



Rithm.start()

What we'll cover

- Curriculum & goals of program
- Daily schedule
- Lectures, exercises, and assessments
- How things work
- Tips for success

The Schedule

Structured Curriculum

Week	Topics	Goals
1	Web Dev & Comp Sci Intro	Algorithms efficiency & web dev
2	JavaScript & Web Development	Deeper understanding of JS & front-end
3	Intro to Python & Flask	Build back-ends for web apps
4	Flask & Databases	Use relational databases with web apps
5	Security/Auth in Flask	Basics of web security & user auth
6	JS on Server: Node.js	JavaScript apps with Express & PostgreSQL
7	Building APIs with Node.js	More sophisticated backend APIs
8	Intro to React.js	Patterns in basic React apps
9	Intro to React Router	Advanced frontend: routing & state management
10	Advanced Frontend Topics	Learn other useful tools in the frontend

Professional Projects & Outcomes

We'll talk about this in detail closer to this period

Week	Topics	Goals
11-13	Professional Projects	Experience contributing to real production codebase
14	DSA	Week-long dive into data structures & algorithms
15	Outcomes Week 1	Phone & whiteboard interviews, essential tech topics
16	Outcomes Week 2	Tools for networking, recruiters, and applying

Daily Schedule

Every M-F during the program

Time	What's Happening?
9:30	Morning Topic
12:15ish	Lunch
1:30	Afternoon Topic
5:30ish	End of Structured Day
5:30-6:00	End of Staff Support

Student Info System

The schedule and resources for your cohort are at [Rithm 31 <https://r31.students.rithmschool.com>](https://r31.students.rithmschool.com):

- Contact info for staff and peers in cohort
- Upcoming lectures, exercises, and events
- Assessments submission

(We typically publish the schedule a few weeks in advance, and publish lecture notes a day or two in advance.)

Lectures

Lecture Hall

- We'll meet every morning at 9:30 *sharp*
 - Most days, we'll have a morning lecture (*typically, 90-120 mins*)
- We'll meet every afternoon at 1:30 *sharp*
 - Many days, we'll have an afternoon lecture (*typically, 90-120 mins*)
- Be rested, attentive, and ready to be called on!
- We will provide lecture notes for each lecture
 - You're *strongly encouraged* to take notes during lecture

IMPORTANT Some things that will help Zoom learning

- To minimize distractions, please quit your internet browser during lectures
 - Applications you should have open: Zoom and Slack
 - Make sure you have a dedicated work space.
 - Arrive a few minutes early — spend some time with your people!

Minimizing Computer Usage

As laptops become smaller & more ubiquitous, and with the advent of tablets, the idea of taking notes by hand seems old-fashioned to many students today. Typing notes is faster—which is handy when there's a lot of info to take down. But it turns out there are advantages to doing things the old-fashioned way.

Research shows laptops & tablets have a tendency to be distracting—it's easy to click over to Facebook. A study has shown that the fact that you have to be slower when you take notes by hand is what makes it more useful in the long run. (Source) <<https://www.npr.org/2016/04/17/474525392/attention-students-put-your-laptops-away>>

Participation

- Have a question? Click *Raise Hand*
- Do not use the chat feature during lectures; it's distracting
- We may occasionally ask for nonverbal feedback as well (*are we going too fast? too slow?*)

Exercises

- Almost all exercises are **paired assignments**
 - You learn & retain more when pairing; we require pairing
 - We'll pick the pairs for you (*you can give us feedback*)
 - One of you will drive, one will navigate (*more on this later*)
- We'll put you into a breakout room with you & your partner
- Instructors will be (mostly) available via Slack; just ping us in Slack if you want us to hop into your room
 - We'll sometimes enter your room to proactively observe while you work
- We'll provide solutions—and recommend you review when finished

Social Fridays

A strong peer network is one of the best things you can leave here with.

We schedule a Friday-after-class social block roughly every other week of the program, where we play games and hang out for a while.

It's not strictly required — but we'd love to have you join!

Weekend Assessments

- On Fridays at 6pm, we'll open your weekend assessment.
- **These are required.** *Need an extension? Talk with your adviser first!*

- Each assessment consists of:
 - Conceptual questions
 - Problem solving (*similar to repl.it problems*)
 - Building a web application
- These should take 6-12 hours. If you spend longer, let us know.
- **Assessments must be returned by Sunday 9pm.**
- **Assessments should represent only your work.**
- Staff will grade & provide feedback

IMPORTANT Our system doesn't accept late submissions

Our web application for receiving submissions stops accepting new submissions at exactly the close time. We suggest you try to submit it at least a few minutes earlier, in case you have any last-minute internet glitches.

You can submit as many times as you'd like, up to the deadline, so if you find a bug or want to add a feature, please do — we'll only look at the last submission we get from you.

We do not grade assessments not received by your deadline; you will receive a failing grade and risk having to leave the program.

Staff Roles

Advisers

One of the instructors will be assigned as your adviser.

You'll meet with your adviser 1:1 during the program for a check-in on your progress and general well-being.

Instructor Availability

Highlights

- We're fully staffed 9:30 am–5:15 pm. We have limited staffing from 5:15 pm–6 pm.
- Expect very limited availability after these hours and over weekends
- If blocking questions come up during these times, please use Slack:
 - Send all tech questions to *rithm-31* (so others can help/benefit)
 - For sensitive things, a joint Slack DM to [@Brian Aston](#) [@Sarah](#) [@Jesse Farmer](#) [@elie](#) [@Kadeem Best](#)

Code of Conduct

- We are a welcoming and inclusive community.

- **Respect each other and the staff.** We have a zero-tolerance policy for threatening or abusive language.
- **Don't be late.** Arrive promptly for the mornings and afternoons.
- **Don't pass off the work of others as your own.**
- **You must complete all assessments and maintain a GPA of at least 65.**

Tips for Success

- **Ask thoughtful questions.** Write answers down in case they come up again.
- **Take care of yourself.** Sleep helps your brain process complex material.
- **We are here to help.** When you feel overwhelmed, let us know.
- **Trust the process.** If we have concerns about you, we'll let you know.
- **You don't have to retain everything the first time.** Know enough to efficiently re-learn the details when you need to (that's what professional devs do!)
- **Recognize when you're blocked.** No progress for 20 minutes? Ask for help.

Good study habits

Preview lectures *beforehand*

Curiosity is very helpful for attention and retention.

Take notes on paper

If possible, print out our lecture notes and take notes on that.

Review lectures afterward

Spend 15-20 minutes in the evening reviewing the day's lecture(s), and re-visit over the weekend.
Spaced memorization is critical for retention.

Be organized in your curiosity

It's tempting to try to learn all the minutiae or internal details of something new—but that often clouds the fundamentals or other topics.

Getting the most out of labs

- Slow down! It's not a race to finish — it's a learning experience 🧠🐢
 - **The most important learning points** are in the first half of a lab
- Practice the debugging, testing, and documentation skills we'll teach
 - These are essential for professional developers — and hard to learn by yourself!
- Review our solution, even if you completed a working version
 - There are many good ideas and expressions in our solutions

Outside interests

“Should I attend a meetup/hackathon during the curriculum portion?” “Should I take this online course in the evenings on another topic?”

You paid a lot to be here! Invest your energy into building a solid foundation.

Meetups, hackathons, and other courses aren't going anywhere. They'll still be around once you complete the program.

“How Am I Doing?”

- 80% of 80% rule 😍
- How do you feel during labs?
- How are your assessment grades?
- Ask your adviser for feedback
- Be honest with us about where you are

Surveys

- After every lecture, we'll have a survey at <http://r31.fb.rithmschool.com/>
[<http://r31.fb.rithmschool.com/>](http://r31.fb.rithmschool.com/)
- It's only 4 questions and only takes 1 minute
- It's very helpful for us to understand how we're doing/where you are
- Please fill this out for every lecture!



Developer Tools

VSCode

Opening VSCode

Let's start with a simple question: how do you open up VSCode?

- Use the Terminal
- Drag and Drop

Open folders instead of files

- Better navigation
- Have one open VSCode tab (*you can always split windows*)

Keyboard shortcuts

- Beginning/end of line
 - macOS: «⌘←» / «⌘→» or ««Ctrl»-«a»» / ««Ctrl»-«e»»
 - Windows: «Home» / «End» or ««Ctrl»-«a»» / ««Ctrl»-«e»»
- Jumping between words
 - macOS: «↖←» / «↖→»
 - Windows: ««Ctrl»+«←»» / ««Ctrl»+«→»»
- Quick jump to file by name
 - macOS: «⌘p»
 - Windows: ««Ctrl»+«P»»
- Select multiple occurrences of word
 - macOS: «⌘d»
 - Windows: ««Ctrl»+«D»»
- Open / hide the left pane
 - macOS: «⌘b»
 - Windows: ««Ctrl»+«B»»
- Move a line
 - macOS: «↖↑» / «↖↓»
 - Windows: ««Alt»+«↑»» / ««Alt»+«↓»»
- Duplicate a line
 - macOS: «↑ ↵↓»

- Windows: ««Shift»+«Alt»+«↓»»
- All commands listed in *Command palette*
- macOS: «⌘ ↑ p»
- Windows: ««Ctrl»+«Shift»+«P»»

You can find more at [Help ▷ Keyboard Shortcuts Reference](#).

TIP **Making the control key easier**

While GUI programs like VSCode often use command keys for keyboard shortcuts, terminal programs don't — they often use control key sequences instead. Unfortunately, the control key is small and out-of-the-way on Mac keyboards, making it impractical to use those shortcuts efficiently.

Not all of our instructors do this, but Joel finds it very useful to re-map the CAPS LOCK key to become a control key (it's very rare that most of us want to type entire words in capital letters, so that key is almost never used, and wastes a premium spot on the keyboard layout).

You can do this remapping on a Mac with [System Preferences ▷ Keyboard ▷ Modifier Keys](#).

“use strict”

What's wrong with this code?

```
function someFunction() {
  for (i = 0; i < 10; i += 1) {
    // do something here
  }
}
```

No *let* or *const* — this makes (or changes) a global variable!

How can we avoid silly mistakes like this: we can enable *strict mode*!

Another silly mistake

```
function add(a, b, b) {
  return a + b + b;
}

console.log(add(2, 2, 3)); // 8
```

This is valid JavaScript!

Strict mode

- Included in the language in 2009
- Meant to prevent silly mistakes and language faults

- Add `"use strict";` at the top of every `.js` file
- Instead of silent failures, strict mode throws errors
- Always include it at the top of your JS files

What it looks like

`demo/strict-mode.js`

```
firstName = "Maddy"; // Uncaught ReferenceError: firstName is not defined

function add(a, b, b) {
  return a + b + b;
}

console.log(add(2, 2, 3));
// Uncaught SyntaxError: Duplicate parameter name not allowed in this context
```

Using the console

How and what to log

- Always include a message
 - `console.log("myFunction x=", x);`
 - `console.log("myFunction usernames=", usernames);`
- Use multiple parameters instead of +
 - `console.log("a=", a, "b=", b);`
- The most helpful log is sometimes the simplest one:
 - `console.log("game.js loaded");`
 - `console.log("myFunction ran");`

TIP Other kinds of logs

You've mostly seen `console.log`, but there are other message functions:

- `console.debug(...)`
- `console.warn(...)`
- `console.error(...)`

These are very useful, since `.warn` and `.debug` appear brightly colored and can make it easy to find the important messages you need.

You can filter which appear with **Default levels** menu. So, you can hide the less-critical `.debug` stuff and only show the more important `.log` stuff. Note that `.debug` is hidden by default — you'll need to enable that with the Default levels dropdown.

Simple testing with `console.assert`

A fine way to test your code is to use `console.assert()`

```
function add(a, b) {
  return a + b;
}

console.assert(add(10,20) === 30);

console.assert(add(10) === 30);
// Assertion failed: console.assert
```

TIP Making your assertions more readable

You can pass a second parameter, a string that appears in output on failure.

```
console.assert(add(10, 20) === 30, "expected: add(10,20) === 30");
```

TIP Tips for JavaScript DOM Work

- The console is wonderful for debugging JavaScript – what about issues with HTML and CSS?
- That's where the *Elements* tab comes in
- Using the elements tab you can:
 - View the DOM
 - Modify HTML and CSS (*not permanently*)
 - Examine what CSS has been computed and applied
 - Examine what event listeners have been added

demo/elements.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet" href="elements.css">
</head>
<body>
  <h1>Open the elements tab with the selector! Command + Shift + C</h1>
  <h2>Oh, the things you'll see!</h2>
  <ul>
    <li>All your elements</li>
    <li>All the CSS you imagine</li>
    <li>You'll even see event listeners added!</li>
  </ul>
  <div id="box">
    I am a box!
  </div>
  <button>Click me and then open the console!</button>
  <script src="elements.js"></script>
</body>
</html>
```

IMPORTANT Preventing Caching

- Chrome will try to cache things for you by default
- This is helpful when loading pages as quickly as possible, but frustrating in development!
- Make sure you check the checkbox next to Disable Cache and **keep your dev tools open when coding**

GitHub Gists

GitHub Gist is a super easy way to instantly share code, notes, and snippets.

<https://gist.github.com/> <<https://gist.github.com/>>

- great for:
 - when you see a snippet of useful code
 - a note you want to find later
 - sharing something quickly with someone else

Your Turn

- What are tools that have helped you?
- Also, we have other important tips in the handout! 



Intermediate JS 1

Goals

- Have a clear understanding of primitive/reference types
- Understand common boolean logic traps
- Learn how to catch errors with *try/catch*
- Learn to write tests with Jasmine

Types in JavaScript

Primitives

- Primitive types are also known as *scalar* or *simple* types.
- Primitive types fit into memory easily.
- JavaScript has six native primitive types:
 - *null*
 - *undefined*
 - *Boolean*
 - *Number*
 - *String*
 - *Symbol* (not very important for now)

NOTE **Symbol type**

The symbol type is fairly obscure and, outside of very specific use cases, you'll probably not use it directly in JavaScript.

Reference Types

- A reference type can contain multiple values.
- Reference types are also known as: *complex* or *container* types.
- Reference types in JavaScript include:
 - *Object*
 - *Array* (a kind of object)
 - *Function* (a kind of object)

Assigning a primitive

When a primitive type is assigned to another variable, a copy of the value of the primitive type is saved in the variable.

Assigning a reference

When a reference type is assigned to another variable, the address of that value is what is copied over.

```
let names = ["Joel", "Brit", "Elie"];
let namesCopy = names;
namesCopy[2] = "Nate";
names[2] === "Nate"; // true or false?
```

Comparing two primitives

- When `==` is used on primitives, it checks the **type and value**.

```
let firstName = "Elie";
let firstNameAgain = firstName;
firstName === firstNameAgain; // true
```

```
firstName === "Elie"; // true
firstNameAgain === "Elie"; // true
```

Comparing two references

- When `==` is used on reference types, it checks the reference.
- If the variables point to the same bit of memory, the comparison is true.

```
let nums = [1, 2, 3, 4];
let nums2 = nums;
nums === nums2; // true
```

```
nums === [1, 2, 3, 4]; // false
nums2 === [1, 2, 3, 4]; // false
```

So how do we compare two objects?

The quick way:

```
let data = {name: "Elie"};
let data2 = {name: "Elie"};

let dataJson = JSON.stringify(data);
let data2Json = JSON.stringify(data2);
dataJson === data2Json; // true
```

Another option would be to recursively loop through the objects and make sure each of the properties/values are the same.

Boolean Logic

JavaScript has three common logical operators: `||`, `&&`, and `!`.

`!` binds very closely, so `! a || b` means “invert a-truth, then OR with b”

To spread NOT across elements, do it like `! (a || b)`

AND and OR

These test “truthy” or “falsy” expressions, not just *true* and *false*

They evaluate to the “*determining subpart*”:

```
let x = 42
let y = 0;

let a = x || y;    // a === 42
let b = x && y;   // b === 0
```

TIP An excellent resource on truthy/falsy

<https://www.sitepoint.com/javascript-truthy-falsy/> <<https://www.sitepoint.com/javascript-truthy-falsy/>>

Ternary Operator

JavaScript also has one *ternary* (“three-way”) operator, `? :`

It’s useful for setting a variable to one of two possible expressions.

```
let outcome = (timer < 0) ? "game over" : "keep playing";
```

If your ternary isn’t short and sweet, break it into lines like this:

```
let outcome = (timer < 0 && numShips > 0)
? "game over"
: "keep playing";
```

It's best to use this for evaluating expressions, and avoid using it for anything with a "side-effect":

Ugh so awful don't do this 😞

```
(piece !== null)
  ? winner = "black"
  : getNextMove();
```

So much better 😊

```
if (piece !== null) {
  winner = "black";
} else {
  getNextMove();
}
```

Error Handling

JavaScript can handle errors in code:

```
try {
  eaten += garden[y][x]; // might be out-of-bounds
} catch (err) {
  // can now print `err`, or do something different
  eaten = 0;
}
```

You can also throw errors to indicate a problem:

```
function getHighestGrade(tests) {
  if (tests.length === 0) {
    throw new Error("No tests provided!");
  }
  // ... rest of code follows ...
}
```

This is often much clearer than returning a "magic" value, like `-1` or `undefined`.

Writing Tests

Why Test?

- Manually testing software is boring
 - So we tend to not re-run things that "work"
 - And therefore don't notice when they break
- Tests often clarify expectations of a function
 - What should legal input/output be?
- Tests are often a great way to understand what code does
- It's a core skill for professional devs

Jasmine

- *Jasmine* is a popular JavaScript testing framework

- We'll use Jasmine for testing until we get to Python
- We'll see another framework, *Jest*, with Node and React

Running Tests in the Browser with Jasmine

```
<!doctype html>
<html>
<head>
<title>Taxes Tests</title>

<!-- include CSS for Jasmine -->
<link rel="stylesheet"
      href="https://unpkg.com/jasmine-core/lib/jasmine-core/jasmine.css" />
</head>
<body>

<!-- include JS for Jasmine -->
<script src="https://unpkg.com/jasmine-core/lib/jasmine-core/jasmine.js"></script>
<script src="https://unpkg.com/jasmine-core/lib/jasmine-core/jasmine-html.js"></script>
<script src="https://unpkg.com/jasmine-core/lib/jasmine-core/boot0.js"></script>
<script src="https://unpkg.com/jasmine-core/lib/jasmine-core/boot1.js"></script>

<!-- include your JS & test file -->
<script src="taxes.js"></script>
<script src="taxes.test.js"></script>
</body>
</html>
```

Then open this HTML page in browser

Writing tests with Jasmine

Write your functions that will be tested:

demo/taxes.js

```
function calculateTaxes(income) {
  if (income > 30000) {
    return income * 0.25;
  } else {
    return income * 0.15;
  }
}

console.log(calculateTaxes(500))
```

Write a test file:

demo/taxes.test.js

```
it('should calculate lower-bracket taxes', function () {
  expect(calculateTaxes(10000)).toEqual(1500);
  expect(calculateTaxes(20000)).toEqual(3000);
});

it('should calculate higher-bracket taxes', function () {
  expect(calculateTaxes(50000)).toEqual(12500);
  expect(calculateTaxes(80000)).toEqual(20000);
});
```

Let's break that down

`demo/taxes.test.js`

```
it('should calculate lower-bracket taxes', function () {
  expect(calculateTaxes(10000)).toEqual(1500);
  expect(calculateTaxes(20000)).toEqual(3000);
});

it('should calculate higher-bracket taxes', function () {
  expect(calculateTaxes(50000)).toEqual(12500);
  expect(calculateTaxes(80000)).toEqual(20000);
});
```

- *Test cases* are functions passed to `it(...)`
- First argument become test case name (shown by Jasmine)

`demo/taxes.test.js`

```
it('should calculate lower-bracket taxes', function () {
  expect(calculateTaxes(10000)).toEqual(1500);
  expect(calculateTaxes(20000)).toEqual(3000);
});

it('should calculate higher-bracket taxes', function () {
  expect(calculateTaxes(50000)).toEqual(12500);
  expect(calculateTaxes(80000)).toEqual(20000);
});
```

- Test cases can contain any normal code plus “expectations”
- Format is `expect(someValue).someMatcher(...)`

Matchers

`.toEqual(obj)`

Has the same values (eg, different lists with same values match)

`.toBe(obj)`

Is the same object (eg, different lists with same items don't)

`.toContain(obj)`

Does object/array contain this item?

`.not.`

Add before matcher to invert (eg `expect(...).not.toEqual(7)`)

<https://jasmine.github.io/api/edge/matchers.html> <<https://jasmine.github.io/api/edge/matchers.html>>

What To Test

- Test every function in at least one way
- Think about “edges”
 - What if the list were empty?
 - What about non-integer numbers?

- What if the file can't be found?
- Is the first case/last case handled differently?

Testable Code

Write code that is easier to test!

- More functions & smaller functions: easier to test (& debug!)
- Don't mix logic & UI in a function

Meh

```
function playTicTacToe() {
  // ...
  let winner = checkForWinner();
}

function checkForWinner() {
  // code for checking board here...

  if (winner) {
    alert(winner + " wins!");
  }
  return winner;
}
```

Yay!

```
function playTicTacToe() {
  // ...
  let winner = checkForWinner();
  if (winner) {
    announceWinner(winner);
  }
}

function checkForWinner() {
  // code for checking board here...

  return winner;
}

function announceWinner(winner) {
  alert(winner + " wins!");
}
```

Debugging

From then on, when anything went wrong with a computer, we said it had bugs in it.

—Rear Admiral Grace Hopper, 1945

While the term “bug” had been in use for several decades to refer to flaws in designs, it gained prominence after the first real bug was a 2-inch-long moth from the Harvard Mark I experimental computer at Harvard in August 1945.

To Assume Makes an ...

- Part of debugging is clarifying assumptions — so you can challenge them
- Even **preparing** to explain problem can reveal the assumptions



Bad Debugging Strategies

Please **don't** do these:

- Make changes based on guesses with no evidence
- Reread your code again and again
- Change several things at once

Learn From Debugging

As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered.

I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

—Maurice Wilkes, 1949

Every debugging step should **teach you something**

It's helpful to think about what a step will teach you before you do it

Print Debugging

The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

—Brian Kernighan, *Unix for Beginners* (1979)

- Yes, it's simple
- Yes, it's usually the right thing to do

```
function checkForWinner(board, lastPlay) {  
    console.log("checkForWinner", board, lastPlay);  
    //           func name ^      vars ^  
  
    // ... rest of code ...  
}
```

Also: *keep your debugging statements!*

Finding the right log

Serious software benefits from *layers of logging*:

`console.log()`
Something low-level and useful for debugging

`console.info()`
More important, and might help understand the flow of things

`console.warn()`
Even more important

`console.error()`
Looks like an error — really emphasizes the danger

Checking the unexpected

You can use `console.assert(check, msg)` to highlight problems:

```
function getHighestGrade(tests) {  
  for (let test of tests) {  
    console.assert(test <= 100, test);  
    // ...  
  }  
}
```

Your Turn



Intermediate JS 2

Goals

- Use new features of JavaScript to write better code
- Compare and contrast `var`, `let`, and `const`
- Learn advanced techniques for manipulating objects and arrays
- Learn common “functional idioms”, like `map`, `filter`, `find`, etc

String Template Literals

Strings may be declared with *back ticks* (the “grave accent” `\``).

These are special strings that allow for *interpolation* of variables and expressions using `${ }` . This is often cleaner than old-school string concatenation.

```
const firstName = "Jian";
const lastName = "Yang";
const title = "web developer";

// ughhh, so much concatenation...
const oldWay = "I'm " + firstName + " " + lastName +
    ", and I'm a " + title + ".";

// much clearer
const newWay = `I'm ${firstName} ${lastName}, and I'm a ${title}.`;
```

Interpolating Expressions

You can interpolate full JavaScript expressions.

Inside the `${ }` block is regular JS — expression here are evaluated and stringified.

```
function addTwo(x, y) {
    return x + y;
}

const a = 99;
const b = 42;

const result = `${a} plus ${b} is ${addTwo(a, b)}.`;
```

Variable Scopes

let

The *let* keyword creates a **block-scoped** variable: a variable that only exists inside a code block.

What Is A Code Block?

Essentially any pair of curly braces (outside of object syntax).

```
{ // this is a code block
  let x = 5;
  var y = 10;
}

console.log(x);
// ReferenceError: x is not defined

console.log(y);
// 10
```

Where Are Code Blocks Commonly Used?

You'll mostly use code blocks in *for* loops and *if* statements.

```
if (x > 10) {
  let happy = true; // happy lives ONLY in block
}

// can't use it outside the block
console.log(happy); // ReferenceError: happy is not defined
```

They are implied for the next line if you leave them out.

```
if (x > 10)
  let happy = true; // still a code block

console.log(happy); // ReferenceError
```

WARNING Not good style!

While it's convenient to leave out the curly braces if the body of your `if` or `for` statement is only one line long, it's risky. Imagine you have this code:

```
if (x > 10)
let happy = true; // this is still a code block
```

And then someone wants to add another line to the `if` body:

```
if (x > 10)
let happy = true; // this is still a code block
console.log("I'm glad you're happy!"); // BUG
```

While it *looks like* that `console.log` only happens if `x` is greater than 10, it will *always* run.

Therefore, most style guides suggest that you either:

- always use curly braces around the bodies of `if` and `for`, even if they are one line long OR
- keep the body of the one-line `if` or `for` on the *same line*, like so:

```
if (x > 10) let happy = true;
// rest of code continues here ...
```

```
for (var i = 1; i < 4; i++) {
  console.log(i);
}

// 1
// 2
// 3

console.log(i);
// 4
```

```
for (let i = 1; i < 4; i++) {
  console.log(i);
}

// 1
// 2
// 3

console.log(i);
// ReferenceError: i is not defined
```

It can be reassigned but not redeclared (unlike `var`).

```
let z = 5;
z = 25;
let z = 10;
// SyntaxError: Identifier 'z' has already been declared
```

`const`

Declaring with `const` prevents being reassigned *or* redeclared.

```
const PI = 3.14;

PI = 15; // TypeError: Assignment to constant variable

const PI = 5; // SyntaxError
```

`const` is also block-scoped, like `let`.

```
{
  const x = 10;
}

console.log(x); // ReferenceError: x is not defined
```

const variables can still be *mutated*:

```
const beatles = ["John", "Paul", "Ringo", "George"];
beatles.push("Yoko");
```

Comparison of Variable Declaration Keywords

Keyword	Can Reassign	Can Redeclare	Can Mutate	Scope Rules
<i>var</i>	✓	✓	✓	function scope
<i>let</i>	✓	✗	✓	block scope
<i>const</i>	✗	✗	✓	block scope

for...of

- The `for...of` statement creates a loop over an **iterable**
- Shorter loop syntax useful when you want just values, not the indexes
- The loop will iterate through the entire length

Iterable

- Something that can be iterated over:
 - *String*
 - *Map*
 - *Set*
 - *Array*

for...of vs for...in

- `for...in` is used to iterate over *keys in an object*
- `for...of` is used to iterate over *values in an iterable*

```

const numbers = [5, 10, 15];

for (let number of numbers) {
    console.log(number);
}

// 5
// 10
// 15

```

```

// arrays are objects!
const numbers = [5, 10, 15];

for (let number in numbers) {
    console.log(number);
}

// 0
// 1
// 2

```

WARNING Don't use `for...in` with arrays

Since JS arrays are built on top of JS objects, they have access to all of the methods and capabilities of objects. Therefore, `for...in` can work with an array, providing the *index* of each item, in turn:

```

const nums = [10, 20];

// Best way to loop by index:
for (let i = 0; i < nums.length; i += 1) {
    console.log(`#${nums[i]} is at ${i}`);
}
// 10 is at 0
// 20 is at 1

// Not good way to loop by index:
for (let i in nums) {
    console.log(`#${nums[i]} is at ${i}`);
}
// 10 is at 0
// 20 is at 1

```

However, as you get more advanced, you'll increasingly come create or use arrays which have properties set on them: arrays can have arbitrary properties, just like objects in general. These properties don't have numeric keys and aren't "items" in the array, so they shouldn't appear when you loop — but they will when using `for...in`:

```

const nums = [10, 20];
nums.allEvens = true;

// Best way to loop by index:
for (let i = 0; i < nums.length; i += 1) {
    console.log(`#${nums[i]} is at ${i}`);
}
// 10 is at 0
// 20 is at 1

// Not good way to loop by index:
for (let i in nums) {
    console.log(`#${nums[i]} is at ${i}`);
}
// 10 is at 0
// 20 is at 1
// true is at allEvens  <-- YECK

```

The essence of this tip: when you want to loop over an array by the index, do it with a traditional `for` loop, not `for-in`.

Default Parameters

If you have parameters that might be undefined, (eg, the user didn't pass arguments to them), you can easily assign default values:

```
function multiply(a, b = 1) {
  return a * b;
}

console.log(multiply(5, 2));
// 10

console.log(multiply(5));
// 5
```

This only works for *undefined* values, not other falsy values like *null*.

Arrow Functions

- Arrow functions are shorthand for anonymous functions
- They cannot be named and they only work as function expressions.
- They are ideal for shortening callbacks.

```
[1, 2, 3].forEach(function (n, idx) {
  console.log(n, idx);
});
```

is the same as

```
[1, 2, 3].forEach((n, idx) => {
  console.log(n, idx);
});
```

TIP Generally, avoid *forEach*

We're using *forEach* here just as an example of a callback-taking function, but you should avoid using it. It predates the addition of *for...of*, which is a far-more capable and flexible way to generally loop over an array, and, without the overhead of a function call, will always outperform *forEach*.

```
[1, 2, 3, 4, 5].filter(function (n) {
  return n % 2 === 0;
});
```

is the same as

```
[1, 2, 3, 4, 5].filter((n) => {
  return n % 2 === 0;
});
```

Arrow functions have an implicit return if curly braces are omitted:

```
/* square everything */

let nums = [1, 2, 3];

let arrSquared = nums.map(n => n ** 2); // [1, 4, 9]
```

Arrow Functions & *this*

Arrow functions do not have their own *this* context: if your function uses the keyword *this*, it cannot be an arrow function.

We'll revisit this when we talk about OOP.

Summary

- Can only be used as shorthand for anonymous function expressions
- Must put parentheses around parameters if there are 0 or 2+ parameters
- Return statement is implied if you leave out curly braces
- They do not make their own *this*

Rest / Spread Operator

```
function sumMany(...nums) {
  let sum = 0;
  for (let n of nums) {
    sum += n;
  }
  return sum;
}

sumMany(5, 10); // 15

sumMany(10, 10, 10, 10, 10, 10, 10); // 70

sumMany(1); // 1
```

`...nums` collects additional arguments into single *nums* array.

Collecting Remaining Arguments

You can also specify several named parameters and collect the rest.

```

function oneOrMoreArguments(first, ...more) {
  console.log(first);

  for (arg of more) {
    console.log(arg);
  }
}

```

Here, *first* will be first item and *more* will be array of everything else.

Spread Operator

The `...` syntax, in a different context, is called the *spread operator*.

For example: when calling a function, you can “spread out” array elements:

```

function takesFour(one, two, three, four) {
  console.log(one);
  console.log(two);
  console.log(three);
  console.log(four);
}

const names = ['Spencer', 'Brit', 'Kate', 'Joel'];

takesFour(...names);
// Spencer
// Brit
// Kate
// Joel

```

NOTE Spreading objects

You can use the same operator to copy over pieces of an object into a new object.

```

const whiskeyTheDog = {
  name: 'Whiskey',
  species: 'canine',
  cool: true,
};

// make a new dog but override the 'name' key
const gandalfTheDog = { ...whiskeyTheDog, name: 'Gandalf' };

```

This gives us this object as *gandalfTheDog*:

```
{
  name: 'Gandalf',
  species: 'canine',
  cool: true,
}
```

Object Shorthand

ES2015 provides quite a few enhancements for objects in ES2015

When key name is the same as variable holding the desired value, you don't have to repeat them:

```
// ES5
const developer = {
  firstName: firstName,
  lastName: lastName,
}
```

```
const firstName = "Spencer";
const lastName = "Armini";

// ES6
const developer = {
  firstName,
  lastName,
}
```

Object Methods

A nice shorthand when a key in an object represents a function.

```
// ES5
var instructor = {
  sayHello: function () {
    return "Hello!";
  }
}
```

```
// ES2015 - do NOT use arrow func!
let instructor = {
  sayHello() {
    return "Hello!";
  }
}
```

Computed Property Names

```
// ES5
var firstName = "Elie";
var instructor = {};
instructor(firstName] = "That's me!";

instructor.Elie; // "That's me!"
```

```
// ES2015
let firstName = "Elie";
let instructor = {
  [firstName]: "That's me!",
}

instructor.Elie; // "That's me!"
```

Array and Object Destructuring

Object Destructuring

JavaScript programmers take things out of objects all the time.

Older way:

```

let userData = {
  username: 'janet',
  id: 12345,
  firstName: 'Janet',
  lastName: 'Cho',
  age: 34,
};

let username = userData.username;
let firstName = userData.firstName;
let lastName = userData.lastName;
let id = userData.id;

```

So they came up with some “*syntactic sugar*”:

```

const userData = {
  username: 'janet',
  id: 12345,
  firstName: 'Janet',
  lastName: 'Cho',
  age: 34,
};

// declare variables: username, firstName, lastName, id
//   values taken from the keys of the same name in userData

const { username, firstName, lastName, id } = userData;
console.log(username); // janet
console.log(id); // 12345

```

Destructuring and Spread

```

const userData = {
  username: 'janet',
  id: 12345,
  firstName: 'Janet',
  lastName: 'Cho',
  age: 34,
};

// extract the password key; collect the rest in 'user'
const { id, ...user } = userData;

console.log(user);
// {
//   username: 'janet',
//   firstName: 'Janet',
//   lastName: 'Cho',
//   age: 34,
// }

```

TIP You Can Apply The Same Concept To Arrays!

```
const hobbies = ['teaching', 'music', 'hiking', 'art'];

const [first, second, ...others] = hobbies;

console.log(first); // 'teaching'
console.log(second); // 'music'
console.log(others); // ['hiking', 'art']
```

You can use array destructuring to do a fancy 1-line value swap.

With this, you can switch the values of two variables without making a third “temp” variable.

```
let a = 1;
let b = 3;

[a, b] = [b, a];

console.log(a); // 3
console.log(b); // 1
```

Array.from()

Sometimes JavaScript gives you array-like objects that you wish were arrays.

This happens all the time with vanilla JS DOM queries.

```
// querySelectorAll returns a NodeList...
let paragraphsNodeList = document.querySelectorAll('p');

// a NodeList is NOT an Array!
Array.isArray(paragraphsNodeList); // false

let paragraphs = Array.from(paragraphsNodeList);
Array.isArray(paragraphs); // true
```

Now you can easily make an array out of those things with *Array.from()*

NOTE *Array.from()* Under The Hood

It works by looking for anything with a `length` property.

Then it iterates over the items and pushes them into an array.

```
const cat = 'cat';
Array.from(cat);
// ['c', 'a', 't']
```

You can use this trick to build empty arrays of a certain size:

```
Array.from({ length: 5 });
// [undefined, undefined, undefined, undefined, undefined]
```

A particularly nice idiom is to combine this with the `Set` feature to remove duplicates from an array. Since sets don't contain duplicates, we can de-duplicate an array easily by putting it into a set and then turning it back into a new array:

```
const nums = [1, 2, 3, 4, 2, 3, 1];
const unique = Array.from(new Set(nums));
```

Functional Idiom Functions

map()

- Run a callback for each value in the array
- Return a new array with returned values from the callback
- Used to transform each item in array into new array

```
const nums = [1, 2, 3, 4];
const doubledValues = nums.map(num => num * 2);
```

filter()

- Run a callback for each value in the array
- Returns new array with elements where callback returns something truthy

```
const nums = [1, 2, 3, 4];
const evens = nums.filter(num => num % 2 === 0);
```

Finding Things In Arrays

What if you just want to find the first or a single thing?

For finding simple values (eg, numbers or strings) in arrays, it's easy:

```
let numbers = [1, 5, 7];

nums.includes(7);      // true or false
nums.indexOf(7);       // index or -1 if not found
```

This doesn't work for finding complex things (eg, arrays/objects):

```
let instructors = [{name: "Brit"}, {name: "Joel"}];

instructors.includes({name: "Joel"});   // false
instructors.indexOf({name: "Joel"});     // -1
```

find()

“Find first matching item”

- Invoked on arrays
- Accepts callback with value, index and array (like *forEach*, *map*, *filter*, etc.)
- Returns the value found or undefined if not found

```
let instructors = [{name: "Brit"}, {name: "Joel"}];

instructors.find(o => o.name === "Joel"); // {name: "Joel"}
```

findIndex()

“Find first matching item, returning index” (Like *find*, but returns index of item or -1 if no match)

```
const instructors = [{name: "Brit"}, {name: "Joel"}];

instructors.findIndex(o => o.name === "Brit"); // 0
```

some()

Does at least one item match?

- Iterates through array, running callback on each value
- If callback returns true for at least one value, return true (else false)
- The result of the callback will ALWAYS be a boolean

```
let numbers = [1, 1, 3, 5, 2];

// are any numbers even?
if (numbers.some(n => n % 2 === 0)) {
  // ...
}
```

every()

Do all items match?

- Iterates through array, running callback on each value
- If callback returns false for any single value, return false (else true)
- The result of the callback will ALWAYS be a boolean

```
let numbers = [1, 1, 3, 5, 2];

// are all numbers even?
if (numbers.every(n => n % 2 === 0)) {
  // ...
}
```



Professional Coding

Goals

Learn critical parts of professional programming:

- How is pro coding different than regular coding?
- Structuring programs
- Good style
- Appropriate documentation

Pro Coding

How is professional coding different?

- Often involves dozens or hundreds of programmers
 - Many of whom will turn over during the codebase lifetime
- Real-world programs need to be maintained for years
 - And new features needed to be added

Structure

“use strict”;

- The first line of your `.js` files should always be `"use strict";`
 - This includes your test files!
 - This ensures your JavaScript is operating in *strict mode*
- Allows JavaScript to catch more mistakes and throw clear errors
 - Without this, some mistakes may “fail silently”
 - Debugging is much, much harder in silence

Functions

- Functions are the most basic unit of thought
- Most new programmers create too few & too long

```

function playTacToe() {
  // setup board
  // ...

  while (!winner) {
    // show board
    for (let y = 0; y < board.height; y++)
      for (let x = 0; x < board.width; x++)
        console.log(board[y][x]);

    // get move
    // ...

