CS 4530: Fundamentals of Software Engineering

Module 4: Architecture of a Web Application

Adeel Bhutta and Mitch Wand

Khoury College of Computer Sciences

Learning Goals for this Lesson

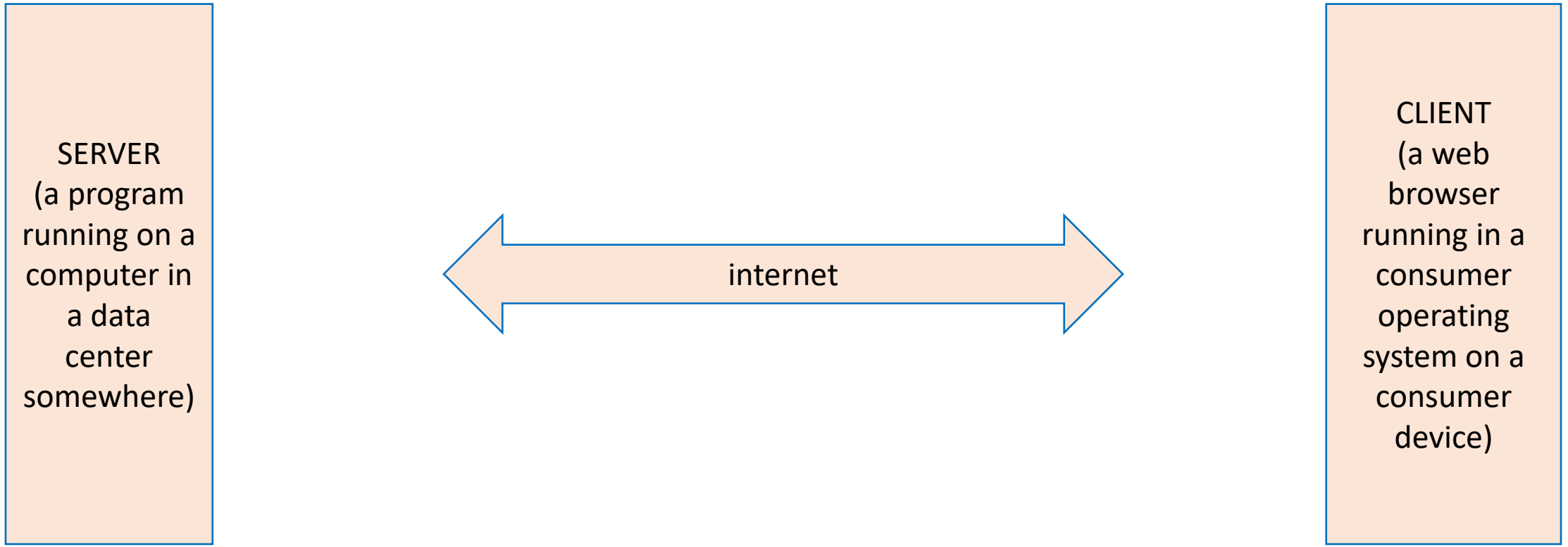
At the end of this lesson, you should be able to

- Explain the role of “client” and “server” in the context of web application programming
- Describe the role of network protocols like http and websockets in a distributed system.
- Describe the fundamental differences between the three layers of the controller, service, and repository layers in a C-S-R architecture
- Use async and await to handle long-running procedures in a persistent repository

Web Applications are Distributed Systems

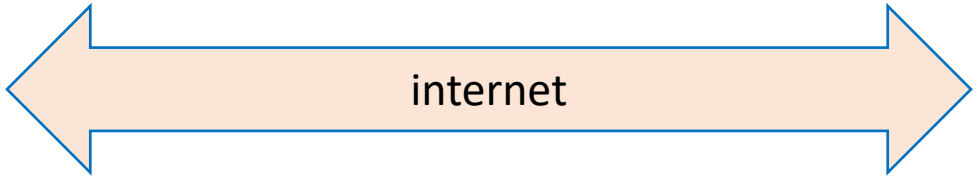
- Distributed systems are applications that perform computation across different computers
- Web applications are distributed systems *because*
  1. Users are literally in different places: you don't live in a data center
  2. Netflix won't fit entirely on one computer
- Distributed systems are hard!
  - Adopting well-understood patterns makes the task easier
  - We'll talk about one kind of pattern in this lecture
  - If you're Google, Netflix, or Amazon, the simple designs in this lecture won't serve your purposes

