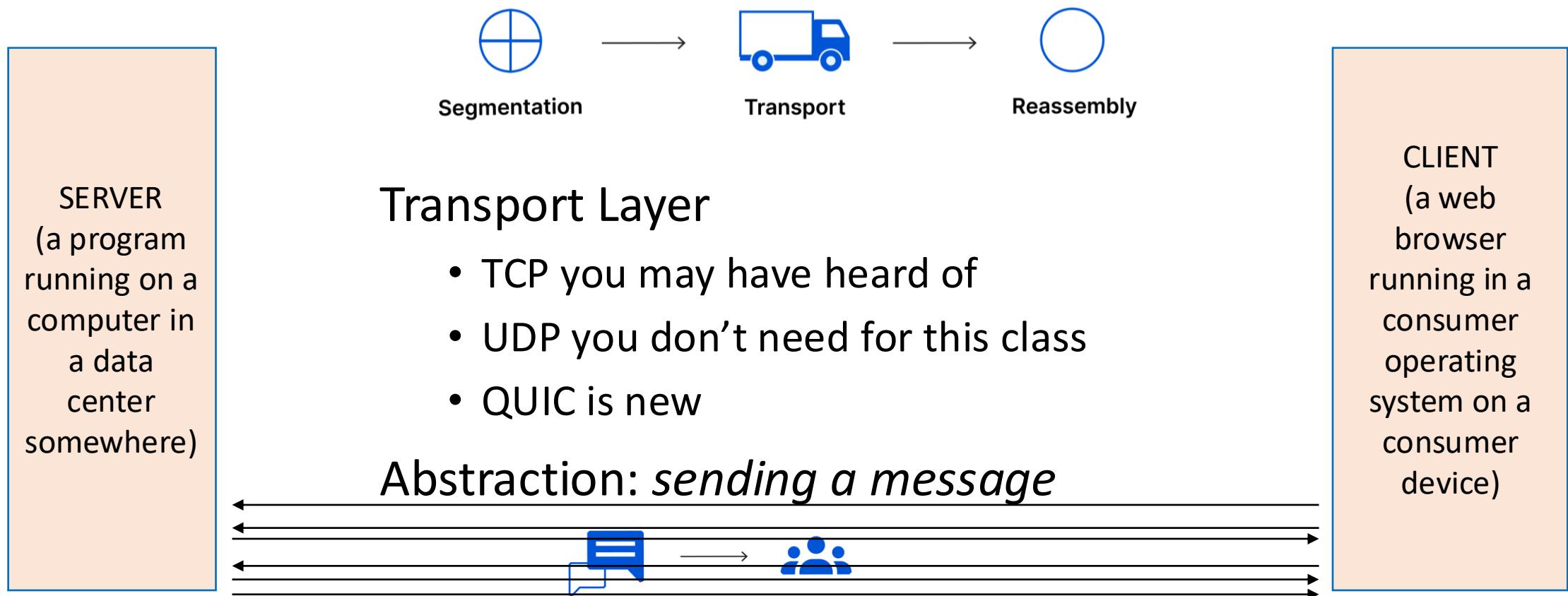
# An Insultingly Shallow Intro to Networking

SERVER  
(a program running on a computer in a data center somewhere)



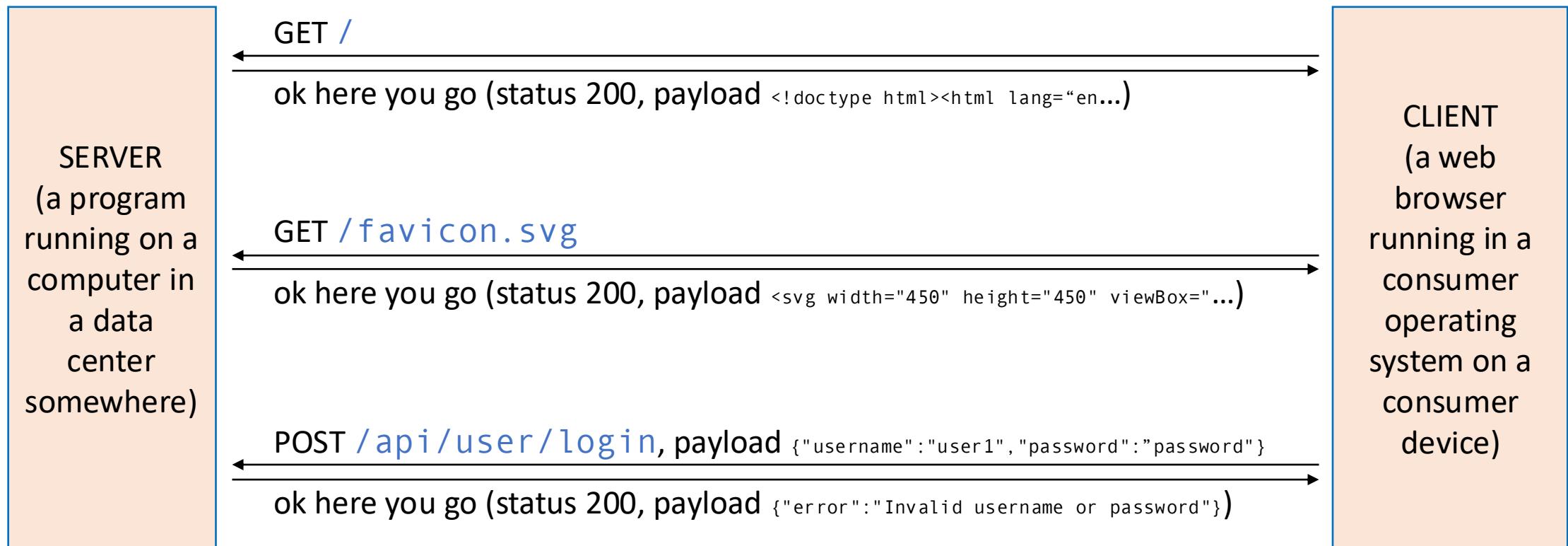
CLIENT  
(a web browser running in a consumer operating system on a consumer device)