    // check for win ...
    // ...
  }
}

```

```

function playTicTacToe() {
  let winner;
  setupBoard();

  while (!winner) {
    displayBoard();

    const move = getValidMove();
    makeMove(move);

    winner = getWinner();
    showWinMsg(winner);
  }

  // other functions follow ...
}

```

- Functions improve readability
- Longer than 20-40 lines? Do it!
- Functions promote code re-use
- Has different logic parts? Do it!
- Functions improve testability
- Mixes different concerns? Do it!
 - User interface
 - Core logic
 - Reading/writing files
 - On the fence? Do it!

Separating Concerns

```

function calcAndShowTaxes() {
  const income = incomeField.value;
  const state = stateField.value;
  const taxes = TAX_LOOKUP[state] * income;
  outputArea.innerHTML = `You owe ${taxes}`;
}

```

This *mixes concerns*: getting UI data, calculating, showing

```

function getFormData() {
  const income = incomeField.value;
  const state = stateField.value;
  return {income, state};
}

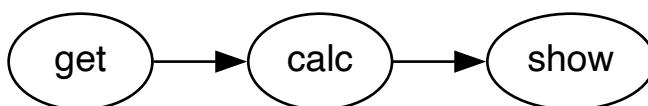
function calcTaxes(income, state) {
  const amount = TAX_LOOKUP[state] * income;
  return amount;
}

function showTaxResults(amount) {
  outputArea.innerHTML = `You owe ${amount}`;
}

```

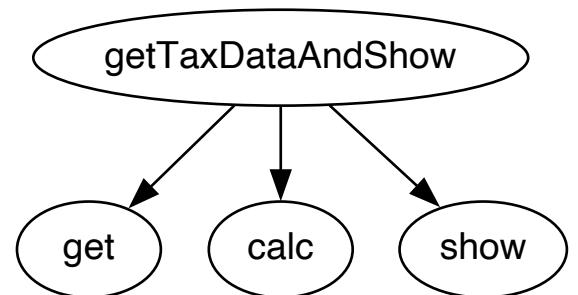
Controller Functions

```
function getFormData() {  
    const income = incomeField.value;  
    const state = stateField.value;  
    calcTaxes(income, state);  
}  
  
function calcTaxes(income, state) {  
    const amount = TAX_LOOKUP[state] * income;  
    showTaxResults(amount);  
}  
  
function showTaxResults(amount) {  
    outputArea.innerHTML = `You owe ${amount}`;  
}
```



Can't use independently or test easily

```
function getFormData() {  
    // ...  
    return {income, state};  
}  
  
function calcTaxes(income, state) {  
    return TAX_LOOKUP[state] * income;  
}  
  
function showTaxResults(amt) {  
    outputArea.innerHTML = `You owe ${amt}`;  
}  
  
function getTaxDataAndShow() { // better!  
    const {income, state} = getFormData();  
    const amount = calcTaxes(income, state);  
    showTaxResults(amount);  
}
```



Real functions over anonymous functions

It's fine to use anonymous functions for short callbacks:

```
// as a traditional anonymous function expression  
let doubled = nums.map(function (num) { return num * 2 });  
  
// as an arrow function --- usually better  
let doubled = nums.map(num => num * 2);
```

But real functions deserve real function declarations

```
button.addEventListener(  
    "click", function () {  
        // ...  
    });  
  
// You can't test this without clicking!  
// It doesn't have a name
```

```
function placePiece(evt) {  
    // ...  
}  
  
button.addEventListener(  
    "click", placePiece);
```

Global Variables

- Variables used through program & not specific to a function

- Often not a great strategy (but sometimes is ok)
- It's often better to pass to functions explicitly

```
let board; // global variable

function playTicTacToe() {
  let winner;
  setupBoard();

  while (!winner) {
    displayBoard();

    const move = getValidMove();
    makeMove(move);

    winner = getWinner();
    showWinMsg(winner);
  }
}
```

```
function playTicTacToe() {
  let winner;
  let board = setupBoard();

  while (!winner) {
    displayBoard(board);

    const move = getValidMove(board);
    makeMove(board, move);

    winner = getWinner(board);
    showWinMsg(winner);
  }
}
```

It's much easier to see which functions use the *board* variable now.

Style

Semicolons

Put semicolons at the end of lines:

```
let x = 42;
```

but not at end of code blocks:

```
if (x === 42) {
  console.log("ok");
} // <- no semicolon here

function double(x) {
  return x * 2;
} // <- no semicolon here
```

NOTE “But I heard semicolons aren’t needed?”

Whoever told you that is wrong.

JavaScript requires semicolons between statements/expressions. It will inject those semicolons itself where it believes they should have gone, but the algorithm it uses for this is not always what people expect. If you let JS add semicolons for you, you will have subtle bugs in some code.

Blank lines

Use blank lines liberally: between functions and “different thoughts”:

```

function getDiscountedPrice(state, age) {
  let price = 5.00;

  if (age < 18 || age >= 65) {
    price -= 1.00;
  }

  if (state === 'CA') {
    price -= 2.50;
  } else if (state === 'OR') {
    price -= 1.00;
  }

  // ...
}

```

Choosing between *let* and *const*

When a variable will never change, make it *const*:

```

function calculateTax(state, income) {
  const rate = (state === "CA") ? 0.08 : 0.06;
  // ...
}

```

This can both make it easier to find bugs, and helps readers.

Global Constants

- Constant in global scope that are **always** the same
- It's very helpful for removing “magic” numbers and strings from code
- Give them names like *ALL_CAPS* to help others understand their nature

```

const HEIGHT = 10;
const WIDTH = 10;

function makeMove(x, y) {
  if (x >= WIDTH) throw new Error("Invalid!");
  // ...
}

```

Naming Things

- Naming things well is a serious, core skill for developers.
- We expect you to learn & practice it here.
- Generally: in JavaScript, names are *camelCase* not *snake_case*
 - Some exceptions: *GLOBAL_CONSTANTS* and *Classes*

Naming Functions

- Be clear – it's fine to be long
- Good function names are “verby”
 - *calculateTax* is much more clear than *tax*

Variables

How could you improve these names?

demo/bad-names.js

```
function evens(arr) {
  let newArr = [];

  for (let num = 0; num < arr.length; num++) {
    if (arr[num] % 2 === 0) {
      newArr.push(num);
    }
  }

  return newArr;
}
```

- What's in the array?
- Is *num* really the number?
- Easy to miss function goal

demo/good-names.js

```
function findIndexesOfEvens(nums) {
  let indexes = [];

  for (let i = 0; i < nums.length; i++) {
    if (nums[i] % 2 === 0) {
      indexes.push(i);
    }
  }

  return indexes;
}
```

- Clear that *i* is the *index* of the number
- More clear what function goal is

- Good names should be clear, but not necessarily long
 - *i* is a fine name for “index in a simple loop”
- Focus on the purpose of data, not structure
 - *nums* or *temperatures* is much better than *arr*
 - Make plural things plural! (*nums* or *num*)
- Global variables need longer & descriptive names
 - Without context of being in function, harder to figure out what they're for

NOTE Naming URLs

When you have a variable that is a URL, you need to name it very carefully — remember, capitalization tells the reader what kind of thing it is.

Very common, good approach:

`demo/url-names-good.js`

```
// A global constant, since I am in ALL_CAPS
const URL = "http://foo.com";

// A class, since I have a leading Capital
// letter but am not in ALL_CAPS
class Url {
}

// A variable, not a global const or a class
let url = process.argv[2];
```

Remember that global consts are ALWAYS the same, so don't do this:

`demo/url-names-bad.js`

```
// DON'T DO THIS
const URL = process.argv[2];
```

That name, `URL`, suggests this is always the same — but it's not always the same; it changes based on how you run the program, and therefore is not a global-const and should not get the name `URL`.

Throwing Errors

```
function calcTaxes(state, income) {
  if (income < 0) return "Invalid";
  // ... more code follows ...
  return amount;
}
```

- What does this function return?
 - An amount? Or is it an error?
- How will you check whether it worked?
 - Add an `if` statement to the caller?
- What happens if string changes?
- This mixes possible UI and logic, too

TIP Also bad: returning -1

```
function calcTaxes(state, income) {
  if (income < 0) return -1;
  // ... more code follows ...
  return amount;
}
```

This might be a bit better — at least it doesn't return a string-or-number — but it's very fragile: you still might propagate the invalid-string by forgetting to check if there was a problem, and someone might edit that string breaking the `if` conditions that checked it.

```
function calcTaxes(state, income) {
  if (income < 0) throw new Error();
  // ... more code follows ...
  return amount;
}
```

You can use *try/catch* to catch an error if you can handle it in your code.

Documenting Programs

“Docstring” Function Comments

- All functions should get a *docstring* comment
 - What does function do?
 - What does it take?
 - What does it return?

```
/** Checks `board` for 3-in-a-row; returns "X", "O", or null */

function checkForWinner(board) {
  // ...
}
```

We expect you to practice this here!

NOTE What’s a “docstring”?

This term comes from the Python community; it refers to a particular feature in the language for creating documentation. However, this *requirement* exists for programmers in any language, and these are often called things like “documentation comments” or “API documentation comments” or other similar terms.

Here at Rithm, we tend to call them “docstrings”, no matter what language we’re talking about: it’s a nice short, simple term for them.

Docstrings as contracts

Consider this docstring and function:

```
/** Convert Kelvin to Fahrenheit. Returns number. */

function getFahrenheit(celsiusTemp) {
  if (kelvinTemp < 0) {
    return "Can't go below absolute zero!";
  }
  // ...
}
```

The docstring promises that this function returns as number — but under some circumstances, it returns a string. This is a violation of the “contract” that docstring suggests.

Other programmers on a project will want to use your functions and expect for them to do *exactly* what the docstring promises, without their having to read the code or keep up with subtle changes you make to its

implementation. Therefore, your docstring should always be very clear about exactly what a function will return:

```
/** Convert Kelvin to Fahrenheit.
 *
 * Returns number or an error message.
 */

function getFahrenheit(celsiusTemp) {
  if (kelvinTemp < 0) {
    return "Can't go below absolute zero!";
  }
  // ...
}
```

This is an important principle: always decide firmly what a function returns and document that clearly.

As a side note: while this function is now documented more accurately, a better approach would be for it to throw an error, rather than returning a string:

```
/** Convert Kelvin to Fahrenheit.
 *
 * Returns number; throws error if invalid.
 */

function getFahrenheit(celsiusTemp) {
  if (kelvinTemp < 0) {
    throw new Error("Can't go below absolute zero!");
  }
  // ...
}
```

Comments

Some comments are pointless:

```
price = price * 2; // double price
```

But some are extremely valuable:

```
price = price * 1.25; // excise tax
```

Focus on “why” comments

- It’s rare you need to explain *what* a line of code does
 - If so, it’s probably too complex
 - Consider breaking it into two or three lines
 - Consider whether better variable names would help

```
if (cust[0] >= 18 &&
  cust[1].length === 0 &&
  reduce((n, acc) => n + acc, sals)
  > 10000)      // wtf?
```

```
const votingAge = cust[0] >= 18;
const numVotes = cust[1].length;
const sumSalary = reduce((n, acc) => n + acc, sals);

if (votingAge &&
  numVotes === 0 &&
  sumSalary > 10000)      // ah!
```

We expect you to demonstrate good commenting while here.

Commenting Out Code

When commenting out code, always say why:

```
// old formula (for ref)  
//  
// const salary = ...
```

```
// not sure if needed; works without?  
//  
// if (reactor.temp > 500) alert("eek");
```

You'll thank yourself later

Keep it simple

Everyone knows that debugging is twice as hard as writing a program in the first place.

If you're as clever as you can be when you write it, how will you ever debug it?

—Brian Kernighan, *The Elements of Programming Style*

(That's a particularly excellent intermediate book, by the way)

Resources

You can find the Rithm Code Requirements in SIS under the Resources tab.



Pair Programming

Goals

Learn critical parts of professional programming:

- Introduce pair programming
- Describe how best to pair
- Understand how and when to ask for help

Working with others

- At Rithm, we're going to pair quite a bit!
- You will often move slower, but be *far more effective*
- You'll get exposure to new ideas & debugging plans
- You'll have the opportunity to teach and explain in order to further your own understanding

The roles



Driver

Writes code



Navigator

Reviews code, asks questions, does side-research

Switch roles frequently (*good rule of thumb: about every 20 minutes*)

The Driver

As the driver, it's your job to talk your partner through each step when you code and make sure that your partner remains engaged.

If you're unsure, ask them to paraphrase what is happening to ensure they do understand.

Other Driver Responsibilities

- **Writing code.** Obvs.
- **Explaining plan.** Don't leave partner in the dark!
- **Asking for help.** Navigator is there to help you find solutions.

The Navigator

As a navigator, make sure you stay engaged and participate!

Keep discussing the assignment with your pair and make sure that you are:

- following instructions
- not going down rabbit holes

Other Navigator Responsibilities

- **Asking clarifying questions.** Make sure both of you know the plan.
- **Trying to debug.** Minimize amount of context switching driver has to do. When you need research, the navigator should be the one to head to Google.
- **Remembering big picture.** Driver may lose the forest for the trees while writing code; make sure you keep team on track.

General Tips

Pairing dynamics

- Some days you will move more quickly than your partner
 - Talk through your process while driving or navigating
 - Explain things to your pair, demonstrate mastery through teaching
- Some days you will move more slowly than your partner
 - You **still need to drive**, don't sit back
 - Ask your pair to explain things to you
- Do not spend more than 15-20 minutes on a problem
- Can't answer a question or google for one? **Grab an instructor!**

How to ask for help

We're here to help you, but we're going to treat you like engineers.

- If you have a bug, saying “my code doesn’t work” is unacceptable
 - What assumptions have you made about the bug?
 - What have you tested?
 - Where is the bug happening?
- If you have a question, make sure you first do your research
 - What exactly are you looking for?
 - What have you searched for?

Other Advice

- **Be kind.** You won't always be paired with your best friend. But this is true in the workplace too, so be kind and professional.
- **It's harder to be a great navigator.** You may be tempted to check out, or steal control to solve it your way. *DON'T DO IT!* Avoid *dravigation* and *steamrolling*.
- **Pairing is valuable, even when there's a skill gap.** Research shows mixed-ability classrooms benefit *all* students.
- **Give us feedback.** Talk with your advisor about pairs (*positive or negative*).



Git and GitHub Fundamentals

Goals

- Describe why version control is useful
- Describe the workflow for using Git locally
- Describe the workflow for using GitHub to collaborate

Git basics

Why version control?

- Keep track of changes to code
- Revert back to previous versions of code
- *Advanced:* manage multiple branches of work
- *Advanced:* manage approvals of code modifications
- *Advanced:* integrate with bug tracking/productivity/security

A bit about Git

Git is an Open Source product, made to manage the code for Linux (the largest collaborative product in the history of software!)

As such, it's large, complex, and has *lots* of weird corners and edges

Stick to what we cover here — there are lots of bad things that can happen if you fall off the path! 🚧

Local workflow: areas

Working Directory

Your directory where you're writing code

Staging Area (*added, but not committed*)

Stored in `.git` directory

Repository (*added and committed*)

Stored in `.git` directory

Creating a Git repository

- `git init`

- This creates a repository in your current directory
- Made it in the wrong place? Just `rm -rf .git`

Committing workflow

- `git status` : What files have been added / modified?
- `git diff` : Among those files, what has been changed?
- `git add {NAME_OF_FILE}` : Add file to staging area
- `git commit -m "message"` : Commit staged work with message
- `git log` : See log of all commits

TIP Avoid `git add *` or `git add .`

You may learn that, instead of adding each file individually, you can add all files in a directory with `git add *` (or all files in a directory, plus normally-hidden files that start with an underscore with `git add ..`).

While convenient, we'll strongly discourage you from doing so. It's very helpful to start training your brain to think about a commit as not just "saving everything as-is", but as "intentionally committing this feature/bug fix/stage of work", and being conscious about exactly which files should be committed will help you start to develop a professional workflow with git early on.

Also: if you add *all* files, it's much too easy to accidentally commit a file that isn't related to your work or wasn't ever supposed to go into your repository at all, like a file with internal notes, passwords, or credit card numbers. **Once something is committed to git, it's part of its history forever**, so it's very smart to keep such information out at all times.

Commit frequently

Commit often.

Every bug or feature should get its own commit.

As a pro, you'll keep your work *atomic* — get in the habit now.

Still a single point of failure

- You still have a single point of failure
 - If your computer dies, you'll lose everything!
- Make sure the code exists in another repository
 - One that is not *local*, but *remote*
- Plus, you can *collaborate*

Git and GitHub

A bit about GitHub

Git is a free Open Source product.

GitHub is a commercial company, not run by the people who make Git.

They offer free and paid plans to share Git repositories.

They are not the only company that does this; there are excellent other players.

Creating remote repositories on GitHub

- Navigate to <https://www.github.com/new> <<https://www.github.com/new>>
- Follow the second block of instructions
- Confirm you have created a remote correctly using `git remote -v`

What's a remote?

- It's a nickname for a URL where your repository lives!
- Instead of typing/remembering the URL every time, we give it a nickname
- `git remote add {NAME_OF_REMOTE} {URL_FOR_REPOSITORY}`

Pushing your code

```
git push {NAME_OF_REMOTE} {NAME_OF_BRANCH}
```

You don't *need* to push after every commit — but don't wait too long, or a computer crash might lose a lot of work.

Working with others

Sharing vs Forking

When working with others you have two options for workflow

Cloning and pushing (*requires collaborator access*)

Most common approach for employees at companies.

It's what we'll do here.

Forking and cloning

More common when contribute to software as a guest.

Sharing a repo

Everyone contributes to the same GitHub repository by having access to it.

- When working with others, make sure you give them collaborator access
- Since they can write to the repo, make sure you communicate

How to work with your pair

1. **Driver:** make project directory on their computer and `git init` in it.
 2. **Driver:** create a new remote repository on GitHub
 3. **Driver:** give navigator collaborator access (settings → manage access)
 4. **Navigator:** Accept access invitation (*check your email*)
 5. **Navigator:** clone the GitHub repository to their computer
1. **Driver:** Working with navigator, write code (follow commit workflow!)
 2. **Driver:** When it's time to switch, do a final `git commit` and `git push`
 3. **Navigator:** Pull latest code with `git pull`

Repeat the role-switching until complete!

Finishing the project

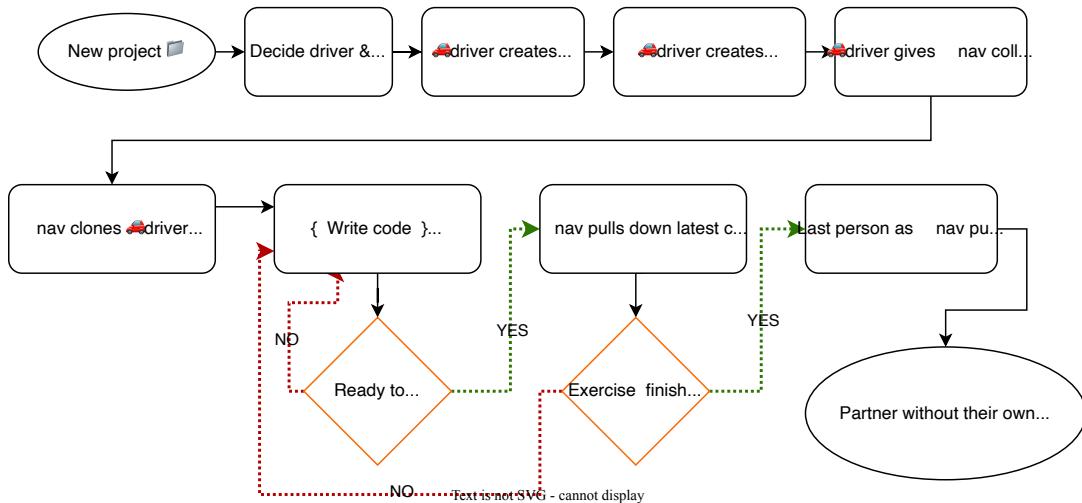
Once done, both of you may want this project in your personal GitHub.

The initial navigator has some steps to do to get their own copy:

1. Make a new repository in GitHub on their own account
2. Switch the on-laptop repository from driver's account to their account:

```
$ git remote rm origin
$ git remote add origin url-github-gave-you
$ git push -u origin main
```

Git workflow flowchart



Thanks to Brit, for making this awesome diagram when she was a TA with us!

Code reviews

- Code reviews are a critical part of our course.
- When exercises say **Get a code review**, we expect you to do that.
 - There are often critical things we want to see/talk about.
- Reviews are listening, not changing code. Hands off the keyboard, please.
- Reviews are for *both of you*, not just the current driver.
- **Always** commit before asking for a code review.
- Sometimes you'll be ready for a review, but instructors will be busy.
 - Keep working!
 - When we're ready, `git stash` to stash your changes since the commit.
 - After review, `git stash pop` to unstash your changes.

Looking ahead

Things to NOT DO

- Experiment with Git commands not covered here:
 - `git branch`, `git reset`, `git rebase`, and all the others
- Use graphical programs to work with Git
 - They're awesome **once you know Git well, but not now**
- Use commands you find on the Internet
 - A lot of people are wrong, sometimes dangerously so

- Even when they’re right, it’s often “right for my company’s workflow”

Next steps

Later, we’ll cover more intermediate/advanced usage:

- Branching
- Merging and merge conflicts
- Forking (the “guest” model, often used in Open Source)
- Pull requests and other GitHub tools



Dev Tools: Debugger

Goals

Learn critical parts of debugging:

- Using the sources tab and debugger
- Using the elements tab for HTML and CSS

JavaScript Debugger

- Watch execution of code and examine at any point
- Built into Chrome (other browsers have similar abilities)
- Can debug in-browser code *or* Node

Starting Debugger

View code or adding “breakpoints”:

View → Developer → Developer Tools → Sources tab

Click left of line of code to add a blue breakpoint

Can put breakpoint into code itself:

```
function myFunction() {  
    let x = 1;  
  
    debugger; // <-- will always stop here  
  
    // rest of code follows ...  
}
```

Let's try this out with an example!

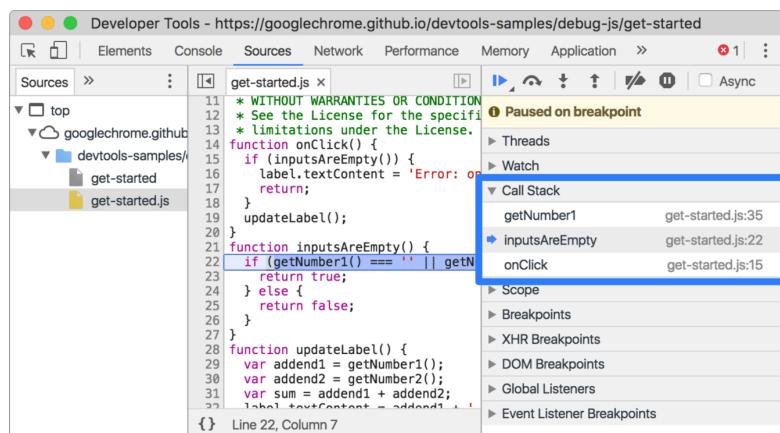
demo/hello-function.js

```
function a() {
  debugger
  console.log("hello");
  b();
  console.log("coding");
}

function b() {
  debugger
  console.log("world");
  c();
  console.log("love");
}

function c() {
  debugger
  console.log("i");
}
```

Call Stack



Shows *stack* of function calls that got you here

Scope

The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. A script named 'get-started.js' is open. In the code editor, line 31 is highlighted: `var sum = addend1 + addend2;`. To the right of the code, the 'Scope' panel is open, showing the current values of variables:

- Local:
 - addend1: "5"
 - addend2: "1"
 - sum: undefined
- Global:
 - Breakpoints
 - XHR Breakpoints
 - DOM Breakpoints
 - Global Listeners

Shows current value of variables

Can click to change value

NOTE Other Dev Tools Buttons

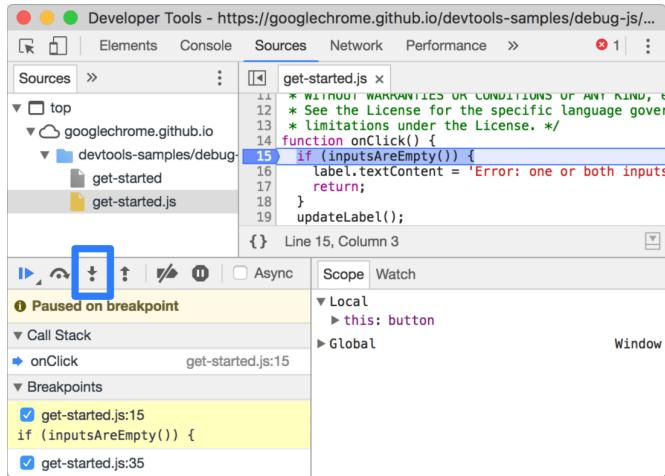
There are quite a few other buttons you can use to debug in more depth, right now we'll be focusing on pausing and resuming code, but here are few other tools you can use:

Step Over

The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. A script named 'get-started.js' is open. In the code editor, line 15 is highlighted: `if (inputsAreEmpty()) {`. Below the code editor, the toolbar buttons are visible, and the 'Step Over' button (represented by a blue square with a white arrow) is highlighted with a blue box.

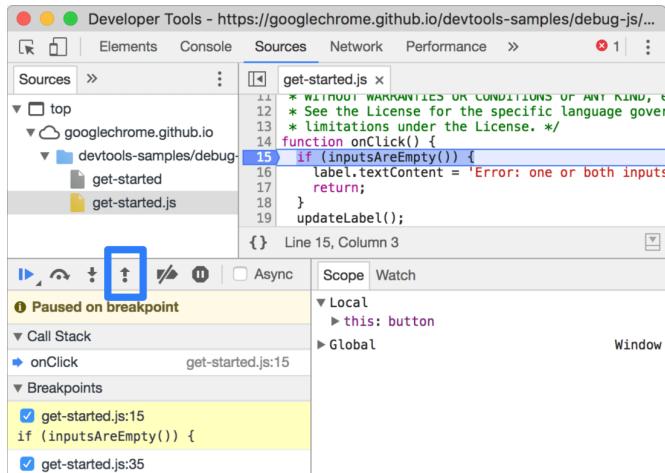
Run current line, but don't debug into any function calls

Step Into



Run current line, stepping into any function calls

Step Out



Return from this function to caller

Tips To Avoid Bugs

Plan First

Software and cathedrals are much the same — first we build them, then we pray.

—Sam Redwine

Common JavaScript Bugs

- == is very loose about comparisons (=== isn't)
 - `7 == "7"`
- Objects & arrays are not equal to similar objects & arrays
 - `[1, 2, 3] != [1, 2, 3]`
- Calling function with missing arguments makes those arguments *undefined*
- Calling function with extra arguments ignored them
- A missing property from object or missing index from array is *undefined*

Good News

If debugging is the process of removing bugs, then programming must be the process of putting them in.

—Edsger W. Dijkstra

- Bugs are an opportunity to improve debugging skills & to learn something
- You will have lots of chances to practice this valuable skill!

JavaScript DOM Work

- The console is wonderful for debugging JavaScript — what about issues with HTML and CSS?
- That's where the *Elements* tab comes in
- Using the elements tab you can:
 - View the DOM
 - Modify HTML and CSS (*not permanently*)
 - Examine what CSS has been computed and applied
 - Examine what event listeners have been added

Let's see a demo here!

demo/elements.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet" href="elements.css">
</head>
<body>
  <h1>Open the elements tab with the selector! Command + Shift + C</h1>
  <h2>Oh, the things you'll see!</h2>
  <ul>
    <li>All your elements</li>
    <li>All the CSS you imagine</li>
    <li>You'll even see event listeners added!</li>
  </ul>
  <div id="box">
    I am a box!
  </div>
  <button>Click me and then open the console!</button>
  <script src="elements.js"></script>
</body>
</html>
```

IMPORTANT Preventing Caching

- Chrome will try to cache things for you by default
- This is helpful when loading pages as quickly as possible, but frustrating in development!
- Make sure you check the checkbox next to Disable Cache and **keep your dev tools open when coding**



JavaScript Object Orientation

Goals

- Understand terminology around instances & classes
- Learn how to create classes
- Learn how to add constructor functions
- Understand inheritance (“*subclassing*”)
- Firm up important terminology of OO

JS objects review

“Plain Old JavaScript Object” (“POJO”):

```
let o1 = {};
let o2 = new Object(); // same thing

o1.name = "Whiskey";
o1["name"] = "Whiskey"; // same thing
```

A property where key is `"sayHi"` and the value is a function:

```
o1.sayHi = function() { return "Hi!" };
o1.sayHi(); // Hi!
```

NOTE Functions are “first-class”

You’ll sometimes hear the phrase “functions are ‘first class’” when people are describing programming languages. This means that a function *itself* can be treated like any other thing; you can assign a function to a variable or put a function in an array (the term “first class” is suggesting that they can do all of the things everything else can do).

TIP Working with JS objects

Can get arrays of keys, values, or `[key, val]` arrays:

```
Object.keys(o1); // ["name", "sayHi"]
Object.values(o1); // ["Whiskey", function () {...}]
Object.entries(o1); // [[{"name": "Whiskey"}, // ["sayHi", function () { ... }]]
```

- Properties that don’t exist evaluate to *undefined*:

```
o1.taco // undefined
```

(if you tried to call this, like `o1.taco()` or tried to access a subobject, like `o1.taco.toppings`, you'd get an error because `undefined` is not callable and you cannot treat it like an object.)

- All keys get “stringified”:

```
o1[1] = "hello";
o1["1"] = "goodbye";
```

- What is `o1[1]`?

```
o1[1]; // "goodbye"
```

(This gets even more confusing when using things like nested arrays as keys)

Mixing data and functionality

Functions and data

Imagine some useful functions:

`demo/triangles.js`

```
/* Area of right triangle */

function getTriangleArea(a, b) {
  return (a * b) / 2;
}

/* Hypotenuse of right triangle */

function getTriangleHypotenuse(a, b) {
  return Math.sqrt(a ** 2 + b ** 2);
}
```

```
getTriangleArea(3, 4);      // 6
getTriangleHypotenuse(3, 4); // 5
```

This gets a bit messy, though — all those functions to keep track of!

Using a POJO

demo/triangle-pojo.js

```
let myTri = {
  a: 3,
  b: 4,
  getArea: function() {
    return (this.a * this.b) / 2;
  },
  getHypotenuse: function() {
    return Math.sqrt(this.a ** 2 + this.b ** 2);
  },
};
```

```
myTri.a;           // 3
myTri.getArea();   // 6
myTri.getHypotenuse(); // 5
```

```
let myTri = {
  a: 3,
  b: 4,
  getArea: function() {
    return (this.a + this.b) / 2;
  },
  getHypotenuse: function() {
    return Math.sqrt(this.a ** 2 + this.b ** 2);
  },
};
```

this references “this object”

So, we can helpfully mix data & functionality 🎉

- This is tidy: related functionality lives together
- Annoying when we have more than one triangle
 - Difficult to maintain
 - If we have 1000 triangles, we’d have 1000 copies of these functions — that’s going to waste memory!

Classes

Classes are a “blueprint” of functionality:

demo/triangle-oo.js

```
class Triangle {

  getArea() {
    return (this.a * this.b) / 2;
  }

  getHypotenuse() {
    return Math.sqrt(this.a ** 2 + this.b ** 2);
  }
}
```

```
let myTri = new Triangle(); // "instantiation"
myTri.a = 3;
myTri.b = 4;
myTri.getArea(); // 6
myTri.getHypotenuse(); // 5
```

TIP “Instantiation”

Instantiation is a fancy term for “making an instance of”.

demo/triangle-oo.js

```
class Triangle {
    getArea() {
        return (this.a * this.b) / 2;
    }
    getHypotenuse() {
        return Math.sqrt(this.a ** 2 + this.b ** 2);
    }
}
```

- Defines the *methods* each instance of *Triangle* will have
- Make a new triangle with `new Triangle()`
- Can still add/look at arbitrary properties
- *this* is “the actual triangle in question”

Class names should be `UpperCamelCase`

Reduces confusion between *triangle* (*individual triangle*) and *Triangle* (*the class of all triangles*)

A triangle is still fundamentally an object:

```
typeof myTri; // 'object'
```

JS knows it's an “instance of” the *Triangle* class:

```
myTri instanceof Triangle; // true
```

Constructors

Consider how we made an instance of our *Triangle* class:

```
let myTri = new Triangle(); // "instantiation"
myTri.a = 3;
myTri.b = 4;
```

demo/triangle-constructor.js

```
class Triangle {  
  constructor(a, b) {  
    this.a = a;  
    this.b = b;  
  }  
  
  getArea() {  
    return (this.a * this.b) / 2;  
  }  
  
  getHypotenuse() {  
    return Math.sqrt(this.a ** 2 + this.b ** 2);  
  }  
}
```

Any method named *constructor* will be called on making a new instance:

```
let myTri2 = new Triangle(3, 4); // <- calls constructor  
myTri2.getArea(); // 6
```

What can you do in the constructor?

- Whatever you want!
- Common things:
 - Validate data
 - Assign properties

```
constructor(a, b) {  
  if (!Number.isFinite(a) || a <= 0)  
    throw new Error(`Invalid a: ${a}`);  
  
  if (!Number.isFinite(b) || b <= 0)  
    throw new Error(`Invalid b: ${b}`);  
  
  this.a = a;  
  this.b = b;  
}
```

(Note that constructor functions always just return *undefined*)

Methods

```
class Triangle {  
  getArea() {  
    return (this.a * this.b) / 2;  
  }  
}
```

Functions placed in a class are *methods* (formally, *instance methods*).

They have access to properties of object with *this*.

They take arguments/return data like any other function.

A method can call another method:

```

class Triangle {
  getArea() {
    return (this.a * this.b) / 2;
  }

  /** Describe triangle and return string. */

  describe() {
    return `Area is ${this.getArea()}.`;
  }
}

```

Note: to call a method, you need to call it on *this*

Without *this*, calling *getArea()* throws a ReferenceError — it's not in scope!

Inheritance and super

Inheritance

Here's another kind of triangle we might want:

```

class ShyTriangle {
  constructor(a, b) {
    this.a = a;
    this.b = b;
  }

  getArea() {
    return (this.a * this.b) / 2;
  }

  getHypotenuse() {
    return Math.sqrt(this.a ** 2 + this.b ** 2);
  }

  describe() {
    return "(runs and hides)";
  }
}

```

Inheritance and extends

```
class ShyTriangle {  
    constructor(a, b) {  
        this.a = a;  
        this.b = b;  
    }  
  
    getArea() {  
        return (this.a * this.b) / 2;  
    }  
  
    getHypotenuse() {  
        return Math.sqrt(this.a ** 2 + this.b ** 2);  
    }  
  
    describe() {  
        return "(runs and hides)";  
    }  
}
```

```
class ShyTriangle extends Triangle {  
  
    // don't repeat if not different:  
    // will "inherit" from "parent"  
  
    describe() {  
        return "(runs and hides)";  
    }  
}
```

```
let shy = new ShyTriangle(3, 4);  
shy.getArea(); // 12
```

```
shy instanceof ShyTriangle; // true  
shy instanceof Triangle; // true!
```

Another subclass: ColorTriangle

```
class Triangle {  
    constructor(a, b) {  
        this.a = a;  
        this.b = b;  
    }  
  
    getArea() {  
        return (this.a * this.b) / 2;  
    }  
  
    getHypotenuse() {  
        return Math.sqrt(  
            this.a ** 2 + this.b ** 2);  
    }  
  
    describe() {  
        return `Area is ${this.getArea()}.`;  
    }  
}
```

```
class ColorTriangle extends Triangle {  
    constructor(a, b, color) {  
        this.a = a;  
        this.b = b;  
        this.color = color;  
    }  
  
    describe() {  
        return `Area is ${this.getArea()}.` +  
            ` Color is ${this.color}!`;  
    }  
}
```

- Syntax Error!
 - Must call parent's constructor!
 - But we want different logic here
 - Also: half of `describe()` is duplicated

Extending a parent's method with super

demo/triangle-extends.js

```
class Triangle {  
  constructor(a, b) {  
    this.a = a;  
    this.b = b;  
  }  
  
  getArea() {  
    return (this.a * this.b) / 2;  
  }  
  
  getHypotenuse() {  
    return Math.sqrt(  
      this.a ** 2 + this.b ** 2);  
  }  
  
  describe() {  
    return `Area is ${this.getArea()}.`;  
  }  
}
```

demo/triangle-extends.js

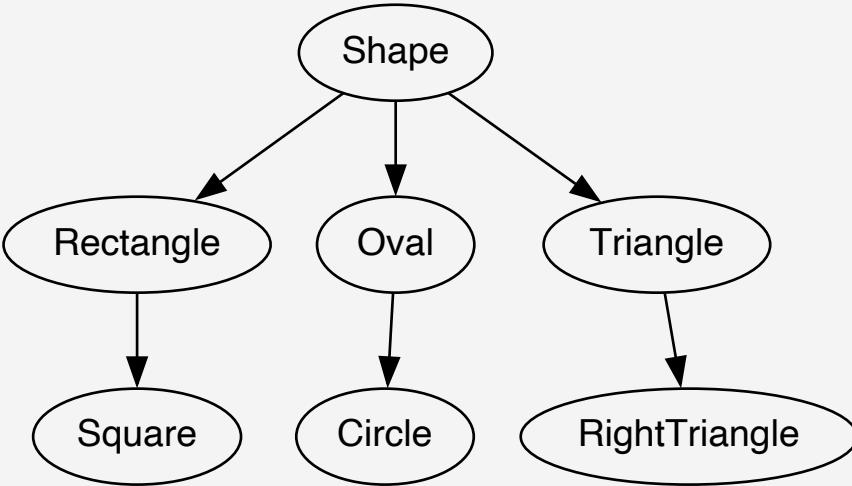
```
class ColorTriangle extends Triangle {  
  constructor(a, b, color) {  
    // call parent constructor w/(a, b)  
    super(a, b);  
    this.color = color;  
  }  
  
  // "inherits" getArea, getHypotenuse  
  
  // "override" describe() w/new version  
  
  describe() {  
    return super.describe() +  
      ` Color is ${this.color}!`;  
  }  
}
```

Multi-level inheritance

We won't go into it now, but it's even possible to subclass a subclass, making a kind of hierarchy. Here's a special kind of *ColorTriangle*:

```
class RainbowTriangle extends ColorTriangle {  
  constructor(a, b) {  
    // call parent constructor  
    super(a, b, "rainbow");  
  }  
  
  // inherits getArea, getHypotenuse  
  
  describe() {  
    return "I'm a beautiful rainbow!";  
  }  
}
```

A lot of things in the world have these kind of relationships:



△ A hierarchy of geometric shapes

We might have a *Shape* class, and since rectangles are a kind of shape, *Rectangle* subclasses *Shape* (`class Rectangle extends Shape`). And, since squares are a special kind of rectangle, *Square* subclasses *Rectangle* (`class Square extends Rectangle`).

JS will understand this:

```

mySquare = new Square();
mySquare instanceof Square;      // true
mySquare instanceof Rectangle;   // true -- squares are a kind of rectangle
mySquare instanceof Shape;       // true -- rectangles are a kind of shape
mySquare instanceof Oval;        // false

```

Terminology

- **Instance**
 - an individual instance; an array is `instanceof Array`
- **Class**
 - “blueprint” for making instances
- **Property**
 - piece of data on an instance (eg `myTriangle.a`)
 - most languages call this idea an *instance attribute*
- **Method**
 - function defined by a class, can call on instance
 - most accurate to call these *instance methods*
- **Parent / superclass**
 - More general class you inherit from
 - *Rectangle* might be parent of *Square*
- **Child / subclass**
 - More specific class (a *Square* is a special kind of *Rectangle*)

- **Inherit**
 - Ability to call methods/get properties defined on ancestors
- **Object-oriented programming**
 - Using classes & instances to manage data & functionality together
 - Often makes it easier to manage complex software requirements

Looking ahead

- More about *this*
- Additional OO Concepts
- Python OO
- Old-school JavaScript OOP



JavaScript This

Goals

- Learn how to stop worrying and love the keyword *this*
- Explain what *.call* does
- Explain what *.bind* does
- Use *.call* and *.bind* to reassign the value of the keyword *this*

The Keyword *this*



Ben Halpern
@bendhalpern

Follow

Sometimes when I'm writing Javascript I want to throw up my hands and say "this is bullshit!" but I can never remember what "this" refers to

Mystery of the Undefined Fluffy

demo/fluffy.js

```
class Cat {
  constructor(firstName) {
    this.firstName = firstName;
  }

  dance(style = "tango") {
    return `Meow, I am ${this.firstName} +
      and I like to ${style}`;
  }
}
```

makes sense...

```
let fluffy = new Cat("Fluffy");

fluffy.firstName; // "Fluffy"

fluffy.dance("tango"); // works!
```

wtf?

```
let fDance = fluffy.dance;

fDance("salsa"); // error?!
```

JavaScript “Functions”

In a sense, JavaScript doesn't have functions.

Everything is called on something, like a method.

```
function whatIsThis() {
  console.log("this =", this);
}
```

```
let o = { myFunc: whatIsThis };  
o.myFunc(); // get "this = o"
```

```
whatIsThis(); // wtf?!
```

Global Object

When you call a function on nothing ...

... you call it on the “global object”

- In browser JS, that’s typically *window* (the browser window)
- in NodeJS, that’s *global* (where some Node utilities are)

You’ve relied on that, even if you didn’t realize it!

```
console.log("Hi!");  
window.console.log("Hi!"); // -- same thing!
```

Therefore, a “function” called at the top level is same as:

```
window.whatIsThis()
```

Undefined Fluffy

demo/fluffy.js

```
class Cat {  
  constructor(firstName) {  
    this.firstName = firstName;  
  }  
  
  dance(style = "tango") {  
    return `Meow, I am ${this.firstName} +  
           ` and I like to ${style};  
  }  
}
```

```
fluffy.dance("tango");
```

so... what's happening here?

```
let fluffy = new Cat("Fluffy");  
  
fluffy.firstName; // "Fluffy"  
  
fluffy.dance("tango"); // works!  
  
let fDance = fluffy.dance;  
  
fDance("salsa"); // error??
```

- Find the *dance* method on *fluffy*
- Call the *dance* method on *fluffy* – yay!

```
let fDance = fluffy.dance;  
fDance("tango");
```

- Find the *dance* method on *fluffy*
- Call the *dance* method on... undefined? *ut oh*

OOP and *this*

When you call a function on nothing, but the function comes from inside a *class*, the value of *this* will be *undefined*, not the *window*.