The simplest distributed system has a server and one or more clients



The server and the client communicate using an agreed-upon protocol

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)

In this lecture we're going to work over here. We'll talk about clients in a couple of weeks

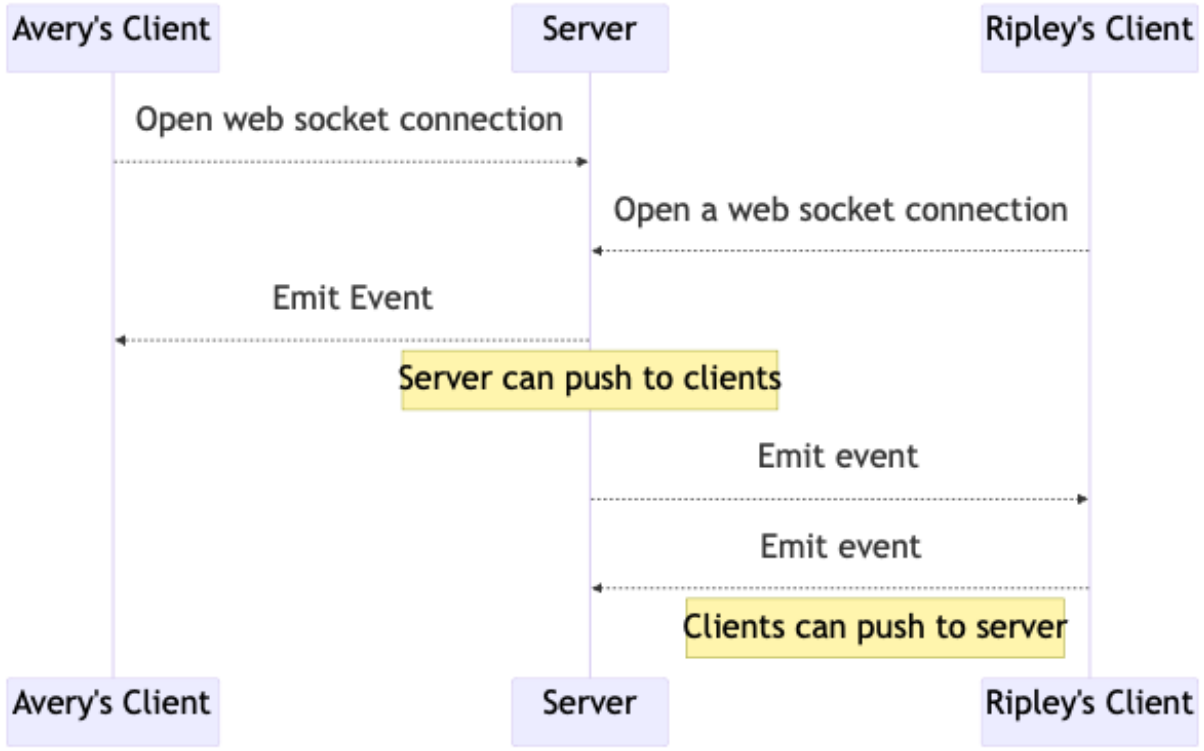
Examples of protocols

- Checking guests off a guest list
- Morse code
- Procedure calls
  - how are arguments passed to procedures?
  - how are results passed back to the caller?