# An Insultingly Shallow Intro to Networking



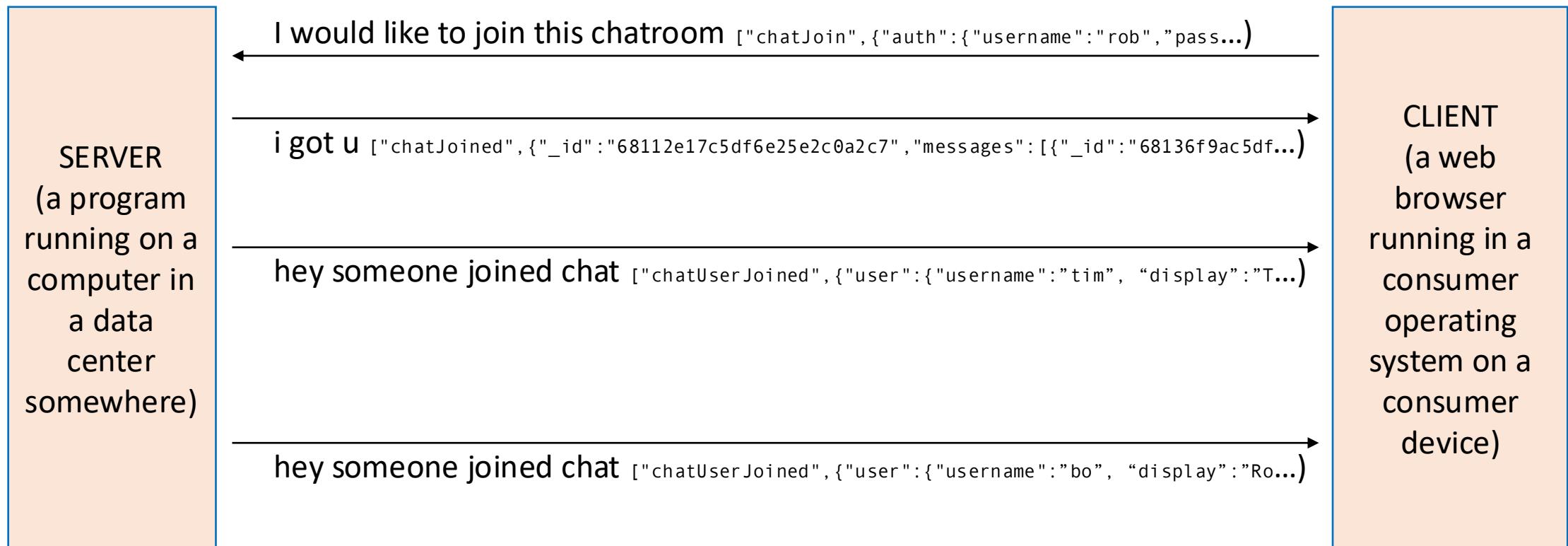
# Application Layer Abstractions: RPC/REST

**Remote procedure calls** happen via HTTP requests (REST)



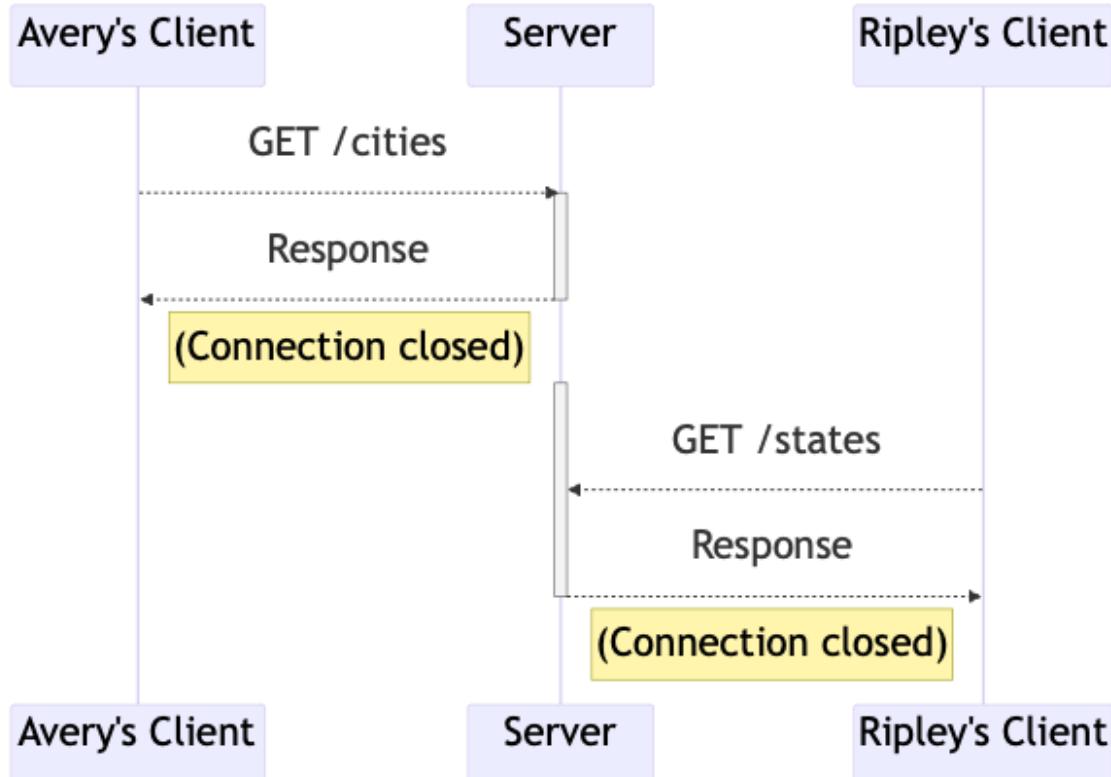
# Application Layer Abstractions: WebSockets