In either case, you'll see this referred to as "losing the *this* context."

Fortunately, there are ways to force the value of *this* to be whatever we want.

Call and Bind

Call

Sometimes, you'll need to say "call this function *on this object*"

That's what *call()* is for!

```
let fDance = fluffy.dance;  
  
// call on fluffy, passing "tango" as arg  
fDance.call(fluffy, "tango");
```

```
whatIsThis.call(fluffy); // this = fluffy
```

NOTE *apply()*

There a related function, *apply()*: for this, you can pass the list of arguments to the function as an array, rather than passing one-by-one.

This used to be a very important technique, since it was the only reasonable way to call a function that expected several individual arguments where you already had those arguments in a list:

```
Math.max(1, 2, 3); // Math.max expects indiv arguments  
  
let myNums = [1, 2, 3]; // If you already have an array ...  
  
Math.max.apply(null, myNums); // pass that array as indiv arguments  
// (don't care what "this" is; pass `null`)
```

Nowadays, however, this is much more easily done with the spread operator:

```
Math.max(...myNums);
```

Bind

You can "perma-bind" a function to a context:

```
fDance("tango");           // error -- this isn't the cat
fDance.call(fluffy, "tango"); // ok but tedious to always do
let betterDance = fDance.bind(fluffy);
betterDance("tango"); // ok -- bound so that `this` is Fluffy
```

bind() is a method on functions that returns a bound copy of the function.

Binding Arguments

You can also bind arguments to functions. That will bake them into the function.

```
function applySalesTax(taxRate, price) {
  return price + price * taxRate;
}

// "null" for "this" means it doesn't matter what "this" is
const applyCASalesTax = applySalesTax.bind(null, 0.0725);
applyCASalesTax(50); // 53.63
```

Where This Comes Up

Callback on Methods

Want to have object method as callback:

```
myBtn.addEventListener("click", fluffy.dance);
```

That won't work – browser will call *dance* on global object :(

```
myBtn.addEventListener("click", fluffy.dance.bind(fluffy));
```

That will work — when it calls that callback, it will always be on Fluffy!

NOTE Pre-Binding Calls

Imagine we want three buttons which call *popUp*, but with different args:

demo/buttons-meh.html

```
<button id="a">A</button>
<button id="b">B</button>
<button id="c">C</button>
```

demo/buttons-meh.html

```
function popUp(msg) {
  alert("Secret message is " + msg);
}

function handleClick(evt) {
  let id = evt.target.id;

  if (id === "a") popUp("Apple");
  else if (id === "b") popUp("Berry");
  else if (id === "c") popUp("Cherry");
}

const get = document.getElementById.bind(document);

get('a').addEventListener("click", handleClick);
get('b').addEventListener("click", handleClick);
get('c').addEventListener("click", handleClick);
```

Meh. *handleClick* needs to scrounge around *evt.target* to determine the argument that should be passed when we call *popUp*.

We can instead bind the applicable argument so the *popUp* function that's passed to each event listener already has the correct argument baked into the function:

demo/buttons.html

```
function popUp(msg) {
  alert("Secret message is " + msg);
}

const get = document.getElementById.bind(document);

get('a').addEventListener("click", popUp.bind(null, "Apple"));
get('b').addEventListener("click", popUp.bind(null, "Berry"));
get('c').addEventListener("click", popUp.bind(null, "Cherry"));
```

Arrow Functions

Arrow functions don't make their own *this*

```
class Cat {  
  constructor(firstName) {  
    this.firstName = firstName;  
  }  
  
  superGreet() {  
    console.log(`#1: I am ${this.firstName}`); // works, obvs  
  
    setTimeout(function () {  
      console.log(`#2 I am ${this.firstName}`); // ut oh  
    }, 500);  
  
    setTimeout(() => {  
      console.log(`#3 I am ${this.firstName}`); // yay!  
    }, 1000);  
  }  
}  
  
let kitty = new Cat("Kitty");  
kitty.superGreet();
```

Key Takeaways

Make sure you don't lose your *this*!

- *this* is a reserved keyword whose value is determined **only at the point of function execution**.
- If you don't call a function yourself (*eg, it's called by a callback*), you need to ensure JS knows what the *this* context should be.

Looking Ahead

- Additional OO Concepts
 - Class properties
 - Static methods
- Python OO



Introduction to Big-O Notation

Goals

- Develop a conceptual understanding of Big-O notation
- Explain need for notation
- Analyze time complexity
- Compare different time complexities

Runtime / Algorithm Efficiency

Big picture:

Big O Notation allows us to measure how code performs as input grows.

- Imagine we have multiple implementations of the same function
- How can we determine which one is the “best?”
- *Function that accepts a string and returns reversed copy*
- It’s important to discuss performance under scale
- Useful for discussing trade-offs between different approaches
- When code slows, identifying inefficient parts helps find pain points
- Less important: it comes up in interviews!
- Calculate sum of numbers from 1 up to (*and including*) some number n

```
function getSumUpTo(n) {  
  let total = 0;  
  
  for (let i = 1; i <= n; i++) {  
    total += i;  
  }  
  
  return total;  
}
```

```
function getSumUpToSecond(n) {  
  return n * (n + 1) / 2;  
}
```

Which is better?

- Faster?
- Less memory-intensive?
- More readable?
- Let’s focus on *scale*

The problem with timers

- Different machines will record different times
- The same machine will record different times!
- For fast algorithms, speed measurements may not be precise enough

If not time, then what?

- Rather than counting *seconds*, which are so variable...
- Let's count the *number* of simple operations the computer has to perform!

Counting operations

```
function getSumUpToSecond(n) {
    return n * (n + 1) / 2;
}
```

3 simple operations, regardless of the size of n

```
function getSumUpTo(n) {

    let total = 0;

    for (let i = 1; i <= n; i++) {
        total += i;
    }

    return total;
}
```

Let's try counting number of operations!

What have we learned?

- Counting is hard!
- Regardless of exact number, number of operations grows proportional to n
 - If n doubles, number of operations will also double
- We can use this idea to measure the performance efficiency of algorithms

Big O Notation

- Big O Notation is a way to formalize fuzzy counting
 - Can use to talk about how the runtime of an algorithm grows as inputs grow
 - We won't care about the details, only the trends

Big-O Notation for *getSumUpTo*

```
function getSumUpToSecond(n) {
    return n * (n + 1) / 2;
}
```

- Always 3 operations
- O(1)

```
function getSumUpTo(n) {
    let total = 0;

    for (let i = 1; i <= n; i++) {
        total += i;
    }

    return total;
}
```

- The number of operations is bounded by a multiple of n (say, $10n$)
- This algorithm “runs in” O(n)

Print All Pairs

```
function printAllPairs(n) {
    for (let i = 0; i < n; i++) {
        for (let j = 0; j < n; j++) {
            console.log(i, j);
        }
    }
}
```

- O(n) operation inside of an O(n) operation
- This algorithm “runs in” O(n^2)

```
> printAllPairs(2);
0,0
0,1
1,0
1,1
```

```
> printAllPairs(3);
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

Worst Case

Big O notation is concerned with *worst case* of algorithm’s performance.

```
function isAllEvens(nums) {
    for (let i = 0; i < nums.length; i++) {
        if (nums[i] % 2 !== 0) {
            return false;
        }
    }
    return true;
}
```

This is O(n): even though it may not always take n times, it **scales with n** .

Simplifying Big O Expressions

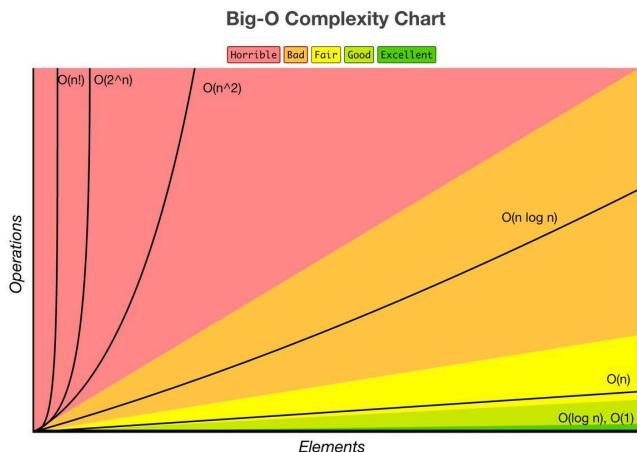
- When determining algorithm time complexity, rules for big O expressions:
 - Constants **do not** matter
 - $O(5) \rightarrow O(1)$
 - $O(10n^2) \rightarrow O(n^2)$
 - Smaller terms **do not** matter
 - $O(n^2 + 10n + 5) \rightarrow O(n^2)$
 - Always make sure you can answer: **what is n ?**

Hints and Common Runtimes

Hints

- Arithmetic operations are constant
 - $x + y / 10 \rightarrow O(1)$
- Variable assignment is constant
 - `const firstName = "Fluffy";` $\rightarrow O(1)$
- Accessing elements in array (*by index*) or object (*by key*) is constant
 - `nums[5]` $\rightarrow O(1)$
- Loops: length of the loop times complexity of whatever happens in loop
 - Think back to the example of `printAllPairs`

Common Runtimes



Logarithms

- We're in base 2 (*think about 0s and 1s*)
- $\log_2 8 = 3$ (*2 to the power of what gives me 8?*)
- The logarithm of a number roughly measures the number of times you can divide that number by 2 before you get a value that's less than or equal to one.
- Logarithmic time complexity is great!
 - You've seen an algorithm to find a value in a sorted array in $\log_2 n$ time
 - Binary Search!

For $n = 1000$:

Type	Function	Result
Constant	1	1
Logarithmic	$\log n$	≈ 10
Linear	n	1000
Linear-logarithmic	$n \log n$	$\approx 10,000$
Quadratic	n^2	1,000,000
Exponential	2^n	$\approx 1.07151 \times 10^{301}$
Factorial	$n!$	$\approx 4.02387 \times 10^{2568}$

Runtime of built-in functions

- What is the time complexity of `.includes()`?
- What is the time complexity of `.indexOf()`?

Must knows for now

- A loop does not mean it's $O(n)$!

```
for (let i = 0; i <= 10; i++) { ... }
```

- A loop in a loop does not mean it's $O(n^2)$!

```
for (let i = 0; i <= n; i++) {
  for (let j = 0; j <= 10; j++) {
    // a constant operation
  }
}
```

- Best runtime for sorting is $O(n \times \log_2 n)$ (also referred to as $n \log_2 n$)
 - That is *not* the same as $\log_2 n$

Space Complexity

So far, we've been focusing on **time complexity**: how can we analyze runtime of an algorithm as size of inputs increase?

Can also use big O notation to analyze **space complexity**: how will memory usage scale as size of inputs increase?

NOTE More on space complexity

Most primitives (booleans, numbers, *undefined*, *null*): constant space

Strings: $O(n)$ space (where *n* is string length)

Reference types: generally $O(n)$, where *n* is length of array (or keys in object)

```
function sum(nums) {  
  let total = 0;  
  
  for (let i = 0; i < nums.length; i++) {  
    total += nums[i];  
  }  
  
  return total;  
}
```

- $O(1)$ space

```
function double(nums) {  
  let doubledNums = [];  
  
  for (let i = 0; i < nums.length; i++) {  
    doubledNums.push(2 * nums[i]);  
  }  
  
  return doubledNums;  
}
```

- $O(n)$ space

Time complexity is more of the focus for now

Recap

- To analyze the performance of algorithm, use Big O Notation
 - Can give high level understanding of time or space complexity
 - Doesn't care about precision, only general trends (linear? quadratic? constant?)
 - Depends only on algorithm, not hardware used to run code
- Big O Notation is everywhere, so get lots of practice!



Problem Solving Process and Patterns

Goals

- Develop problem solving process & learn fundamental patterns
- Use frequency counters to solve problems more efficiently
- Use multiple pointers to solve problems more efficiently
- Compare different runtimes

Problem solving process

The process

1. Understand the Problem
2. Explore Concrete Examples
3. Break It Down
4. Solve a Simpler Problem
5. Use Tools Strategically
6. Look Back and Refactor

1. Understand the problem

- Can I restate the problem in my own words?
- What are the inputs that go into the problem?
- What are outputs that are required by the problem?
- Do I have enough information?
- How should I name the important data in this problem?

2. Explore concrete examples

- Start with simple examples
- Progress to more complex examples
- Explore examples with empty inputs
- Explore examples with invalid inputs

3. Break it down

- Explicitly write out the steps you need to take.
- You can write or draw this as pseudocode
- This forces you to think about the solution rather than the syntax
- This helps you catch any lingering misunderstandings
- Don't write code!

4. Solve a simpler problem

If there is a problem you can't solve, then there is an easier problem you *can* solve: find it.
—George Pólya

- Find the core difficulty in what you're trying to do
- Temporarily ignore that difficulty
- Write a simplified solution
- Then incorporate that difficulty back in

NOTE **Easier said than done.**

This fourth strategy (solve a simpler problem) is easier said than done.

If you simplify too much, you may make the problem too simple, in which case solving the simpler problem provides little insight into the original.

But if you don't simplify enough, you still might be stuck on a problem that is too challenging. Finding the right sub-problem to isolate takes a decent amount of practice.

5. Use tools strategically

- Use your debugging tools.
- Don't guess and check!
- Scientific approach: make hypotheses, test, draw conclusions. Repeat.

6. Look back and refactor

- Does the result match your expected output?
- Can you improve the clarity of your solution?
- What other ideas could you have pursued?

The only way to get better is to practice using this plan!

Problem solving patterns

- Frequency Counter
- Multiple Pointers
- Sliding Window
- Divide and Conquer
- Greedy Algorithms
- Backtracking
- Many more!

Frequency counters

- “hello” => {“h”: 1, “e”: 1, “l”: 2, “o”: 1}
- This pattern tracks the frequency of values
- Using an object can often avoid nested loops or O(n²) operations

Write a function called *squares*, which accepts two arrays.

It should return true if every value in the array has its corresponding value squared in the second array.

The order doesn't matter, but the frequency must be the same.

```
squares([1, 2, 3], [4, 1, 9]); // true
squares([1, 2, 3], [1, 9]); // false
squares([1, 2, 1], [4, 4, 1]); // false (must be same freq)
```

A naive solution

```
function squares(nums1, nums2) {
  if (nums1.length !== nums2.length) {
    return false;
  }

  for (let i = 0; i < nums1.length; i++) {
    const foundAt = nums2.indexOf(nums1[i] ** 2);

    if (foundAt === -1) {
      return false;
    }

    nums2.splice(foundAt, 1);
  }

  return true;
}
```

```
squares([1, 2, 3], [4, 1, 9]); // true
squares([1, 2, 3], [1, 9]); // false
squares([1, 2, 1], [4, 4, 1]); // false
```

Time Complexity: O(n²)

Using a frequency counter: solution

```
function getFrequencyCounter(items) {
  const freqs = {};

  for (const item of items) {
    const curr = freqs[item] || 0;
    freqs[item] = curr + 1;
  }

  return freqs;
}
```

```
function squaresWithFreqCounter(nums1, nums2) {
  if (nums1.length !== nums2.length) return false;

  const freqs1 = getFrequencyCounter(nums1);
  const freqs2 = getFrequencyCounter(nums2);

  for (let key in freqs1) {
    const squared = key ** 2;

    if (!squared in freqs2) {
      return false;
    }

    if (freqs2[squared] !== freqs1[key]) {
      return false;
    }
  }

  return true;
}
```

Time Complexity: O(n)

NOTE Maps vs. objects

You can also use a Map to make a frequency counter:

```
// a function to create a simple
// frequency counter using a map

function getFrequencyCounter(items) {
  let freqs = new Map();

  for (let item of items) {
    let curr = freqs.get(item) || 0;
    freqs.set(item, curr + 1);
  }

  return freqs;
}
```

Maps and objects are similar in JavaScript, as both can be used to store collections of key-value pairs. While objects have been around since the beginning of JavaScript, Maps came to the language as part of ES2015. You can read more about the difference between these two data structures at [MDN <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map).

Given two words, write a function called *validAnagram*, which determines if the second word is an *anagram* of the first.

An *anagram* is a word formed by rearranging another word. For example: *cinema*, formed from *iceman*.

```
validAnagram("", "");           // true
validAnagram("noon", "nono");   // true
validAnagram("Noon", "nono");   // false (not case insensitive)
validAnagram("rat", "car");    // false
```

Multiple pointers

Creating *pointers* (values that correspond to an index) and moving those pointers based on a certain condition.

An example

`sumZero` accepts a **sorted** array of integers. It should find the first pair where the sum is 0.

Return an array that includes both values that sum to zero or throw an error if a pair does not exist.

```
sumZero([-3, -2, -1, 0, 1, 2, 3]); // [-3,3]
sumZero([-2, 0, 1, 3]);           // ✗ throws error
sumZero([1, 2, 3]);             // ✗ throws error
```

```
sumZero([-3, -2, -1, 0, 1, 2, 3]); // [-3,3]
```

```
function sumZero(nums) {
  for (let i = 0; i < nums.length; i++) {
    for (let j = i + 1; j < nums.length; j++) {
      if (nums[i] + nums[j] === 0) {
        return [nums[i], nums[j]];
      }
    }
  }

  throw new Error("Pair not found");
}
```

Time Complexity: $O(n^2)$

Using multiple pointers

```
sumZeroMultiplePointers([-3, -2, -1, 0, 1, 2, 3]); // [-3,3]
```

```

function sumZeroMultiplePointers(nums) {
  let left = 0;
  let right = nums.length - 1;

  while (left < right) {
    const sum = nums[left] + nums[right];

    if (sum === 0) {
      return [nums[left], nums[right]];
    } else if (sum > 0) {
      right--;
    } else {
      left++;
    }
  }

  throw new Error("Pair not found");
}

```

Time Complexity: $O(n)$

Recap

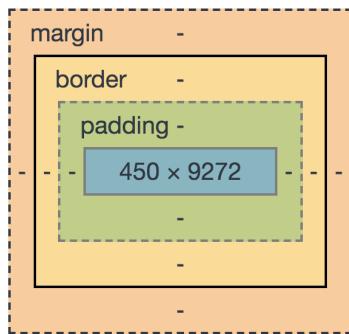
- Developing a *problem solving approach* is incredibly important
- Planning before writing code will always make you solve problems faster
- Be mindful about problem solving patterns
- Frequency counters and multiple pointers are just the start
- Do not over-fit!

CSS Display + Positioning

Goals

- Review the Box Model
- Compare different values for the *display* property
- Compare different values for the *position* property
- Provide an overview of responsive design and CSS transitions
- Highlight features of production-level CSS

Box Model



- Margin is outside of border (“space between”)
- Padding is inside of border (“space inside”)

Direction Shorthands

margin, *padding*, and *border-width* all support four different ways to set their value:

- *one number* for top, right, bottom, and left
 - e.g. `margin: 10px;`
- *two numbers*, for top & bottom, right & left
 - e.g. `margin: 10px 20px;`
- *three numbers*, for top, right & left, bottom
 - e.g. `margin: 10px 20px 30px;`
- *four numbers*, for top, right, bottom, left
 - e.g. `margin: 10px 20px 30px 40px;`

Box Sizing

```
div {  
  box-sizing: content-box;  
  width: 100px;  
  padding: 1em;  
  border: solid 5px black;  
}
```

- *content-box* is the default
- Content area is 100px
- Don't know how wide element is

```
div {  
  box-sizing: border-box;  
  width: 100px;  
  padding: 1em;  
  border: solid 5px black;  
}
```

- Element width is 100px
- Don't know width of content area
- Often, easier to work with

Outline

Outlines are like borders ...

- but are outside of borders
- but never take up space

```
div:hover {  
  outline: solid 5px red;  
  outline-offset: 5px;  
}
```

Often most used for hover effects, so things don't "move"

Learning More

Intro to the Box Model <https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Box_Model/Introduction_to_the_CSS_box_model>

Mastering Margin Collapsing <https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Box_Model/Mastering_margin_collapsing>

Display

"How should this be displayed?"

- *inline* (e.g. `<i>`, ``, ``)
 - Only takes as much space as needed
 - Ignores *width* and *height* properties
 - Next inline item is side-by-side
- *block* (e.g. `<form>`, `<div>`, `<section>`)
 - 100% of parent width *unless* set via *width* or *max-width*

- Can modify *width* and *height* with CSS
- Next item is on a separate line
- *inline-block*
 - Side-by-side, like inline; respects width and height, like block

Feature	block	inline	inline-block
Side-by-side layout	✗	✓	✓
Respects width / height properties	✓	✗	✓

- *none*
 - Don't show, don't take up any space
- *flex*
 - Enables Flexbox

Plus lots of specific ones related to lists, tables, and so on

Display Docs <<https://developer.mozilla.org/en-US/docs/Web/CSS/display>>

Flexbox

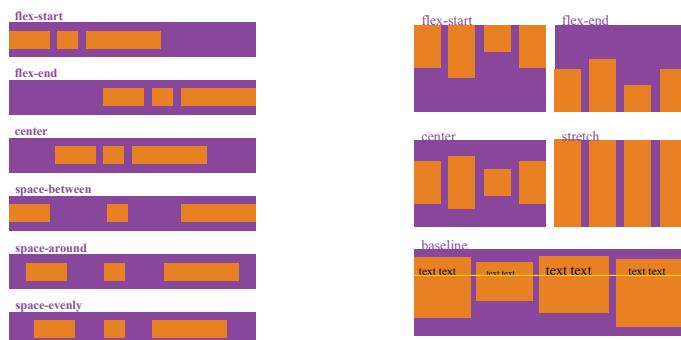
Flexbox is a large topic, beyond the scope of a single lecture.

TL;DR: flexbox makes some common layouts easier to create

Common Flexbox Properties

On an element with *display* set to *flex*:

- *justify-content: along main axis*
- *align-items: along cross axis*



Flexbox Resources

Why Flexbox <https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Flexbox#Why_Flexbox>

A Guide to Flexbox <<https://css-tricks.com/snippets/css/a-guide-to-flexbox/>>

Flexbox Froggy <<https://flexboxfroggy.com/>>

Flexbox Defense <<http://www.flexboxdefense.com/>>

Position

“How should this be positioned?”

- *static*
 - Default until you say otherwise
 - Appears at normal place in “flow”
- “*Positioned*”
 - All non-static are considered “positioned”
- *relative*
 - *relative* to normal position (via *top*, *right*, *bottom*, *left*)
 - takes same space as if it were static & in normal flow
- *absolute*
 - *absolute* position from *closest positioned ancestor*
 - other relative settings, like width, come from that now, too
 - removed from flow, takes up no space
- *fixed*
 - absolutely positioned from viewport (stays there on scroll)
 - removed from flow, takes up no space

NOTE **Other Position Values**

There are other possibilities here, but many are uncommon to work with.

Another useful one is:

sticky

A combination of *absolute* and *fixed*; this is useful for allowing something to scroll on the screen, but then fix to the viewport before it scrolls out of sight.

Responsive Design

- Bad old days: separate sites for desktop & mobile
- Semi-bad semi-old days: every site should have a native app
- Responsive design: same site with adjustments in CSS

Media Queries

```
/* default to smaller devices ("mobile-first") */

img.headshot {
  display: block;
  width: 70%;
}

@media (min-width: 800px) {
  img.headshot {
    display: inline-block;
    width: 10em;
  }

  /* other "desktop-size" stuff here...*/
}
```

Mostly, we'll use Bootstrap for this

Transitions

transition: property duration function delay [, ...]

- What property changes? (`all` for all)
- Over what time (eg, `3s`)
- What function determines values? (`ease`, `linear`)
- What delay before this? (eg, `0.5s`)
- Can list several properties

Transition Intro <https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Transitions/Using_CSS_transitions#Simple_example>

Professional CSS

Industry practice varies, but some common ideas are:

- avoid `!important` where reasonable
- generally, prefer classes over IDs (*IDs for JS is fine!*)
- “why” comments in CSS are often very important
- adopt & use a naming scheme

Class Naming Conventions

- CSS about a “component” is named (via class) after component:

```
.Tweet { /* ...declarations go here ... */ }
```

- Variations/small subcomponents get “dashed” name:

```
.Tweet-likes { /* stuff about the photo ... */ }
```

- Used-lots-of-places “utilities” get lowercase classes:

```
.muted {  
  font-size: 80%;  
  color: gray;  
}
```

- This is a light version of BEM <<http://getbem.com/introduction/>>

“Meta-CSS” Systems

There are meta-systems for CSS, doing things like:

- Letting you associate colors with names (*eg, “our logo color”*)
- Factor out common CSS declarations
- Calculate things (*eg, “our logo color, but 10% darker”*)
- Common systems are: SASS and LESS



Introduction to jQuery

Goals

- Develop a conceptual understanding of jQuery and its methods
- Explain why you would or would not use a library like jQuery
- Compare and contrast jQuery with vanilla JavaScript

jQuery

It's a library for:

- Manipulating the DOM
- Adding event listeners
- Animating elements
- Making HTTP requests (*AJAX*)

Why should you learn jQuery?

- Brevity and clarity
- Cross-Browser support
- AJAX
- 77% of the top 1,000,000 most visited pages use it

Vanilla JavaScript

```
const tasks = document.querySelectorAll(".Task");
for (const task of tasks) {
  task.style.color = "red";
  task.innerText = "Please do this!";
  task.addEventListener("click", completeTask);
}
```

jQuery

```
$(".Task")
.css("color", "red")
.text("Please do this!")
.on("click", completeTask);
```

Including jQuery and Selecting Elements

- <https://code.jquery.com/> <https://code.jquery.com/>
- Once you include jQuery script, you have access to global \$

TIP \$ in the browser console

Just because `$` has a value in your browser's console, this does *not* mean that the page you're on necessarily uses jQuery. Most browsers reserve `$` as a sort of shorthand for `document.querySelector`, unless some library overrides this behavior.

If you see something like `f $(selector, [startNode]) { [Command Line API] }` as the value for `$`, this means that jQuery is *not* installed. On the other hand, if you see something like `f (e,t){return new he.fn.init(e,t)}`, this means that a (minified) version of jQuery has been installed.

`$` is just a shorter alias for a global `jQuery` object when jQuery is loaded, so another test is just to check in the console whether there's a global variable called `jQuery`.

Selecting elements

It's as easy as using CSS selectors! (except you need to remember your CSS selectors)

```
$( "ul" )
// like document.querySelectorAll,
// this will select ALL uls

$( "#todo-container" )

$( ".carousel-image" )
// like document.querySelectorAll, this will
// select ALL the elements with that class
```

What does this give you?

A jQuery object

jQuery objects are NOT the same as DOM elements

To get a plain vanilla DOM element/list from jQuery objects:

```
let $listItems = $("li");
$listItems; // a jquery object

$listItems.get();
// an array of HTMLLIElements

$listItems.get(0);
// the first HTMLLIElement
```

It's uncommon to want to do this, though — use the jQuery object!

jQuery Methods

A great way to learn these is to compare them to vanilla JS methods!

- `.val()`
- `.text()`
- `.attr()`
- `.html()`
- `.css()`
- `.addClass() / .removeClass() / .toggleClass()`
- `.empty() / .remove()`
- `.append() / .prepend()`
- `.find() / .closest() / .parent() / .next() / .prev()`

Storing jQuery Objects in variables

It's a common convention to store jQuery objects in variable names that begin with \$. This isn't necessary, but clearly communicates your intent.

```
let $firstName = $("#firstName");
let firstName = $("#firstName").val();

// 200 lines later...

console.log($firstName);
// the jQuery object for the input element

console.log(firstName);
// not the jQuery object for the element!
```

jQuery getter / setter pattern

- Vanilla JS: `.getAttribute(attrName)` and `.setAttribute(attrName, newValue)`
- jQuery: `.attr(attrName, newValue)` (*second param is optional*)
- This is common with jQuery methods

Chaining with jQuery

Almost all jQuery methods return a jQuery object, which allows for *method chaining*.

Instead of performing DOM operations line-by-line, we can chain method calls together on a single jQuery object.

Vanilla JavaScript

```
const tasks = document.querySelectorAll(".Task");
for (const task of tasks) {
  task.style.color = "red";
  task.innerText = "Please do this!";
  task.addEventListener("click", completeTask);
}
```

jQuery

```
$(".Task")
  .css("color", "red")
  .text("Please do this!")
  .on("click", completeTask);
```

Creating elements

Instead of using `document.createElement("li")` we can simply create an element using `$()`

- `$()` Create a new *li*
- `$("li")` Select existing *li* elements

Vanilla JavaScript

```
let footer = document.getElementById("footer");
let newItem = document.createElement("b");

newItem.classList.add("message");
newItem.innerText = "Hey!";
newItem.insertBefore(footer);
```

jQuery

```
let $footer = $("#footer");
$(<b class='message'>Hey!</b>).insertBefore($footer);
```

TIP Waiting for DOM to load

With vanilla JS we have *DOMContentLoaded*; the jQuery equivalent is:

```
// waits for the DOM to load
$(mainFunction);
```

You may see this version:

```
// waits for the DOM to load
$(document).ready(mainFunction);
```

Events and Delegation with jQuery

jQuery's `on()` works similarly to `addEventListener`. It lets you specify the type of event to listen for.

```
// logs when item with id "submit" clicked
$("#submit").on("click", function() {
  console.log("Another click");
});
```

```
//alerts when ANY button is clicked
$("button").on("click", function() {
  console.log("button clicked!");
});
```

TIP Why Use `on()`?

In most cases, `click()` and `on("click")` will both get the job done. However, there is one key difference:

- `.click(callback)` is a shorthand for `.on("click", callback)`
- `on()` accepts optional argument between type of event and callback
- This flexibility allows us to leverage *event delegation*.

Event Delegation

Event delegation allows us to attach an event listener to a parent element, but only invoke the callback if the event target matches a certain selector.

Add listener to meme container

```
function deleteMeme(evt) {  
    // delete meme...  
}  
  
$("#meme-container").on("click", ".meme", deleteMeme);
```

Add listener to each meme

```
function deleteMeme(evt) {  
    // delete meme...  
}  
  
$(".meme").on("click", deleteMeme);
```

- More performant

This will work *even if elements matching the selector don't exist yet!*

Add listener to meme container

```
function deleteMeme(evt) {  
    // delete meme...  
}  
  
// deletes a meme when it is clicked  
// even if it doesn't exist on page load  
  
$("#meme-container").on("click", ".meme", deleteMeme);
```

Add listener to each meme

```
function deleteMeme(evt) {  
    // delete meme...  
}  
  
// deletes a meme when it is clicked  
// ONLY if it exists on page load  
  
$(".meme").on("click", deleteMeme);
```

Wrap Up

Do you need jQuery?

- The DOM API is much more standardized than it used to be
- It doesn't do anything you can't do on your own
- It's an extra dependency

NOTE **You might not need jQuery**

If you're ever on the fence about whether you should include jQuery or not, here's a website that shows you how to implement a lot of jQuery functionality with vanilla JavaScript: [You Might Not Need jQuery <http://youmightnotneedjquery.com/>](http://youmightnotneedjquery.com/).

Their general philosophy is that if you want to use jQuery because it makes building your app better, great! Go for it. But if you're building a library, it's worth asking whether you *need* a dependency like jQuery.

Your turn!

jQuery has some of the best documentation out there. Check it out at <https://api.jquery.com/> <<https://api.jquery.com/>> if you're curious!



How the Web Works

Goals

- High level: what happens when you visit URL in browser
- Explain what IP and DNS are
- Describe the different parts of a URL
- Describe the request / response cycle
- Compare *GET* vs *POST* requests

What Happens When...

When I type *http://site.com/some/page.html*

into a browser, what really happens?

This is a common interview question for software engineers.

How the Web Works

The internet is complicated.

Really, really complicated.

Fortunately, to be a software developer, you only need to know a bit.

For people who want to work in “development operations”, or as a system administrator, it’s typical to have to learn more about the details here.

Networks

A *network* is a set of computers that can intercommunicate.

The internet is just a really, really big network.

The internet is made up of smaller, “local” networks.

Hostnames

We often talk to servers by “hostname” — *site.com* or *computer-a.site.com*.

That’s just a nickname for the server, though — and the same server can have many hostnames.

IP Addresses

On networks, computers have an “IP Address” — a unique address to find that computer on the network.

IP addresses look like `123.77.32.121`, four numbers (0-255) connected by dots.

There are a lot of advanced edges here that make this more complicated, but most of these details aren’t important for software engineers:

- there another whole way to specify networks, “IPv6”, that use a different numbering scheme.
- some computers can have multiple IP addresses they can be reached by
- under some circumstances, multiple computers can share an IP address and have this be handled by a special kind of router. If you’re interested in system administration details, you can learn about this by reading about “Network Address Translation”.

127.0.0.1

`127.0.0.1` is special — it’s “this computer that you’re on”.

In addition to their IP address on the network, all computers can reach themselves at this address.

The name `localhost` always maps to `127.0.0.1`.

URLs

`http://site.com/some/page.html?x=1`

turns into:

Protocol	Hostname	Port	Resource	Query
http	site.com	80	/some/page.html	?x=1

Protocols

Protocol	Hostname	Port	Resource	Query
http	site.com	80	/some/page.html	?x=1

“Protocols” are the conventions and ways of one thing talking to another.

http

Hypertext Transfer Protocol (standard web) (How browsers and servers communicate)

https

HTTP Secure (How browsers and servers communicate with encryption)

ftp

File transfer protocol (An older protocol for sending files over internet)

There are many others, but these are the common ones.

In this lecture, we'll be focusing only on HTTP.

Hostname

Protocol	Hostname	Port	Resource	Query
http	site.com	80	/some/page.html	?x=1

DNS (domain name service) turns this into an IP address

So *site.com* might resolve to `123.45.67.89`

Port

Protocol	Hostname	Port	Resource	Query
http	site.com	80	/some/page.html	?x=1

- Every server has 65,535 unique “ports” you can talk to
- Services tend to have a default port <https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers>
 - For HTTP, is port 80
 - For HTTPS, is port 443
 - You don't have to specify in URL unless you want a different port
 - To do: *http://site.com:12345/some/page.html*

Resource

Protocol	Hostname	Port	Resource	Query
http	site.com	80	/some/page.html	?x=1

- This always talks to some “web server” program on the server
 - For some servers, may just have them read an actual file on disk: */some/page.html*
 - For many servers, “dynamically generates” a page

Query String

Protocol	Hostname	Port	Resource	Query
http	site.com	80	/some/page.html	?x=1

- This provides “extra information” — search terms, info from forms, etc
 - The server is provided this info; might use to change page
 - Sometimes, JavaScript will use this information in addition/instead
- Multiple arguments are separated by &: `?x=1&y=2`

- Argument can be given several times: `?x=1&x=2`

So...

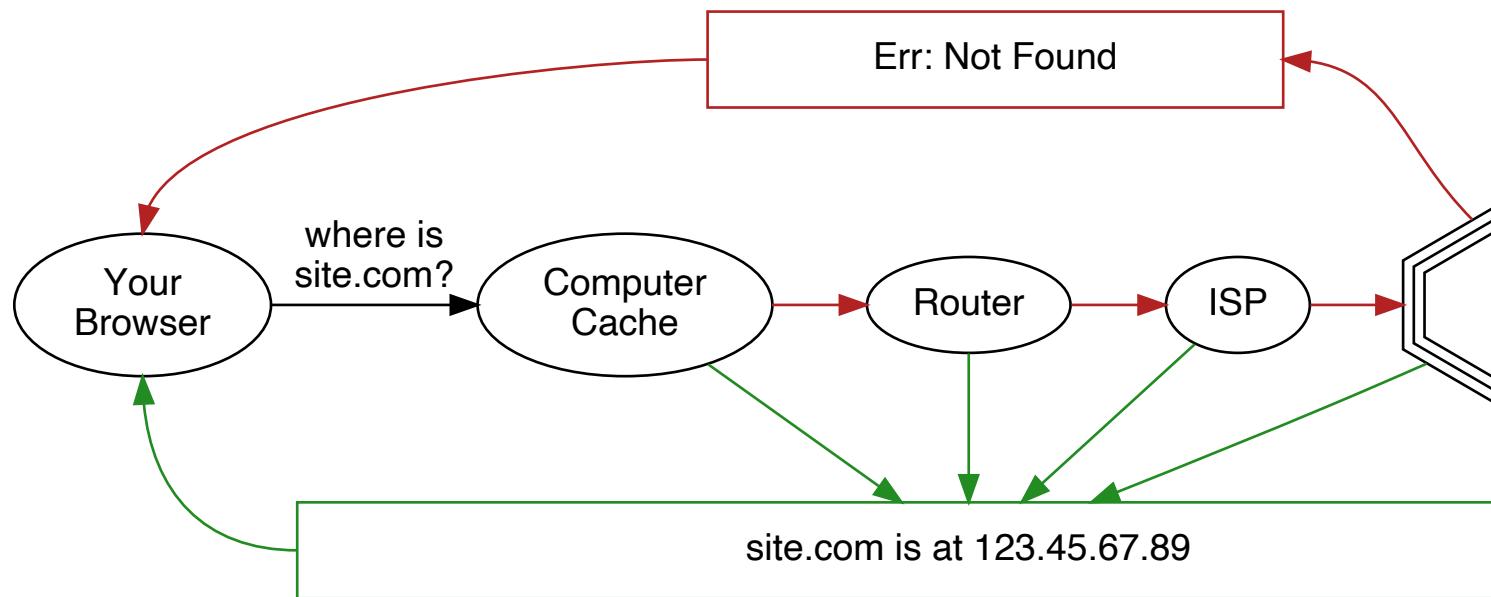
`http://site.com/some/page.html?x=1`

means

- Turn “site.com” into `123.45.67.89`
- Connect to `123.45.67.89`
- On port 80 (the default)
- Using the HTTP protocol
- Ask for `/some/page.html`
- Pass along query string: `x=1`

DNS

“I want to talk to site.com”



Unix (and OSX and Linux) systems ship with a utility, `host`, which will translate a hostname into an IP address for you, and provide debugging information about the process by which it answered this.

