CS 4530: Fundamentals of Software Engineering
Module 3, Lesson 1
Web Applications

Rob Simmons

Khoury College of Computer Sciences

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 primary options for client-server communication
- Identify places where TypeScript does — and doesn’t! — help with writing correctly-behaving web applications, and identify some of the solutions to functionality TypeScript doesn’t provide

So, software engineering must encompass:

	PEOPLE	PROCESSES	PROGRAMS
PLANNING			
ORGANIZING			We're gonna be stuck over here for a bit.
IMPLEMENTING			

Web Applications are Distributed Systems

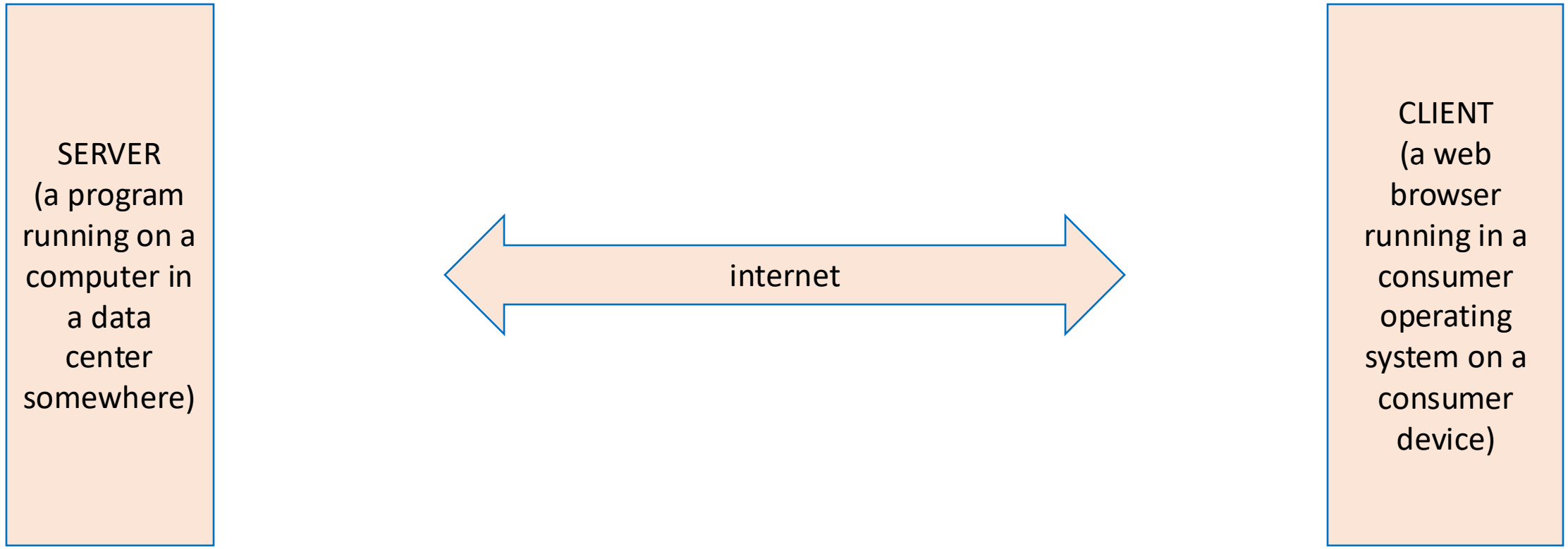
Distributed systems are hard!