- The system designer gets to choose the protocol, and then builds the programs to match

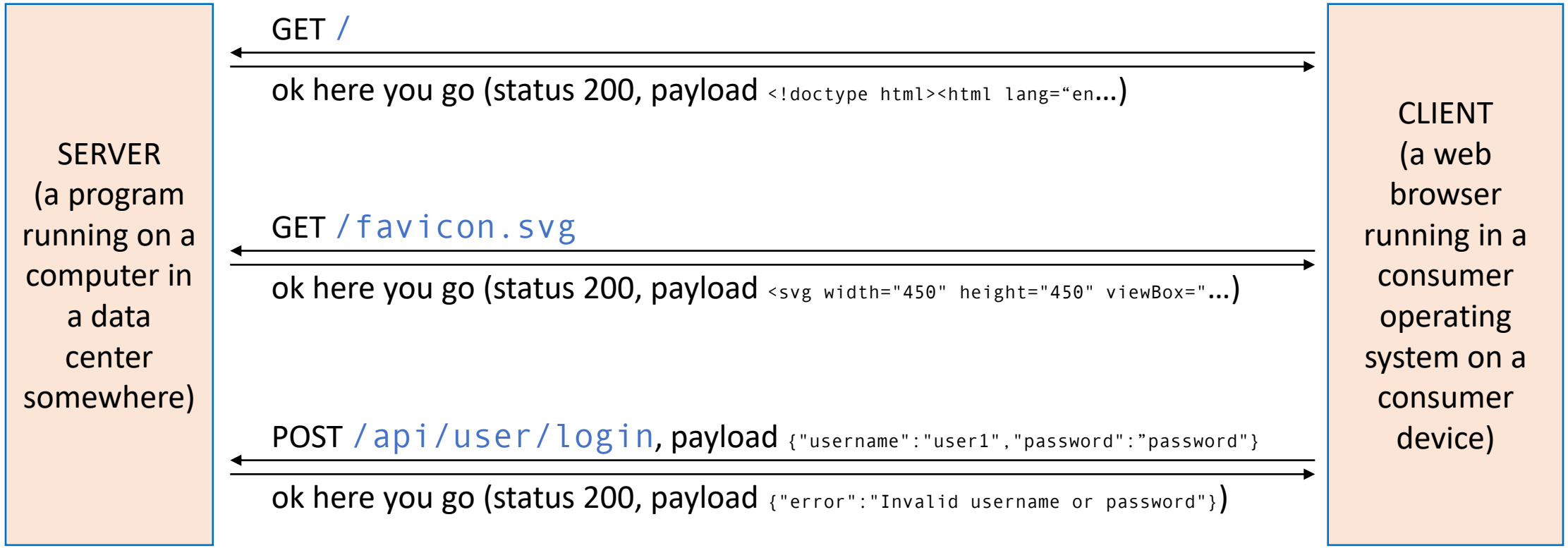
WebSockets is a useful protocol for event-driven applications

- A server can communicate with many clients
- A chat message is an “event” that the server wants everyone to know about.
- The web server is a *centralized* point that allows distributed clients to talk to each other.
  - Centralization is a cheat code for making distributed systems manageable