```
$ host -t A site.com
site.com has address 54.193.183.51
```

```
$ host -v -t A site.com
Trying "site.com"

;; QUESTION SECTION:
;site.com.           IN      A

;; ANSWER SECTION:
site.com.        300     IN      A      54.193.183.51
```

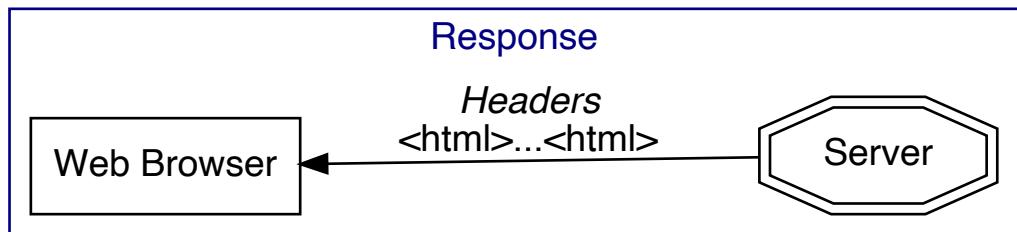
Browsers and Servers

Request and Response

When you point your browser to a webpage on a server, your browser makes a *request* to that server. This is almost always a *GET* request and it contains the exact URL you want.

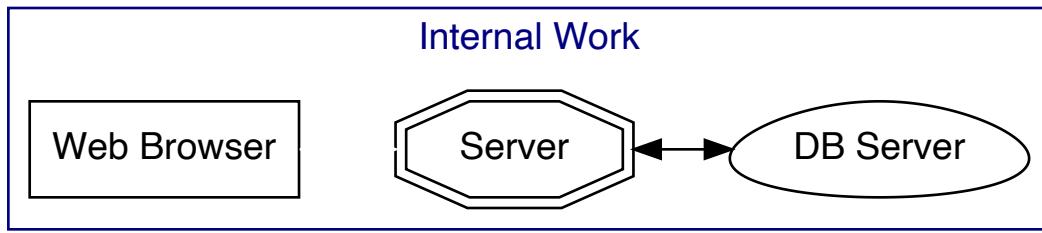


The server then responds with the exact HTML text for that page:

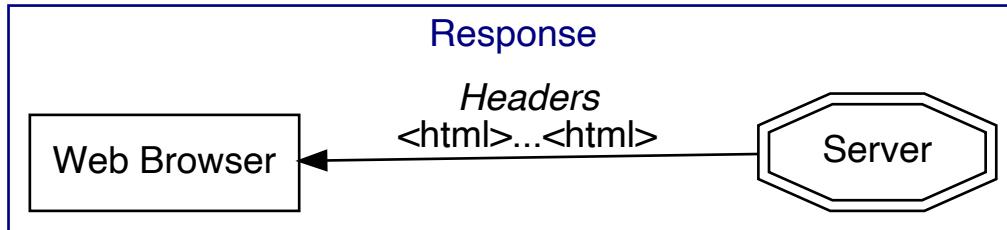


It's often the case, though, that the web server itself will have to do some work to get the page you want, often interacting with other things, such as database servers.





And then it can give back the response you want:



What's in a Request?

- Method (eg *GET*)
- HTTP protocol version (almost always `1.1`)
- Resource URL you want
- Headers
 - Hostname you're asking about
 - Date your browser thinks it is
 - Language your browser wants information in
 - Any cookies that server has sent
 - And more!

What's in a Response

- HTTP protocol version (almost always `1.1`)
- Response Status Code (*200, 404*, etc)
- Headers
 - Content Type (typically `text/html` for web pages)
 - Date/time the server thinks it is
 - Any cookies server wants to set
 - Any caching information
 - And more!

Watch a Request/Response

Request

```
GET / HTTP/1.1          # GET request for /
Host: site.com           # Header in request
```

Response

```
HTTP/1.1 200 OK          # HTTP Ver & Response Code
Date: Mon, 20 Apr 2018 07:09:16 GMT   # Date Header
Server: Apache             # Server version
Content-Type: text/html    # This is HTML content

<!doctype html>           # Body of response
<html>
  <head>
    <title>The Site</title>
  </head>
  <body>
    `...`:tan:
  </body>
</html>
```

TIP Trying this out

You can install a program called **nc**, which will allow you to interact with other computers over the IP network in a raw fashion. This can be useful to get some hands-on feeling about the web protocols.

Here, this shows an interactive session:

```
$ nc 123.45.67.89 80
Trying 123.45.67.89...
Connected to site.com.
Escape character is '^]'.
GET / HTTP/1.1          # GET request for /
Host: site.com           # Header in request

HTTP/1.1 200 OK          # HTTP Ver & Response Code
Date: Mon, 20 Apr 2018 07:09:16 GMT   # Date Header
Server: Apache             # Server version
Content-Type: text/html    # This is HTML content

<!doctype html>           # Body of response
<html>
  <head>
    <title>The Site</title>
  </head>
  <body>
    ...
  </body>
</html>
```

Response Codes

200

OK

301

What you requested is elsewhere

404

Not Found

500

Server had an internal problem

There are [more <https://en.wikipedia.org/wiki/List_of_HTTP_status_codes>](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes), but these are the most important ones.

Serving Over HTTP

Just opening an HTML file in browser uses *file* protocol, not *http*

Some things don't work same (esp security-related stuff)

It's often useful to start a simple HTTP server for testing

You can start a simple, local HTTP server with Python:

```
$ python3 -m http.server
```

Serve files in current directory (& below):

```
$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Multiple Requests

Sample HTML

demo/demo.html

```
<!doctype html>
<html>
<head>
  <link rel="stylesheet" href="demo.css">
</head>
<body>
  <h1>Hi There!</h1>
  
  <script src="demo.js"></script>
</body>
</html>
```

CSS

demo/demo.html

```
<!doctype html>
<html>
<head>
<link rel="stylesheet" href="demo.css">
</head>
<body>
<h1>Hi There!</h1>

<script src="demo.js"></script>
</body>
</html>
```

Connects to *site.com* on port 80 and requests:

```
GET /demo.css HTTP/1.1
Host: site.com
Date: [date browser thinks it is]
Cookie: [any cookies browser is storing for that host] # ...
```

Response:

```
HTTP/1.1 200 OK
Date: [date server thinks it is]
Content-Type: text/css
Cookie: [any cookies server wants you to set] # ...

body {
    background-color: lightblue;
}
...
```

Image

demo/demo.html

```
<!doctype html>
<html>
<head>
<link rel="stylesheet" href="demo.css">
</head>
<body>
<h1>Hi There!</h1>

<script src="demo.js"></script>
</body>
</html>
```

Connects to *tinyurl.com* on port 80 and requests:

```
GET /rithm-logo HTTP/1.1
Host: tinyurl.com
Date: [date browser thinks it is]
Cookie: [any cookies browser is storing for tinyurl.com] # ...
```

Javascript

demo/demo.html

```
<!doctype html>
<html>
<head>
  <link rel="stylesheet" href="demo.css">
</head>
<body>
  <h1>Hi There!</h1>
  
<script src="demo.js"></script>
</body>
</html>
```

Connects to *site.com* on port 80 and requests:

```
GET /demo.js HTTP/1.1
Host: site.com
Date: [date browser thinks it is]
Cookie: [any cookies browser is storing for that host] # ...
```

Hey, That's a Lot of Work!

- Yes, it is
- Requesting 1 webpage often involves many separate requests!
- Browsers issue these requests asynchronously
 - They'll assemble the final result as requests come back
- You can view this in the browser **console ▾ Network**

Trying on Command Line

Curl (OSX)

OSX systems come with a utility, **curl**, which will make an HTTP request on the command line.

```
$ curl -v http://site.com/some/page.html
* Rebuilt URL to: http://site.com/
*   Trying 123.45.67.89...
* Connected to site.com (123.45.67.89) port 80 (#0)
> GET /some/page.html HTTP/1.1
> User-Agent: curl/7.41.0
> Host: site.com
>
< HTTP/1.1 200 OK
< Date: Mon, 20 Apr 2018 08:28:50 GMT
< Server: Apache/2.4.7 (Ubuntu)
< Vary: Accept-Encoding
< Transfer-Encoding: chunked
< Content-Type: text/html; charset=UTF-8
<

<!doctype html>
<html>
<head>
  <link rel="stylesheet" href="demo.css">
</head>
...

```

Hey...

Everything is a string!

Methods: GET and POST

GET vs POST

- *GET*: requests without side effects (ie, don't change server data)
 - Typically, arguments are passed along in query string
 - If you know the arguments, you can change the URL
 - Entering-URL-in-browser, clicking links, and *some* form submissions
- *POST*: requests with side effects (ie, change data on server)
 - Typically, arguments sent as body of the request (not in query string)
 - *Some* form submissions (but never entering-URL-in-browser or links)
 - Always do this if there's a side-effect: sending mail, charge credit card, etc
 - “Are you sure you want to resubmit?”

Sample GET Requests

```
<a href="/about-us">About Us</a>  
  
<a href="/search?q=lemurs">Search For Lemurs!</a>  
  
<!-- will submit to URL like /search?q=value-in-input -->  
<form action="/search" method="GET">  
  Search for <input name="q">  
  <button type="submit">Search!</button>  
</form>
```

Sample POST Request

POST requests are always form submissions:

```
<!-- will submit to URL add-comment, with value in body -->  
<form action="add-comment" method="POST">  
  <input name="comment">  
  <button type="submit">Add</button>  
</form>
```

HTTP Methods

GET and *POST* are “HTTP methods” (also called “HTTP verbs”)

They’re the most common, by far, but there are others

Further Study

- How the Internet Works: A Code.org Video Series <<https://www.youtube.com/playlist?list=PLzdnOPI1jJNfMRZm5DDxco3UdsFegvuB7>>
- How Does the Internet Work? <<https://web.stanford.edu/class/msande91si/www-spr04/readings/week1/InternetWhitepaper.htm>>
- A cute web comic about how DNS works <<https://howdns.works/>>

How The Web Works

Two Quick Things

1. Cold-calling
 2. Hand-raising
-

*And down under all those piles of stuff, the secret was written: We build our computers the way we build our cities — **over time, without a plan, on top of ruins.***

— Ellen Ullman, Salon Magazine (1998)
https://www.salon.com/1998/05/12/feature_321/

How, Why, and History

No how without why, no why without (some) history.

I can't emphasize this enough, e.g., why does the `` tag use `src` and not `href`?

Answer: <http://1997.webhistory.org/www.lists/www-talk.1993q1/0182.html>

Outline

1. A (very) brief history of the internet
 2. Networking, in general
 3. DNS, aka the *Domain Name System*
 4. 🕸️ The web 🕸️
-

~~There's No Magic~~

Typically, your computer's doing it, there's a way to see it.

We can make the invisible, visible using tools and commands already on your computer.

If you remember anything, remember how to use the tools to get answers to your questions.

What is networking?

What is a networking protocol?

What is a networking protocol?

A set of rules for how computers communicate with each other.

- What do messages look like?
 - What can they contain?
 - What are the rules for when and how messages are sent back and forth?
-

What's an IP address?

What is an IP address?

IP stands for Internet Protocol. An IP address is like a "phone number" or "mailing address" for a computer communicating using the internet protocol.

What is a port?

All your network traffic is coming and going through a single interface, whether that's a physical ethernet cable or WiFi or something else.

How does your computer know to which traffic is associated with which program?

Answer: Ports

(Much) more on this later, but briefly...when your computer connects to another computer over the internet, the hostname isn't enough because that computer can be running many different services.

You need to specify something called a *port*. This is a number which identifies the service you want to communicate with on the other computer.

For now, just remember...

Three Ingredients

Communicating with another computer over the internet requires three things:

1. *What* computer, i.e., the IP address
 2. *Which* service, i.e., the port
 3. *How* to communicate once connected, i.e., the protocol
-

Many Protocols

There are many, *many* protocols that run on top of the internet.

The web uses a protocol called **HTTP (HyperText Transfer Protocol)**. More on that HTTP and the web later.

I just want to emphasize “the web” is one of *thousands* of internet-based protocols.

What does an IP address look like?

IP addresses are four numbers, each between `0` and `255`, separated by periods.

Examples:

- `142.251.45.238`
 - `17.253.144.10`
 - `23.63.55.204`
-

Two things:

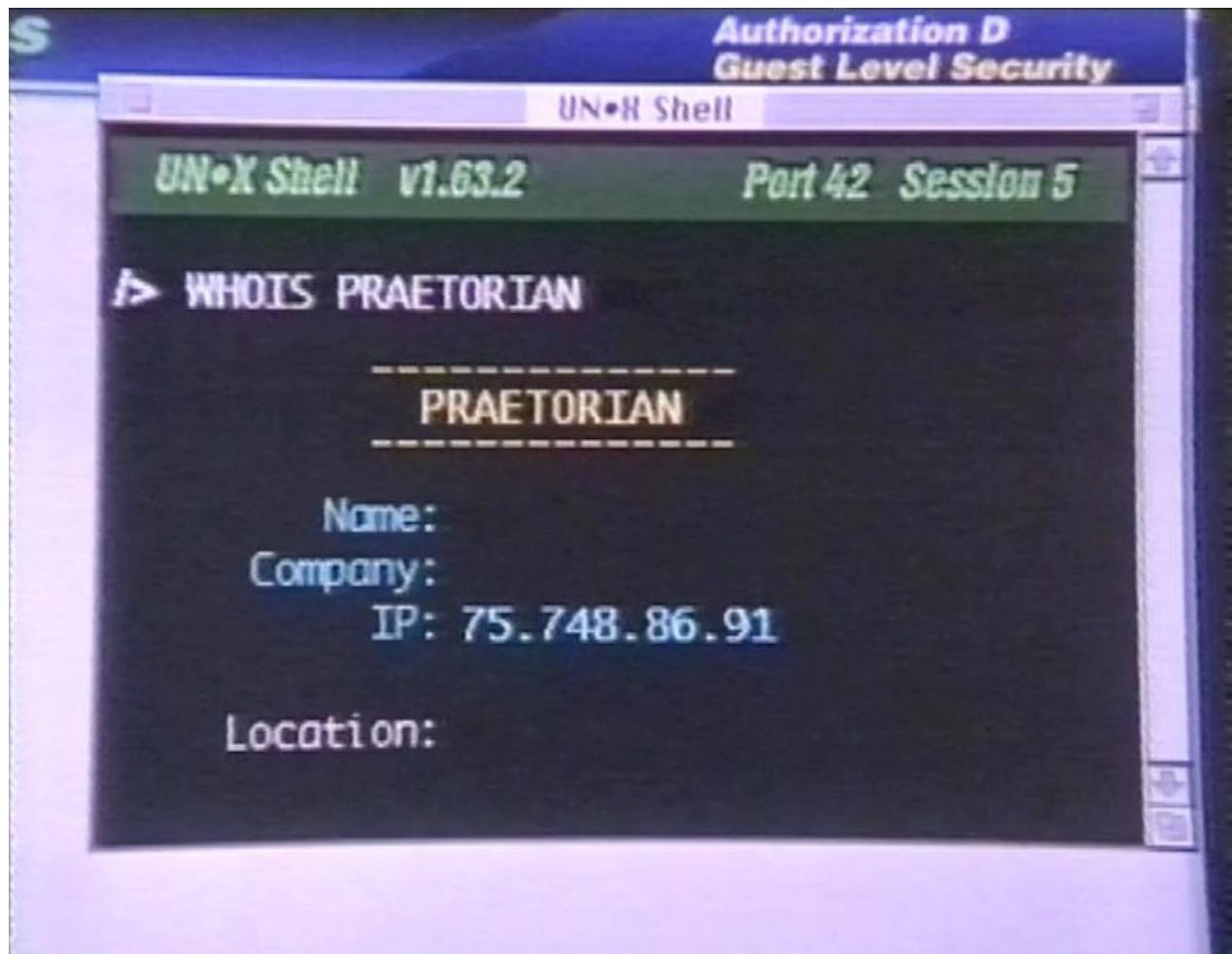
1. An IP address like `879.64.8.1` isn't valid because each number must be between `0` and `255`
 2. You might see shorthand like `17.253.y.z` which means any IP address beginning with `17.253`
-

Special IP Addresses

Some IP addresses have special meaning and aren't "publicly routable". That means a computer can't have one of these IP addresses as their *public* IP address.

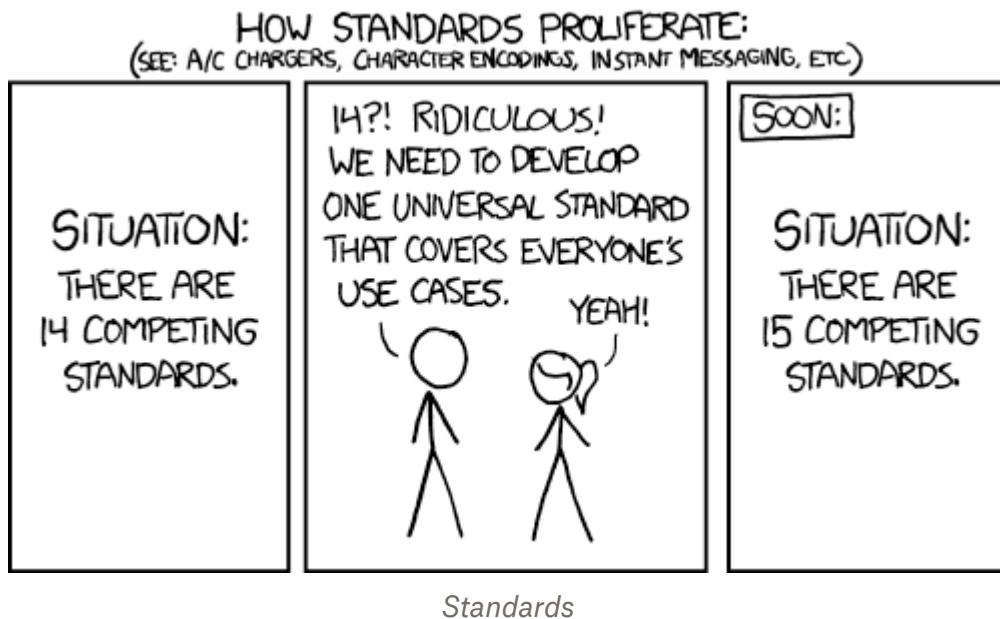
- `127.0.0.1` - your own computer, the first-person pronoun of IP addresses
 - `169.254.y.z` - fallback IP address range if a machine doesn't have an IP address configured
 - `192.168.y.z` - used for internal networks
 - `10.x.y.z` - also used for internal networks
-

POP QUIZ



The American President – The Net – tape 2397 | VHISStory

A (Very) Brief History of the Internet



The internet is a *network of networks*

...like New York or Tokyo are “cities of cities”.

The internet grew out of the need for computers in isolated-but-growing networks to communicate with each other.

Our largest cities are often consolidated versions of smaller cities that grew so big and so close they decided to consolidate, e.g., New York City, London, Tokyo, and Montréal.

Isolated Networks (Late 1960s - Early 1970s)

- Countries often had their own, often tied to national security concerns, e.g., ARPANET in the US, CDNet in Canada, CYCLADES in France
- Companies, too, e.g., if you worked at IBM you could use VNet
- Regional, e.g., Michigan’s *Merit Network*

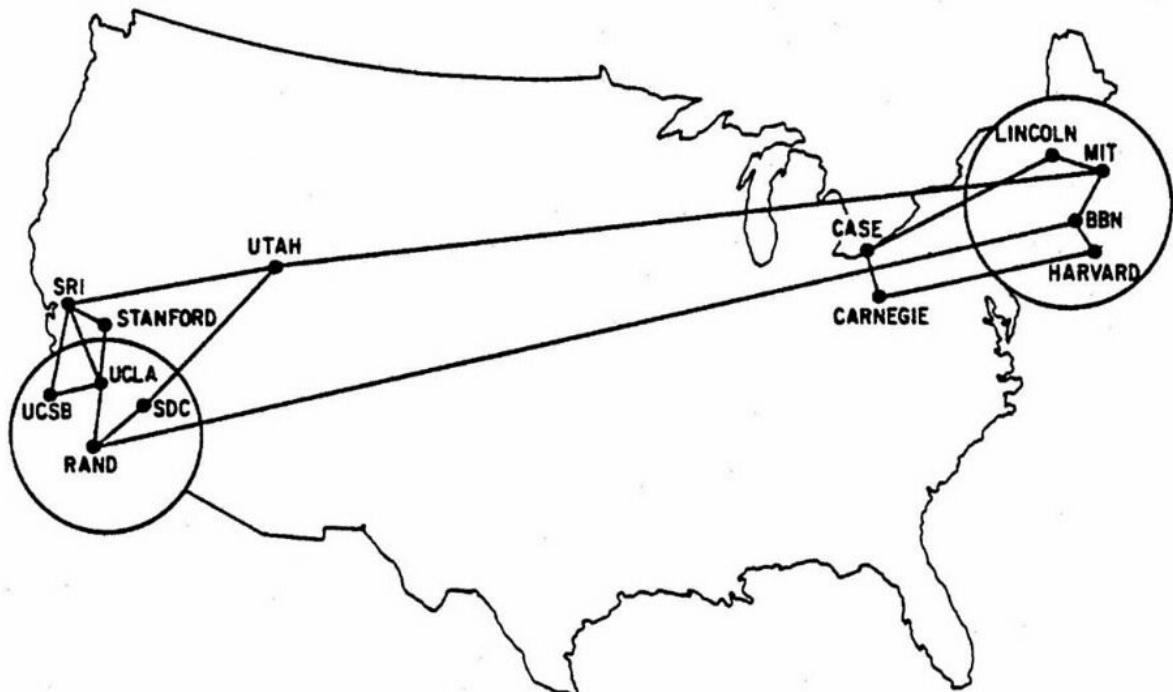
Note

The first email systems are defined + created during this time — *before the internet*.

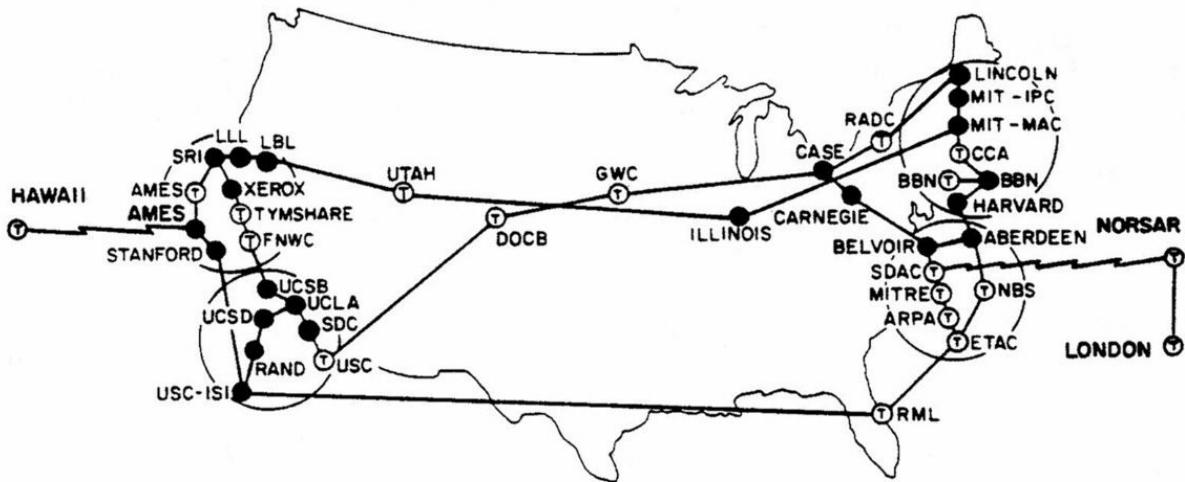


The ARPANET in December 1969

Before the internet, there was the ARPANET



1970: ARPANET expands



1973: ARPANET goes international

Mid-1970s: Proposing The Internet

In 1974, inspired largely by the design of France's CYCLADES, **Vint Cerf** and **Bob Kahn** propose what eventually becomes **TCP/IP** as a protocol for inter-network communication.

TCP/IP stands for **T**ransmission **C**ontrol **P**rotocol and **I**nternet **P**rotocol.

This is the same system we use today.

Early 1980s - The Internet

In 1982, ARPANet adopts **TCP/IP** and “the internet” becomes defined as connected **TCP/IP** networks.

Proprietary networks try to compete throughout the 1980s and even into the 1990s, but you’re nothin’ if you’re not on the internet.

1983 - Birth of the Internet

Two very important things happen in 1983



1983: DNS

DNS stands for Domain Name System

It's time to talk about how computers actually communicate over a network

Hostnames

Referring to other computers by IP address is tedious and error-prone.

If someone's IP address changes, how is anyone else supposed to know?

We'd like to refer to computers by name (called the *hostname*).

Before DNS

In the 1970s and early 1980s, if you wanted to refer to a computer by name you'd use a simple text file called a `hosts` file. Each line contained an IP address and a human-readable name like `jim.cs.ucla.edu`.

If you told your computer to connect to `jim.cs.ucla.edu` it'd first look in the `hosts` file to see if it could find an IP address.

If not, error.

The hosts file still exists!

On UNIX-based systems like macOS and Linux, the `hosts` file lives at
`/etc/hosts`

On Windows it lives at

```
C:\Windows\System32\drivers\etc
```

Downsides?

DNS

DNS was designed to solve these problems. It's a decentralized system for assigning **DNS records** a given hostname / domain.

There are many types of DNS records, but the main one is called an **A Record** and it tells you

You can use the `host` command to find the DNS records for a given domain.

Let's try it!

Open your terminal and type each of the following commands in turn:

- 1 host facebook.com
 - 2 host rithmschool.com
 - 3 host localhost
-

Back to Ports

Remember, you need to specify *which* service you want to use when trying to connect with another computer over the internet.

To do that, you need to specify something called a *port*. This is a number ranging between `0` and `65535` which identifies the service you want to communicate with on

the other computer.

No Magic: Ports

You can use the `netstat` command to see all the currently active network connections on your computer. This includes the IP addresses and ports.

```
netstat -anvp tcp
```

Binding to Ports

When a program wants to listen for an incoming connection it asks the operating system to give it a connection and “bind to a port”.

Any program can ask to bind to any port, but only one program can claim a given port at a time.

Which service corresponds to which port is a matter of *convention*.

Ports Are A *Convention*

The protocol used by the web is called **HTTP** which stands for **Hypertext Transfer Protocol**. Related is the **HTTPS** protocol which stands for **HTTP Secure**, used for secure web communication.

When using your browser day-to-day you don’t often see ports. This is because *by convention* HTTP and HTTPS have standard ports.

- The standard HTTP port is `80`
 - The standard HTTPS port is `443`
-

URLs

Notice the `:3000` below. That's a port!

Everyone open two tabs in your browser:

1. <http://hello.20bits.com:3000/waffles/182>
2. <http://hello.20bits.com:3000/hello.html>

If `hello.20bits.com` doesn't load, visit these instead:

1. <http://requests.20bits.com/waffles/182>
 2. <http://requests.20bits.com/hello.html>
-

Making HTTP Requests

1. The browser
 2. Using `curl` on the command line
 3. Using `nc` on the command line
 4. Using code
-

Request / Response Cycle

The web was designed first and foremost by academics to share (HTML) documents. The document-centric nature of the web affects everything.

Think of an HTTP server as a document depot. You call it up and ask to do different things with the documents.

What are some things you might want to do?

CRUD

These are the four main things you want to do with a document and HTTP was designed to accommodate all of them:

1. Create

2. Read
 3. Update
 4. Destroy
-

Your Own Personal Live Server

Run this command in any directory and it will launch a localhost-only web server that serves files from that directory:

```
python -m http.server
```

Demo!

AJAX

Goals

- Describe what AJAX is
- Compare AJAX requests to non-AJAX requests
- Make GET and POST AJAX requests with axios
- Use `async / await` to manage asynchronous code with axios
- Describe what JSON is

AJAX

Traditional Requests

Traditional browser requests happen in response to:

- Entering a URL in the browser bar
- Clicking on a link
- Submitting a form

In all cases:

- Browser makes request
- Receives response
- Replaces *entire resource* with result

[Traditional Demo <http://localhost:5000/trad>](#)

```
<!-- EXAMPLE 1: SIMPLE GET REQUEST -->

<h2>Simple GET Request</h2>

<a href="/card" class="btn btn-primary">Get Card</a>

<!-- EXAMPLE 2: SIMPLE POST REQUEST -->

<h2>Simple POST Request</h2>

<form action="/borrow" method="POST">
  <input name="amount" placeholder="Amount" />
  <button class="btn btn-warning">Borrow</button>
</form>
```

AJAX

AJAX web request:

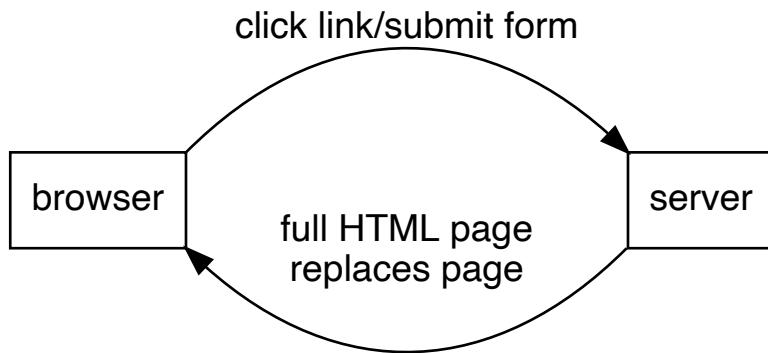
- Made from JavaScript in browser
- JavaScript makes request (*GET*, *POST*, or other)
- You receive a response
- Do whatever you want with result!

AJAX is a technique in Javascript for sending requests and receiving responses from a server *without* having to reload the browser page.

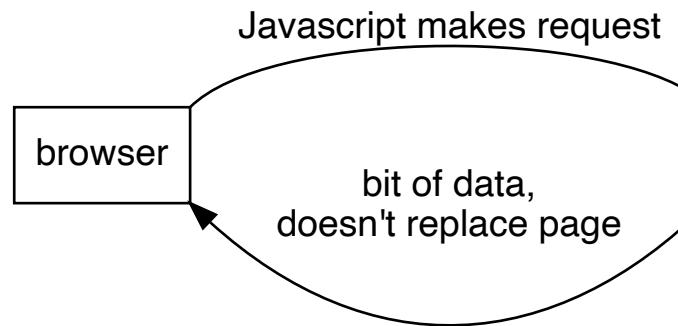
What Does AJAX stand for?

AJAX originally was an acronym for “Asynchronous Javascript and XML”. However many people don’t send XML over AJAX nowadays; it’s more common to send HTML or JSON. The technology is still the same, though, even if the data payload is commonly different. Ultimately, AJAX is a cooler sounding acronym than AJAJ or AJAH.

Regular Request



AJAX Request



Why Use AJAX?

- Don’t need to reload entire page if just 1 thing is changing
- Interactive web sites
- Fewer full page loads from server
 - Your JS can talk to other servers directly
- Less info has to go across network

AJAX with Axios

You don’t *have* to use Axios for this

- There is an old, clunky built-in tool: (*XMLHttpRequest*)

- Or a newer-but-still-clunky built-in tool: (*fetch*)
- Or lots of other libraries (including *jQuery*)
- ... but we'll use *axios* for now! It's featureful & popular

Getting Axios

Can easily include it using a CDN link:

```
<script src="https://unpkg.com/axios/dist/axios.js"></script>
```

Making a Simple Request

```
axios.get(url)
```

Make a *GET* request to that URL

Not What We Expected

```
let card = axios.get("/api/card");
console.log(card);
// "Promise {<pending>}"
```

What's A Promise???

- We'll talk about it in more detail when we get to Node.
- For now, all you need to know is that a promise is like a placeholder for a future value.
- We want to wait for the promise to have that value before proceeding.
- But we don't know when the promise will receive its value!

Handling Asynchronous Code

Asynchronicity

AJAX requests are *asynchronous*

- The *axios.get()* completes *before* the response is received
- This means that if we want to use the data we get back from our AJAX requests, we need to *wait* until the response has been given to us
- We're going to use two newer keywords in JS to do this: *async* and *await*!

Await

Here's what it looks like:

```
await axios.get('/api/card');

// returns response object, with `data` as response body
```

The code is asynchronous, but it “waits” for the AJAX request to complete.

Async

To use *await* in a function, you must mark that function as *async*:

```
async function getCardInfo() {
  let response = await axios.get("/api/card");
  console.log("got", response);
  return response.data;
}
```

When calling *async* function, you should *await* it:

```
let card = await getCardInfo();
```

NOTE Using *await* outside of all functions

Normally, you can't use *await* when you're not in an *async* function. However, as a special case, many web browsers (including Chrome) allow you to do so only if you enter that line in the console directly.

Callbacks Vs Async/Await

- Callbacks are what we've used for event handlers and timers
 - But they're tricky to nest or do other complex things
- *async/await* makes it easier to handle chains of requests
 - Modern libraries like Axios return “promises”, which you await

Axios API

.get

```
axios.get(url, ?config)
```

config is an optional object many Axios methods use

It holds specific configuration for what you need.

demo/templates/index.html

```
<h2>Simple GET Request</h2>

<button class="btn btn-primary"
        id="card-btn"> Get Card </button>

<div id="card" class="box"></div>

<script src="/static/card.js"></script>
```

demo/static/card.js

```
/* show result directly in card box */

async function getCard() {
  let response = await axios.get(
    "/api/card");

  console.log("getCard resp=", response);
  $("#card").html(response.data);
}

$("#card-btn").on("click", getCard);
```

To make request for `/resource?a=1&b=2`, can either use:

```
axios.get("/resource?a=1&b=2")
```

or

```
axios.get("/resource", {params: {a: 1, b: 2}})
```

Second form is better: you don't have to worry about how to "url safe quote" characters that aren't normally legal in URLs.

.post

Similar to `axios.get`, but uses a *POST* request

```
axios.post(url, ?data, ?config)

axios.post(url, {a: 1, b: 2})
```

This is passed as JSON to the server

demo/templates/index.html

```
<h2>Simple POST Request</h2>

<input id="amount" placeholder="Amount" />
<button class="btn btn-warning"
        id="borrow-btn"> Borrow </button>

<div id="borrowed" class="box"></div>

<script src="/static/borrow.js"></script>
```

demo/static/borrow.js

```
/* show result of borrowing in box */

function showBorrow(res) {
  $("#borrowed").html(res);
}

async function borrowMoney() {
  let amount = Number($("#amount").val());

  let response = await axios.post(
    "/api/borrow", { amount });

  console.log("borrow resp=", response);
  showBorrow(response.data)
}

$("#borrow-btn").on("click", borrowMoney);
```

JSON

- JSON is a string that looks like a JS object
- Most APIs use JSON to communicate
 - What's an API? We'll talk about it soon!
- By default, Axios recognizes JSON response & turns into JS object
- By default, Axios sends *POST* data as JSON

demo/templates/index.html

```
<h2>Getting JSON Responses</h2>

Get <input id="ncards" value="5" /> Cards
<button class="btn btn-primary"
        id="hand-btn">Go!</button>
<div id="hand" class="box"></div>
<script src="/static/hand.js"></script>
```

demo/static/hand.js

```
/* show result of hand in box */

function showHand(hand) {
  let $box = $("#hand");
  $box.empty();

  for (let {rank, suit} of hand) {
    let t = `<p>${rank} of ${suit}</p>`;
    $box.append($(t));
  }
}

async function getHand() {
  let ncards = Number($("#ncards").val());

  let response = await axios.get(
    "/api/hand", { params: { ncards } });

  console.log("getHand resp=", response);
  showHand(response.data.hand);
}

$("#hand-btn").on("click", getHand);
```

NOTE Global JSON object

JavaScript comes with a global `JSON` object which can convert strings of JSON into JavaScript objects, and vice versa. These methods are `JSON.stringify` (`object` → JSON) and `JSON.parse` (JSON → `object`).

```
JSON.stringify({
  name: "Whiskey",
  favFood: "popcorn",
  birthMonth: 7
});
// '{"name": "Whiskey", "favFood": "popcorn", "birthMonth": 7}'

JSON.parse('{"name": "Whiskey", "favFood": "popcorn", "birthMonth": 7}');
// {name: "Whiskey", favFood: "popcorn", birthMonth: 7}
```

NOTE “Form Encoded” POST requests

By default, Axios sends POST data as JSON. This is what almost all modern APIs expect.

When web browsers submit POST forms in the traditional way (ie, not using AJAX), they don't send this data in JSON — they send it in an older format, “form-encoded”.

It's not common that you'd want Axios to send POST data this way. But you may be working with older APIs that expect data in this format, or you may want to work on switching over an older, non-AJAX application to an AJAX one, and find it helpful for the server to receive traditional form-encoded data. For an example of how to do so, see

[<https://www.npmjs.com/package/axios#browser>](https://www.npmjs.com/package/axios#browser)

Wrap Up

Big Ideas

- Traditional web requests:
 - Made by browser (via link, form, URL bar, etc)
 - Replace *entire page* with thing linked to
- AJAX requests:
 - Made via JS AJAX calls
 - JS get data; JS decides what to do with it
- Axios is the popular AJAX client we'll use
- AJAX calls are asynchronous & return a “promise”
 - You need to *await* those to get real results
 - Functions that use *await* must be *async*
- JSON
 - Axios parses JSON responses automatically for us

Axios Docs

[<https://www.npmjs.com/package/axios>](https://www.npmjs.com/package/axios)



Working with APIs

Goals

- Define what an API is
- Compare and contrast different kinds of APIs
- Understand the limitations
- Use Terminal and GUI clients for making HTTP requests

APIs

What Is An API?

A set of clearly defined methods of communication between various components.

An API may be for a web-based system, operating system, database system, computer hardware, or software library.

APIs You Have Used

Axios API https://axios-http.com/docs/api_intro <https://axios-http.com/docs/api_intro>

The jQuery API <https://api.jquery.com/> <<https://api.jquery.com/>>

JavaScript Array API https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array>

Third Party APIs

Companies will provide access to their data (sometimes not for free)

- Twitter API, give me all tweets that mention “ice cream”
- Facebook API, send me the current user’s profile picture
- Weather API, what is the weather in Missoula Montana?
- Reddit API, what is the current top post?
- GooglePlaces API, what gas stations are near the user?
- Yelp API, give me 10 restaurants in the zipcode 94110

Data Formats

- When we browse on the web, we make HTTP requests and get HTML back.
- APIs don't respond with HTML.
 - HTML contains info about page structure. APIs respond with data, not structure.
- They use different data formats like XML and JSON.
 - These are still text based formats—remember, HTTP is text based!

XML

Syntactically similar to HTML, but does not describe presentation like HTML, and many of the tags are custom.

```
<person>
  <name>Elie</name>
  <favoriteColor>purple</favoriteColor>
  <city>San Francisco</city>
</person>
```

JSON

JSON stands for **JavaScript Object Notation**.

JSON looks similar to JS objects, but all the keys must be “double-quoted”.

```
{
  "person": {
    "name": "Elie",
    "favoriteColor": "purple",
    "city": "San Francisco",
    "favoriteNumber": -97,
    "interests": ["CEOing", "eating Mediterranean food"],
    "futureDreams": null
  }
}
```

A JSON payload must be sent as a string over HTTP requests.

To convert JavaScript object to JSON string:

```
JSON.stringify(myObject) // "...string of JSON..."
```

To convert JSON string to JavaScript object:

```
JSON.parse(jsonString) // {prop: value, ...}
```

Most libraries do this for you.

JSON vs XML

We'll primarily use JSON: it's easier to parse & works great with JavaScript!

JSON is also the contemporary standard for most modern APIs.

API Security

Of course, some APIs require you to have an “API key” or password to use them.

You'll need to read their docs to understand how to get those.

Separate from that concern, many APIs cannot be used via AJAX because of the *same origin policy*.

AJAX & Same Origin Policy

By default, you can only get a resource via AJAX if the page making the request is the “same origin” as the API URL.

To be “same origin”, it needs the same hostname, protocol, *and* port.

From a page served at <https://site.com/users>:

- `https://api.site.com/books`: **not same**, different hostname
- `http://site.com/api/books`: **not same**, different protocol
- `https://site.com:5000/api/books`: **not same**, different port
- `https://site.com/api/books`: **same origin**

When an AJAX request is blocked by the “same origin policy”, the *browser* refuses to make the request.

You don't get things like a 404 error (*those come from the server*). Instead, you get messages like this:

```
Access to fetch at *https://api.twitter.com/* from origin  
*https://site.com* has been blocked by CORS policy.
```

If you get any status code (400, 500, etc), the same origin policy didn't block the AJAX request

Some APIs allow themselves to be used by other sites via AJAX:

So, from an AJAX call from [Rithm School's Website](https://rithmschool.com) <<https://rithmschool.com>>, this works:

```
await fetch("https://api.github.com")
```

Other APIs do not:

```
await fetch("https://api.twitter.com")
```

(That would work if request was made from `https://api.twitter.com`, because that's the “same origin” as the AJAX call)

TIP A way around this will come later!

We will see later in this course how to work around this policy!

For now, with the APIs we will be working with, if you are seeing errors that mention the Same Origin Policy or something called CORS (Cross Origin Resource Sharing), you most likely have something wrong with your request.

Insomnia

A GUI for making HTTP requests.

<https://insomnia.rest/> <<https://insomnia.rest/>>

Practicing with Insomnia

If you want extra practice, check out <https://jsonplaceholder.typicode.com/>

Curl

curl is used in command lines or scripts to transfer data.

Open source & comes with your computer—so it's easy to use right out of the box

Making a request using Curl

We do it in the Terminal!

Simplest & most common request/operation made using HTTP is to GET a URL:

```
$ curl 'https://curl.haxx.se'
```

This will return the entire resource from the server.

```
$ curl 'https://api.github.com/users/elie'
```

This will return a JSON response from the Github API

Flags with Curl

- `-d` or `--data` to send information to a server
`-d ' {"username":"xyz", "password":"xyz"} '`
- `-X` or `--request` to specify HTTP verb (`-X POST`)
- `-H` or `--header` to specify additional headers

```
-H "Content-Type: application/json"
```

Example of a larger request

```
curl --header "Content-Type: application/json" \
--request POST \
--data '{"username":"xyz","password":"xyz"}' \
'https://myapplication.com/login'
```

When to use Curl

- When you are making a simple HTTP(S) request
- When you don't have any other option
- When you're doing scripting
- You will also see it in almost all API documentation for examples



Bootstrap

Goals

- Learn benefits of Bootstrap & how to include it
- Learn Bootstrap layout/grid system
- Meet some of more important components
- Learn to integrate Bootstrap Icons into sites

Benefits

- Make tricky column layouts easy
- Provides consistent results across browsers
- Make responsive design much easier
- Includes useful interactive components
 - Modals, dropdowns, popovers, etc
- *Declarative* look and feel
 - Easier to theme
 - Familiar for writing custom CSS

Big Ideas: Responsive & Semantic

Responsive Groups

xs

Cell phones in portrait mode

sm

Cell phones in landscape / small tablets

md

Tablets

lg

Average sized-laptops

xl

Wide screens

xxl

Ultra-wide screens

Semantic Colors

Name	Purpose	Default
<i>primary</i>	Brand color	blue
<i>secondary</i>	Neutral brand-appropriate color	grey
<i>success</i>	Operation successful	green
<i>danger</i>	Dangerous operation / error	red
<i>warning</i>	Risky operation	orange
<i>info</i>	FYI message	light blue

Used for table cells, text, buttons, & more

Using Bootstrap

Include their CSS:

```
<link rel="stylesheet"  
      href="https://unpkg.com/bootstrap@5/dist/css/bootstrap.css">
```

To use interactive components, include JS:

```
<script  
      src="https://unpkg.com/bootstrap@5/dist/js/bootstrap.bundle.js">  
</script>
```

Layout

All content should descend from a *container* element:

.container-fluid

Full-browser-width container (with small amount of breathing room).