- 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*

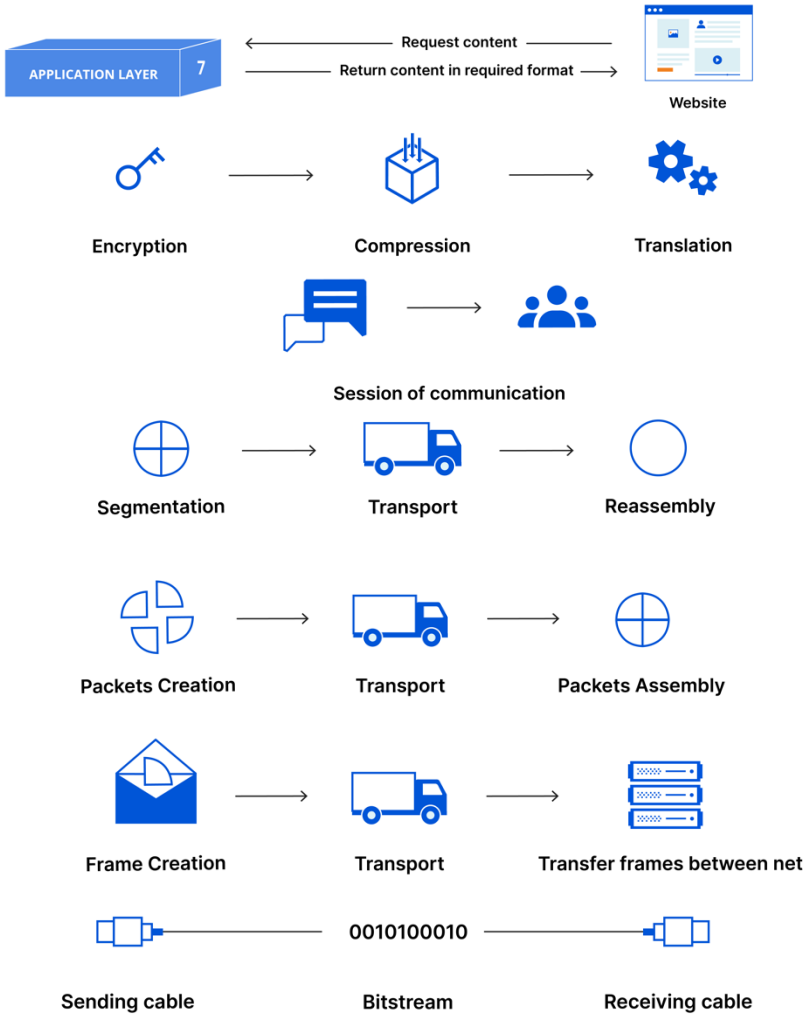
1. **You don't live in the cloud**
2. Scalability: Netflix needs at *least* two computers

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)

https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/

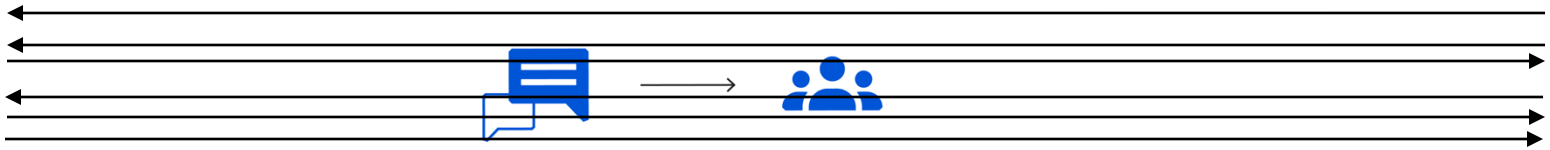
An Insultingly Shallow Intro to Networking

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

Transport Layer

- TCP you may have heard of
- UDP you don't need for this class
- QUIC is new

Key abstraction: *sending a message*



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

https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/

Application Layer Abstractions

Remote procedure calls happen via HTTP requests (REST)

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

GET /

ok here you go (status 200, payload <!doctype html><html lang="en...)

GET /favicon.svg

ok here you go (status 200, payload <svg width="450" height="450" viewBox="...)