**Message Passing** happen via WebSockets

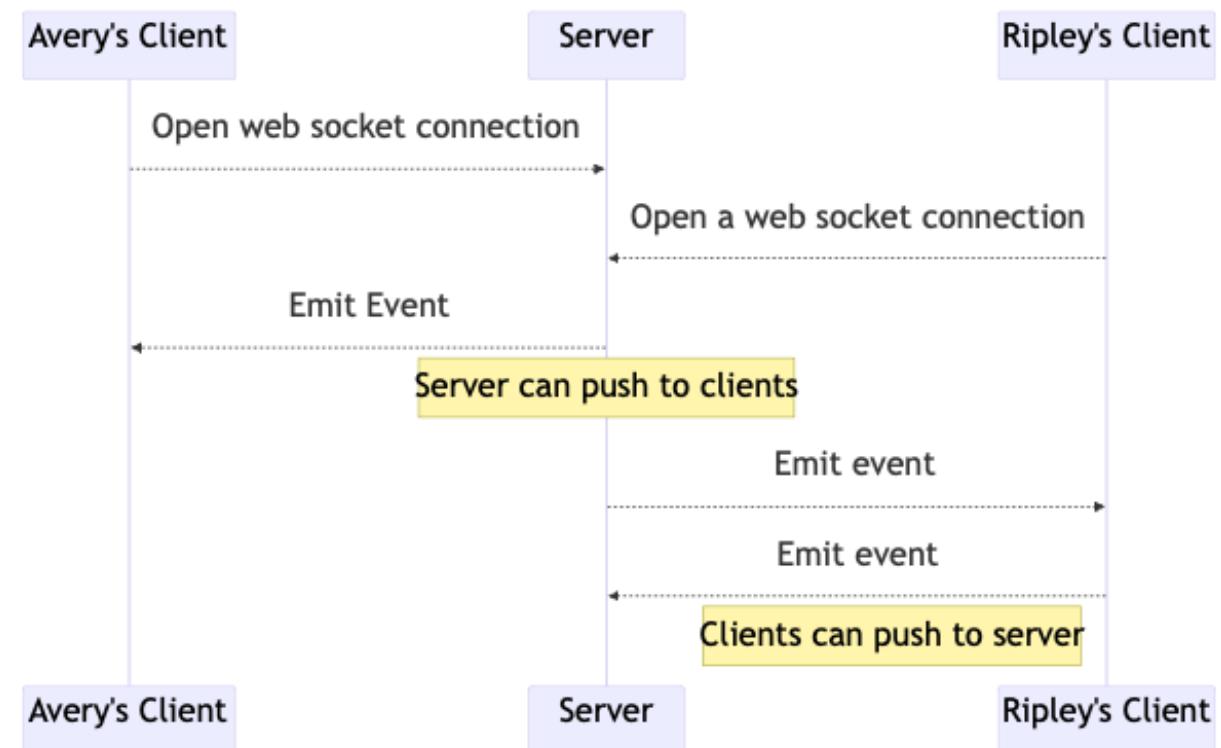


# Application Layer Abstractions: REST versus Web Sockets

## REST

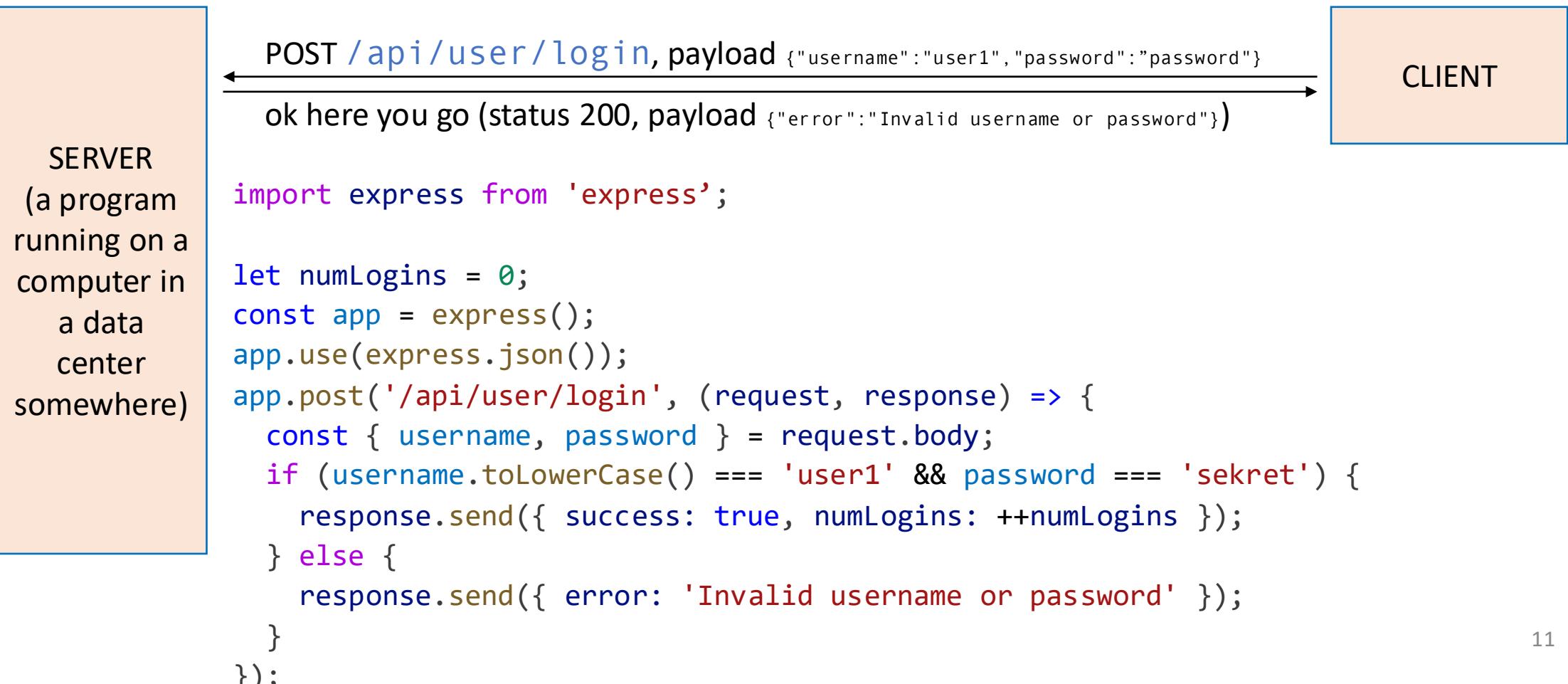


## Web Sockets



# RPC/REST in Express

## How implementing RPC/REST looks for an Express server



# The Controller: Knows about HTTP

```
// app.ts
import express from 'express';
import * as user from './controller.ts';
const app = express();
app.use(express.json());
app.post('/api/user/login', user.postLogin);

// controller.ts
import type { Request, Response } from 'express';
let numLogins = 0;

function postLogin(request: Request, response: Response) {
  const { username, password } = request.body;
  if (username.toLowerCase() === 'user1' && password === 'sekret') {
    response.send({ success: true, numLogins: ++numLogins });
  } else {
    response.send({ error: 'Invalid username or password' });
  }
}
```