.container

Full-browser-width but at specific breakpoints. Makes UI testing easier: far fewer possible layouts to test.

Content that doesn't need to be in columns can go directly in this.

Grids

- 12 Column Layout
- Cells can span any number of columns
- After all columns are used, will become new row
- To use: all columns must be in a *.row*

```

<div class="container">
  <div class="row">
    <div class="col-4">A</div>
    <div class="col-4">B</div>
    <div class="col-4">C</div>
    <div class="col-4">D</div>
    <div class="col-4">E</div>
    <div class="col-4">F</div>
  </div>
</div>

```

two rows of 3 columns

Responsive Grid

- Can specify a breakpoint: that size and above use this
- Specification without breakpoint is for *xs*

different # columns

```

<div class="container">
  <div class="row">
    <div class="col-6 col-md-4">A</div>
    <div class="col-6 col-md-4">B</div>
    <div class="col-6 col-md-4">C</div>
    <div class="col-6 col-md-4">D</div>
    <div class="col-6 col-md-4">E</div>
    <div class="col-6 col-md-4">F</div>
  </div>
</div>

```

(2 columns on cell phones; 3 columns for larger devices)

Auto-Columns

- Can leave off numbers & divide by available size
- Useful when you don't know how many items there will be

auto-columns

```

<div class="container">
  <div class="row">
    <div class="col">A</div>
    <div class="col">B</div>
  </div>

  <div class="row">
    <div class="col">C</div>
    <div class="col">D</div>
    <div class="col">E</div>
  </div>
</div>

```

Learning More

Grid Docs <<https://getbootstrap.com/docs/5.0/layout/grid/>>

Images

.img-fluid

Make image responsive; won't be wider than parent

Learn more: [Images Docs <https://getbootstrap.com/docs/5.0/content/images/>](https://getbootstrap.com/docs/5.0/content/images/)

Tables

.table

Get nice standard table look (use this plus other classes)

.table-hover

Hover-effect over a row

.table-sm

Tighten up margin around cells

.table-striped

Stripe alternative rows

Learn more: [Table Docs <https://getbootstrap.com/docs/5.0/content/tables/>](https://getbootstrap.com/docs/5.0/content/tables/)

Alerts

Useful for providing feedback/warnings:

.alert

(use this plus other classes)

.alert-[semantic-color]

Use color scheme for this level of message.

Learn more: [Alerts <https://getbootstrap.com/docs/5.0/components/alerts/>](https://getbootstrap.com/docs/5.0/components/alerts/)

Buttons

.btn

(Use this plus other classes)

.btn-[semantic-color]

Use color scheme for this level of message

.btn-link

Make button look like a `<a>` link

.btn-lg / .btn-sm

Make larger or smaller button

Can use on `<a>` links to look like buttons—very useful!

Learn more: [Button Docs <https://getbootstrap.com/docs/5.0/components/buttons/>](https://getbootstrap.com/docs/5.0/components/buttons/)

UI Components

- Breadcrumbs
- Forms
- Lists
- Media cards
- Pagination sets
- and more!

Learn more: [Components <https://getbootstrap.com/docs/5.0/components/>](https://getbootstrap.com/docs/5.0/components/)

JavaScript Components

Need to add JS CDN link for Bootstrap

- Carousels
- Collapse
- Dropdown
- Modals
- Popovers
- Tooltips
- and more!

Learn more: [Components <https://getbootstrap.com/docs/5.0/components/carousel/>](https://getbootstrap.com/docs/5.0/components/carousel/)

Bootstrap Wrap-Up

Does Everyone Use Bootstrap?

No

But almost everyone uses *some* CSS framework

Other popular frameworks:

- Foundation <<https://foundation.zurb.com/>>
- Tailwind <<https://tailwindcss.com>>

Theming Bootstrap

- Can write your own CSS to change things
- Can make your own Bootstrap with SASS (*advanced*)
- Can find thousands of Bootstrap themes
- Can easily use Bootswatch <<https://bootswatch.com/>>

Bootstrap Icons

- Excellent image icons: [Bootstrap Icons <https://icons.getbootstrap.com/>](https://icons.getbootstrap.com/),
- Icons come as fonts allowing them to scale easily

Include this:

```
<link rel="stylesheet"  
      href="https://unpkg.com/bootstrap-icons/font/bootstrap-icons.css">
```

Use icons by name on a *i* or *span* tag

```
<!-- bi stands for bootstrap icon -->  
<i class="bi bi-apple"></i> <!-- Apple icon -->  
<i class="bi bi-star-fill"></i> <!-- solid star -->  
<i class="bi bi-star"></i> <!-- regular (outline) of star -->  
  
<!-- font-size can be applied to an icon to resize -->  
  <i class="bi bi-search" style="font-size: 30px"></i>  
  <i class="bi bi-search" style="font-size: 80px"></i>  
  
<!-- icons scale in size to fit their parent element -->  
  
<!-- search icon inside a small button -->  
<button type="submit" class="btn btn-primary btn-sm">  
  <span class="bi-search"></span>  
  Search  
</button>  
  
<!-- search icon inside a large button -->  
<button type="submit" class="btn btn-primary btn-lg">  
  <span class="bi-search"></span>  
  Search  
</button>
```

NOTE Font Awesome

Another popular font library is [Font Awesome <https://fontawesome.com>](https://fontawesome.com). While this has historically been very popular, the core open source part of it has become harder to use outside of their commercial offering.



Introduction to Python

Intro

- a general purpose programming language
- fast, powerful, widely used
- *high level*: express concepts at a high level (*a little more than JS*)
- *dynamic*: can change variable types, make new functions in your code, etc
- runs on servers (*but not in a browser*)
- particularly used for data science, machine learning, servers, et al

Python Versions

Python 2

- Latest is 2.7
- What some people still use
- What comes by default on OSX

Python 3

- Latest is 3.11 (*you may have a slightly older one; that's fine*)
- Slightly different language & syntax
- What we'll use at Rithm

Installing Python

```
$ brew install python
```

Test that it works: in a *new Terminal window*

```
$ which python3  
/usr/local/bin/python3
```

Install other Python utility: *ipython*:

```
$ brew install ipython
```

Test that it works:

```
$ which ipython  
/usr/local/bin/ipython
```

Interactive Python

IPython is a program for interactive exploring of Python

```
$ ipython
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0: An enhanced Interactive Python. Type '?' for help.

In [1]: print("Hello, World!")
Hello, World
```

(«^d» to exit)

Printing

```
print(value, value, ...)
```

- Puts spaces between values
- Puts return character (*newline*) at the end

```
x = "awesome"

print("Python is", x)
```

Indentation

In *many* programming languages, you use `{` and `}` to show blocks:

```
if (voter) {
    console.log("Please go vote!");
    vote();
}
```

Programmers indent code for readers, but this would work the same:

This would work the same:

```
if (voter) {
    console.log("Please go vote!");
    vote();
}
```



(So ugly. Please don't do that.)

In Python, you don't use `{ / }` for blocks; the indentation *is* what matters:

```
if voter:
    print("Please go vote!")
    vote()
```

That's very different than:

```
if voter:  
    print("Please go vote!")  
vote()
```

In JS, people often use 2 or 4 spaces for indentation (styles vary)

In Python, *everyone* agrees: it should always be 4 spaces

Variables

- Python variable name style is `like_this` (lower-snake-case)
- There is no keyword for declaring variables; ie no `let` or `var`
- No specific way to make un-re-bindable like `const`
 - It's good style to write global constants `LIKE_THIS`
- “Lexical function scoped”

```
x = 42  
  
def my_function():  
    x = 12  
    print(x)    # 12  
  
print(x)        # 42
```

Strings

- Like JS, can use `"` or `'` as delimiters
- Can be multi-line by using triple-quotes: `"""` or `'''`
- Can interpolate expressions with *f-strings*:

```
food = "cheese"  
  
print(f"I love {food}")
```

- To debug an interpolation & see expression, put `=` at end:

Numbers

Very much like JavaScript!

- Separate types for integers (can be any size) or floating-point
 - In JS, there are only floating-point numbers
 - Separate type for complex numbers

- `+`, `-`, `*`, `/` (true division), `//` (integer division)
- `%` (modulo: remainder after division)
- Dividing by zero is an error (JS: is *Infinity*, except 0/0, which is *Nan*)
- Can use `+` and `*` on strings: `"cat" + "food"` or `"yay" * 3`

Lists

Like JS arrays:

- ordered
- can be heterogeneous: `[1, "apple", 13.5]`

Equality

JavaScript

- `==` loose equality
 - `7 == "7"`
- `===` strict equality
 - `7 === "7" // false`
- Objects & arrays only equal when same identity

Python

- `==` equality (strict about types)
 - `7 == "7" # False`
- Structures with same items *are* equal
 - `[1, 2, 3] == [1, 2, 3]`
- Use `is` to check obj identity
 - `[1, 2] is [1, 2] # False`

Truthiness

- In JS, these things are falsy:
 - `0, 0.0, "", undefined, null, NaN, false`
- In JS, these things are (⚠ perhaps unexpectedly) truthy:
 - `[], {}`
- In Python, these things are falsy:
 - `0, 0.0, "", None, False`
 - ⚠ `[]` (empty list), `{} (empty dictionary)`, `set()` (empty set)
- In Python, these things are truthy:
 - Any non-empty string, non-empty list/dict/set, non-0 number
 - `True`

And/Or/Not

- JS: `&&`, `||`, `!`

- Python: `and`, `or`, `not`
- Just like in JS, these “short circuit”

If

```
if grade == "A":
    print("awesome job!")

elif grade == "F":
    print("ut oh")

else:
    print("don't worry too much")
```

(parens around condition aren’t required, unlike JS)

```
if age >= 18:
    if unregistered:
        print("please register")

    else:
        print("keep voting!")

else:
    print ("Wait a bit")
```

Ternary

JavaScript

```
let msg = (age >= 18) ? "go vote!" : "go play!"
```

Python

```
msg = "go vote!" if (age >= 18) else "go play!"
```

(in both, parens are optional but often helpful)

Loops

While Loops

```
count = 10

while count > 0:
    print(count)
    count = count - 1    # or "count -= 1", but not "count--"

print("Liftoff!")
```

For Loops

Python for loops are like JS *for ... of* loops:

```
for snack in ["Peanut", "Twizzler", "Mars Bar"]:
    print("I ate a", snack)
```

To loop 5 times:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

Can also use *range()* function:

```
for num in range(5):    # makes [0, 1, 2, 3, 4]
    print(num)
```

Functions

```
def add_numbers(a, b):
    sum = a + b
    print("doing math!")
    return sum
```

Functions that don't explicitly *return* return *None*

Can pass arguments by name:

```
def order_pizza(size, flavor):
    print(f"{size} pizza with {flavor} topping")

order_pizza("large", "mushroom")

order_pizza(size="small", flavor="sausage")

# Same thing
order_pizza(flavor="sausage", size="small")
```

Can provide defaults for parameters:

```
def send_invite(name, city="SF", state="California"):
    print(f"mailing invitation to {city}, {state}")

send_invite("Jenny", "Portland", "Oregon")

send_invite("Joel")
```

Providing too many/too few arguments is an error (in JS, this is ignored / becomes *undefined*):

```
def add_three_numbers(a, b, c):
    return a + b + c

add_three_numbers(10, 20, 30)          # 60, yay!
add_three_numbers(10, 20)             # error!
add_three_numbers(10, 20, 30, 40)    # error!
```

Comments and Docstrings

- `#` : rest of line is comment (use to explain complex code)
- String as very first thing in file/function is “docstring”
 - Use to document what the function/file does
 - Shown when you ask for `help(some_function)`

```
def add_limited_numbers(a, b):
    """Add two numbers, making sure sum caps at 100."""

    sum = a + b

    # If this required explanation, comment like this

    if sum > 100:
        sum = 100

    return sum
```

Modes

Running a Source File

```
$ python3 mygame.py
You win! Your score is 10

$ # back in shell
```

- runs Python
- loads `mygame.py`
- executes the code
- returns to the terminal when done.

Running in IPython

```
$ ipython
In [1]: %run mygame.py
```

- runs `mygame.py`
- stays in IPython, variables are still set

Play in the Console

It's. The. Best. Way. to. Learn.

Good idea: open a console at the same time as your editor!

Getting Help

`dir()`

“Show me the methods and attributes of this object”

```
>>> dir([])
>>> ['__add__', '__append__', '__count__', '__extend__', '__index__', '__insert__',
    '__pop__', '__remove__', '__reverse__', '__sort__']
```

NOTE __methods__

You'll notice many objects provide a lot of methods that have names starting and ending with double-underscores (Python programmers often call these “special methods” or “dunder [for ‘double-underscore’] methods”.

These aren't methods you call directly (ie, you wouldn't ever say `mylist.__add__()`) – instead, these work behind-the-scenes to support other operations of the object.

Generally, you can ignore them when examining an object.

`help()`

“Show me help about how to use this object”

```
>>> help([])
```

«`q`» to quit that



Python Data Structures

Includes excellent, high-performance data structures as part of language.

Length of Structure

Generic `len(x)` returns length of x:

- # chars in string
- # items in list
- # items in dictionary
- # items in a set

Lists

Like JS arrays:

- Mutable, ordered sequence
- $O(n)$ to search, add, delete
 - Except when at end: $O(1)$

Making Lists

```
alpha = ['a', 'b', 'c']
```

Can use constructor function, `list()`

This will make list from iterating over argument:

```
letters = list("apple")    # ['a', 'p', 'p', 'l', 'e']
```

Membership

Can check for membership with `in`:

```
if "taco" in foods:  
    print("Yum!")  
  
if "cheese" not in foods:  
    print("Oh no!")
```

Retrieving By Index

Can retrieve/mutate item with `[n]`:

```
print(fav_foods[0])
```

```
fav_foods[0] = "taco"
```

```
fav_foods[-1] # last item
```

```
fav_foods[-3] # third from end
```

Slicing

Can retrieve list from list:

```
lst[start:stop:step]
```

- `start`: Index to begin retrieval (*default start*)
- `stop`: Index to end retrieval before (*default: end*)
- `step`: Number to step (*default: 1*)

```
alpha = ['a', 'b', 'c', 'd', 'e']

alpha[2:]      # ['c', 'd', 'e']
alpha[2:4]     # ['c', 'd']
alpha[:3]      # ['a', 'b', 'c']
alpha[::-2]    # ['a', 'c', 'e']
alpha[3:0:-1]  # ['d', 'c', 'b']
alpha[::-2]    # ['e', 'c', 'a']
```

TIP Slice assignment

You can use the slicing syntax to *splice* from a list (insert or remove things from the list in-place):

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

Core API

<code>l.append(x)</code>	Add x to end of list
<code>l.copy()</code>	Return shallow copy of list l
<code>l.count(x)</code>	Return # times x appears in l
<code>l.extend(l2)</code>	Add items of $l2$ to l
<code>l.index(x)</code>	Return (0-based) index of x in l
<code>l.insert(i, x)</code>	Insert x at position i
<code>l.pop(i)</code>	Remove & return item at i (default last)
<code>l.reverse()</code>	Reverse list (change in place)
<code>l.sort()</code>	Sort list in place

Differences From JS Arrays

Can't add new item with `[]`:

```
alpha = ['a', 'b', 'c']
alpha[3] = 'd'           # error!
```

```
alpha.append('d')        # ok!
```

Functions that mutate list return `None`, not data:

JavaScript

```
let letters = ["c", "a", "b"];
letters.sort(); // in-place; returns it
```

Python

```
letters = ["c", "a", "b"]
letters.sort() # in-place; returns None
```

Strings

Immutable sequence of characters (like JS)

Making Strings

```
msg = "Hello!"
also = 'Oh hi!'

long_msg = """This can continue on for several
lines of text"""

greet = f"Hi, {fname} {lname}"

email = f"""Dear {user},
You owe us ${owed}. Please remit."""
```

```

nums = [1, 2, 3]
str(nums)      # "[1, 2, 3]"

```

Membership / Substrings

- Can use `in` for membership (`"e" in "apple"`)
- Can slice to retrieve substring (`"apple"[1:3] == "pp"`)
 - Cannot splice; strings are immutable!
- Can iterate over, get letter-by-letter:

```

for letter in word:
    print(letter)

```

Core API

<code>s.count(t)</code>	Returns # times <code>t</code> occurs in <code>s</code>
<code>s.endswith(st)</code>	Does <code>s</code> end with string <code>t</code> ?
<code>s.find(t)</code>	Index of first occurrence of <code>t</code> in <code>s</code> (-1 for failure)
<code>s.isdigit()</code>	Is <code>s</code> entirely made up of digits?
<code>s.join(seq)</code>	Make new string of <code>seq</code> joined by <code>s</code> (<code>" ".join(nums)</code>)
<code>s.lower()</code>	Return lowercase copy of <code>s</code>
<code>s.replace(t,u,count)</code>	Replace <code>count</code> (default: all) occurrences of <code>t</code> in <code>s</code> with <code>u</code>
<code>s.split(sep)</code>	Return list of items made from splitting <code>s</code> on <code>sep</code>
<code>s.splitlines()</code>	Split <code>s</code> at newlines
<code>s.startswith(t)</code>	Does <code>s</code> start with <code>t</code> ?
<code>s.strip()</code>	Remove whitespace at start/end of <code>s</code>

Dictionaries

Mutable, ordered mapping of keys → values
 $O(1)$ runtime for adding, retrieving, deleting items
 (like JS object or *Map*)

Making Dictionaries

```
fruit_colors = {  
    "apple": "red",  
    "berry": "blue",  
    "cherry": "red",  
}
```

- Values can be *any type*
- Keys can be any **immutable** type

```
my_dict = {  
    "ok": "yes",  
    42: "all good",  
    [1,2]: 2  
} # ERR: not immutable
```

Membership & Retrieval

- `in` checks for membership of key (`"apple" in fruit_colors`)
- `[]` retrieves item by key (`fruit_colors['apple']`)
 - Cannot use dot notation, though (no `fruit_colors.apple`)
 - Failure to find is *error* (can say `.get(x, default)`)
 - If omit *default* argument, *None* is returned

Looping over Dictionaries

```
ages = {"Whiskey": 6, "Fluffy": 3, "Ezra": 7}  
  
for name in ages.keys():  
    print(name)  
  
for age in ages.values():  
    print(age)  
  
for name_and_age in ages.items():  
    print(name_and_age)
```

Can unpack `name_and_age` while looping:

```
for (name, age) in ages.items():  
    print(name, "is", age)
```

JS calls this same idea *destructuring*.

Core API

<code>d.copy()</code>	Return new copy of <i>d</i>
<code>d.get(x, default)</code>	Return value of <i>x</i> (or optional <i>default</i> if missing)
<code>d.items()</code>	Return iterable of (key, value) pairs
<code>d.keys()</code>	Return iterable of keys
<code>d.values()</code>	Return iterable of values

Sets

Unordered, unique collection of items, like JS *Set*

$O(1)$ runtime for adding, retrieving, deleting

Making Sets

Use `[]`, but with only keys, not `key: value`

```
colors = {"red", "blue", "green"}
```

Can use constructor function to make set from iterable:

```
set(pet_list)    # {"Whiskey", "Fluffy", "Ezra"}  
set("apple")     # {"a", "p", "l", "e"}
```

Any immutable thing can be put in a set

Membership

Use `in` for membership check:

```
"red" in colors
```

Core API

<code>s.add(x)</code>	Add item <i>x</i> to <i>s</i>
<code>s.copy()</code>	Make new copy of <i>s</i>
<code>s.pop()</code>	Remove & return arbitrary item from <i>s</i>
<code>s.remove(x)</code>	Remove <i>x</i> from <i>s</i>

Set Operations

```
moods = {"happy", "sad", "grumpy"}  
  
dwarfs = {"happy", "grumpy", "doc"}  
  
moods | dwarfs      # union: {"happy", "sad", "grumpy", "doc"}  
moods & dwarfs      # intersection: {"happy", "grumpy"}  
moods - dwarfs      # difference: {"sad"}  
dwarfs - moods      # difference: {"doc"}  
moods ^ dwarfs      # symmetric difference: {"sad", "doc"}
```

(These are so awesome! 😍)

Tuples

Immutable, ordered sequence
(like a list, but immutable)

Making Tuples

```
t1 = (1, 2, 3)  
  
t2 = ()           # empty tuple  
  
t3 = (1,)         # one-item tuple: note trailing comma
```

Can use constructor function to make tuple from iterable:

```
ids = [1, 12, 44]  
  
t_of_ids = tuple(ids)
```

What Are Tuples Good For?

Slightly smaller, faster than lists

Since they're immutable, they can be used as dict keys or put into sets

Files

You can open an on-disk file with `open(filepath, mode)`

- *filepath*: absolute or relative path to file
- *mode*: string of how to open file (eg, `"r"` for reading or `"w"` for writing)

This returns an file-type instance.

Reading

Line-by-line:

```
file = open("/my/file.txt")

for line in file:
    print("line =", line)

file.close()
```

All at once:

```
file = open("/my/file.txt")

text = file.read()

file.close()
```

Writing

```
file = open("/my/file.txt", "w")

file.write("This is a new line.")
file.write("So is this.")

file.close()
```

NOTE ***with* blocks**

Python has an intermediate bit of syntax called a *with block*.

For example:

```
with open("/my/file.txt", "r") as file:  
    for line in file:  
        print("line=", line)  
  
    # our file is still open here  
  
    # but it will be automagically closed here
```

Python will keep that file open as long as you're inside the with block. At the point the your code is no longer indented inside that block, it will automatically close the file you've used.

These with-blocks are used for other kinds of resources besides files; to learn more about them, you can search for *python context managers*.

Comprehensions

Python has *map()* and *filter()*, like JS

But *comprehensions* are even more flexible

Mapping Into List

Instead of this:

```
doubled = []  
  
for num in nums:  
    doubled.append(num * 2)
```

You can say this:

```
doubled = [num * 2 for num in nums]
```

Filtering Into List

Instead of this:

```
evens = []  
  
for num in nums:  
    if num % 2 == 0:  
        evens.append(num)
```

You can say this:

```
evens = [num for num in nums if num % 2 == 0]
```

Combining Filtering and Mapping

If we want only doubled, even numbers, can do in two steps:

```
evens = [num for num in nums if num % 2 == 0]
doubled_evens = [num * 2 for evens in nums]
```

Or can combine filtering and mapping into one comprehension:

```
doubled_evens = [num * 2 for num in nums if num % 2 == 0]
```

Can make lists via comprehensions from *any kind of iterable*:

```
vowels = {"a", "e", "i", "o", "u"}
word = "apple"

vowel_list = [ltr for ltr in word if ltr in vowels]
```

Can make *dictionary comprehensions* and *set comprehensions*:

```
evens_to_doubled = {n: n * 2 for n in nums if n % 2 == 0}

a_words = {w for w in words if w.startswith("a")}
```



Python Tools & Techniques

Packing / Unpacking

Unpacking

Can “unpack” iterables:

```
point = [10, 20]
x, y = point
```

```
a = 2
b = 3

b, a = (a, b)
```

Can gather rest using * before variable:

```
letters = ["a", "b", "c"]
first, *rest = letters
```

Spread

Can “spread” iterables:

```
fruits = {"apple", "berry", "cherry"}
foods = ["kale", "celery", *fruits]
```

Unpacking with functions

You can also use unpack/spread capabilities with parameters to functions. This is an intermediate idea, and doesn’t come up very often for many developers.

Positional unpacking

```
def draw(x, y, z):
    "Draw point at x/y/z coord"
    ...
coords = [10, 20, 15]
draw(*coords)
```

Keyword unpacking

** is like rest/spread for keywords:

```
def log(msg, **kwargs):
    "Log msg with flexible keyword args"

    # at this point, kwargs is a dictionary

log("Failure", item="car", model="Ford")
```

```
def greet(prefix, fname, lname):
    "Return greeting for user."

    return f"Hi, {prefix} {fname} {lname}"

user = {"fname": "Jane", "lname": "Goodall"}

greet("Dr", **user)
```

Error Handling

In general, Python raises errors in places JS returns undefined:

- provide too few/too many arguments to a function
- index a list beyond length of list
- retrieve item from dictionary that doesn't exist
- use missing attribute on an instance
- conversion failures (eg, converting “hello” to an int)
- division by zero
- *and more!*

In general, in Python: **explicit is better than implicit**

Catching Errors

```
# try to convert this to a number

try:
    age = int(data_we_received)
    print("You are", age)

except:
    print("Hey, you, that's not an age!")

# next line is run either way
```

It's risky, though, to just say *except* – that catches *all* errors!

```

data_we_received = "42"

try:
    age = int(data_we_received)
    print("You are", Age) # <-- error here!

except:
    print("Hey, you, that's not an age!")

```

Better to catch the specific error you're looking for:

```

data_we_received = "42"

try:
    age = int(data_we_received)
    print("You are", Age) # <-- error here!

except ValueError:
    print("Hey, you, that's not an age!")

```

Common Exception Types

<i>AttributeError</i>	Couldn't find attr: o.missing
<i>KeyError</i>	Couldn't find key: d["missing"]
<i>IndexError</i>	Couldn't find index: lst[99]
<i>NameError</i>	Couldn't find variable: not_spelled_right
<i>OSError</i>	Operating system error: can't read/write file, etc
<i>ValueError</i>	Incorrect value (tried to convert "hello" to int, etc)

Raising Errors

In Python, it's common for you to “raise” errors to signal problems: (*JavaScript calls this same idea “throwing”*)

```

if color not in {"red", "green", "blue"}:
    raise ValueError("Not a valid color!")

```

Error Handling Pattern

Raise exception when you know it should be an error Handle it at the point you can give good feedback

```

def bounded_avg(nums):
    """Return avg of nums (makes sure nums are 1-100)"""

    for n in nums:
        if n < 1 or n > 100:
            raise ValueError("Outside bounds of 1-100")

    return sum(nums) / len(nums)

def handle_data():
    """Process data from database"""

    ages = get_ages(from_my_db)

    try:
        avg = bounded_avg(ages)
        print("Average was", avg)

    except ValueError as exc:
        # exc is exception object -- you can examine it!
        print("Invalid age in list of ages")

```

Docstrings & Doctests

Docstrings

Docstrings are the strings at top of function or file that document it:

```

def bounded_avg(nums):
    """Return avg of nums (makes sure nums are 1-100)"""

    for n in nums:
        if n < 1 or n > 100:
            raise ValueError("Outside bounds of 1-100")

    return sum(nums) / len(nums)

```

It's incredibly good style for every function to have one!

TIP Use triple-quotes

Docstrings are just strings, so if they don't have newlines in them (are one line long), you could use `"` instead of `"""` around them. However, it's very common practice to always use triple-quotes here:

- it helps the reader at a glance start to recognize docstrings
- a docstring that starts as a single line long will often become multiline as more documentation is added to it

Doctests

Doctests are snippets of interactive demonstration in a docstring:

```

def bounded_avg(nums):
    """Return avg of nums (makes sure nums are 1-100)

    >>> bounded_avg([1, 2, 3])
    2

    >>> bounded_avg([1, 2, 101])
    Traceback (most recent call last):
    ...
    ValueError: Outside bounds of 1-100
    """

    for n in nums:
        if n < 1 or n > 100:
            raise ValueError("Outside bounds of 1-100")

    return sum(nums) / len(nums)

```

Can run this test:

```
$ python3 -m doctest -v your-file.py
```

(use the *doctest* module, verbosely showing tests found & run)

Doctests are **awesome**

Testable documentation and readable tests.

Importing

Python includes a “standard library” of dozens of useful modules.

These are not in the namespace of your script automatically.

You have to *import* them

choice(seq) is a useful function: given a sequence, it returns a random item

```

from random import choice

print("Let's play", choice(games))

```

“From *random*, pull in *choice* function as *choice*”

```

# can pull in several things from a place

from random import choice, randint

# can change the local name of it

from random import choice as pick_a_thing

pick_a_thing(games)

```

Sometimes, it helpful to pull in the *library itself*:

```
import random

# now, we have the obj `random`, with all the funcs/classes
# within available to us

random.choice(games)
```

Exporting/Importing Your Code

score.py

```
def get_high_score():
    ...
def save_high_score():
    ...
```

game.py

```
from score import get_high_score

high = get_high_score()
```

(unlike JS, nothing needed to “export”)

Installing Libraries

Python includes dozens of useful libraries

There are over 270,000 additional available ones :)

Using Pip

To install a new package:

```
$ pip3 install forex_python
# ... pip output here...

$ ipython
In [1]: from forex_python.converter import convert
In [2]: convert("USD", "GBP", 10)
7.6543
```

Virtual Environments

Normally, *pip* makes the installed library available everywhere

This is convenient, but a little messy:

- you might not need it for every project
- you might want to more explicitly keep track of which libraries a project needs
- you might want a new version of a library for one project, but not another

Python can help us by using a “virtual environment”

Creating a Virtual Environment

```
$ cd my-project-directory  
$ python3 -m venv venv
```

(“using *venv* module, make a folder, *venv*, with all the needed stuff”)

That makes the virtual environment folder — but you’re not *using it* yet!

Using Your Virtual Environment

```
$ source venv/bin/activate  
(venv) $ # <-- notice shell prompt!
```

- You only need to **create** the virtual environment once
- You need to use *source* every time you open a new terminal window

What does it mean to be “using” a virtual environment?

- It makes certain *python* is the version of Python used to create the venv
- You have access to the standard library of Python
- You **don’t** have access to globally installed pip stuff
- You get to explicitly install what you want — and it will be only for here!

Installing into Virtual Environment

- Make sure you’re using your venv — do you see it in your prompt?
- Use *pip install*, as usual
 - But now it’s downloaded & installed into that *venv* folder
 - It won’t be available/confuse global Python or other venvs — tidy!

Tracking Required Libraries

To see a list of installed libraries in a venv:

```
$ pip3 freeze  
# ... list of installed things...
```

It’s helpful to save this info in a file (typically named “requirements.txt”):

```
$ pip3 freeze > requirements.txt
```

Using Virtual Environments

- Virtual environments are large & full of stuff you didn't write yourself
- You don't want this to get into git / Github
- So, add `venv/` to your project's `.gitignore`
 - Use `git status` to make sure it's being ignored

Recreating a Virtual Environment

When using a new Python project:

```
$ git clone http://path-to-project
$ cd that-project
$ python3 -m venv venv
```

Then, as usual when working with a venv:

```
$ source venv/bin/activate
(venv) $ pip3 install -r requirements.txt
# ... pip output here ...
```

Leaving Virtual Environments

Use the `deactivate` shell command to leave the virtual environment:

```
$ source venv/bin/activate
(venv) $ deactivate
$ # ... back to regular terminal ...
```



Python Object Orientation

Intro

OO Review

```
class
    blueprint for new objects, defines attributes & methods
method
    function defined on class, can see/change attributes on instance
class method ("static method" in JS)*
    function defined on class, called on class, not individual instance
```

OO Terminology Deep Dive

Why do we use Object Orientation?
To help organize our code!
Often makes it easier to manage complex software requirements

Abstraction

OO can offer **abstraction** (*to hide implementation details when not needed*)

- Not everyone should have to understand everything

Encapsulation

OO can offer **encapsulation** (*to group functionality into logical pieces*)

- To get in a “capsule”
 - Everything related to cat data/functionality lives in *Cat*

Inheritance

OO can offer **inheritance** (*ability to call methods/get properties defined on ancestors*)

- One class subclasses another, inheriting properties and methods
 - A *ColoredTriangle* is a kind of *Triangle*; it should have sides a and b, and can use *Triangle* methods for getting the area and hypotenuse

- These are *inherited*; we only need to define what is different about *ColoredTriangle*

Polymorphism

Meh...

```
class Dog {
  bark() { return "woof!" }
}

class Cat {
  meow() { return "meow!" }
}

class Snake {
  hiss() { return "Sssssss...." }
}

// make all animals make noise
for (const pet of pets) {
  if (pet instanceof Dog) pet.bark();
  if (pet instanceof Cat) pet.meow();
  if (pet instanceof Snake) pet.hiss();
}
```

OO can offer **polymorphism** (*making classes interchangeable when similar*)

- The ability to make similar things work similarly
 - We could have other kinds of animals with same API
 - eg, dogs and cats could both have a *speak()* method, even though it works differently ("Meow" vs "Woof")

Yay, polymorphism!

```
class Dog {
  speak() { return "woof!" }
}

class Cat {
  speak() { return "meow!" }
}

class Snake {
  speak() { return "Sssssss...." }
}

// make all animals make noise --- MUCH BETTER
for (const pet of pets) {
  pet.speak();
}
```

Instances

Like in JS, you make an instance by calling the class:

```
from collections import Counter

# make instance of a counter
counts = Counter("hello world")

type(counts)      # 'collections.Counter'
isinstance(counts, Counter)      # True
```

Get/set attributes or find methods with `.` (like JS):

```
# get most common letter
counts.most_common(1)
```

JavaScript:

- get/set attribute of object: `o.name` or `o['name']`
- call method: `o.method()` or `o['method']()`

Python:

- get/set attribute of object: `o.name`
- call method: `o.method()`
- retrieve value from dictionary: `o['my-key']`
 - not the same thing!

What Can I Do With This Object?

`help(obj)`
Show help about object and methods

`dir(obj)`
List methods/attributes of object

Classes

Making classes is similar to JS:

```

class Triangle:
    """Right triangle."""

    def __init__(self, a, b):
        """Create triangle from a and b sides."""
        self.a = a
        self.b = b

    def get_hypotenuse(self):
        """Get hypotenuse (length of 3rd side)."""
        return math.sqrt(self.a ** 2 + self.b ** 2)

    def get_area(self):
        """Get area of triangle."""
        return (self.a * self.b) / 2

    def describe(self):
        """Return description of area."""
        return f"My area is {self.get_area()}"

```

Self

self is similar to *this*

- *this* is a bit magical: it automatically gets created
- *self* is explicit: you must list it as the first argument of methods
 - It's just a normal variable, otherwise

Inheritance

Like in JS, classes can subclass other objects:

```

class ColoredTriangle(Triangle):
    """Triangle that has a color."""

    def __init__(self, a, b, color):
        # get parent class [super()], call its __init__()
        super().__init__(a, b)

        self.color = color

    def describe(self):
        """Return description of area and color."""
        return super().describe() + f" I am {self.color}"

```

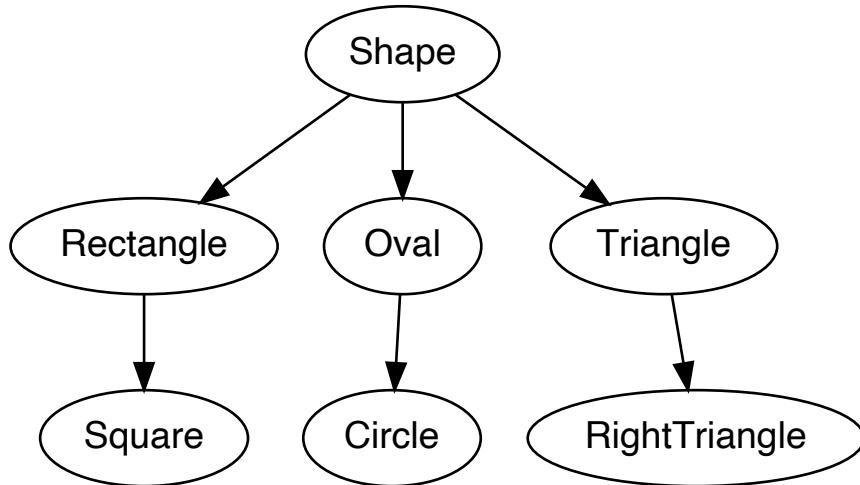
Super

Like in JS, *super* finds parent class:

- JS: `super` is parent, `super(...)` calls parent constructor function
- Python: `super()` is parent, `super().__init__(...)` is parent initializer.

Multi-Level Inheritance

Like in JS, you can have multiple levels of inheritance



△ A hierarchy of geometric shapes

Documenting Classes

As always, good style to have comment explaining purpose of class & methods:

```
class Triangle:  
    """Right triangle."""  
  
    def __init__(self, a, b):  
        """Create triangle from a and b sides."""  
        self.a = a  
        self.b = b  
  
    def get_hypotenuse(self):  
        """Get hypotenuse (length of 3rd side)."""  
        return math.sqrt(self.a ** 2 + self.b ** 2)  
  
    def get_area(self):  
        """Get area of triangle."""  
        return (self.a * self.b) / 2
```

Documenting Instance

When you print an instance/examine in Python shell, often not helpful:

```
>>> tri = Triangle(3, 4)  
  
>>> tri  
<__main__.Triangle object at 0x1012a6358>
```

Would be nicer to see values for *a* and *b*

We can do this by making a `__repr__` (*representation*) method:

```
class Triangle:  
    """Right triangle."""  
  
    def __init__(self, a, b):  
        """Create triangle from a and b sides."""  
        self.a = a  
        self.b = b  
  
    def __repr__(self):  
        return f"<Triangle a={self.a} b={self.b}>"  
  
    def get_hypotenuse(self):  
        """Get hypotenuse (length of 3rd side)."""  
        return math.sqrt(self.a ** 2 + self.b ** 2)  
  
    def get_area(self):  
        """Get area of triangle."""  
        return (self.a * self.b) / 2
```

```
>>> tri = Triangle(3, 4)  
>>> tri  
<Triangle a=3 b=4>
```

Flask Intro

Goals

- Describe the purpose and responsibilities of a web framework
- Build small web applications using Python and Flask
- Set environmental variables for local Flask development
- Handle GET and POST requests with Flask
- Extract data from different parts of the URL with Flask

Web Frameworks

```
(venv) $ FLASK_DEBUG=True flask run
* Environment: development
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 160-080-703
```

What is a Web Server?

A program that's running on a machine and waiting for a web request.

Note: A web server is a technology that can process requests and issue responses via HTTP, a protocol used to distribute information on the world wide web. Though it's also used to refer to computer systems and even internet "appliances", we'll use web server to refer to the software running on a machine that's waiting to respond to HTTP requests.



△ Browser makes request to server

△ Server responds w/headers & HTML

The ability to start a server in listening for requests, and then issue responses:

```
GET /hello      (http://server/hello)
```

↓

```
<html>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```

Note: To keep code samples short in the presentation, we're eliding some less-important HTML markup. The shortest valid HTML skeleton in modern HTML would actually be:

```
<!doctype html>
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```

Flask is a Web Framework

- Set of functions and classes that help you define:
 - Which requests to respond to
 - *http://server/about-us*
 - *http://server/post/1*
 - How to respond to requests
 - Shows an “About Us” page
 - Show the 1st blog post
- Like a library, but bigger and more opinionated
- Usage is similar to the Python Standard Library.

Using the Python Standard Library

```
from random import choice, randint
```

Using Flask

```
from flask import Flask, request
```

What Do Web Applications Need To Do?

- handle web requests
- produce dynamic HTML
- handle forms
- handle cookies
- connect to databases
- provide user log-in/log-out
- cache pages for performance
- & more!

Flask Apps

Installing Flask

```
$ python3 -m venv venv
$ source venv/bin/activate

(venv)$ pip3 install flask
*... lots of stuff ...
Successfully installed flask Werkzeug Jinja2 ...
Cleaning up...
```

Making An App

Need to create a “flask application”:

```
from flask import Flask

app = Flask(__name__)
```

When we create a Flask application, it needs to know what module to scan for things like routes (covered later)—so the `__name__` is *required* and should always be written like that.

Running Flask App

```
(venv) $ flask run
```

(««Control»-«C»» to quit)

NOTE Name your application file `app.py`

It's common convention to name your application file `app.py`. If, for some reason, you call it something else, you'll need to tell Flask that when you try to start it up by using an *environmental variable*:

```
(venv) $ FLASK_APP=other.py flask run
```

TIP Make sure you're starting up in *development mode*

While building your application, you want to make sure Flask is running in *development mode*. This is a special mode for the server. It gives you:

- Much better error messages
- Automatically re-loads server when code changes on disk

Both of these are very helpful when developing (and very bad for working on a live, production server, but we'll talk about this when we teach you to deploy completed applications to a real server).