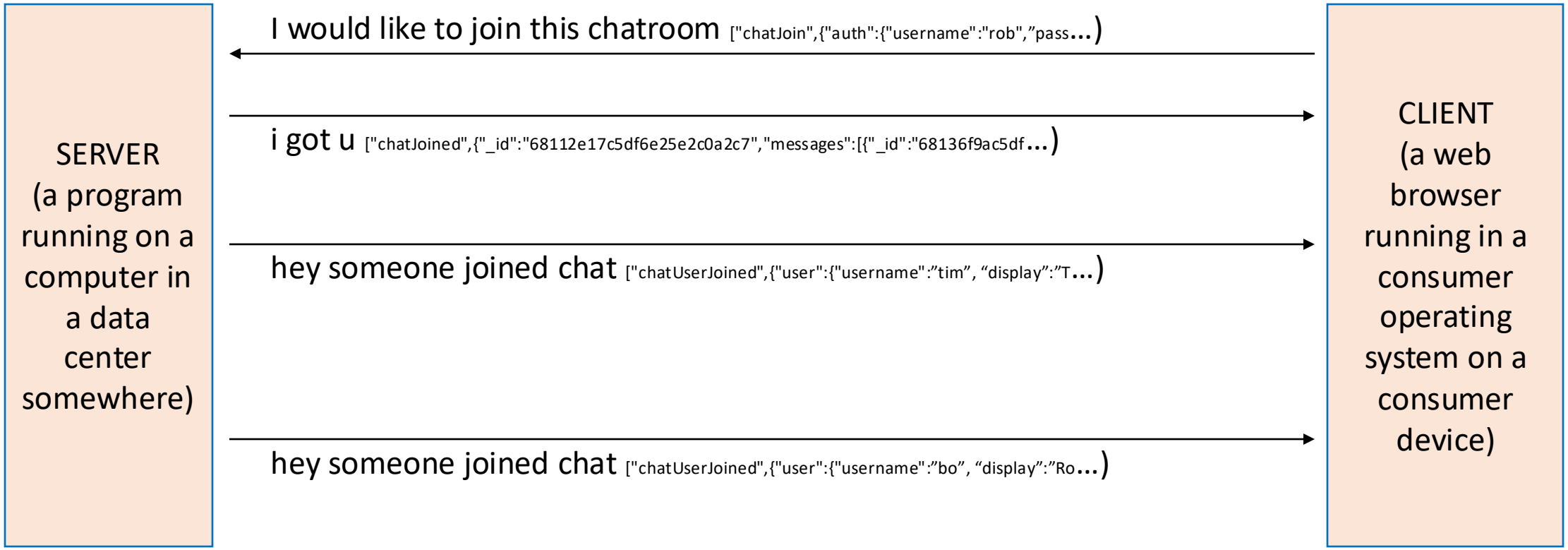
POST /api/user/login, payload {"username":"user1","password":"password"}

ok here you go (status 200, payload {"error":"Invalid username or password"})