# The Controller (with Zod)

---

```
// controller.ts
import type { Request, Response } from 'express';
import { z } from 'zod';

let numLogins = 0;
const zLoginReq = z.object({ username: z.string(), password: z.string() });
export function postLogin(request: Request, response: Response) {
  const auth = zLoginReq.safeParse(request.body);
  if (!auth.success) {
    response.send({ error: 'Bad request' });
  } else if (auth.data.username.toLowerCase() === 'user1' && auth.data.password === 'secret') {
    numLogins += 1;
    response.send({ success: true, numLogins });
  } else {
    response.send({ error: 'Invalid username or password' });
  }
};
```

# The Controller Relies on the Service Layer

```
// controller.ts
import type { Request, Response } from 'express';
import { z } from 'zod';
import { isAuthorized, incrementLogins } from './service.ts';

const zLoginReq = z.object({ username: z.string(), password: z.string() });
export function postLogin(request: Request, response: Response) {
  const auth = zLoginReq.safeParse(request.body);
  if (!auth.success) {
    response.send({ error: 'Bad request' });
  } else if (isAuthorized(auth.data.username, auth.data.password)) {
    const numLogins = incrementLogins();
    response.send({ success: true, numLogins });
  } else {
    response.send({ error: 'Invalid username or password' });
  }
};
```

# The Service Layer's Business Logic

```
// service.ts

export function isAuthorized(username: string, password: string) {
  return username.toLowerCase() === 'user1' && password === 'secret'
}

let numLogins = 0;
export function incrementLogins() {
  const oldNumLogins = numLogins;
  numLogins += 1;
  return oldNumLogins;
}
```

- What user needs *aren't* being met here (and in the IP1 starter code?)  
**(Change password? Save numLogins when rebooting?)**
- How can we do better?  
**(Add a database)**

# The Repository Layer Provides Persistence

---

- Logins should be cumulative even if we restart the server
- Adding users and changing passwords shouldn't necessarily require updating code
- Lots of ways to achieve this:
  - MongoDB
  - PostgreSQL
  - SQLite
  - A file on the hard drive

# Foreshadowing

---

- Adding a persistent repository makes one big difference!
  - almost every action that reads or writes data is now *hundreds* of times slower, and involves reading to disk
  - this involves a relatively long delay, during which the CPU isn't doing anything useful
- JavaScript handles this with *asynchronous programming*; that's a topic we'll return to in a few weeks.

# Review: three parts of a web server

---

- The **repository** is the only part that stores information long-term
  - This is pretty much a synonym for “database”
- The **service** doesn’t know how we connect to the client
  - HTTP? REST? WebSockets? The service shouldn’t know!
- The **controller** doesn’t know how we store data
  - Are we actually “stateless,” or storing things in memory like IP1?



# Review: three parts of a web server

---

- Everything we saw from the transcript server is the **business logic** — a *boring-sounding* name that refers to most of the *interesting* stuff a web server does
  - “Is this HTTP request coming from a recognized user?” — not business logic
  - “Does this user have permission to access student records” — business logic!
  - “Do new grades go at the front or back of the list” — business logic!

# Testing Controller-Service-Repository Servers

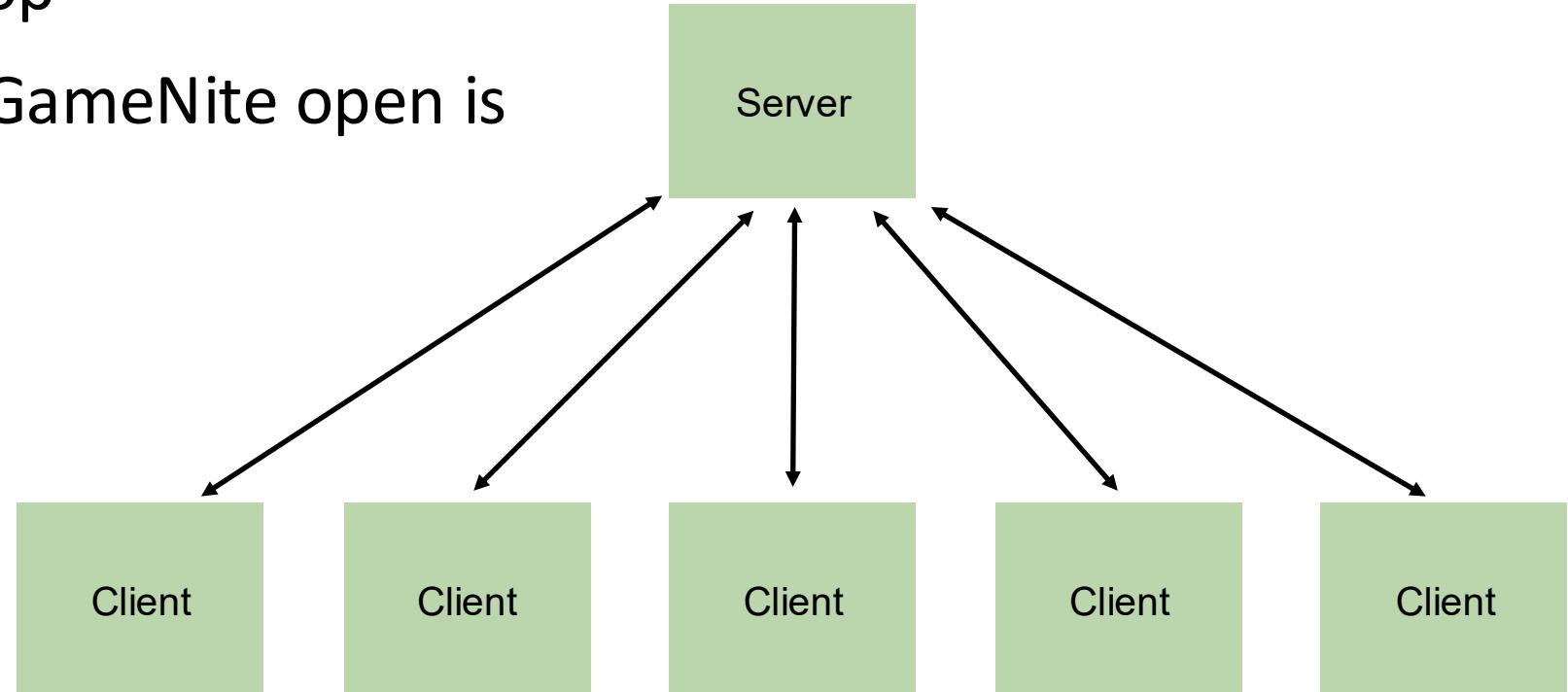
---

- We can test at both the service layer and the controller layer
  - What are the pros and cons of each?
- Sometimes we'll want to test the service layer and/or controller layer *without* the repository layer!
  - We'll come back to this.