Our installation instructions should set this automatically for you. However, when you start Flask, if it says you're in *production mode*, you can fix this by starting Flask up like this:

```
(venv) $ FLASK_DEBUG=True flask run
```

You can make that the default for your current shell session by setting that environmental variable for the length of the shell session:

```
(venv) $ export FLASK_DEBUG=True
```

Then you can just use `flask run`.

Adding Routes

Making Responses

- A function that returns web response is called a **view function**
 - Response is a **string**
 - Usually, a **string** of HTML
- So, our function returns an HTML string:

```
@app.get('/hello')
def say_hello():
    """Return simple "Hello" Greeting."""

    html = "<html><body><h1>Hello</h1></body></html>"
    return html
```

Handling Requests

On requesting `http://localhost:5000/hello` in browser, function is called:

```
@app.get('/hello')
def say_hello():
    """Return simple "Hello" Greeting."""

    html = "<html><body><h1>Hello</h1></body></html>"
    return html
```

- Flask lets you **route** a URL to a function

- `@app.get("/hello")` is a Python *decorator*
 - `"/hello"` in the decorator maps directly to the URL the user requested

Now we can get to this at <http://localhost:5000/hello> <http://localhost:5000/hello>

Serving at the Root (Homepage)

```
@app.get('/')
def index():
    """Show homepage"""

    return """
        <html>
            <body>
                <h1>I am the landing page</h1>
            </body>
        </html>
    """
```

This function will get called if the user requests <http://localhost:5000/>.

Now we can reach this page at <http://localhost:5000> <http://localhost:5000>

What Routes Return

Routes should return strings!

GET and POST

Requests

Flask provides an object, *request*, to represent web requests

```
from flask import request
```

Handling Query Arguments

For a url like `/search?term=fun`

```
@app.get('/search')
def search():
    """Handle GET requests like /search?term=fun"""

    term = request.args["term"]
    return f"<h1>Searching for {term}</h1>"
```

`request.args` is a dict-like object of query parameters.

Handling POST Requests

To accept POST requests, must specify that:

```
@app.post("/my/route")
def handle_post_to_my_route():
    ...
```

Example POST Request

```
@app.get("/add-comment")
def add_comment_form():
    """Show comment-add form."""

    return """
        <form method="POST">
            <input name="comment">
            <button>Submit</button>
        </form>
    """
```

```
@app.post("/add-comment")
def add_comment():
    """Handle adding comment."""

    comment = request.form["comment"]

    # TODO: save into database

    return f'<b>Got "{comment}"</b>'
```

`request.form` is a dict-like object of POST parameters.

NOTE Polymorphism

This is an excellent example of the coding concept of *polymorphism*: `request.form` isn't exactly a Python dictionary, but it's a specialized class that, in almost every way, *act like* a Python dictionary in terms of how users can use it.

Variables in a URL

Motivation

- Want user info pages for each user:
 - <http://localhost:5000/users/whiskey> <<http://localhost:5000/users/whiskey>>
 - <http://localhost:5000/users/spike> <<http://localhost:5000/users/spike>>
 - We don't want every possible username as a separate route
- Want to show blog posts (read from database) by id:
 - <http://localhost:5000/posts/1> <<http://localhost:5000/posts/1>>
 - <http://localhost:5000/posts/2> <<http://localhost:5000/posts/2>>

Variables in a URL

Argument capture in Flask:

```
USERS = {  
    "whiskey": "Whiskey The Dog",  
    "spike": "Spike The Porcupine",  
}  
  
@app.get('/users/<username>')  
def show_user_profile(username):  
    """Show user profile for user."""  
  
    name = USERS[username]  
    return f"<h1>Profile for {name}</h1>"
```

- <variable_name> in `@app.route`
- View function must have same *variable_name* as parameter

Can also match and convert *int* URL parameters:

```
POSTS = {  
    1: "Flask is pretty cool",  
    2: "Python is neat-o",  
}  
  
@app.get('/posts/<int:post_id>')  
def show_post(post_id):  
    """Show post with given integer id."""  
  
    print("post_id is a ", type(post_id))  
  
    post = POSTS[post_id]  
  
    return f"<h1>Post # {post_id} </h1><p>{post}</p>"
```

- <int:variable_name> in `@app.route`
- Converts to integer when calling function

Can have more than one:

```
@app.get('/products/<category>/<int:product_id>')  
def product_detail(category, product_id):  
    """Show detail page for product."""  
  
    ...
```

Query Params vs URL Params

`http://localhost:5000/shop/spinning-top?color=red` <`http://localhost:5000/shop/spinning-top?color=red`>

```

@app.get("/shop/<toy>")
def toy_detail(toy):
    """Show detail about a toy."""

    # Get color from req.args, falling back to None
    color = request.args.get("color")

    return f"<h1>{toy}</h1>Color: {color}"

```

URL Parameter	Query Parameter
/shop/<toy>	/shop?toy=elmo
Feels like “subject of page”	Feels like “extra info about page”
	Often used when coming from form

Looking Ahead

- HTML templates
- Handling cookies
- APIs and Flask
- Using databases with Flask
- Auto-generating forms
- Handling users and log in

Flask Documentation

The Flask documentation (<http://flask.pocoo.org/> <<http://flask.pocoo.org/>>)



Flask 2: Templates & Jinja

Goals

- Explain what HTML templates are, and why they are useful
- Use Jinja to create HTML templates for our Flask applications
- Debug Flask applications more easily by installing *Flask Debug Toolbar*
- Serve static files (CSS, JS, etc) with Flask

Review

Views

Views are functions that return a **string** (a string of HTML)

Routes

Routes define the URL that will run a view function.

They are declared by using *decorators*.

A route and view function:

```
@app.get('/simple')
def show_simple_form():
    """Show greeting form."""

    return """
<!DOCTYPE html>
<html>
    <head>
        <title>Hi There!</title>
    </head>
    <body>
        <h1>Hi There!</h1>
        <form action="/greet">
            What's your name? <input name="person">
            <button>Go!</button>
        </form>
    </body>
</html>
"""
```

This is kind of messy to read through (and we don't get nice things like color-highlighting in editors).
Much better to keep HTML in a separate file.

Templates

How Can Templates Help?

- Keep HTML in an HTML file
- Allows HTML to have dynamic parts
 - Can use variables passed from your views
 - For loops, if/else statements
- Can inherit from other templates to minimize repetition

Jinja

Jinja is a popular template system for Python, used by Flask.

There are many template systems for Python. Jinja is a particularly popular one. Django has its own template system, which served as an inspiration for Jinja.

Templates Directory

Your templates directory lives under your project directory. Flask knows to look for them there.

```
my-project-directory/
  venv/
  app.py
  templates/
    hello.html
```

Our Template

demo/templates/hello.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>This is the hello page</title>
</head>
<body>
  <h1>HELLO!</h1>
</body>
</html>
```

You can't view this at <http://localhost:5000/hello.html> <<http://localhost:5000/hello.html>>

hello.html is the template filename – we still have to have a route.

Rendering a Template

```
@app.get('/')
def index():
    """Return homepage."""

    html = render_template("hello.html")
    print("html is: ", html)

    return html
```

Will find `hello.html` in `templates/` automatically.

Flask Debug Toolbar

Ultra-useful add-on for Flask that shows, in browser, details about app.

Install add-on product:

```
(venv) $ pip3 install flask-debugtoolbar
```

Add the following to your Flask `app.py`:

```
from flask import Flask, request, render_template
from flask_debugtoolbar import DebugToolbarExtension

app = Flask(__name__)
app.config['SECRET_KEY'] = "oh-so-secret"

debug = DebugToolbarExtension(app)

... # rest of file continues
```

NOTE SECRET_KEY

For now, that secret key doesn't really have to be something secret (it's fine to check this file into your GitHub, and you can use any string for the SECRET_KEY).

Later, when we talk about security & deployment, we'll talk about when and how to actually keep this secret.

Using The Toolbar

Request Vars

Explore what Flask got in request from browser

HTTP Headers

Can be useful to explore all headers your browser sent

Templates

What templates were used, and what was passed to them?

Route List

What are all routes your app defines?

Dynamic Templates

Example normal template string in JavaScript ...

```
const lucky_num = 12;
`Lucky num is ${lucky_num}`; // "Lucky num is 12"
```

To do the same kind of thing in Jinja, use `{{ lucky_num }}`

templates/lucky.html

```
<p>Lucky number: {{ lucky_num }}</p>
```

app.py

```
@app.get("/lucky")
def show_lucky_num():
    "Show lucky number."

    num = randint(1, 100)

    return render_template(
        "lucky.html",
        lucky_num=num)
```

Example: Greeting

Let's make a form that gathers a user's name.

On form submission, it should use that name & compliment the user.

Our Form

demo/templates/form.html

```
<!DOCTYPE html>
<html lang="en">
<body>
    <h1>Hi There!</h1>
    <form action="/greet">
        <p>What's your name? <input name="person"></p>
        <button>Go!</button>
    </form>
</body>
</html>
```

Our Template

demo/templates/compliment.html

```
<!DOCTYPE html>
<html lang="en">
<body>
    <p>Hi {{ name }}! You're so {{ compliment }}!</p>
</body>
</html>
```

Our Route

```
@app.get('/greet')
def offer_greeting():
    """Give player compliment."""

    player = request.args["person"]
    nice_thing = choice(COMPLIMENTS)

    return render_template(
        "compliment.html",
        name=player,
        compliment=nice_thing)
```

Example 2: Better Greeting!

Let's improve this:

- We'll ask the user if they want compliments & only show if so
- We'll show a list of 3 random compliments, like this:

```
You're so:
<ul>
  <li>clever</li>
  <li>tenacious</li>
  <li>smart</li>
</ul>
```

Our Form

demo/templates/form-2.html

```
<!DOCTYPE html>
<html lang="en">
<body>
  <h1>Better Hi There!</h1>
  <form action="/greet-2">
    <p>What's your name? <input name="person"></p>
    <p>Want compliments?
      <input type="checkbox" name="wants_compliments">
    </p>
    <button>Go!</button>
  </form>
</body>
</html>
```

Our Route

```
@app.get('/greet-2')
def offer_better_greeting():
    """Give player optional compliments."""

    player = request.args["person"]

    # if they didn't tick box, `wants_compliments` won't be
    # in query args -- so let's use safe `.get()` method of
    # dict-like things
    wants = request.args.get("wants_compliments")

    nice_things = sample(COMPLIMENTS, 3) if wants else []

    return render_template(
        "compliments.html",
        compliments=nice_things,
        name=player)
```

Conditionals in Jinja

```
compliments = ["c1", "c2", "c3"]
# or
compliments = []
```

```
{% if CONDITION_EXPR %} ... {% endif %}

{% if compliments %}
    You're so:
    ...
{% endif %}
```

Loops in Jinja

```
compliments = ["c1", "c2", "c3"]
```

```
{% for VAR in ITERABLE %} ... {% endfor %}

<ul>
    {% for compliment in compliments %}
        <li>{{ compliment }}</li>
    {% endfor %}
</ul>
```

Our Template

`demo/templates/compliments.html`

```
<!DOCTYPE html>
<html lang="en">
<body>
<h1>Hi {{ name }}!</h1>
{% if compliments %}
  <p>You're so:</p>
  <ul>
    {% for compliment in compliments %}
      <li>{{ compliment }}</li>
    {% endfor %}
  </ul>
{% endif %}
</body>
</html>
```

Template Inheritance

Motivation

Different pages on the same site often have the same headers, footers, and overall page structure.

Repetition is Boring

Your templates have many things in common

Example site with common header/footer

```
<html>
<head>
  <title> TITLE GOES HERE </title>
  <link rel="stylesheet" href="/static/css/styles.css">
</head>
<body>
  <header>Fluffy Co</header>
  BODY CONTENT GOES HERE
  <footer>Copyright by Fluffy</footer>
</body>
</html>
```

- Make a `base.html` that will hold all the repetitive stuff
- “Extend” that base template in your other pages
- Substitute blocks in your extended pages

If you want the same stylesheet everywhere, you have to remember to include it in every template. If you forget in one template, that page won’t have your custom css that you spent so much time getting right. The same goes for scripts. If you want jquery everywhere, do you really want to have to remember to include it in the head in every template.

Sample Base Template

```
{% block BLOCKNAME %} ... {% endblock %}
```

templates/base.html

```
<html lang="en">
<head>
    <title>{% block title %}TITLE #HERE{% endblock %}</title>
    <link rel="stylesheet" href="/static/css/styles.css">
</head>
<body>
    <header>Fluffy Co</header>
    {% block content %}BODY CONTENT GOES HERE{% endblock %}
    <footer>Copyright by Fluffy</footer>
</body>
</html>
```

Page Using Template

```
{% block BLOCKNAME %} ... {% endblock %}
```

templates/my-page.html

```
{% extends 'base.html' %}

{% block title %}My Page{% endblock %}

{% block content %}

    <h2>I'm a header!</h2>
    <p>I'm a paragraph!</p>

{% endblock %}
```

Result of rendering template

```
<html lang="en">
<head>
    <title>My Page</title>
    <link rel="stylesheet" href="/static/css/styles.css">
</head>
<body>
    <header>Fluffy Co</header>
    <h2>I'm a header!</h2>
    <p>I'm a paragraph!</p>
    <footer>Copyright by Fluffy</footer>
</body>
</html>
```

Where Other Project Files Go

Do I Need Routes for CSS (or JS, etc)?

```
@app.get("my-css.css")
def my_css():
    return """
        b { color: red }
        ...
    """
```

No! That would be tedious – plus, everyone gets the *same* CSS

Static Files: CSS and JS

In *static/* directory:

```
my-project-directory/
  venv/
  app.py
  templates/
    hello.html
  static/
    my-css.css
    my-script.js
    image.gif
```

Find files like:

```
<link rel="stylesheet" href="/static/my-css.css">
```

The static directory is where you put files that don't change, unlike templates, which are parsed. The static directory can be divided into the types of files that it contains: js for javascript, css for css files, img for images, etc., but that isn't necessary.



Flask Tools

Goals

- Explore other common features of web frameworks like Flask, including:
 - Redirecting
 - Flash messaging
 - Returning JSON data
- Debug Flask errors from the error page
- Set breakpoints in Python code with **pdb**

Redirecting

What is an HTTP redirect?

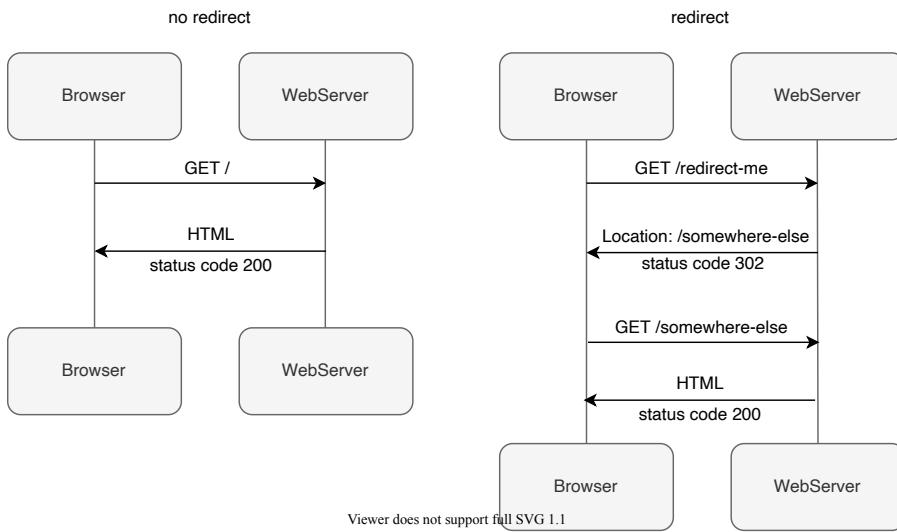
- An HTTP response
- The status code is a “redirect code” (often, 302)
- It contains a URL for browser to re-request
- Typically, for ancient browsers, contains HTML with a link

```
$ curl -v http://localhost:5000/redirect-me

< HTTP/1.0 302 FOUND
< Content-Type: text/html; charset=utf-8
< Location: http://localhost:5002/somewhere-else <http://localhost:5002/somewhere-else>

<h1>Redirecting...</h1>
<p>You should be redirected automatically to target URL:
<a href="/somewhere-else">/somewhere-else</a>.
If not click the link.</p>
```

Your browser won't typically show you this page — it makes the re-request so fast you don't even notice it happened!



Flask Debug Toolbar & Redirects

The Debug Toolbar makes redirects explicit

This is very useful for debugging!

If you don't want this, you can turn it off:

```
app.config['DEBUG_TB_INTERCEPT_REDIRECTS'] = False
```

Common Pattern: Redirect POST to GET

- POST requests are often from a form
 - And change data on the server
- If you return HTML from a POST request, the browser shows it fine
 - But if the user hits “Refresh”, they get weird “ok to resubmit” dialog
- Better strategy:
 - Do the work you want inside your POST route
 - Then *redirect* to a page that shows the confirmation

```
demo/app.py
```

```
@app.post("/post-example")
def post_example():
    """An example of good POST handling."""

    isbn = request.form["isbn"]

    if isbn in VALID_ISBNs:
        print(f"\n\nBuying Book: {isbn}\n\n")

        # flash message: we'll talk about this soon
        # flash(f"Book {isbn} bought!")

        return redirect("/thanks")
    else:
        error = "Invalid isbn"
        return render_template("post-form.html", isbn=isbn, error=error)

@app.get("/thanks")
def say_thanks():
    """Thank user for buying a book."""

    return render_template("thanks.html")
```

Message Flashing

Often you want to provide feedback at “the next page user sees”

This is most common when you will redirect

```
from flask import flash

@app.get("/your/route")
def your_route():
    """Some route that redirects."""

    flash("Message for user!")
    return redirect("/somewhere/else")
```

template used by /somewhere/else

```
{% for msg in get_flashed_messages() %}
  <p>{{ msg }}</p>
{% endfor %}
```

Returning JSON

JSON is just a string – so you don’t need to do anything special

```
@app.get("/some/route")
def some_route():
    """Route that returns JSON."""

    return '{"name": "Whiskey", "cute": "hella"}'
```

Two considerations:

- It's finicky to hand-write JSON and get it right
- It's sometimes helpful to send header to browser that "this is JSON"
 - Some AJAX libraries are better than others in guessing in absence of this

demo/app.py

```
@app.get("/example-json")
def example_json_route():
    """Return some JSON."""

    info = {"name": "Whiskey", "cute": "Hella"}
    return jsonify(info)

    # Alternate syntax
    # return jsonify(name="Whiskey", cute="Hella")
```

Flask Debugging

Strategies:

- as always *print()* (*appears in terminal*)
- Flask Debug Toolbar
- Get an error? You can debug on the error page!

Debugging Errors

Click the black "Terminal" symbol in a traceback

You'll need to enter "PIN code" (printed out to terminal at start)

That will give you a Python interpreter right there!

You can examine variables, try out code, etc.

Python Debugger

Python includes a built-in debugger, *pdb*

To add a breakpoint to your code:

```
def my_function():
    ...
    breakpoint()
    ...
```

This is like *debugger* in JavaScript.

When you hit that, Python will stop so you can debug this.

Don't forget the parentheses!

Debugger Basics

Key	Command
?	Get help
l	List code where I am
p	Print this expression
pp	Pretty print this expression
n	Go to next line (step over)
s	Step into function call
c	Continue to next breakpoint or end
w	Print "frame" (where am I?)



Cookies & Sessions

Goals

- Define what it means for HTTP to be stateless
- Compare different strategies for persisting state across requests
- Explain what a cookie is, and how client-server cookie communication works
- Compare cookies and sessions
- Implement session functionality with Flask

Motivation: Saving State

HTTP is what's called a "stateless" protocol.

On its own, it remembers nothing.

It's like a goldfish. Every time it circles around, what it sees is brand new.

- Passing info in a query param / POST form hidden field
 - `/step-zero?fav-color=blue → /step-one?fav-color=blue → ...`
- Keeping info in URL path
 - `/fav-color/blue/step-zero → /fav-color/blue/step-one → ...`
- Using JS `localStorage API` <<https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>>
 - Nice, but only JS can access this — you can't get data on server
 - Useful for single-page applications or heavily AJAX-driven apps
- Using cookies / sessions

Cookies

Flask's `session` is powered by cookies; let's start there

Cookies Save “State”

Cookies are a way to store small bits of info on client (browser)

What is a Cookie?

Cookies are **name/string-value pair** stored by the **client** (browser).

The server tells client to store these.

The client sends cookies to the server with each request.

Site	Cookie Name	Value
rithmschool.com	number_visits	"10"
rithmschool.com	customer_type	"Enterprise"
localhost:5000	favorite_food	"taco"

Cookies, A Conversation

- *Browser*: I'd like to get the resource */upcoming-events*.
- *Server*: Here's some HTML. Also, please remember this piece of information: *favorite_food* is `"taco"`.
- *Browser* (stores this somewhere on the computer)
- *Browser*: I'd like to get the resource */event-detail*. Also, you told me to remind you that *favorite_food* is `"taco"`.
- *Server*: Here's the HTML for that.
- *Browser*: I'd like to get the resource */calendar.jpg*. Also, you told me to remind you that *favorite_food* is `"taco"`.
- ...

Seeing Cookies in Chrome

Dev Tools → Application → Storage → Cookies

Settings Cookies in Flask

demo/app.py

```
@app.get('/handle-form-cookie')
def handle_form():
    """Return form response; include cookie for browser."""

    fav_color = request.args['fav_color']

    # Get HTML to send back. Normally, we'd return this, but
    # we need to do in pieces, so we can add a cookie first
    html = render_template(
        "response-cookie.html",
        fav_color=fav_color)

    # In order to set a cookie from Flask, we need to deal
    # with the response a bit more directly than usual.
    # First, let's make a response obj from that HTML
    resp = make_response(html)

    # Let's add a cookie to our response. (There are lots of
    # other options here--see the Flask docs for how to set
    # cookie expiration, domain it should apply to, or path)
    resp.set_cookie('fav_color', fav_color)

    return resp
```

Reading Cookies in Flask

demo/app.py

```
@app.get('/later-cookie')
def later():
    """An example page that can use that cookie."""

    fav_color = request.cookies.get('fav_color', '<unset>')

    return render_template(
        "later-cookie.html",
        fav_color=fav_color)
```

Cookie Options

- **Expiration:** how long should the browser remember this?
 - Can be set to a time; default is “as long as web browser is running” (*session cookie*)
- **Domain:** which domains should this cookie be sent to?
 - Send only to *books.site.com* or everything at *site.com*?
- **HttpOnly** - HTTP-only cookies aren’t accessible via any kind of JavaScript
 - Useful for cookies that contain server-side information and don’t need to be available to JavaScript.

Site	Cookie	Expiration	Domain	Value
www.rithmschool.com	num_visits	(browser)	*.rithmschool.com	“10”
shop.rithmschool.com	cus_type	2015-12-31	shop.rithmschool.com	“Enterprise”
localhost:5000	fav_color	(browser)	localhost:5000	“blue”

Comparison of Types of Browser Storage

- LocalStorage
 - Doesn’t have expiration date; only through JS or clearing browser cache
 - Domain specific
 - Storage limit is much larger than a cookie
- SessionStorage
 - Stores data only until browser window/tab is closed
 - Storage limit is much larger than a cookie
- Cookie
 - Sent from browser to server for *every request* to the domain
 - Set (*usually*) from server; can be read by browser or server
 - Can prevent JS from accessing by setting *httpOnly* flag on cookie

A Visual Display

△ Credit: <https://stackoverflow.com/questions/19867599/> <<https://stackoverflow.com/questions/19867599/>>

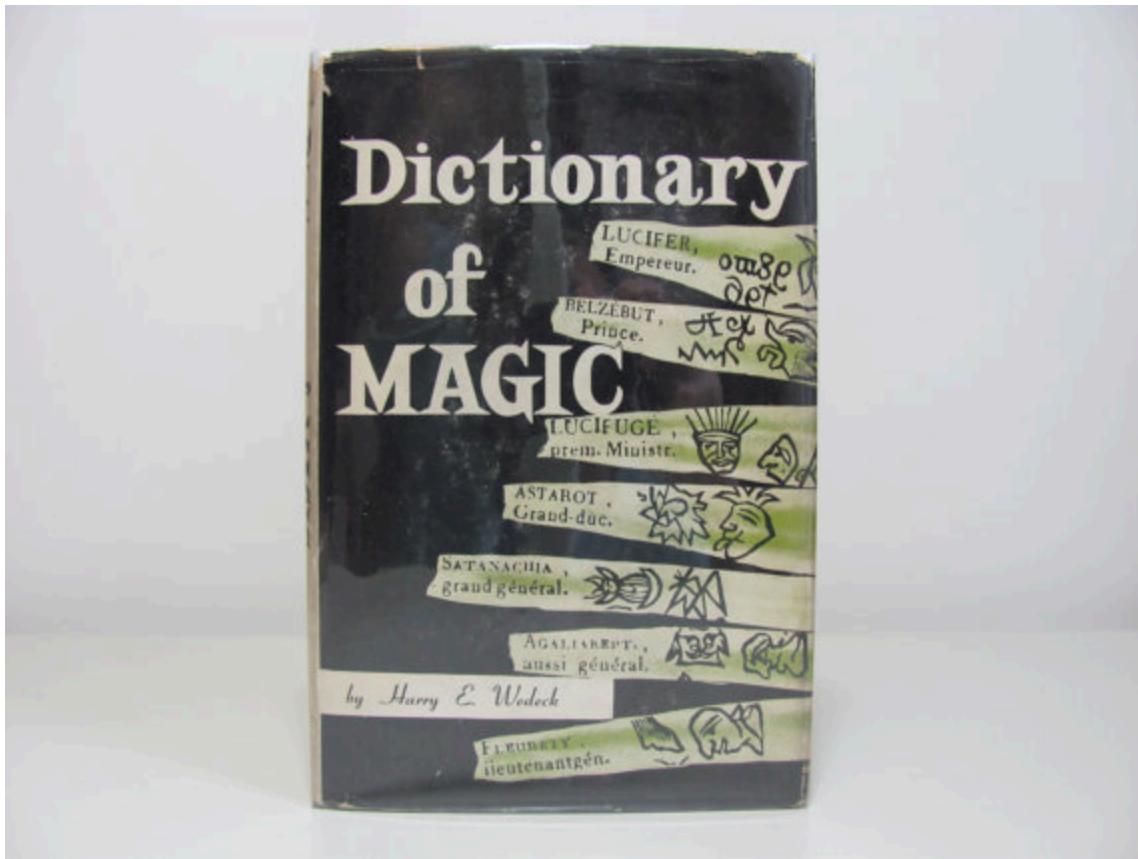
Sessions

Cookies Can Be Tricky

- Cookies are just strings
- Cookies are limited by how much information you can store
- Cookies are a bit low-level in how you use them

Sessions

Flask sessions are a “magic dictionary”



- Contain info for the current browser
- Preserve type (lists stay lists, etc)
- Are *signed*, so users can't modify data

Using Session in Flask

- Import `session` from `flask`
- Set a `secret_key`

```
from flask import Flask, session

app = Flask(__name__)
app.config["SECRET_KEY"] = "SHHHHHHHHHHH SEEKRIT"
```

Now, in routes, you can treat `session` as a dictionary:

```
@app.get('/some-route')
def some_route():
    """Set fav_number in session."""

    session['fav_number'] = 42
    return "Ok, I put that in the session."
```

To get things out, treat it like a dictionary:

```

from flask import session

@app.get('/my-route')
def my_route():
    """Return information using fav_number from session."""
    return f"Favorite number is {session['fav_number']}"

```

It will stay the same kind of data (in this example, an integer)

You also have direct access to `session` automatically in Jinja templates:

```
Your favorite number is {{ session['fav_number'] }}
```

How Do Sessions Work?

- Different web frameworks handle this differently
- In Flask, the sessions are stored in the browser as a cookie
 - `session = "eyJjYXJ0IjLDIsMiwyLDJdfQ.CP0ryA2EMSZdE"`
 - They're *serialized* and *signed*
 - So users could see, but can't change their actual session data—only Flask can

Advanced details: Flask by default uses the *Werkzeug* provided “secure cookie” as session system. It works by serializing the session data, compressing it and base64 encoding it.

Are “Sessions” Related to “Session Cookies”?

Not directly, no.

They both just use the term “session” but to mean something different.

By default: Flask sessions use browser-lifetime cookies (“session cookies”). So a Flask session lasts as long as your browser window.

Yes, you can change that (read the Flask docs!)

This distinction isn't too important right now, but the terminology sometimes comes up in interviews, so be sure to review this material!

NOTE Server-Side Sessions

Some web frameworks store session data on the server instead.

- Often, in a relational database
- Sends a cookie with *session key*, which tells server how to get the real data
- Useful when you have lots of session data, or for complex setups
- Flask can do this with the add-on [Flask-Session/](http://pythonhosted.org/Flask-Session/)

Which Should I Use? Cookies or Sessions?

Generally, sessions.

It's important to know how cookies work, but if your framework provides sessions (as Flask does), they're easier to work with.



Testing

Goals

- Discuss the benefits of writing tests
- Compare unit, integration, and end-to-end tests
- Compare different ways to write tests in Python:
 - *assert* statements
 - doctests
 - *unittest* module
- Write integration tests for our Flask apps
- Use tests to inform how we write application code

Whys and Wherefores

Can't I Just Test Code Myself?

Yes. You probably do so now.

Testing is

- #1 thing employers ask us about
- Something ALL engineers do
- Automating the boring stuff
- Fascinating and highly skilled art
- Peace of mind: develop with confidence!

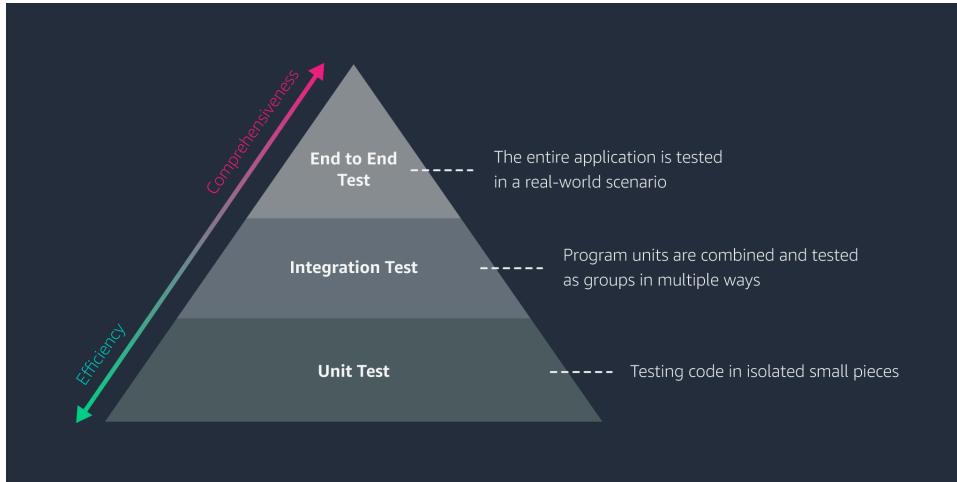
Automated Tests Are Particularly Good For

- Testing things that “should work”
- Testing the edge cases (anticipate the unexpected)
- New things don’t break things that were working (*regression*)

Kinds of Tests

Testing a Dryer

- **Unit Test:** does this individual component work?
- **Integration Tests:** do the parts work together?
- **End-to-end Test:** wet clothes → dry clothes?



Some people call and include other notions of testing levels, like “acceptance tests”, “system tests”, and others.

Unit Tests

- Test one “unit” of functionality
 - Typically, one function or method
- Don’t test integration of components
 - Don’t test framework itself (*e.g., Flask*)
- Promote modular code
 - Write code with testing in mind

You Can Do This By Hand

```
def adder(x, y):
    """Add two numbers together."""
    return x + y

assert adder(1, 1) == 2, '1 + 1 is not 2'
```

- `assert` raises `AssertionError` if expression is False
- Can provide optional exception message
- Code exits as soon as an exception is raised

DocTests

DocTests are awesome!

“Testable documentation” “Documented testing”

doctest module in Python standard library

Our Adder

```
def adder(x, y):
    """Add two numbers together."""
    return x + y
```

Let's try it out!

```
>>> from arithmetic import adder
>>> adder(1, 1)
2
>>> adder(-1, 1)
0
```

```
def adder(x, y):
    """Adds two numbers together.

    >>> adder(1, 1)
    2

    >>> adder(-1, 1)
    0
    """

    return x + y
```

Running DocTests

```
$ python -m doctest arithmetic.py
$ (nothing output for success)
```

Everything worked!

Running Verbosely

```
$ python -m doctest -v arithmetic.py
Trying:
    adder(1, 1)
Expecting:
    2
ok
Trying:
    adder(-1, 1)
Expecting:
    0
ok
1 items had no tests:
    arithmetic
1 items passed all tests:
2 tests in arithmetic.adder
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Let's Make it Fail

```
def adder(x, y):
    """Adds two numbers together.

    >>> adder(1, 1)
    2

    >>> adder(-1, 1)
    0
    """
    return x + y + 1 # this is wrong
```

Running DocTests

```
$ python -m doctest arithmetic.py
*****
File "arithmetic.py", line 10, in arithmetic.adder
Failed example:
    adder(1, 1)
Expected:
    2
Got:
    3
*****
File "arithmetic.py", line 15, in arithmetic.adder
Failed example:
    adder(-1, 1)
Expected:
    0
Got:
    1
*****
1 items had failures:
  2 of   2 in arithmetic.adder
***Test Failed*** 2 failures.
```

TIP DocTest options

You can also add special comments in your doctests to say “be a little less strict about how the output matches”.

For example, sometimes you might have so much output it would be overwhelming to show it all:

```
>>> range(16)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

By using the `ELLIPSIS` option, you can elide part of that in your test, like this:

```
>>> print range(16)
[0, 1, ..., 14, 15]
```

Or, if your output may have awkward linebreaks and whitespace that might make it hard to use in a test, you can use `NORMALIZE_WHITESPACE` to ignore all whitespace differences between the expected output and the real output:

```
>>> poem_line
'My father moved through
dooms of love'
```

Python unittest module

Unit testing via classes! In the Python standard library.

`demo/tests_arithmetic.py`

```
import arithmetic
from unittest import TestCase

class AdditionTestCase(TestCase):
    """Examples of unit tests."""

    def test_adder(self):
        assert arithmetic.adder(2, 3) == 5
```

- Test cases are a bundle of tests
 - In a class that subclasses `TestCase`
 - Test methods **must** start with `test_`
- `python -m unittest NAME_OF_FILE` runs all cases

TestCase assertions

`demo/tests_arithmetic.py`

```
class AdditionTestCase(TestCase):
    """Examples of unit tests."""

    def test_adder(self):
        assert arithmetic.adder(2, 3) == 5

    def test_adder_2(self):
        # instead of assert arithmetic.adder(2, 2) == 4
        self.assertEqual(arithmetic.adder(2, 2), 4)
```

- Provides better output, including expected value

Method	Checks that
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x) is True</code>
<code>assertFalse(x)</code>	<code>bool(x) is False</code>
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assert IsNot(a, b)</code>	<code>a is not b</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assert IsNotNone(x)</code>	<code>x is not None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertNotIn(a, b)</code>	<code>a not in b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>

DocTest or unittest Class?

- DocTests keep tests close to code
 - Too many tests can drown out code
- *unittest* classes good for when you have lots of tests
 - Or interesting hierarchies of tests

Integration Tests

Test that components work together

Integration Testing Flask App

What kinds of things do we want to test in our Flask applications?

- “Does this URL path map to a route function?”
- “Does this route return the right HTML?”
- “Does this route return the correct status code?”
- “After a POST to this route, are we redirected?”
- “After this route, does the session contain expected info?”

Writing Integration Tests

You write them with *unittest* framework.

Yeah, I know. Weird.

test_client

demo/tests_flask.py

```
from app import app
```

demo/tests_flask.py

```
class ColorViewsTestCase(TestCase):  
    """Examples of integration tests: testing Flask app."""  
  
    def test_color_form(self):  
        with app.test_client() as client:  
            # can now make requests to flask via `client`
```

Technically, this comes from “Werkzeug”, a library that Flask uses.

This doesn’t start a real web server — but we can make requests to Flask via *client*.

GET Request

demo/tests_flask.py

```
def test_color_form(self):
    with app.test_client() as client:
        # can now make requests to flask via 'client'
        resp = client.get('/')
        html = resp.get_data(as_text=True)

        self.assertEqual(resp.status_code, 200)
        self.assertIn('<h1>Color Form</h1>', html)
```

POST and Form Data

demo/tests_flask.py

```
def test_color_submit(self):
    with app.test_client() as client:
        resp = client.post('/fav-color',
                           data={'color': 'blue'})
        html = resp.get_data(as_text=True)

        self.assertEqual(resp.status_code, 200)
        self.assertIn('Wow! I like blue, too', html)
```

Testing Redirects

demo/tests_flask.py

```
def test_redirection(self):
    with app.test_client() as client:
        resp = client.get("/redirect-me")

        self.assertEqual(resp.status_code, 302)
        self.assertEqual(resp.location, "http://localhost/")
```

`follow_redirects=True` makes new request when response redirects:

demo/tests_flask.py

```
def test_redirection_followed(self):
    with app.test_client() as client:
        resp = client.get("/redirect-me", follow_redirects=True)
        html = resp.get_data(as_text=True)

        self.assertEqual(resp.status_code, 200)
        self.assertIn('<h1>Color Form</h1>', html)
```

Testing the Session

To test value of session:

```
demo/tests_flask.py
```

```
from flask import session
```

```
demo/tests_flask.py
```

```
def test_session_info(self):
    with app.test_client() as client:
        resp = client.get("/")

        self.assertEqual(resp.status_code, 200)
        self.assertEqual(session['count'], 1)
```

To set the session before the request, add block like this:

```
demo/tests_flask.py
```

```
def test_session_info_set(self):
    with app.test_client() as client:
        # Any changes to session should go in here:
        with client.session_transaction() as change_session:
            change_session['count'] = 999

        # Now those changes will be in Flask's `session`
        resp = client.get("/")

        self.assertEqual(resp.status_code, 200)
        self.assertEqual(session['count'], 1000)
```

setUp and tearDown

setUp and *tearDown* methods are called before/after each test.

```
class FlaskTests(TestCase):

    def setUp(self):
        """Stuff to do before every test."""

    def tearDown(self):
        """Stuff to do after each test."""

    def test_1(self):
        ...

    def test_2(self):
        ...
```

Runs, in order: *setUp*, *test_1*, *tearDown* *setUp*, *test_2*, *tearDown*

Often useful to add/remove data in test database before/after each test

Making Testing Easier

Add these before test case classes:

```
demo/tests_flask.py
```

```
# Make Flask errors be real errors, not HTML pages with error info
app.config['TESTING'] = True

# This is a bit of hack, but don't use Flask DebugToolbar
app.config['DEBUG_TB_HOSTS'] = ['dont-show-debug-toolbar']
```

TIP Seeing Errors In Tests

If a route raises an error, it can be hard to debug this in a test.

For example, in your `server.py`:

```
@app.get('/')
def homepage():
    raise KeyError("Foo")
```

In your `tests_flask.py`:

```
class MyTest(unittest.TestCase):
    def test_home(self):
        with app.test_client() as client:

            result = client.get('/')
            self.assertEqual(result.status_code, 200)
```

When you run your tests, it will fail, as that route returns a 500 (Internal Server Error), not a 200 (Ok). However, you won't see the error message of the server.

To fix this, you can set the Flask app's configuration to be in `TESTING` mode, and it will print all Flask errors to the console:

This what `app.config['TESTING'] = True` does.

Breaking Down Code

Intermixed Concerns

How do we test this?

```
@app.post('/taxes')
def taxes():
    """Calculate taxes from web form."""

    income = request.form.get('income')

    # Calculate the taxes owed
    owed = income / 45.3 * random.randint(100) / other_stuff

    return render_template("taxes.html", owed=owed)
```

Breaking Down Code

Very often, you'll want to separate web interface from logic

```

def calculate_taxes(income):
    """Calculate taxes owed for this income."""

    ...