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

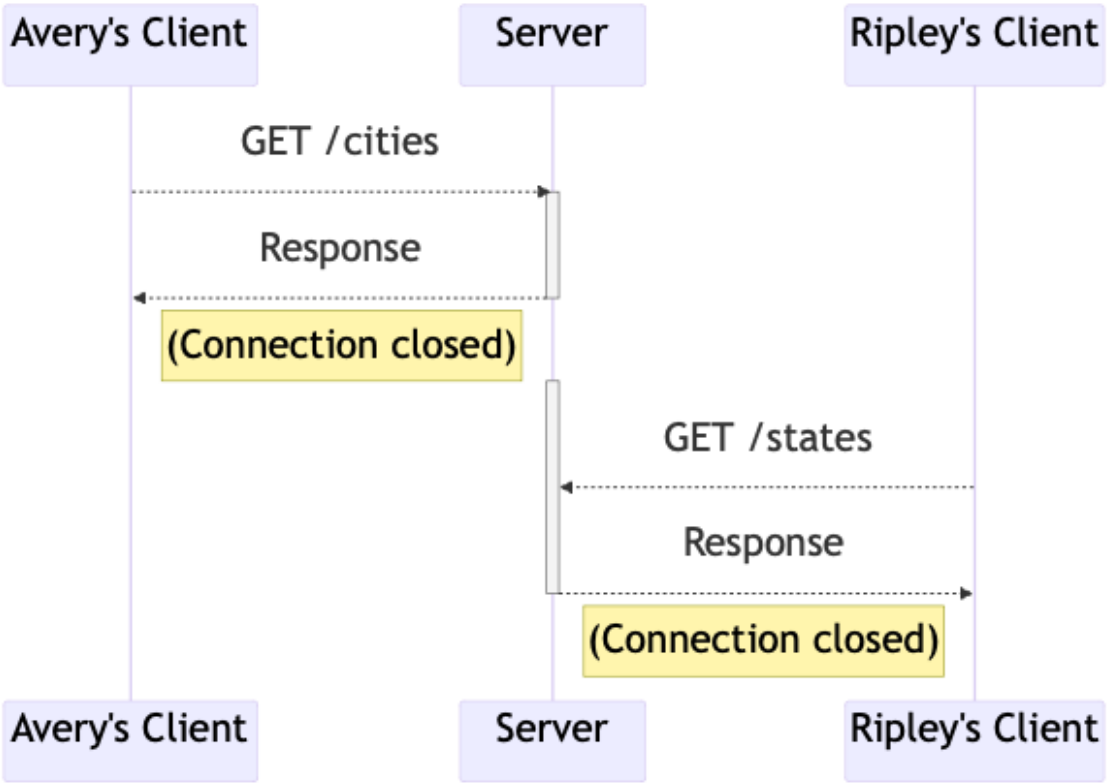
Application Layer Abstractions

Message Passing happen via WebSockets

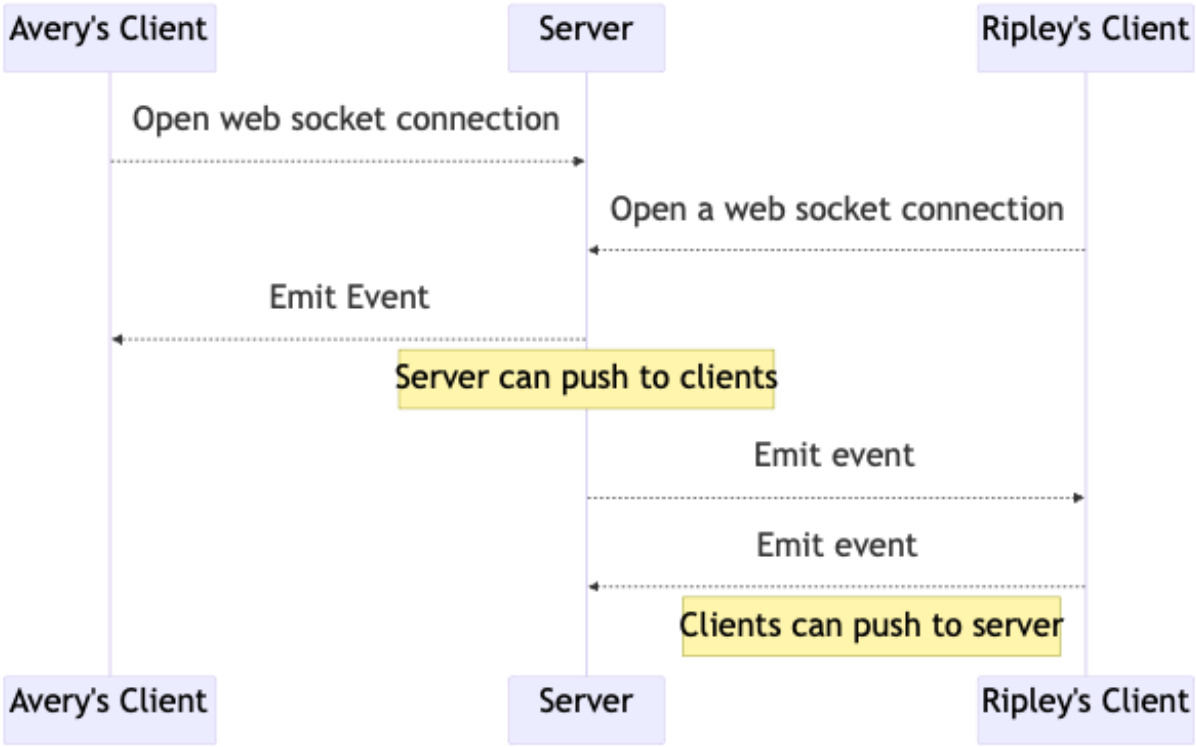


Application Layer Abstractions

REST



Web Sockets



Implementing REST APIs

"Express" is good for implementing servers in NodeJS

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

POST /api/user/login, payload {"username":"user1","password":"password"}

ok here you go (status 200, payload {"error":"Invalid username or password"})

CLIENT

```
import express from 'express';

let numLogins = 0;
const app = express();
app.use(express.json());
app.post('/api/user/login', (request, 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' });
  }
});
```

Implementing REST APIs

The “any” type and “as” are common in TypeScript

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

POST /api/user/login, payload {"username":"user1","password":”password"}

ok here you go (status 200, payload {"error":"Invalid username or password"})

CLIENT

```
import express from 'express’;

let numLogins = 0;
const app = express();
app.use(express.json());
app.post('/api/user/login', (request, 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' });
  }
});
```

creators of the express types chose to make this type “any”

Implementing REST APIs

The “any” type and “as” are common in TypeScript

TypeScript: looks good!
ESLint: “unsafe assignment of any value”

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

POST /api/user/login, payload {"username":"user1","password":"password"}

ok here you go (status 200, payload {"error":"Invalid username or passwor

CLIENT

```
import express from 'express';

let numLogins = 0;
const app = express();
app.use(express.json());
app.post('/api/user/login', (request, 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' });
  }
});
```

creators of the express types chose to make this type “any”

Implementing REST APIs

The “any” type and “as” are common in TypeScript

TypeScript: looks good!
ESLint: looks good!