# Web Applications: One Server, Many Clients

---

- We've mostly ignored so far that one server is supposed to connect to many clients
- This isn't always apparent when everything's running on your laptop
- Each web page with GameNite open is a different client



# Web Applications and Scalability

---

Distributed systems are hard!

- Web applications are designed to only be *kinda* difficult-to-build distributed systems
- Most of this lecture is bad advice if you're Google, Netflix, or Amazon

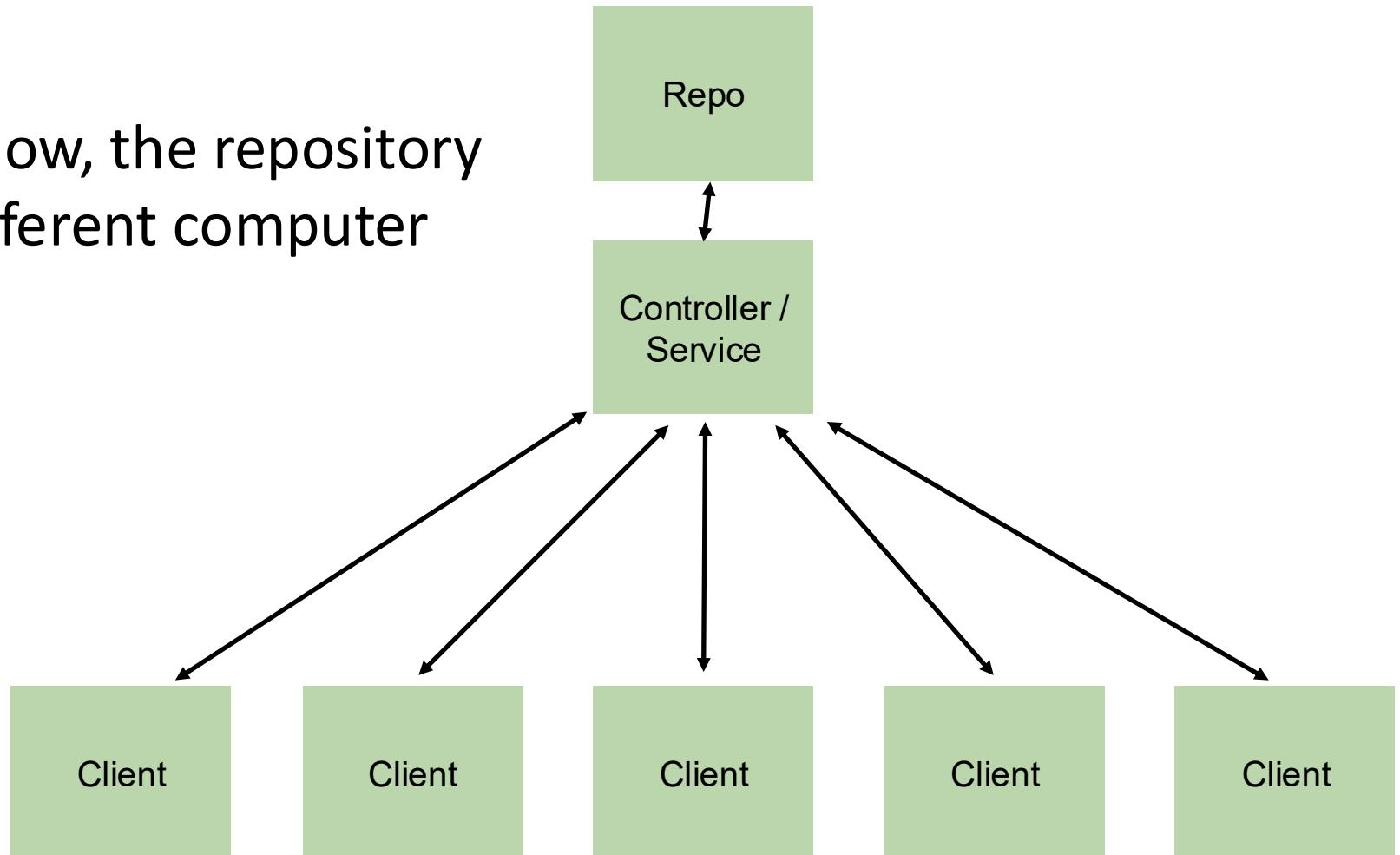
Web applications are distributed systems *because*