@app.post('/taxes')
def taxes():
    """Calculate taxes from web form."""

    income = request.form.get('income')
    owed = calculate_taxes(income)

    return render_template("taxes.html", owed=owed)

```

How Many Tests??

- Ask yourself: is there too much logic in your view function?
- When you test, you don't need one assertion per test function
- Remember to test failing things, like forms that don't validate

Organizing / Running Tests

Small Projects

For tiny projects, keep tests in one file, *tests.py*:

```

├── app.py
└── requirements.txt
└── tests.py

```

Run them like this:

```
(venv) $ python -m unittest
```

Larger Projects

For more complex projects, organize in files named *tests_something.py*:

```

├── app.py
└── requirements.txt
└── tests_cats.py
└── tests_dogs.py

```

Run all of them like this:

```
(venv) $ python -m unittest
```

Can also run individual files / cases / test methods:

```
(venv) $ python -m unittest tests_cats  
(venv) $ python -m unittest tests_cats.CatViewTestCase  
(venv) $ python -m unittest tests_cats.CatViewTestCase.test_meow
```

Looking Ahead

Resources

- Doctests: <https://docs.python.org/3/library/doctest.html> <<https://docs.python.org/3/library/doctest.html>>
- Unittest: <https://docs.python.org/3/library/unittest.html> <<https://docs.python.org/3/library/unittest.html>>
- Flask Testing <https://flask.palletsprojects.com/en/2.0.x/testing/> <<https://flask.palletsprojects.com/en/2.0.x/testing/>>
- Test Client <https://werkzeug.palletsprojects.com/en/2.0.x/test/> <<https://werkzeug.palletsprojects.com/en/2.0.x/test/>>

Future Topics

- Unit & Integration Testing for JS
- End-to-end Tests
 - Does it work? In a real browser? For real?
 - “Test-driven development”



Introduction to Databases

Intro

Goals

- Understand core ideas of relational databases:
 - Schemas and tables (records and columns)
 - Queries and result rows
 - Relationships
- Be able to create, drop, and interact with databases
- Learn how to use *psql*
- Be able to dump and restore databases from SQL

What's a database?

A database is a collection of data organized to be efficiently stored, accessed, and managed.

People often say “database” to mean any of these:

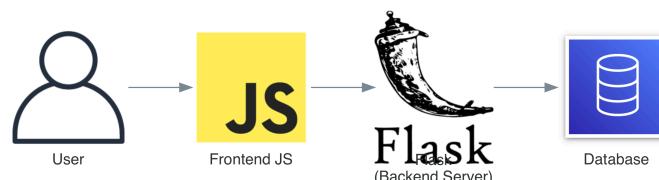
- server software (*database server software*)
- physical computer running that software (*database server*)
- a particular collection of tables & data (*database*)

An organization may have different databases for different projects:

- *employees*: salary, payroll, performance evaluations
- *sales*: customer invoices, billing notifications

One database server can serve many different databases.

Databases in web app stacks



- The backend server queries the database and crafts responses based on data
- The browser or frontend JS do not connect directly to the database

Relational Databases

Different ways to model data

Imagine you want to keep track of blog posts and the comments that people can make on a blog post. One blog posts could have many comments.

You could model this in a hierarchical fashion, where you nest the comments for a blog post inside of a comment.

Another possibility is that you could keep track of these two separately, but use IDs for the comments to refer to which post they were about.

Here are visualizations of these two strategies in JavaScript:

Hierarchical: uses nesting

```
posts_and_comments = [
  {
    title: "Tech Worker Reading List",
    author: "Matt",
    comments: [
      {
        msg: "Awesome!",
        username: "aliya",
      },
    ],
  },
  {
    title: "Why JS Matters",
    author: "Joel",
    comments: [
      {
        msg: "So helpful",
        username: "alex",
      },
    ],
  },
]
```

Relational: connect via IDs

```
posts = [
  {
    postId: 1,
    title: "Tech Worker Reading List",
    author: "Matt",
  },
  {
    postId: 2,
    title: "JavaScript in 2020",
    author: "Elie",
  },
]

comments = [
  {
    postId: 1,
    msg: "Awesome!",
    username: "aliya",
  },
  {
    postId: 2,
    msg: "So helpful",
    username: "alex",
  },
]
```

Relational Databases

Posts table

post_id	title	author
1	Tech Worker Reading List	Matt
2	JavaScript in 2020	Elie

Comments table

comment_id	post_id	msg	username
1	1	Awesome!	aliya
2	2	So helpful	alex

- The most common kind of database
- Model data as rows and columns of tabular data (*like spreadsheets*)
- The collection of data for one thing in a table is a *record*
- There are two records shown for posts, and two for comments

NOTE “RDBMS”

Relational databases are sometimes called by the industry buzzword *RDBMS*, “relational database management systems”.

There are other kinds of databases, which don’t store things in tables and use relationships to connect data. These are less common, and aren’t covered in this lecture.

SQL

SQL Structured Query Language: language for working with relational dbs.

- Standardized across different vendors (but not perfectly)
- Both *SEE-kwell* and *ESS-kyoo-ell* are acceptable pronunciations
- English-like, high-level, and *declarative*:

```
SELECT title, author, pub_date
FROM posts
WHERE author = 'Zach' AND pub_date >= '2020-05-01'
ORDER BY pub_date;
```

PostgreSQL

PostgreSQL is the database we’ll use

- Open Source and free to use
- Powerful
- Popular

TIP Pronouncing “PostgreSQL”

It’s pronounced *post-gres-kyoo-ell*. Please don’t pronounce it *post-grey-see-kwell*; that’s just beastly.

Exploring with psql

psql

PostgreSQL comes with a “database console” program, *psql*.

You can use this to investigate and interact with the database.

List databases

See list of databases on your server with `psql -l` in shell:

```
$ psql -l  
  
      Name   |  Owner  
-----+-----  
rithm  | rithm  
customers | rithm  
hack_or_snooze | rithm  
blog    | rithm
```

(there are other columns actually shown, but you can ignore them)

There are two databases that are always listed, but you shouldn't use directly: `template0` and `template1`. Ignore them.

Getting into a database

```
$ psql blog  
blog=# # prompt waiting for your commands
```

Here, we can enter special PostgreSQL command and SQL to explore our database and read/change the data.

To quit and return to shell, use «**Control-D**»

Your “user” database

In addition to project-specific databases, it's traditional to have one database for each user on a server with the same name as the user.

This database tends to be used for experimenting or trying out DB commands.

If you enter `psql` without giving it a database name, it defaults to this database:

```
$ psql  
me=# # assuming your username is "me", of course ;-)
```

List tables in a database

Use `\dt` (*describe tables*) command:

```
blog=# \dt  
  
      List of relations  
 Schema |   Name   | Type  | Owner  
-----+-----+-----+-----  
 public | comments | table | rithm  
 public | posts   | table | rithm
```

Seeing schema of a table

The *schema* of a table defines the *structure of the data* it can hold.

To view the schema of a table, use `\d tablename`:

```
blog=# \d posts

Table "public.posts"
 Column | Type    | Nullable | Default
-----+-----+-----+-----+
post_id | integer | not null | nextval
title   | text    | not null |
author  | text    | not null |
pub_date| date   |            |
num_likes| integer | not null | 0
```

- *post_id*: auto-incrementing unique integer
- *title*: text, must be known
- *author*: text, must be known
- *pub_date*: date, can be unknown
- *num_likes*: integer, must be known, default 0

SQL

```
SELECT title, pub_date FROM posts WHERE author = 'Matt';
```

- Commands can go over several lines, but must end with a semicolon

```
SELECT title, pub_date
      FROM posts
     WHERE author = 'Matt';
```

- Strings are case-sensitive and must use single (*not double*) quotes:

- `'Matt'` (right!) — not `'matt'` (wrong case) or `"Matt"` (not right)

- Keywords conventionally written in ALL CAPS but are case-insensitive

```
SELECT author FROM posts; -- is the same as:
select author From posts;
```

Types of SQL Statements

DML: Data Manipulation Language

The commands to create, read, modify, or delete data:

Examples: *SELECT, INSERT, UPDATE, DELETE*

DDL: Data Definition Language

The commands to create and delete tables and modify schemas:

Examples: *CREATE TABLE, ALTER TABLE, DROP TABLE*

SELECT queries

Query

```
SELECT title, author FROM posts WHERE author = 'Joel';
```

Result

title	author
Learning a New Language	Joel
Working from Home: A Guide to Success	Joel
(2 rows)	

- Queries just show the results, they do not change the table data
- Each line in the result of a query is a *row*

Select all columns and all records

```
SELECT * FROM posts;
```

Select some columns and all records

```
SELECT title, author FROM posts;
```

Select all columns and some records

```
SELECT * FROM posts WHERE author = 'Joel';
```

Select some columns and some records

```
SELECT title, pub_date, num_likes FROM posts WHERE author = 'Joel';
```

Relationships

Second table: *comments*

```
blog=# \d comments
```

Table "public.comments"			
Column	Type	Nullable	Default
comment_id	integer	not null	nextval
post_id	integer	not null	
msg	text	not null	
username	text	not null	
approved	boolean	not null	false

- *comment_id*: auto-incrementing unique int
 - *post_id*: int referring to post table ID
 - *msg*: text
 - *username*: text
 - *approved*: boolean
- (all of these fields cannot be unknown)

posts			comments			
post_id	title	author	comment_id	post_id	msg	username
1	Tech Worker Reading List	Matt	1	1	Awesome!	aliya
2	JavaScript in 2020	Elie	2	2	So helpful	alex
3	Rithm School FAQs	Tim	3	2	Buy drugs at ...	bot-135

Each comment record tells us which post is being commented on by following the `comments.post_id` to `posts.post_id`.

Querying across relationships

RDBMSs are good at asking questions that span a relationship.

For each post with a comment, show title, message, and username

```
SELECT title, msg, username
  FROM posts
  JOIN comments USING(post_id)
```

title	msg	username
Tech Worker Reading List	Awesome!	aliya
JavaScript in 2020	So helpful	alex
JavaScript in 2020	Buy drugs at ...	bot-135
(3 rows)		

Querying and joining is what RDBMSs are designed for, so they're extremely efficient at doing it well

Creating & deleting DBs

Creating a database

```
$ createdb my_database_name
```

- We'll make a different database for each project/exercise.
- That creates a database, but it has no tables or data yet.
- Database names should be lowercase *snake_case* – otherwise, there will be challenges later in using it.

The database is **not** a file in your current directory

It's a bunch of files/folders elsewhere on your computer

These aren't human-readable (*they're optimized for speed!*)

Saving your project in Git won't save your database!

Dropping a database

A database that is “dropped” is completely deleted (schema and data):

```
$ dropdb my_database_name
```

There is no way to undo that without your having kept a copy of it.

Exporting & importing DBs

Exporting to SQL

You can make a backup of your database by “dumping it” to a file:

```
$ pg_dump -c -O blog > blog.sql
```

This makes a file in the current directory, `blog.sql`

It contains commands to recreate the schema and data when needed.

To give others starter data for a project, include this in your Git repo.

Importing from SQL

You can feed `*.sql` scripts into `psql`:

```
$ createdb blog          # needed if it doesn't already exist
$ psql -f blog.sql blog
```

Exporting to CSV

CSV (*comma-separated values*) is a common file format for sharing data:

`posts.csv`

```
post_id,title,author,pub_date,num_likes
1,"A Tech Worker Reading List","Matt","2020-06-20",1212
2,"JavaScript in 2020","Elie","2020-05-27",907
3,"Rithm School FAQs","Tim","2020-04-01",696
```

Spreadsheet apps and many other programs can import data from CSV.

Exporting posts table to posts.csv file

```
blog=# \COPY posts TO 'posts.csv' CSV
```

Exporting to JSON

Exporting posts table to posts.json file

```
blog=# \copy (SELECT json_agg(posts) FROM posts) TO 'posts.json'
```

Tips and resources

Restarting PostgreSQL server

If your database server isn't running, you'll get a message like this:

This error means PostgreSQL's server isn't running

```
$ psql blog
psql: error: could not connect to server: No such file or directory
Is the server running locally and accepting
connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

You can restart it with the shell command `pg_restart`:

```
$ pg_restart
```

Common commands in psql

- `\l` — List all databases
- `\c DB_NAME` — connect to *DB_NAME*
- `\dt` — List all tables (*in current db*)
- `\d TABLE_NAME` — Get details about *TABLE_NAME* (*in current db*)
- `\q` — Quit *psql* (even easier: <Control-D>)
- `\h` — Get help on SQL commands
- `\?` — Get help on special *psql* commands

Coming Up

- SQL Querying
- Designing and creating tables
- Working with table relationships
- Using databases in Python and JavaScript Node.js
- *Later:* an brief introduction to non-relational (“*NoSQL*”) databases



SQL Querying

[Download Demo SQL Code <..../demo.zip>](#)

Intro

- Learn getting data with `SELECT`
- Learn insertion, updating, and deletion
- Learn about transactions

Types of SQL statements

DML: Data Manipulation Language

The commands to create, read, modify, or delete records:

Examples: `SELECT, INSERT, UPDATE, DELETE`

DDL: Data Definition Language

The commands to create and delete tables and modify schemas:

Examples: `CREATE TABLE, ALTER TABLE, DROP TABLE`

This lecture will focus on DML.

Data Manipulation Language

Most DML is about **CRUD** operations on records.

Letter	Verb	SQL Commands
C	Create	<code>INSERT INTO table</code>
R	Read	<code>SELECT ... FROM table ...</code>
U	Update	<code>UPDATE table SET ...</code>
D	Delete	<code>DELETE FROM table ...</code>

SELECT

`SELECT` is the most flexible and powerful command in SQL.

It selects data (including summary data, roll-up data, etc) from table(s).

It doesn't change data.

`SELECT` statements have subclauses, which are *performed* in this order:

#	Clause	Purpose	Required?
1	FROM	Select and join together tables where data is	—
2	WHERE	Decide which rows to use	—
3	GROUP BY	Place rows into groups	—
4	HAVING	Determine which grouped results to keep	—
5	SELECT	Determine values of result	✓
6	ORDER BY	Sort output data	—
7	LIMIT	Limit output to n rows	—
8	OFFSET	Skip n rows at start of output	—

All select statements start with *SELECT*.

```
SELECT 'math is fun', 2 * 3;
```

That's not very interesting, though, until we get data from tables!

FROM

Determine which table(s) to use to get data:

title/author/pub_date from all records in posts

```
SELECT title, author, pub_date
  FROM posts;
```

You can get data from multiples tables by *joining* them — we'll discuss this later

You can get all columns by using *:

all columns from all records in posts

```
SELECT *
  FROM posts;
```

Be cautious doing so in actual code — this might return sensitive data!

WHERE

Filter result rows on a condition:

only posts >= May 1st

```
SELECT *
  FROM posts
 WHERE pub_date >= '2020-05-01';
```

GROUP BY

Reduce the amount of rows returned by grouping rows together:

group by author: each author shows up once

```
SELECT author
  FROM posts
 GROUP BY author;
```

group by author: name & num posts

```
SELECT author, COUNT(*)
  FROM posts
 GROUP BY author;
```

WHERE and GROUP BY

Remember, *WHERE* happens first, so it filters *before* we group:

group by author: name & num posts

```
SELECT author, COUNT(*)
  FROM posts
 WHERE pub_date >= '2020-05-01'
 GROUP BY author;
```

HAVING

Decide which group(s), if grouped, to keep:

only show groups with more than 1 post

```
SELECT author, COUNT(*)
  FROM posts
 GROUP BY author
 HAVING COUNT(*) > 1;
```

You can't use *HAVING* without *GROUP BY*.

SELECT

This is where the *SELECT* clause is used (*even though it appears first*)

The select clause can have `*`, column names, or even SQL functions or expressions:

get title and time-since-published

```
SELECT title, AGE(pub_date)
  FROM posts;
```

For a products table, you could calculate and show a sale price:

show title and 50% off price

```
SELECT product_name, price / 2 AS sale_price
  FROM products;
```

To give a row column a name, use *AS* last this.

ORDER BY

Sort rows before returning:

order results by author name (A → Z)

```
SELECT *
  FROM posts
 ORDER BY author;
```

Can sort descending with *DESC*:

order results by author name (Z → A)

```
SELECT *
  FROM posts
 ORDER BY author DESC;
```

If you don't provide an *ORDER BY* clause, the returned row order is not predictable.

You can provide multiple expressions to sort on:

order results by author name, within that, date:

```
SELECT *
  FROM posts
 ORDER BY author, pub_date;
```

If you want to sort descending, put the word *DESC* after any sort expression to invert that sort:

ORDER BY author DESC, pub_date returns rows first sorted by author name, in Z → A order, and within that, by publication date sorted oldest to newest.

LIMIT

Only show *n* number of rows:

only show first 5 rows

```
SELECT *
  FROM posts
 ORDER BY pub_date
 LIMIT 3;
```

It only makes sense to use *LIMIT* if you have an *ORDER BY* clause.

OFFSET

Skip *n* number of rows.

skip first row

```
SELECT *
  FROM posts
 ORDER BY pub_date
 OFFSET 1;
```

Often used with *LIMIT* to paginate results:

get "next batch of 3"

```
SELECT *
  FROM posts
 ORDER BY pub_date
  LIMIT 3
OFFSET 3;
```

SQL Operators

Operators are used to build more complicated queries They are reserved SQL keywords or symbols:

Common ones: `=, <>, IN, NOT IN, BETWEEN, AND, OR, NOT`

get posts if $\geq 5/1$ or by Zach:

```
SELECT * FROM posts WHERE author='Zach' OR pub_date >= '2020-05-01';
```

get posts if $\geq 5/1$ and by Zach:

```
SELECT * FROM posts WHERE author='Zach' AND pub_date >= '2020-05-01';
```

get posts where post id is in a list

```
SELECT * FROM posts WHERE post_id IN (1, 3, 4, 5);
```

get posts between 4/1 and 4/30 (inclusive)

```
SELECT *
  FROM posts
 WHERE pub_date BETWEEN '2020-04-01' AND '2020-04-30';
```

SQL Aggregates

Aggregates are used to combine records together to extract data:

Common aggregate functions include `COUNT, AVG, SUM, MIN, and MAX`

count of posts

```
SELECT COUNT(*) FROM posts;
```

count posts by Zach

```
SELECT COUNT(*) FROM posts WHERE author = 'Zach';
```

get most recent post date

```
SELECT MAX(pub_date) FROM posts;
```

sum of all likes

```
SELECT SUM(num_likes) FROM posts;
```

average (mean) number of likes

```
SELECT AVG(num_likes) FROM posts;
```

GROUP BY

The *GROUP BY* and *HAVING* clauses are often used with aggregate functions

count of posts

```
SELECT author, COUNT(*)
  FROM posts
 GROUP BY author;
```

count, by author, of recent posts

```
SELECT author, COUNT(*)
  FROM posts
 WHERE pub_date >= '2020-05-01'
 GROUP BY author;
```

only show groups with more than 1 post

```
SELECT author, COUNT(*)
  FROM posts
 GROUP BY author
 HAVING COUNT(*) > 1;
```

NULL

NULL is a special value that means *unknown*.

The schema for the table controls which columns can allow NULL values.

Designers tend to mark columns as *NOT NULL* if they know there will be data.

NULL is **very different** from Python's *None* and JavaScript's *null*.

SQL *NULL* vs Python *None*

Let's say there are two people who we don't know much about:

```
donald = None
ivanka = None
```

We can check if *donald* is equal to *None* or a known number:

```
donald == None    # True
donald == 10      # False --- but how do we know that?
```

Python raises an error if we use *None* in an inequality check:

```
donald > 100    # TypeError: can't compare int and None
```

Python treats both "unknown values" as the same

```
donald == ivanka  # True, even though we don't know that
```

SQL **NULL** vs JavaScript **null**

Same setup:

```
let donald = null;  
let ivanka = null;
```

We can check if *donald* is equal to null or a known number:

```
donald === null;    // true  
donald === 10;      // false -- but how do we know that?
```

JavaScript just assumes null is less than any number:

```
donald > 100;    // false -- but do we really know this?
```

JavaScript also treats both “unknown values” as the same

```
donald === ivanka;    // true, but we don't know that
```

SQL **NULL**

In SQL, it's very rigorously logical

```
SELECT donald = 10;    -- NULL, we don't know!
```

```
SELECT donald > 100;    -- NULL, we don't know!
```

```
SELECT donald = ivanka;    -- NULL, we don't know!
```

```
SELECT donald = null;    -- NULL; are different unknowns
```

Comparing *anything* to a NULL value returns NULL:

```
SELECT NULL = NULL;    -- NULL: different unknowns!
```

Therefore, to check if something is NULL, write it like:

```
SELECT NULL IS NULL;    -- gives true or false
```

Be careful! What's the problem with these two queries:

find new posts?

```
SELECT *  
FROM posts  
WHERE pub_date >= '2020-05-01';
```

find old posts?

```
SELECT *  
FROM posts  
WHERE pub_date < '2020-05-01';
```

Neither finds non-published posts!

find new posts PLUS unpublished

```
SELECT *
  FROM posts
 WHERE pub_date >= '2020-05-01' OR pub_date IS NULL;
```

NULL values are tricky and sometimes a pain. Be careful with them.

NULL and text fields

The same kind of logic applies with text fields. Imagine a table with fields for first, middle, and last names. If middle names are left NULL for some people, an expression like `fname || ' ' || mname || ' ' || lname` won't work—since `mname` is NULL, the entire expression can only evaluate to NULL.

SQL includes a *coalesce* function: it takes several arguments and returns the first non-NULL argument. We could instead write our name problem as `fname || coalesce(mname, '') || lname`.

While it's not part of the SQL standard, PostgreSQL includes an easier function, `concat_ws`. This is used to join together many arguments with a separator string, ignoring any nulls. We could write the above as `concat_ws(' ', fname, mname, lname)`.

You can learn about other string functions at [String Functions and Operators](https://www.postgresql.org/docs/current/functions-string.html) <<https://www.postgresql.org/docs/current/functions-string.html>>.

Modifying Data

Creating data with INSERT

insert new post

```
INSERT INTO posts (title, author)
  VALUES ('Why We Teach Node', 'Elie');
```

can insert multiple records at once

```
INSERT INTO posts (title, author) VALUES
  ('Why We Teach Python', 'Matt'),
  ('Debugging in Chrome', 'Nate');
```

TIP INSERT via SELECT

You can combine INSERT and SELECT to copy data from another table.

```
INSERT INTO posts (title, author)
  SELECT title, author
    FROM some_other_table;
```

Updating Data with UPDATE

Matt is a prolific writer

```
UPDATE posts SET author = 'Matt';
```

Not that prolific!

```
UPDATE posts SET author = 'Matt' WHERE post_id = 2;
```

Deleting Data with DELETE

delete a post

```
DELETE FROM posts WHERE post_id = 3;
```

delete unpublished posts

```
DELETE FROM posts WHERE pub_date IS NULL;
```

delete all posts

```
DELETE FROM posts;
```

Transactions

SQL statements run independently and commit changes immediately.

Instead, you can work in a *transaction*.

Changes made in a transaction are limited to that transaction; they are not saved for others until you commit them.

These are good for:

- Rolling back from accidental deletions/updates
- Ensuring operation is *atomic* – all parts must succeed, else it isn't committed

Working in a Transaction

Start a transaction with the command *BEGIN*:

```
BEGIN;  
DELETE FROM posts;  
  
-- within this transaction, your query:  
SELECT COUNT(*) FROM posts;    -- finds none!  
  
-- (other connections to the database still see them)
```

To *commit* changes, use *COMMIT*:

```
COMMIT;  
-- commits & closes transaction; now everyone has no posts
```

If you did something you don't want, or had an error, *ROLLBACK*:

```
ROLLBACK;  
-- closes transaction without committing
```

TIP Fouled Transactions

If you have any kind of error in a transaction, it is *fouled* — no further SQL statements can succeed. *ROLLBACK* the transaction (and then open a new one if still want to be in a transaction) and start over.

Relationships in SQL

Goals

- Learn what makes SQL databases “relational”
- Understand one-to-many and many-to-many relationships
- Describe and make use of the different types of joins (inner, outer)

Example: Movies

Data we want to store in a database

id	title	studio	founded_in
1	Star Wars: The Force Awakens	Walt Disney Studios	1953-06-23
2	Avatar	20th Century Fox	1935-05-31
3	Black Panther	Walt Disney Studios	1953-06-23
4	Jurassic World	Universal Pictures	1912-04-30
5	Marvel's The Avengers	Walt Disney Studios	1953-06-23

- So much duplication!
- What if we want other info about studios besides founding date?

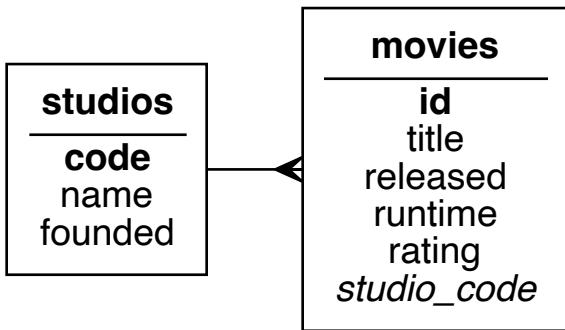
movies

id	title	studio_code
1	Star Wars: The Force Awakens	disney
2	Avatar	fox
3	Black Panther	disney
4	Jurassic World	universal
5	Marvel's The Avengers	disney

studios

code	name	founded_in
disney	Walt Disney Studios	1953-06-23
fox	20th Century Fox	1935-05-31
universal	Universal Pictures	1912-04-30

One-to-Many (1:M)



This type of schema drawing uses *crow's foot notation* – the feet go on the “many” side.

- `studio_id` column gives us reference to record in `studios` with that id.
 - This makes a **foreign key constraint**; ensuring every `studio_id` is in `studios`.
- One-to-Many: one studio *has many* movies, but each movie *has one* studio.
 - `movies` is *referencing* table, and `studios` is *referenced* table.

Foreign key constraints

Setting up a foreign key constraint with DDL:

```
CREATE TABLE studios (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    founded_in DATE);
```

```
CREATE TABLE movies (
    id SERIAL PRIMARY KEY,
    title TEXT NOT NULL,
    release_year INT,
    runtime INT,
    rating TEXT,
    studio_id INT REFERENCES studios);
```

Constraints are specified by the DDL, but affect DML query behavior.

```
INSERT INTO studios (code, name, founded) VALUES
    ('disney', 'Walt Disney Studios Motion Pictures', '1953-06-23'),
    ('fox', '20th Century Fox', '1935-05-31'),
    ('universal', 'Universal Pictures', '1912-04-30');
```

```
-- reference Disney's primary key
INSERT INTO movies (title, studio_code)
    VALUES ('Star Wars: The Force Awakens', 'disney');
```

```
-- Throws an Foreign Key Constraint Error...
-- There is no studio with a primary key of "blah"
INSERT INTO movies (title, studio_code)
VALUES ('Black Panther', 'blah');
```

Deleting data

When trying to delete a studio...

We cannot delete it outright while movies still reference it.

```
DELETE FROM studios WHERE code = 'disney'; -- error
```

Option 1: NULL *studio_code* field of movies that reference it (*if can be NULL*)

```
UPDATE movies SET studio_code=NULL WHERE studio_code = 'disney';
DELETE FROM studios WHERE code = 'disney';
```

Option 2: Delete the movies associated with that studio first.

```
DELETE FROM movies WHERE studio_code = 'disney';
DELETE FROM studios WHERE code = 'disney';
```

Joining tables

- *JOIN* allows us to get results combining info from multiple tables
- Data from tables is matched according to a *join condition*
- Usually, join condition involves comparing:
 - a *foreign key* from one table (*movies.studio_id*), and
 - *primary key* in another table (*studios.id*)

Setting up data

```
CREATE TABLE studios (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  founded_in DATE);
```

```
CREATE TABLE movies (
  id SERIAL PRIMARY KEY,
  title TEXT NOT NULL,
  release_year INT,
  runtime INT,
  rating TEXT,
  studio_id INT REFERENCES studios);
```

```

INSERT INTO studios
  (code, name, founded)
VALUES
  ('disney', 'Walt Disney Studios Motion Pictures', '1953-06-23'),
  ('fox', '20th Century Fox', '1935-05-31'),
  ('universal', 'Universal Pictures', '1912-04-30');

```

```

INSERT INTO movies
  (title, released, runtime, rating, studio_code)
VALUES
  ('Star Wars: The Force Awakens', 2015, 136, 'PG-13', 'disney'),
  ('Avatar', 2009, 160, 'PG-13', 'fox'),
  ('Black Panther', 2018, 140, 'PG-13', 'disney'),
  ('Jurassic World', 2015, 124, 'PG-13', 'universal'),
  ('Marvel's The Avengers', 2012, 142, 'PG-13', 'disney');

```

Joining tables

Inner join

```

SELECT title, name
  FROM movies
  INNER JOIN studios
    ON studios.code = movies.studio_code;

```

Also a inner join

```

SELECT title, name
  FROM movies
  JOIN studios
    ON studios.code = movies.studio_code;

```

JOIN and *INNER JOIN* are the same; the *INNER* keyword is optional.

Types of joins

There are two primary types of joins: *inner* and *outer*.

- **Inner**

Only the rows that match the condition in both tables.

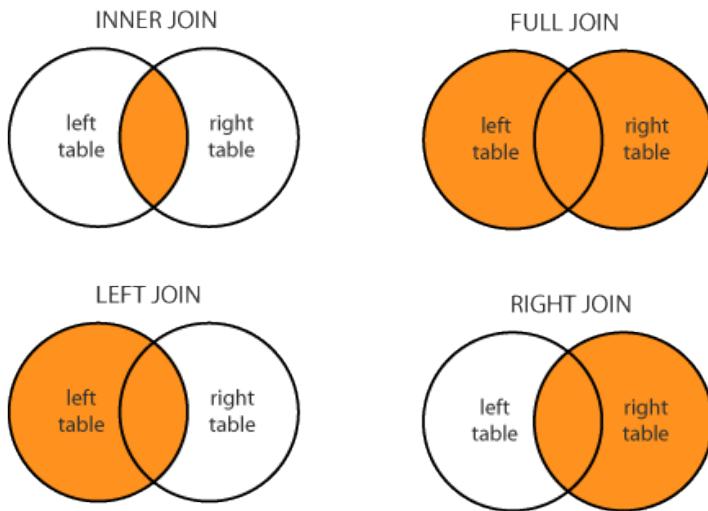
- **Outer**

Left: all rows from left table, combined with matching rows from right.

Right: all rows from right table, combined with matching rows from left.

Full - All the rows from both tables (left and right).

Join diagrams



Joins in practice

- You'll use *INNER* joins most often
- *OUTER* joins will find rows in one table even with no match in other table
 - eg, an movie with a NULL studio *or* a studio with no films
- Outer join example:

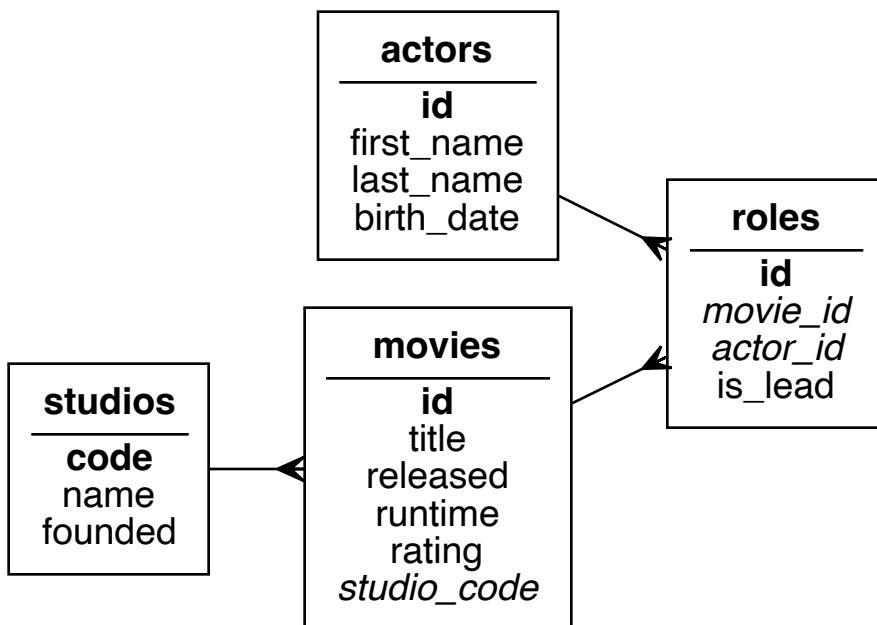
```
-- insert an indie movie (the studio_id is left out)
INSERT INTO movies (title, released, runtime, rating, studio_code)
VALUES ('Memento', 2000, 113, 'R', NULL);
```

```
-- this query will include the indie movie
SELECT name
FROM movies
LEFT OUTER JOIN studios
ON studios.code = movies.studio_code;
```

Many-to-Many (M:M)

Actors and roles

- We've seen a *one-to-many* relationship: one studio has many movies
- But not all relationships can be expressed in this way...
- eg, one movie has many actors, but each actor has roles in many movies!
- This is an example of a *many-to-many* relationship.
- A *many-to-many* is just two *one-to-many*s back-to-back!



- Each actor has many roles (1:M)
- Each movie has many roles (1:M)
- Actor and movie don't relate directly – but through roles! (M:M)

Set up: actors and roles

```
CREATE TABLE actors (
  id SERIAL PRIMARY KEY,
  first_name TEXT,
  last_name TEXT,
  birth_date DATE);
```

```
CREATE TABLE roles (
  id SERIAL PRIMARY KEY,
  movie_id INTEGER REFERENCES movies,
  actor_id INTEGER REFERENCES actors);
```

```
INSERT INTO actors
(first_name, last_name, birth_date)
VALUES
('Scarlett', 'Johansson', '1984-11-22'),
('Samuel L', 'Jackson', '1948-12-21'),
('Kristen', 'Wiig', '1973-08-22');
```

```
INSERT INTO roles
(movie_id, actor_id)
VALUES
(1, 1),
(1, 2),
(3, 2);
```

Many-to-Many (M:M)

Let's see what the movies, actors and roles tables look like!

Movies					
id		title	release_year	runtime	rating
1	Marvel's The Avengers		2012	142	PG-13
2	Avatar		2009	160	PG-13
3	Star Wars: Episode I		1999	133	PG

Actors				Roles			
id		first_name	last_name	birth_date	id	movie_id	actor_id
1	Scarlett	Johansson	1984-11-22		1	1	1
2	Samuel L	Jackson	1948-12-21		2	1	2
3	Kristen	Wiig	1973-08-22		3	3	2

Join tables

- The *roles* table is often called a *join table* or *association table*
- A join table connects two tables in a many-to-many relationship.
 - It has two foreign key columns to the two other tables in the relationship.
 - It can have other columns (eg, how much was actor paid for role).
- Join tables are often named in one of a few ways:
 - If there's a good, obvious name for it: *roles*
 - If not, use both other tables: *movies_actors*
 - For complex databases, some people use both: *movies_actors_roles*

Querying a many-to-many

Connecting movies and actors:

```
SELECT title, first_name, last_name
  FROM movies
  JOIN roles
    ON movies.id = roles.movie_id
  JOIN actors
    ON roles.actor_id = actors.id;
```

Selecting certain columns, using table alias shorthand:

```
SELECT m.title, a.first_name, a.last_name
  FROM movies AS m
  JOIN roles AS r
    ON m.id = r.movie_id
  JOIN actors AS a
    ON r.actor_id = a.id;
```

The `AS` keyword is optional (you can leave it out, like `FROM movies m`), but it makes it read nicer to write it in.

Get `(id, first name, last name)` of actors who have been in 2+ movies:

```
SELECT a.id, a.first_name, a.last_name
  FROM movies AS m
    JOIN roles AS r
      ON m.id = r.movie_id
    JOIN actors AS a
      ON r.actor_id = a.id
 GROUP BY a.id, a.first_name, a.last_name
 HAVING count(*) >= 2;
```

All actors & their movies

A trickier problem: list *all* actors, along with any movies they've been in:

```
SELECT a.id, a.first_name, a.last_name, m.title
  FROM actors AS a
    JOIN roles AS r
      ON a.id = r.actor_id
    JOIN movies AS m
      ON r.movie_id = m.id;
```

We won't get actors who've never been in a movie!

```
SELECT a.id, a.first_name, a.last_name, m.title
  FROM actors AS a
    LEFT OUTER JOIN roles AS r
      ON a.id = r.actor_id
    LEFT OUTER JOIN movies AS m
      ON r.movie_id = m.id;
```



DDL & Schema Design

Goals

- Learn SQL commands to create, alter, and drop databases/tables
- Understand the basics of database schema design
- Learn how to properly model relational data

DDL Basics

Creating and Dropping Databases

Can do in **psql** as SQL:

```
=# CREATE DATABASE new_db;  
=# DROP DATABASE new_db;
```

Can do from shell (not recommended):

```
$ createdb new_db  
$ dropdb new_db
```

Creating Tables

```
jane=# CREATE DATABASE movies;  
CREATE DATABASE  
  
jane=# \c movies  
You are now connected to database "movies" as user "jane".  
movies=#
```

```
CREATE TABLE movies (  
    id SERIAL PRIMARY KEY,  
    title VARCHAR(50) NOT NULL,  
    released DATE,  
    runtime INT NOT NULL,  
    rating VARCHAR(5) NOT NULL  
        CHECK (rating IN  
            ('G', 'PG', 'PG-13', 'R', 'NC-17')),  
    studio_code VARCHAR(10) NOT NULL REFERENCES studios,  
    UNIQUE (title, released));
```

Inspecting Tables in PostgreSQL

Listing the tables in the database

```
movies=# \dt
```

Showing schema for a database object (*eg a table*)

```
movies=# \d movies
```

Dropping Tables

```
DROP TABLE movies;
```

DANGER Dependent Objects

If that table is the referenced table in a relationship, it cannot be dropped until the referencing table is dropped.

To do so automatically, you can use `DROP movies CASCADE`. Be careful, though — that could delete a bunch of tables!

Column Specifications

Data Types

Text

Text Strings

Varchar(*n*)

Text Strings, but limited to *n* characters

Other than the length restriction, they're the same.

Int

Integer numbers

Numeric

Fixed-precision numbers (*for money and other fixed things*)

Float

Non-fixed-precision floating-point numbers (*for other things*)

What Is Fixed Precision?

Computer floating point numbers typically can represent incredibly small numbers or incredibly large numbers, and can also have lots of precision (*you can think of this right of the decimal place*).

However, floating point values have a small degree of imprecision: in databases and most programming languages, if you add together the floating point numbers 0.1 and 0.2, you don't get 0.3 — instead, you get something closely approximating that (typically, something like 0.3000000000000004). While that's *almost* the same as 0.3, a logical condition like `WHERE num <= 0.3` would fail for that number.

This isn't a bug in PostgreSQL. It's just a side-effect of the way computers store floating point numbers and the differences in binary representation (what computers use) and human representation in tens. In tens numbers, the number 1/3 (one third) can't be written directly in digits: we tend to write 0.333333333333 for it, but that's *close to* 1/3rd, but not exactly so. Computer floating point math, though, can accurately hold that

exact value in memory. So there are different values that are easier or harder to represent in different systems.

To get around this, relational databases tend to offer a *NUMERIC* type where you give it a fixed precision, like *NUMERIC(10,2)*. This can store a 10-digit number perfectly, with exactly two perfectly-accurate decimal places to the right. This is a good choice for storing monetary amounts, because those often require fixed, perfect-precision. The downside of this type is that it's slower for computation than floating point numbers, but the fixed precision usually makes that a good trade-off.

Some programming languages have these kind of numbers, too, including Python, which has a *Decimal* type with the same properties.

The TL;DR: use `NUMERIC(10,2)` for currency amounts and it will work as expected.

Boolean

True or False

Date

Date (without time)

Timestamp

Date and time without time zone (*the idea of a date time*)

Timestamp with time zone

... with time zone (*an actual point in date time in a place*)

NOTE The Idea of A Timestamp Versus a Point in Time

Timestamp is good for the “idea of a date and time”, like “Trick-or-Treating Start Time All Over World in 2020” could be October 31, 2020 at 6:00pm. Of course, the actual point in time when it’s 6pm in Paris is different than when it’s 6pm in Mumbai. When you want to store an actual point in time, you can use *Timestamp with time zone* and provide a time zone, so you can know “when that timestamp actually is”.

Serial

Auto-incrementing numbers (*generally used for primary keys*)

NOTE Serials

Serials are *never* reused for the same table — even if that record is deleted, or even if on inserting, an error happened. Therefore, the highest serial can be higher than the number of items in a table, and there can be gaps between serial numbers. All that is guaranteed are that they are unique.