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

POST /api/user/login, payload {"username":"user1","password":”password"}

ok here you go (status 200, payload {"error":"Invalid username or passwor

CLIENT

```
import express from 'express’;
type UserAuth = { username: string; password: string };
let numLogins = 0;
const app = express();
app.use(express.json());
app.post('/api/user/login', (request, response) => {
const { username, password } = request.body as UserAuth;
  if (username.toLowerCase() === 'user1' && password === 'sekret') {
    response.send({ success: true, numLogins: numLogins++ });
  } else {
    response.send({ error: 'Invalid username or password' });
  }
});
```

Implementing REST APIs

Anyone can send an HTTP request containing anything

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

POST /api/user/login, payload {"lol":["owned"],"password":4,"note":"loser"}

???

```
import express from 'express';
type UserAuth = { username: string; password: string };
let numLogins = 0;
const app = express();
app.use(express.json());
app.post('/api/user/login', (request, response) => {
const { username, password } = request.body as UserAuth;
  if (username.toLowerCase() === 'user1' && password === 'sekret') {
    response.send({ success: true, numLogins: numLogins++ });
  } else {
    response.send({ error: 'Invalid username or password' });
  }
});
```

Implementing REST APIs

Anyone can send an HTTP request containing anythin

Uncaught TypeError: username.toLowerCase is not a function

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

POST /api/user/login, payload {"lol":["owned"],"password":4,"note":"loser"}

i have no idea what is going on (500 Internal Server Error)

```
import express from 'express';
type UserAuth = { username: string; password: string }
let numLogins = 0;
const app = express();
app.use(express.json());
app.post('/api/user/login', (request, response) => {
const { username, password } = request.body as UserAuth;
  if (username.toLowerCase() === 'user1' && password === 'sekret') {
    response.send({ success: true, numLogins: numLogins++ });
  } else {
    response.send({ error: 'Invalid username or password' });
  }
});
```

Implementing REST APIs

```
import express from 'express';
import { z } from 'zod';

type UserAuth = { username: string; password: string };
const zUserAuth = z.object({
  username: z.string(),
  password: z.string(),
});
let numLogins = 0;
const app = express();
app.use(express.json());
app.post('/api/user/login', (request, response) => {
  const { username, password }: UserAuth = zUserAuth.parse(request.body);
  if (username.toLowerCase() === 'user1' && password === 'sekret') {
    response.send({ success: true, numLogins: numLogins++ });
  } else {
    response.send({ error: 'Invalid username or password' });
  }
});
```

Throws an error if the input is unexpected (safeParse is the non-exception-raising option)

Implementing REST APIs

```
import express from 'express';
import { z } from 'zod';

type UserAuth = z.infer<typeof zUserAuth>;
const zUserAut
  username: z.strin
  password: z.string
});
let numLogins = 0;
const app = expres
app.use(express.js
app.post('/api/user/login', (request, response) => {
  const { username, password }: UserAuth = zUserAuth.parse(request.body);
  if (username.toLowerCase() === 'user1' && password === 'sekret') {
    response.send({ success: true, numLogins: numLogins++});
  } else {
    response.send({ error: 'Invalid username or password' });
  }
});
```

```
type UserAuth = {
    username: string;
    password: string;
}
```

Testing and TypeScript

- It makes sense *often* to treat your TypeScript types as not-needing-to-be-tested
- It *never* makes sense to assume anything about information coming to your server from a REST API call as having the TypeScript type you expect.
  - Your project is set up so that inputs to REST APIs are treated as **unknown**, not **any**.
  - Don't use as assertions — validate!
- It makes sense *sometimes, maybe* to treat information coming back to your web app from a server as having the TypeScript type you expect.