1. You don't live in the cloud
2. **Scalability: Netflix needs at *least* two computers**

# Scaling The C-S-R Architecture

---

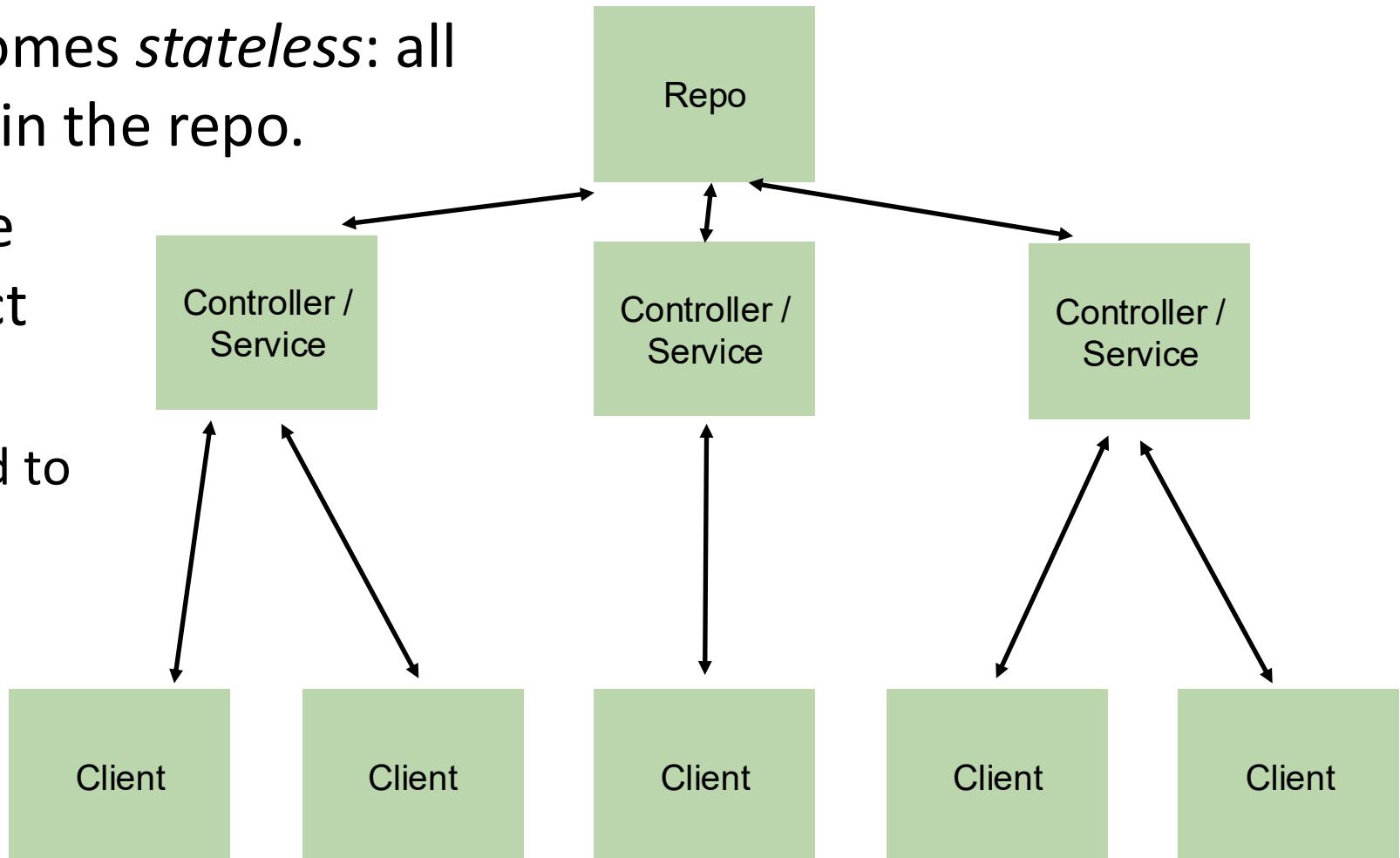
- Web services often start on a single computer
- When that gets too slow, the repository is easy to put on a different computer



# Horizontal Scaling

---

- By separating out the repository layer, the service layer becomes *stateless*: all the state it uses lives in the repo.
- Multiple copies of the controller can connect to multiple clients!
  - AWS will be delighted to help, only real limit is money