HTTP is a useful protocol for "function-call"-like patterns

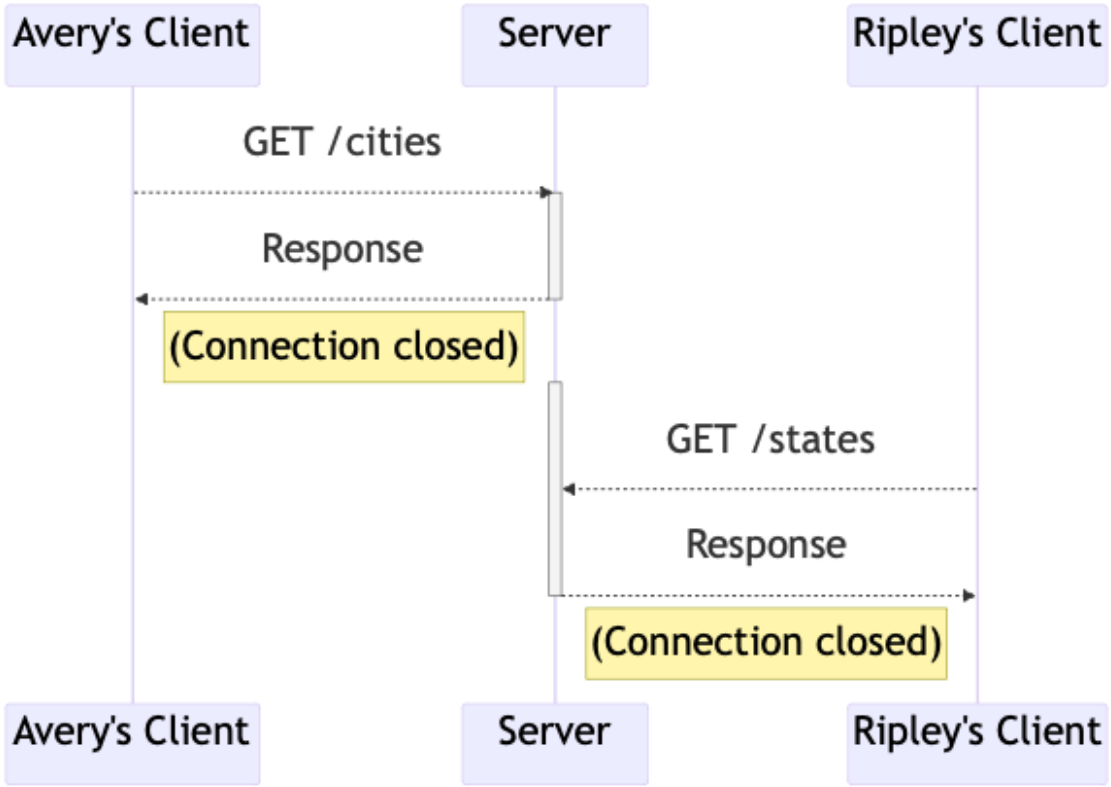
HTTP gathers messages in pairs: client sends, server responds



HTTP is a protocol useful for remote procedure calls, like REST.

HTTP supports common design patterns:

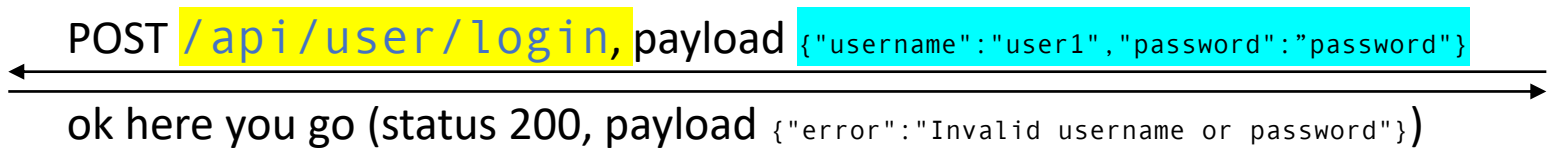
- *Remote Procedure Call* (RPC): one computer is calling a function on another computer
- *REpresentational State Transfer* (REST): an RPC where server doesn't maintain a session or remember specific clients.
- Most of our web apps will use a REST-style design.



http requests typically contain a **route** and a **payload**.

Here's a very simple express server

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



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' });
  }
});
```

The Controller-Service-Repository Design

- This is the design we will use for all our web servers.
- It's only one way to organize a web server.
- But it's good enough for us.
- It emphasizes an important SE principle: Information Hiding
- If you are Google, you probably want something more sophisticated.

Three parts of our web servers

- The Controller-Service-Repository design separates the server into three parts
  - The **controller** knows how we communicate with client, doesn't know how we store data
  - The **service** doesn't know how we connect to the client or store data, but does as much of the interesting work ("business logic") as possible without knowing these things
  - The **repository** is the only part that stores information long-term

4.1 The Controller

- The controller is responsible for communication with the outside world
- We'll start by looking at the very simple controller we saw a few slides ago.
- Then we will add a little structure.

First, we add validation to the controller (here using Zod)

```
import express from "express";
import { type Request, type Response } from "express";
import { z } from "zod"
const app = express();
let numLogins = 0;

const zLogin = z.object({
  username: z.string().min(4),
  password: z.string().min(8)
})

app.post('/api/user/login', (request:Request, response:Response) => {
  const validatedRequest = zLogin.safeParse(request)
  if (validatedRequest.error) {response.send({error: "Bad Request"})}
  else {
    const username:string = validatedRequest.data.username
    const password:string = validatedRequest.data.password
    // action goes here
  }
})
```

Description of the valid payloads

safeParse.data returns correctly typed components

The controller relies on the service layer to perform the appropriate actions