Parse, don't validate

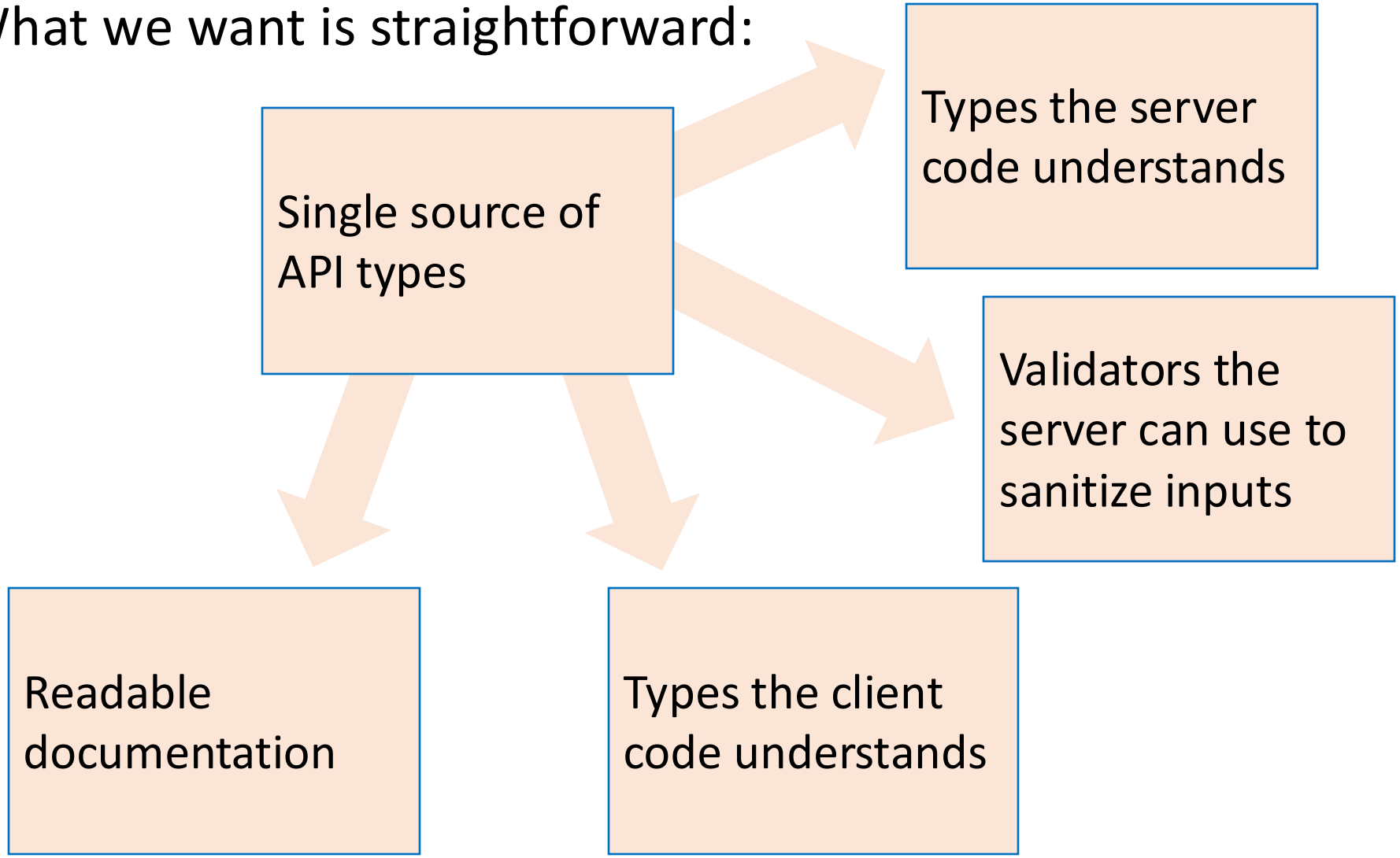
2019-11-05 functional programming, haskell, types

Historically, I've struggled to find a concise, simple way to explain what it means to practice type-driven design. Too often, when someone asks me "How did you come up with this approach?" I find I can't give

https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/

Testing and TypeScript

What we want is straightforward:



Testing and TypeScript

What we want is straightforward:

- A single place to explain the API interface that produces docs, types, and validators
- TSOA
  - requires writing the API as classes, not as functions
- GraphQL
  - shoves your entire API into one endpoint that accepts HTTP POST requests
  - has some other advantages we won’t talk about here
- Hono
  - Uses Zod, is interesting! But doesn’t work with express

Review

It’s the end of the lesson, so you should be able to:

- Explain the role of “client” and “server” in the context of web application programming
- Explain the primary options for client-server communication
- Identify places where TypeScript does — and doesn’t! — help with writing correctly-behaving web applications, and identify some of the solutions to functionality TypeScript doesn’t provide