# State and statelessness

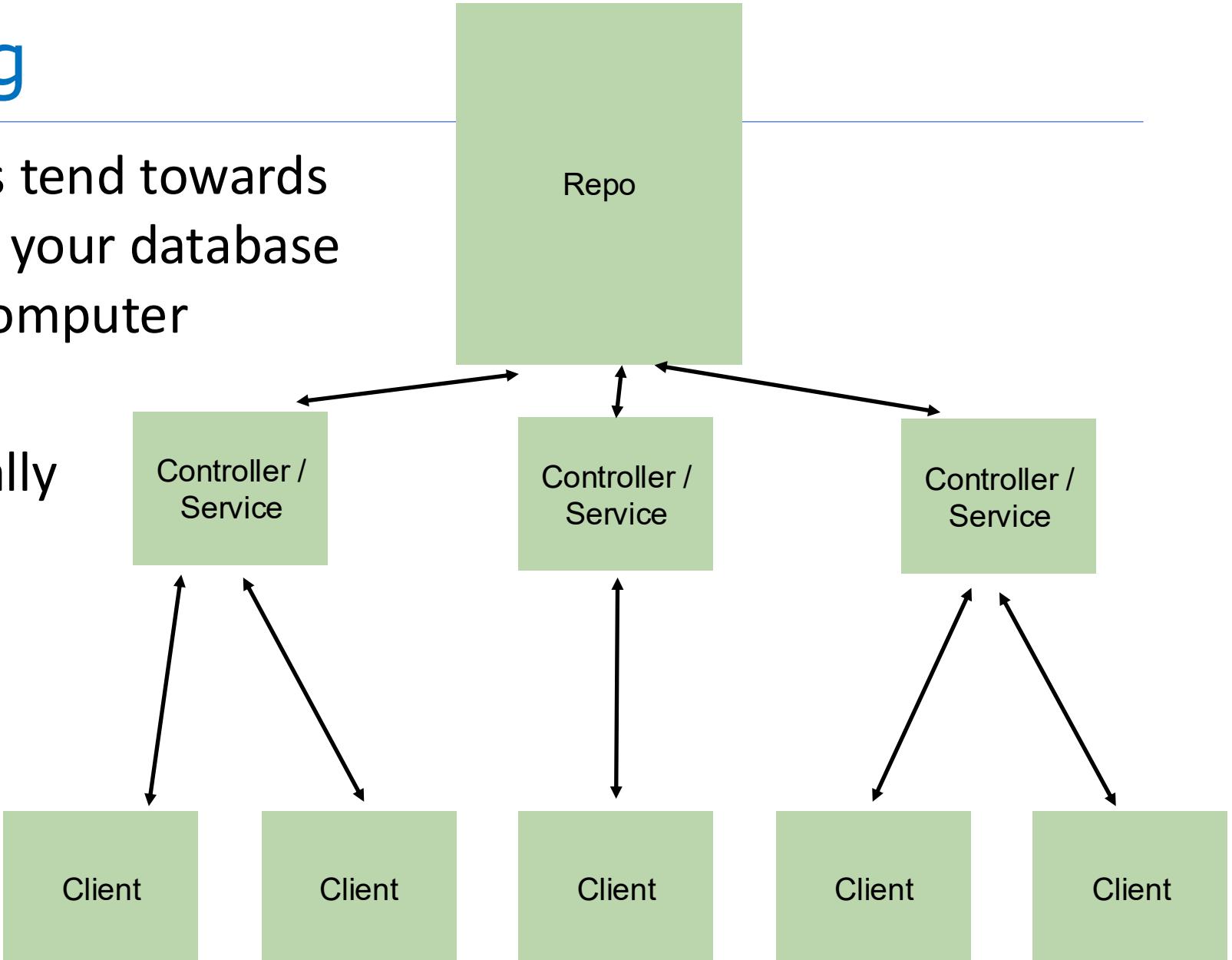
---

- A web server or web service should be *stateless*
  - Every REST request should be indifferent to whether the node application has been *running* for several hours or five seconds
  - Our silly application, and the IP1 code, is *not* stateless (why?)
- If the web server is going to be stateless, and the web application has state, the server has to phone a friend to:
  - Access the filesystem
  - Query a database
  - Initiate some other remote procedure call to another server
- Common case: the repository layer and the database is the point of centralization
  - Centralization (& hierarchical centralization) is a cheat code for making distributed systems manageable

# Vertical Scaling

---

- Centralized databases tend towards *vertical scaling*: move your database to a more powerful computer
- This has limits: the database will eventually become a bottleneck



# Scaling Past the Database Bottleneck (1)

---

Maybe the problem is that you're occasionally doing big, expensive analysis on your database, like analyzing your sales data

- Database *read-only-replicas* are an easy solution here — seconds to minutes behind reality (and can add reliability in case of failure!)

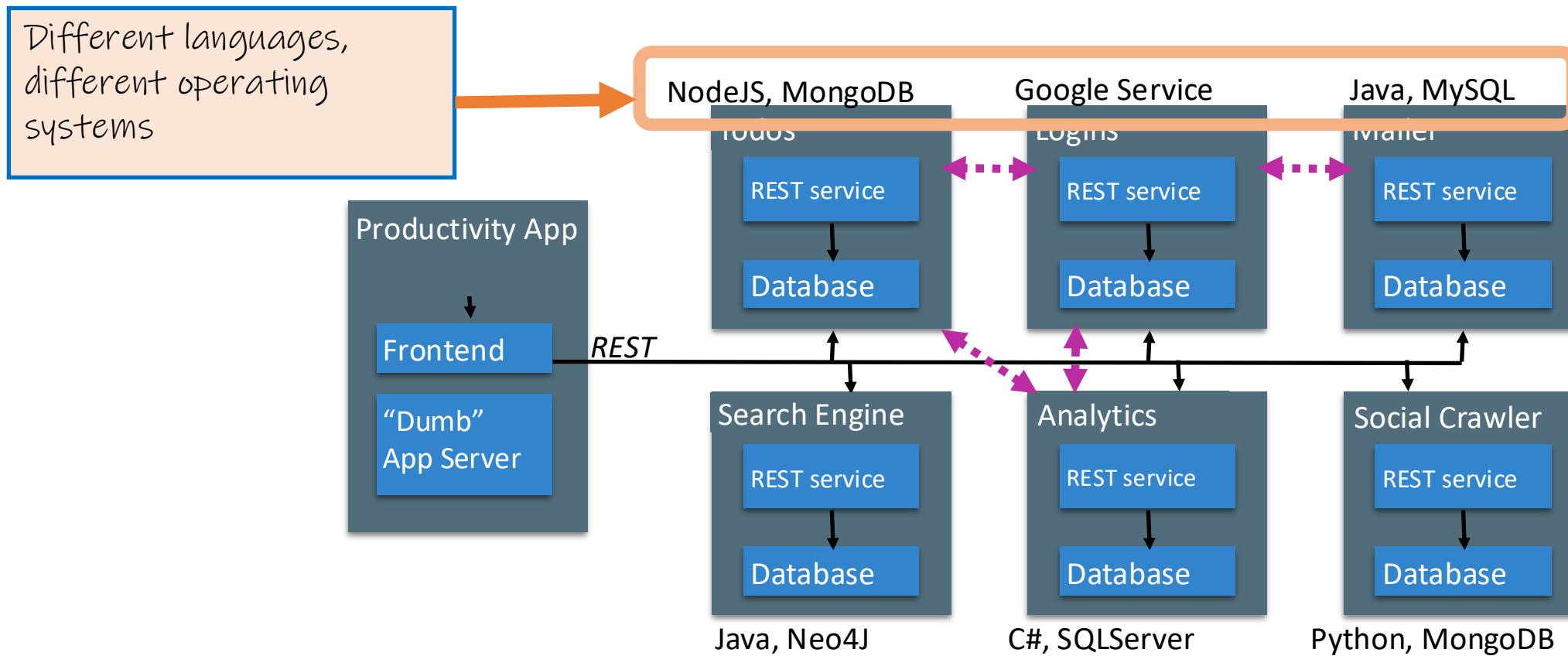
# Scaling Past the Database Bottleneck (2)

---

Maybe you can identify parts of your data that are independent, and don't need to be synchronized or stored together

- Chat and game information in GameNite could live be in separate places
- Games could have their business logic running on different servers, written in different programming languages, and accessed (by the server the client is connected to) through their own REST API!
- This way lies microservices