```
import express from "express";
import { type Request, type Response } from "express";
import { z } from "zod"
const app = express();
import { isAuthorized, incrementLogins} from "./loginService.ts“

    // validation goes here

    const username:string = validatedRequest.data.username
    const password:string = validatedRequest.data.password
    if (isAuthorized(username, password)) {
      const numLogins = incrementLogins()
      response.send({success:true, numLogins})
    } else
      response.send({error: "Invalid username or password"})
}
```

The controller uses the service layer to assemble the data for an appropriate response

The controller sends the actual response itself

4.2 The Service Layer Provides the Business Logic

```
var numLogins = 0

export function isAuthorized(username:string, password:string):boolean {
    return ((username == "user1") && (password == "secret"))
}

export function incrementLogins(): number {
    numLogins++;
    return numLogins
}
```

Looking back at the Transcript Server from the TDD Lecture

- Most of we saw from the transcript server was the **business logic** — the boring-sounding name that refers to most of the interesting stuff a web server does

But there was no clear separation of a repository layer

src/transcript.service.ts

```
/** Adds a new student to the database
 * @param {string} newName - the name of the student
 * @returns {StudentID} - the newly-assigned ID for the new student
 */
  addStudent(newName: string): StudentID {
    const newID = this._lastID++;
    const newStudent: Student = { studentID: newID,
                                  studentName: newName };
    this._transcripts.push({
          student: newStudent, grades: []
    });
    return newID;
  }
```

NOT business logic!

4.3 A Repository Layer should provide 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

Adding a persistent repository requires some new programming ideas

- 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.*

Typescript uses the type system to distinguish long-running computations

- A long-running computation gets type **Promise<Grade>**, rather than **Grade.**

We need to change the interface to accommodate this change

```
export interface ITranscriptService {
  addStudent(studentName: string): Promise<StudentID>;
  getTranscript(id: StudentID): Promise<Transcript>;
        // promise is if id invalid
  deleteStudent(id: StudentID): Promise<void>;
        // promise is rejected if id invalid
  addGrade(id: StudentID, course: Course, courseGrade: CourseGrade): Promise<void>;
  getGrade(id: StudentID, course: Course): Promise<CourseGrade>;
  nameToIDs(studentName: string): Promise<StudentID[]>;
}
```

src/withPersistence/Itranscript.service.ts

Use **await** to get the result of a promise (1)

```
describe('addStudent', () => {
  it('should add a student to the database and return their id',
   async () => {
    expect(await service.nameToIDs('blair')).toStrictEqual([]);
    const id1 = await service.addStudent('blair');
    expect(await serice.nameToIDs('blair')).toStrictEqual([id1]);
  });
```

- service.nameToIDs returns only a promise to compute a list of IDs
- here we need to see the actual list of IDs, so we need to wait for that computation to complete.

Use **await** to get the result of a promise (2)

```
describe('addStudent', () => {
  it('should add a student to the database and return their id',
   async () => {
    expect(await sevice.nameToIDs('blair')).toStrictEqual([]);
    const id1 = await service.addStudent('blair');
    expect(await service.nameToIDs('blair')).toStrictEqual([id1]);
  });
```

- So long as we have a single-threaded app, this works just fine!
- If our app is juggling multiple threads, there are dangers-- more in M08.

Any function that calls a long-running function is itself long-running

```
async function numberOfSharedNames(name: string): Promise<number>
{
    const ids = await service.nameToIDs(name);
    return ids.length
}
```

- Of course, this means that any function that waits for the result of **numberOfSharedNames** must now also be async.
- Eventually, you will run out of functions-that-wait-for-an-async, so you will eventually run out of functions that need to be made async ☺

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
- Describe the role of network protocols like http and websockets in a distributed system.
- Describe the fundamental differences between the three layers of the controller, service, and repository layers in a C-S-R architecture
- Use async and await to handle long-running procedures in a persistent repository