# Microservices

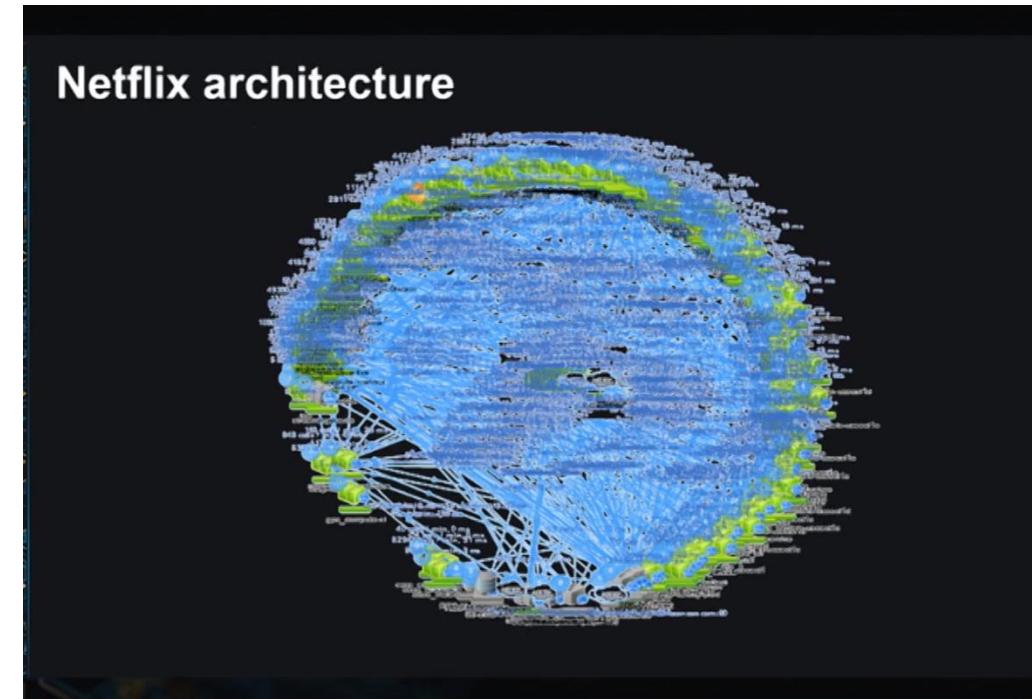


# Microservices

---

Netflix is the microservices darling

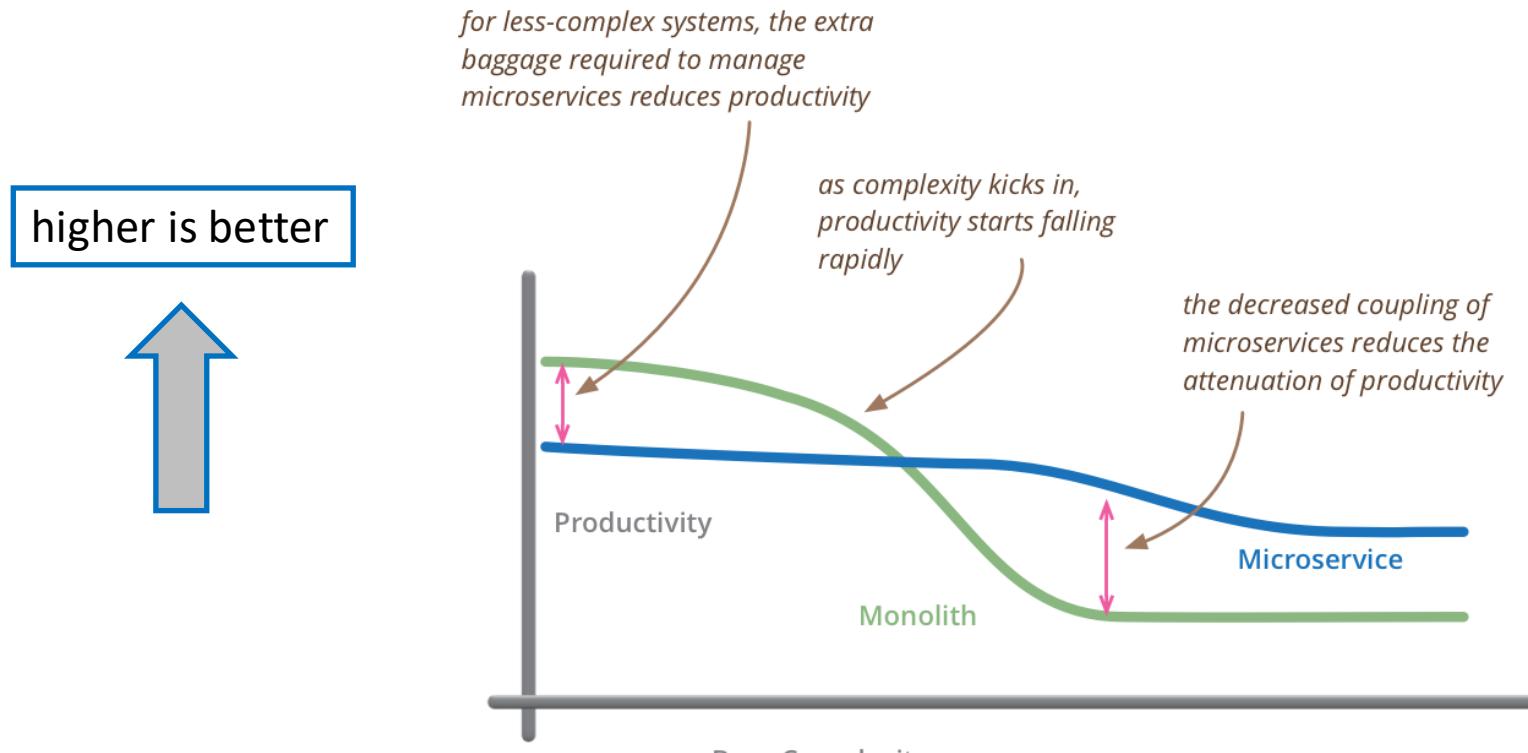
- 100s of microservices
- 1000s of daily production changes
- 10,000s of instances
- BUT:
- only 10s of operations engineers



<https://medium.com/refraction-tech-everything/how-netflix-works-the-hugely-simplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254b>

# Microservices

The opposite of “microservices” is “monolith”



*but remember the skill of the team will outweigh any monolith/microservice choice*

<https://martinfowler.com/microservices/>

# GameNite is Monolithic

---

- GameNite is a monolithic application
- It's not perfect: there's probably a bit too much business logic in the controller layer (service layer doesn't quite do enough)
- You'll start IP2 with a more proper web app
  - MongoDB is the database used for repository layer, by way of a general-purpose adapter called Keyv
  - Changing the repository greatly changes the service layer
  - The controller doesn't change much (the controller is mostly unaware of the repository later)

# Review

---

At the end of this lesson, you should be able to

- Explain the role of “client” and “server” in the context of web application programming
- Explain the role of REST versus WebSocket communication
- Describe the fundamental differences between the three layers of the controller, service, and repository layers in a C-S-R architecture
- Understand how the C-S-R architecture works in the context of a basic Express application
- Be able to answer an interview question about “business logic,” “horizontal and vertical scaling,” or “microservices”