NOTE Other Types

There are lots of other types for handling geospatial information, IP addresses, arrays, and more!

Common Type Choices

- Amount of money:
 - *NUMERIC(10, 2)* or *INT* — but not *FLOAT*
- Title of book:

- `TEXT` or `VARCHAR(50)` if you want to limit size
- Zip code:
 - `VARCHAR(5)` (or 9 for ZIP+4) — they’re not really numbers
- Meeting start time:
 - `TIMESTAMP WITH TIME ZONE`

NULL

`NULL` is a special value in SQL for *unknown*.

`NULL` values are ok when you might have missing/unknown data

But generally, they’re a pain, so it’s good to make columns `NOT NULL`

Default Values

A column can be given a default value:

```
CREATE TABLE invoices (
  id SERIAL PRIMARY KEY, -- actually make INT with a default
  amt NUMERIC(10,2) NOT NULL,
  discount NUMERIC(10,2) NOT NULL DEFAULT 0,
  sent_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP),
  notes TEXT NOT NULL DEFAULT ''');
```

Primary Keys

Every table should have a *primary key*, a unique way to identify records

- Primary keys *must be*:
 - Unique
 - Not Null
- Primary keys *should be*:
 - Unchanging (*it’s a pain when primary keys change*)

Choosing Primary Keys

- If there’s a unique stable identifier in the real world, use it
 - country codes for countries or taxpayer id numbers for companies
- If the company has an internal code that’s unique and stable, use it
 - Offices often have a room code for every office in buildings
- If the “real name” is unique and stable, might be ok to use
 - “JavaScript” and “Python” are fine PKs in a table of languages
- If there isn’t a good choice: use a `SERIAL` (auto-incrementing int)

Since PKs should be stable, *email* is a bad choice for user tables.

But: people don't like remembering username, and like logging in by email

Common solution: use a *SERIAL* for the PK & make email address unique

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email TEXT NOT NULL UNIQUE
    -- other things );
```

You can use the *id* in relationships and still let users log in by email or change the email

Constraints

Constraints are a basic form of validation. The database can prevent basic types of unintended behavior.

- Primary Key (*every table must have a unique identifier*)
- Not Null (*prevent null in the column*)
- Foreign Key (*column values must reference values in another table*)
- Unique (*prevent duplicates in the column*)
- Check (*check a condition before inserting / updating*)

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email TEXT NOT NULL,
    phone_number TEXT UNIQUE,
    acct_bal NUMERIC(15,2) CHECK (acct_bal >= 0),
    country_code VARCHAR(2) REFERENCES countries);
```

Constraints can apply over multiple columns:

```
CREATE TABLE people (
    id SERIAL PRIMARY KEY,
    first_name NOT NULL TEXT,
    last_name NOT NULL TEXT,
    born DATE NOT NULL,
    died DATE,

    UNIQUE(first_name, last_name, born),
    CHECK(born < died)
);
```

CHECK conditions fail if the result expression is false — if it is true or null, it will succeed (assuming no other constraints come into play).

Column Manipulation

Adding / Removing / Renaming columns

```

ALTER TABLE books ADD COLUMN in_paperback BOOLEAN;

ALTER TABLE books DROP COLUMN in_paperback;

ALTER TABLE books RENAME COLUMN page_count TO num_pages;

```

NOTE Controlling Delete Behavior with DDL

(You can ignore this now, but it can be an interesting thing later for you to explore).

Normally, you wouldn't be able to delete a studio that has related movies: *referential integrity* prevents that. In order to delete the Disney studio, you'd need to make sure that there are no movies referencing that studio. (You could do that by either deleting those movies or changing the *studio_code* to another studio or to NULL [if that field can be null]).

In some cases, you may want the database to do that for you automatically:

```

CREATE TABLE movies (
    -- ...
    studio_code VARCHAR(10)
    REFERENCES studios
    ON DELETE SET NULL);

```

Deleting a studio sets related movies' *studio_code* to null

```

CREATE TABLE movies (
    -- ...
    studio_code VARCHAR(10) NOT NULL
    REFERENCES studios
    ON DELETE CASCADE);

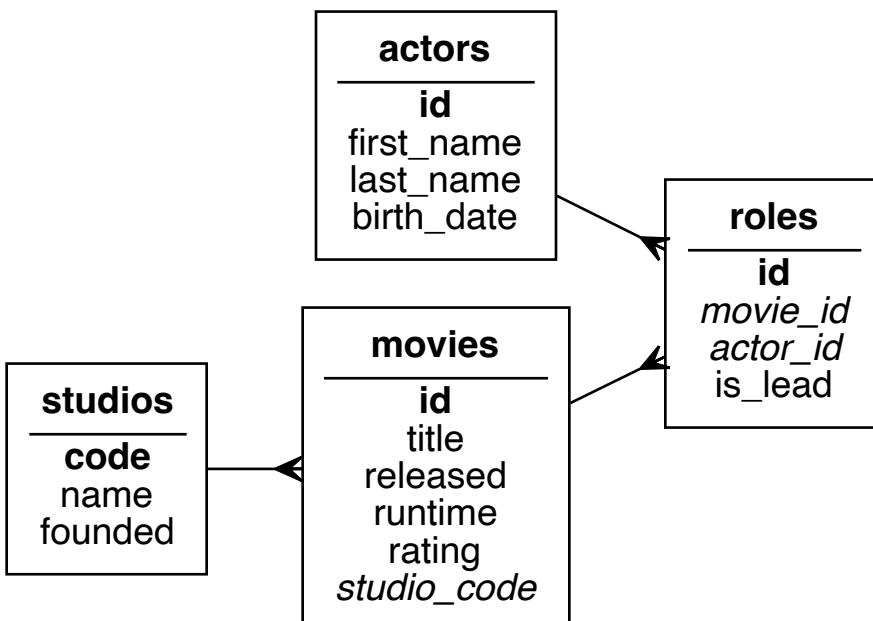
```

Deleting a studio deletes related movies first

However, it's often a good choice to keep the default — if someone really wants to delete a studio, this will force them to fix/delete movies first themselves, preventing casual accidental deletes.

Structuring Relational Data

Movies Database

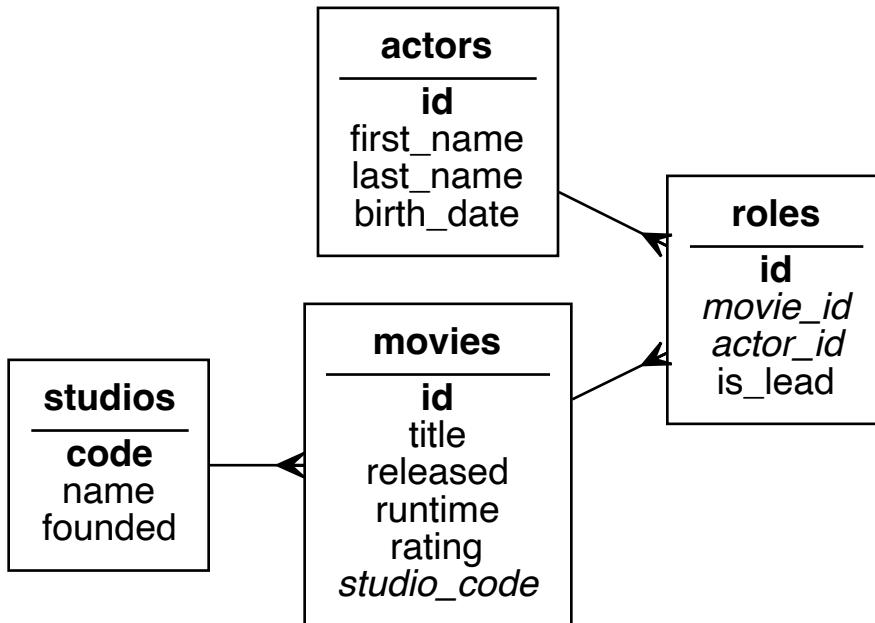


- one studio has many movies
- one actor has many movies
- one movie has many actors

```
CREATE TABLE studios (
    code VARCHAR(10) PRIMARY KEY,
    name VARCHAR(50) NOT NULL UNIQUE,
    founded DATE NOT NULL);
```

```
CREATE TABLE movies (
    id SERIAL PRIMARY KEY,
    title VARCHAR(50) NOT NULL,
    released DATE,
    runtime INT NOT NULL,
    rating VARCHAR(5) NOT NULL
    CHECK (rating IN
        ('G', 'PG', 'PG-13', 'R', 'NC-17')),
    studio_code VARCHAR(10) NOT NULL REFERENCES studios,
    UNIQUE (title, released));
```

Many-to-Many DDL



```
CREATE TABLE movies (
    id SERIAL PRIMARY KEY,
    title VARCHAR(50) NOT NULL,
    released DATE,
    runtime INT NOT NULL,
    rating VARCHAR(5) NOT NULL
    CHECK (rating IN
        ('G', 'PG', 'PG-13', 'R', 'NC-17')),
    studio_code VARCHAR(10) NOT NULL REFERENCES studios,
    UNIQUE (title, released));
```

```
CREATE TABLE roles (
    id SERIAL PRIMARY KEY,
    movie_id INT NOT NULL REFERENCES movies,
    actor_id INT NOT NULL REFERENCES actors,
    is_lead BOOLEAN NOT NULL DEFAULT FALSE);
```

```
CREATE TABLE actors (
    id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    birth_date DATE);
```

```
CREATE TABLE studios (
    code VARCHAR(10) PRIMARY KEY,
    name VARCHAR(50) NOT NULL UNIQUE,
    founded DATE NOT NULL);
```

This would let an actor have multiple roles in the same movie:

```
CREATE TABLE roles (
    id SERIAL PRIMARY KEY,
    movie_id INT REFERENCES movies,
    actor_id INT REFERENCES actors);
```

If you want to prevent that:

```
CREATE TABLE roles (
    id SERIAL PRIMARY KEY,
    movie_id NOT NULL INT REFERENCES movies,
    actor_id NOT NULL INT REFERENCES actors,
    UNIQUE(movie_id, actor_id));
```

Best Practices

Normalization

Normalization is a database design technique which organizes tables in a manner that reduces redundancy and dependency of data.

It divides larger tables to smaller tables and links them using relationships.

Normalization Bad Example

The *color* column is multivalued string (not *scalar*), making it hard to query:

products			
id	item	colors	price
1	frisbee	red, green	5.00
2	top	green	10.00

Normalized Example

colors		products_colors			products		
color		id	color	product_id	id	item	price
red		1	red	1	1	frisbee	5.00
green		2	green	1	2	top	10.00
		3	green	2			

Another Bad Example

<i>purchases</i>				
purchase_id	customer_id	store_id	store_city	item_id
1	1	1	New York	4545
2	1	3	New York	342
3	2	2	San Francisco	222
4	3	1	New York	222
5	3	4	Boston	222

Do you see the dependency, where one value already tells us another value?

store_city is fully dependent on *store_id*.

Normalized Example

<i>stores</i>		<i>purchases</i>			
store_id	store_city	purchase_id	customer_id	store_id	item_id
1	New York	1	1	1	4545
2	San Francisco	2	1	3	342
3	New York	3	2	2	222
4	Boston	4	3	1	222
		5	3	4	222

Design Wrap Up

Designing complex data is a process requiring experience and thought.

You'll get better at it with practice.



SQL Alchemy

Goals

Learn to use object-oriented techniques with relational DBs.

Without writing any SQL.

```
>>> whiskey = Pet(name='Whiskey', species="dog", hunger=50)  
>>> whiskey.hunger  
50  
>>> whiskey.hunger = 20
```

SQLAlchemy intro

- Popular, powerful, Python-based ORM (object-relational mapping)
- Translation service between OOP in our server language and relational data in our database
- Can use by itself, with Flask, or other web frameworks
- Can talk to SQLite, PostgreSQL, MySQL, and more
- You (almost) never have to change code if you change databases

Installing SQLAlchemy

Need the program that lets Python speak PostgreSQL: *psycopg2*

Need the program that provides SQLAlchemy: *flask-sqlalchemy*

```
(venv) $ pip3 install psycopg2-binary  
(venv) $ pip3 install flask-sqlalchemy
```

Sample model

A model like this:

Would turn into this SQL:

Would generate this SQL

```
CREATE TABLE pets (  
    id SERIAL PRIMARY KEY,  
    name VARCHAR(50) NOT NULL UNIQUE,  
    species VARCHAR(30),  
    hunger INTEGER NOT NULL  
)
```

demo/models.py

```
class Pet(db.Model):
    """Pet."""

    __tablename__ = "pets"

    id = db.Column(
        db.Integer,
        primary_key=True,
        autoincrement=True)

    name = db.Column(
        db.String(50),
        nullable=False,
        unique=True)

    species = db.Column(
        db.String(30),
        nullable=True)

    hunger = db.Column(
        db.Integer,
        nullable=False,
        default=20)
```

Setup

demo/models.py

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

def connect_db(app):
    """Connect to database."""

    app.app_context().push()
    db.app = app
    db.init_app(app)

class Pet(db.Model):
    """Pet."""

# ...
```

demo/app.py

```
import os

from flask import Flask, request, redirect, render_template
from flask_debugtoolbar import DebugToolbarExtension

from models import db, connect_db, Pet

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get(
    "DATABASE_URL", 'postgresql://sqla_intro')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.config['SQLALCHEMY_ECHO'] = True

connect_db(app)
```

make sure
to use 3 ///

- *SQLALCHEMY_DATABASE_URI*: Where is your database?
- *SQLALCHEMY_TRACK_MODIFICATIONS*: Always set to `False`
- *SQLALCHEMY_ECHO*: Print SQL to terminal (*helpful for debugging*)

An *environmental variable* is a variable in the shell. These are often used to configure programs.

Setting/viewing in the shell

```
$ export MY_VAR="hello"      # setting an env var
$ echo $MY_VAR              # showing an env var
```

Setting/viewing in Python

```
import os
os.environ["MY_VAR"] = "hello"
print(os.environ["MY_VAR"])
```

demo/app.py

```
# get env var `DATABASE_URL`, using default if not found

app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get(
    "DATABASE_URL", 'postgresql://sqla_intro')
```

Models

demo/models.py

```
class Pet(db.Model):
    """Pet."""

    __tablename__ = "pets"

    id = db.Column(
        db.Integer,
        primary_key=True,
        autoincrement=True)

    name = db.Column(
        db.String(50),
        nullable=False,
        unique=True)

    species = db.Column(
        db.String(30),
        nullable=True)

    hunger = db.Column(
        db.Integer,
        nullable=False,
        default=20)
```

- Models must subclass `db.Model`

- Name SQL table with `__tablename__`

demo/models.py

```
class Pet(db.Model):
    """Pet."""

    __tablename__ = "pets"

    id = db.Column(
        db.Integer,
        primary_key=True,
        autoincrement=True)

    name = db.Column(
        db.String(50),
        nullable=False,
        unique=True)

    species = db.Column(
        db.String(30),
        nullable=True)

    hunger = db.Column(
        db.Integer,
        nullable=False,
        default=20)
```

- Specify the type of column

- Columns can be `NULL` unless `nullable=False`

- Common options:

- `default`
- `unique`
- `primary_key`
- `autoincrement`

Creating the database

- First, create database with `createdb sqlalchemy_intro`
- Create table for each model in IPython:

```
In [1]: %run app.py  
In [2]: db.create_all()
```

- Only have to do once (no effect if tables already exist)
- If you change table schema, drop table and re-run

NOTE **Do I always have to drop the table?**

Dropping all of your tables may seem like an extreme move every time you make a change to your schema. There are tools that can help you update your schema more smoothly. *Database migrations* are a common way to do this, but this topic is beyond our scope.

Using models

```
>>> fluffy = Pet(name='Fluffy', species="cat", hunger=30)  
>>> fluffy.hunger  
30  
  
>>> db.session.add(fluffy)      # required to add to database!  
>>> db.session.commit()        # commit the transaction
```

You only have to use `db.session.add()` to add a new object once – you don't need to keep adding it to the session each time you change it.

Querying

`Pet.query.all()`

```
SELECT *  
FROM pets
```

`Pet.query.get(1)`

```
SELECT *  
FROM pets  
WHERE id = 1
```

`Pet.query.filter_by(species='dog').all()`

Works equivalently

`Pet.query.filter(Pet.species == 'dog').all()`

`Pet.query.filter(Pet.hunger < 10).all()`

```
SELECT *  
FROM pets  
WHERE species = 'dog'
```

```
SELECT *  
FROM pets  
WHERE hunger < 10
```

```
Pet.query.filter(  
    Pet.hunger < 15,  
    Pet.species == 'dog').all()
```

```
SELECT *  
FROM pets  
WHERE hunger < 15  
AND species = 'dog'
```

Fetching records

```
.get(pk)  
Get single record with that primary key value
```

```
.all()  
Get all records as a list
```

```
.first()  
Get first record or None
```

```
.one()  
Get first record, error if 0 or if > 1
```

```
.one_or_none()  
Get first record, error if >1, None if 0
```

Methods

demo/models.py

```
class Pet(db.Model):  
    """Pet.""""  
  
    __tablename__ = "pets"  
  
    id = ...  
    name = ...  
    species = ...  
    hunger = ...  
  
    def greet(self):  
        """Greet using name.""""  
  
        name = self.name  
        species = self.species or "thing"  
        return f"I'm {name} the {species}"  
  
    def feed(self, units=10):  
        """Nom nom nom.""""  
  
        self.hunger -= units  
        self.hunger = max(self.hunger, 0)
```

Working with model

```
>>> fluffy.greet()  
'I am Fluffy the cat'  
  
>>> fluffy.feed()  
>>> fluffy.hunger  
>>> db.session.commit() # save
```

Class methods

NOTE Class Methods

- Most methods are “instance methods”
 - These are called on an instance of a class (ie, a single cat)
 - They can refer to/change attributes of that instance
- Some methods are “class methods”
 - They are called on the class itself
 - They can’t refer to instance attributes
 - Often used as “factories” to return instances

demo/models.py

```
class Pet(db.Model):
    """Pet."""

    __tablename__ = "pets"

    id = ...
    name = ...
    species = ...
    hunger = ...

    def greet(self): ...
    def feed(self, units=10): ...

    @classmethod
    def get_by_species(cls, species):
        """Get all pets by species."""

        return Pet.query.filter_by(
            species=species).all()
```

Working with model

```
>>> Pet.get_by_species("dog")
[<Pet ...>, <Pet ...>]
```

SQLAlchemy with Flask

Flask-SQLAlchemy

- Add-on product to integrate Flask and SQLAlchemy
- Benefits
 - Ties SQLAlchemy session to Flask response
 - Simplifies finding things in SQLAlchemy API
- With Flask-SQLAlchemy, all useful methods are on *db*.
 - With vanilla SQLAlchemy, stuff is spread all over
 - There are useful web-related methods, like `Pet.objects.get_or_404(pk)`

Demo

demo/app.py

```
@app.get("/")
def list_pets():
    """List pets and show add form."""

    pets = Pet.query.all()
    return render_template("list.html", pets=pets)
```

demo/templates/list.html

```
<ul>
    {% for pet in pets %}
        <li><a href="/{{ pet.id }}">{{ pet.name }}</a></li>
    {% endfor %}
</ul>
```

demo/templates/list.html

```
<form method="POST">
    <p>Name: <input name="name"></p>
    <p>Species: <input name="species"></p>
    <p>Hunger: <input name="hunger"></p>
    <button>Save</button>
</form>
```

demo/app.py

```
@app.post("/")
def add_pet():
    """Add pet and redirect to list."""

    name = request.form['name']
    species = request.form['species']
    hunger = request.form['hunger']
    hunger = int(hunger) if hunger else None

    pet = Pet(name=name, species=species, hunger=hunger)
    db.session.add(pet)
    db.session.commit()

    return redirect(f"/{pet.id}")
```

demo/app.py

```
@app.get("/<int:pet_id>")
def show_pet(pet_id):
    """Show info on a single pet."""

    pet = Pet.query.get_or_404(pet_id)
    return render_template("detail.html", pet=pet)
```

```
demo/templates/detail.html
```

```
<h1>{{ pet.name }}</h1>

<p>Species: {{ pet.species }}</p>
<p>Hunger: {{ pet.hunger }}</p>
<p>{{ pet.name }} says {{ pet.greet() }}!</p>

<a href="/">Go back</a>
```

```
demo/seed.py
```

```
"""Seed file to make sample data for pets db."""

from models import Pet, db
from app import app

# Create all tables
db.drop_all()
db.create_all()

# If table isn't empty, empty it
# An alternative if you don't want to drop
# and recreate your tables:
# Pet.query.delete()

# Add pets
whiskey = Pet(name='Whiskey', species="dog")
bowser = Pet(name='Bowser', species="dog", hunger=10)
spike = Pet(name='Spike', species="porcupine")

# Add new objects to session, so they'll persist
db.session.add(whiskey)
db.session.add(bowser)
db.session.add(spike)

# Commit--otherwise, this never gets saved!
db.session.commit()
```

Coming Up

SQLAlchemy II: relationships and joins

Learning more

SQLAlchemy Docs <<http://docs.sqlalchemy.org/en/latest/>>

Flask-SQLAlchemy Docs <<https://pythonhosted.org/Flask-SQLAlchemy/>>

Rithm SQLA Cheatsheet (in the demo folder)

Steps To Get Your Application To Connect To Your Database Using Flask-SQLAlchemy

1. Install psycopg2 and flask-sqlalchemy modules and all other pertinent dependencies that your application needs (**Please ensure that you are in your virtual environment before you run the below commands**) :
 - a. pip3 install psycopg2-binary
 - b. pip3 install flask-sqlalchemy
 - c. pip3 install flask
 - d. pip3 install

2. Set up your **models.py** file (Part #1)

```
a. from flask_sqlalchemy import SQLAlchemy  
  
b.  
  
c. db = SQLAlchemy() #create a SQLAlchemy instance  
  
d. #create a function that ties your db object to your app object  
  
e. #thus, allows your flask app to connect to the specified db  
  
f. def connect_db(app):  
  
g.     """Connect to database."""  
  
h.     app.app_context().push()  
  
i.     db.app = app  
  
j.     db.init_app(app)  
  
k.
```

3. Set up your **models.py** file (Part #2)

- a. Create all your Model Classes and ensure that each class inherits from db.model. For example:

```
class Pet(db.Model):  
    """Pet."  
  
    __tablename__ = "pets"  
  
    id = db.Column(  
        db.Integer,  
        primary_key=True,  
        autoincrement=True)  
  
    name = db.Column(  
        db.String(50),  
        nullable=False,  
        unique=True)
```

4. Set up your `app.py`

- a. Ensure that you import all the usual modules, classes, functions, objects that you need . For example :

```
import os
from flask import Flask, request, redirect, render_template
from flask_debugtoolbar import DebugToolbarExtension
```

- b. Ensure that you import **ALL** your **Model Classes**, your **db object** and your **connect_db** function. For example :

```
from models import db, connect_db, Pet, Superhero, OtherModelClasses...
```

- c. Provide the needed SQLAlchemy config values to your app : For example :

```
app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get(
    "DATABASE_URL", 'postgresql:///sqla_intro')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.config['SQLALCHEMY_ECHO'] = True
```

Note, the most important of the 3 config values is the **SQLALCHEMY_DATABASE_URI**. This allows you to set the name of the database that you want to connect to. Pay attention to the syntax :
postgresql://database_name

- d. Call the `connect_db` function . For Example :

```
connect_db(app)
```

This allows your app to connect to your db specified in the `SQLALCHEMY_DATABASE_URI` value

- e. Create all your get and post routes. For example

```
@app.get("/")
def show_homePage():
    """Display Home Page"""

// Listens to a POST request
@app.post("/")
def process_incomming_data():
```

5. Create your database in psql. Ensure your db name matches the name that you specified in `app.config['SQLALCHEMY_DATABASE_URI']`
6. Create your TABLES
 - a. Go to ipython and the below commands :
`%run app.py`
`db.create_all()`
7. Start up your server
 - a. `flask run -p 5001`



SQLAlchemy Cheat Sheet

Making Models

models.py

```
from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy()

class Pet(db.Model):
    __tablename__ = "pets"

    # can specify multi-column unique or check constraints like:
    # __table_args__ = (
    #     db.UniqueConstraint("col1", "col2"),
    #     db.CheckConstraint("born <= died") )

    id = db.Column(
        db.Integer,
        primary_key=True,
        autoincrement=True)
    name = db.Column(
        db.String(50),
        db.CheckConstraint('len(name) >= 5'),
        nullable=False,
        unique=True)
    species = db.Column(
        db.String(30),
        nullable=True,
        default="cat")
    hunger = db.Column(
        db.Integer,
        nullable=False,
        default=20)
    created_at = db.Column(
        db.DateTime,
        nullable=False,
        default=db.func.now())
```

SQLAlchemy types:

- *Integer, String(len), Text, Boolean, DateTime, Float, Numeric*

Field options (*all default to False*):

- *primary_key, autoincrement, nullable, unique, default* (value or callback)

Creating/Dropping Tables:

- `db.create_all()`, `db.drop_all()`

Making and Deleting Instances

Making an instance and adding (*only need to do 1st time adding*):

- `fluffy = Pet(name="Fluffy", species="cat")`
- `db.session.add(fluffy)` or `db.session.add_all([fluffy, bob])`

Deleting instance or deleting all matching data:

- `fluffy.query.delete()` or `Pet.query.filter(...).delete()`

Getting and Filtering

Getting record by primary key:

- `fluffy = Pet.query.get("fluffy")` or `Pet.query.get_or_404("fluffy")`

Simple Filtering: (*returns a “query”, not the answer—see fetching below*)

- `Pet.query.filter_by(species="cat")`

Flexible filtering: (*returns “query”*)

- `Pet.query.filter(Pet.species == "dog")`
- also: `Pet.hunger != 10`, `.filter(Pet.hunger < 10)`
- also: `Pet.name.like('%uff%')`, `Pet.name.like('%uff%')`, `Pet.hunger.in_([2, 7])`
- OR: `expr | expr`, AND: `expr & expr`, NOT: `~ expr`

Grouping, Ordering, Offsetting, Limiting:

- `.group_by('species', 'age')`
- `.group_by('species').having(db.func.count() > 2)`
- `.order_by('species', 'age')`, `.offset(10)`, `.limit(10)`

Getting lightweight tuples, not instances of model class:

- `db.session.query(Pet.name, Pet.hunger) → [("fluffy", 10), ("bob", 3)]`

Fetching:

- `query.get(pk)`
- `query.get_or_404(pk)` (*Flask-specific: get or raise 404*)
- `query.all()` (*get all as list*)
- `query.first()` (*get first record or None*)
- `query.one()` (*get first record, error if 0 or if > 1*)
- `query.one_or_none()` (*get first record, error if > 1, None if 0*)
- `query.count()` (*returns # of elements*)

Transactions

“Flushing” (sending SQL to database, but doesn’t commit transaction yet)

- `db.session.flush()`

Committing or rolling back transactions:

- `db.session.commit()`, `db.session.rollback()`

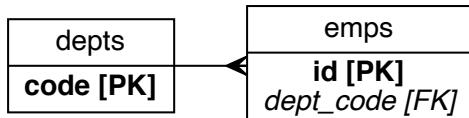
Handling Errors

Import SQLA exception classes from `sqlalchemy.exc`, like this:

```
from sqlalchemy import exc

try:
    User.query.delete() # delete all users
except exc.IntegrityError:
    print("Cannot delete users because of ref integrity!")
```

Relationships



```
class Employee(db.Model):
    __tablename__ = "emps"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    dept_code = db.Column(db.Text, db.ForeignKey('depts.dept_code'))
    # remember to make foreign keys `nullable=False` if required!

class Department(db.Model):
    __tablename__ = "depts"
    dept_code = db.Column(db.Text, primary_key=True)
    employees = db.relationship('Employee', backref='department')
```

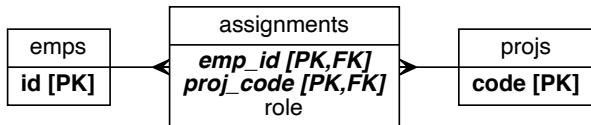
Can navigate like:

```
>>> jane.department      # <Department finance>
>>> finance.employees   # [<Employee jane>, <Employee bob>]
```

Can add/remove/clear foreign key data via relationships:

```
>>> finance.employees.append(bob)
>>> finance.employees.remove(bob)
>>> finance.employees.clear()
```

Many to Many Relationships



```
class Employee(db.Model):
    __tablename__ = "emps"
    id = db.Column(db.Integer, primary_key=True, auto_increment=True)
    # can nav from employee to projects, or project to employees
    projects = db.relationship(
        'Project', secondary='assignments', backref='employees')

class Project(db.Model):
    __tablename__ = "projs"
    code = db.Column(db.Text, primary_key=True)

class Assignment(db.Model):
    __tablename__ = "assignments"
    emp_id = db.Column(
        db.Integer,
        db.ForeignKey("emps.id"),
        primary_key=True)
    proj_code = db.Column(
        db.Text,
        db.ForeignKey("projs.code"),
        primary_key=True)
    role = db.Column(db.Text, nullable=False, default='')

    # if you want to nav emp<->assignment and project<->assignment
    project = db.relationship("Project", backref="assignments")
    project = db.relationship("Employee", backref="assignments")
```

Can navigate like:

```
>>> jane.projects      # [<Project A>, <Project B>]
>>> proj_a.employees  # [<Employee 1>, <Employee 2>]

>>> jane.assignments  # [<Assignment jane A>, <Assignment jane B>]
>>> proj_a.assignments # [<Assignment jane A>, <Assignment bob A>]
>>> asn_jane_a.employee # <Employee 1>
>>> asn_jane_a.project  # <Project A>
```

Can add/edit/remove foreign key data via relationships:

```
>>> jane.projects.add(project_a)
>>> jane.projects.remove(project_a)
>>> jane.projects.clear()

>>> jane.assignments.add(Assignment(proj_code='a', role='Chair'))
```

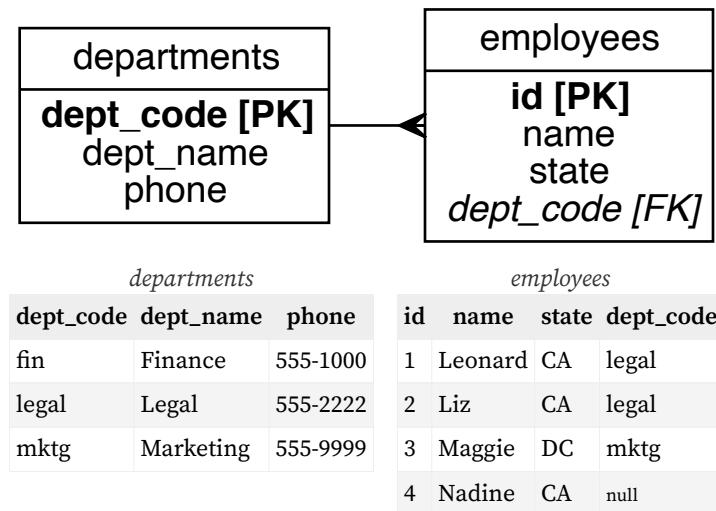
Written and maintained by Joel Burton <joel@joelburton.com> for Rithm School.

SQL Alchemy Associations

Goals

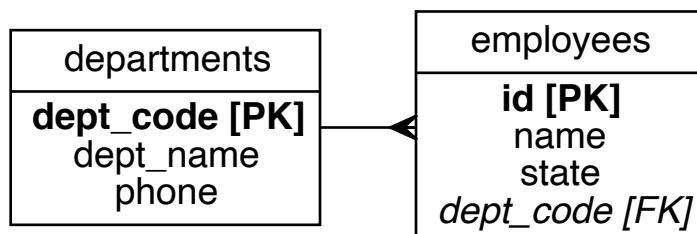
- Translate relationships between tables to relationships between Python classes
- Deeper dive into SQLAlchemy querying
- Compare different approaches to querying in SQLAlchemy

Example: employees & departments



Relationships

Related tables



```
demo/models.py
```

```
class Department(db.Model):
    """Department. A department has many employees."""

    __tablename__ = "departments"

    dept_code = db.Column(
        db.Text,
        primary_key=True)

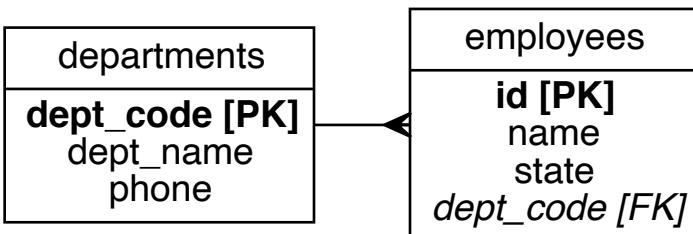
    dept_name = db.Column(
        db.Text,
        nullable=False,
        unique=True)

    phone = db.Column(db.Text)
```

```
demo/models.py
```

```
class Employee(db.Model): # ...
    dept_code = db.Column(
        db.Text,
        db.ForeignKey('departments.dept_code'))
```

- Add an actual field, *dept_code*
- *ForeignKey* makes primary/foreign key relationship
 - Parameter is string “tablename.fieldname”
 - Database will handle referential integrity

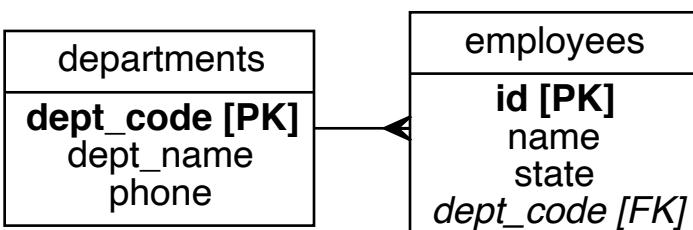


```
demo/models.py
```

```
class Employee(db.Model): # ...
    dept_code = db.Column(
        db.Text,
        db.ForeignKey('departments.dept_code'))

    dept = db.relationship('Department')
```

- *relationship* allows SQLAlchemy to “navigate” this relationship
 - Using the name *dept* on an *Employee*



```
class Department(db.Model):    # ...
    employees = db.relationship('Employee')
```

- Can get list of employee objects from dept with `.employees`

TIP Backreference

You can specify both “ends” of a database relationship as shown above: going from an employee to their department with `.dept` and from a department to their employees with `.employees`.

SQLAlchemy also allows a shortcut that some people prefer—that you can just declare one relationship, and note the “backreference” for it.

To do this, you wouldn’t need `.employees` attribute on the `Department` class and could just put this on the `Employee` class:

```
dept = db.relationship('Department', backref='employees')
```

Both give the same results—you can navigate from an employee to their department and from a department to its employees, so which you use is a matter of aesthetic preference.

Navigating

```
class Employee(db.Model):    # ...
    dept = db.relationship('Department')

class Department(db.Model):  # ...
    employees = db.relationship('Employee')
```

can navigate `emp → dept` with `.dept`

```
>>> leonard = Employee.query.filter_by(name='Leonard').one()

>>> leonard.dept_code
'legal'

>>> leonard.dept
<Department legal Legal>
```

can navigate `dept → emp` with `.employees`

```
>>> legal = Department.query.get('legal')

>>> legal.employees
[<Employee 1 Leonard CA>, <Employee 2 Liz CA>]
```

Short-hand defining with backref

longer way

```
class Employee(db.Model):    # ...
    dept = db.relationship('Department')

class Department(db.Model):  # ...
    employees = db.relationship('Employee')
```

short-hand way using backref

```
class Employee(db.Model):    # ...
    dept = db.relationship('Department', backref='employees')

class Department(db.Model):  # ...
    # don't need to specify here; will auto-magically get
    # .employees to navigate to employees because of backref
```

There are subtle benefits to the short-hand method: **always do it this way.**

Using relationships

Goal: “Show phone directory of employees and their dept.”

Name	Department	Phone
Leonard	Legal	555-2222
Liz	Legal	555-2222
Maggie	Marketing	555-9999
Nadine	-	-

Navigating

demo/models.py

```
def phone_dir_nav():
    """Phones of emps & depts."""

    emps = Employee.query.all()

    for emp in emps:  # [<Emp>, <Emp>]
        if emp.dept is not None:
            print(
                emp.name,
                emp.dept.dept_code,
                emp.dept.phone,
            )
        else:
            print(emp.name, "--", "--")
```

- Yay! So pretty! So easy!

This is inefficient because SQLAlchemy fires off several queries:

- one for the list of employees
- one for *each* department

Querying

demo/models.py

```
class Employee(db.Model):
    """Employee."""

    __tablename__ = "employees"

    id = db.Column(
        db.Integer,
        primary_key=True,
        autoincrement=True)

    name = db.Column(
        db.Text, nullable=False,
        unique=True)

    state = db.Column(
```

```
SELECT * FROM employees
WHERE name = 'Liz';
```

shorter form, for simple cases

```
Employee.query.filter_by(name='Liz')
```

longer form, can use other operators

```
Employee.query.filter(Employee.name == 'Liz')
Employee.query.filter(Employee.id > 1)
```

Chaining

```
new_emps = Employee.query.filter(Employee.id > 1)

just_ca = new_emps.filter(Employee.state == 'CA')
```

Remember: nothing runs until we get results:

```
>>> just_ca
<flask_sqlalchemy.BaseQuery at 0x105234750>

>>> just_ca.all()
[<Employee 2 Liz CA>, <Employee 4 Nadine CA>]
```

More flexible filtering

```
SELECT * FROM employees
WHERE name = 'Liz';
```

Simple version: *ClassName.query*

```
Employee.query.filter_by(name='Liz')
Employee.query.filter(Employee.name == 'Liz')
```

More flexible version: `db.session(thing, ...).query`

```
db.session.query(Employee).filter_by(name='Liz')
db.session.query(Employee).filter(Employee.name == 'Liz')
```

This doesn't seem to gain us anything, but this general form of `db.session.query(...)` allows us to query more flexibly than a single model class.

Returning rows

```
SELECT id, name FROM employees;
```

```
>>> emps = db.session.query(Employee.id, Employee.name).all()
[(1, 'Leonard'), (2, 'Liz'), (3, 'Maggie'), (4, 'Nadine')]

>>> type(emps[0])
sqlalchemy.engine.row.Row
```

This is a list of rows (it may *look like* a tuple, but don't be fooled!).

This is useful if we just need to display data and don't need to call methods on our models (we'd need full instances for that).

A row can be turned into a tuple:

```
>>> emp_tuples = [(e.id, e.name) for e in emps]

>>> type(emp_tuples[0])
tuple
```

Fetching records

`.all()`

Get all records

`.first()`

Get first record, ok if there is none

`.one()`

Get only record, error if 0 or more than 1

`.one_or_none()`

Get only record, error if >1, None if 0

`.count()`

Get number of records found without fetching all

Get by PK

```
>>> Department.query.filter_by(dept_code='fin').one()
<Department fin Finance>
```

```
>>> Department.query.get('fin')
<Department fin Finance>

>>> Department.query.get_or_404('fin')
<Department fin Finance>
```

Common Operators

Operators

```
Employee.query.filter(Employee.name == 'Jane')

Employee.query.filter(Employee.name != 'Jane')

Employee.query.filter(Employee.id > 65)

Employee.query.filter(Employee.name.like('%Jane%'))      # LIKE

Employee.query.filter(Employee.id.in_([22, 33, 44]))    # IN ()

Employee.query.filter(Employee.state == None)           # IS NULL
Employee.query.filter(Employee.state.is_(None))        # also IS NULL

Employee.query.filter(Employee.state != None)          # IS NOT NULL
Employee.query.filter(Employee.state.isnot(None))       # also IS NOT NULL
```

```
q = Employee.query
```

AND:

```
q.filter(Employee.state == 'CA', Employee.id > 65)

q.filter( (Employee.state == 'CA') & (Employee.id > 65) )
```

OR:

```
q.filter( db.or_(Employee.state == 'CA', Employee.id > 65) )

q.filter( (Employee.state == 'CA') | (Employee.id > 65) )
```

NOT:

```
Employee.query.filter( db.not_(Employee.state.in_('CA', 'OR')) )

Employee.query.filter( ~ Employee.state.in_('CA', 'OR') )
```

Learning more

Self-learning

```
q = Employee.query  
q.group_by('state')  
q.group_by('state').having(db.func.count(Employee.id) > 2)  
q.order_by('state')  
q.offset(10)  
q.limit(10)
```

All described at Query Docs <http://docs.sqlalchemy.org/en/rel_1_0/orm/query.html#sqlalchemy.orm.Query.offset>

Learning more

SQLAlchemy Docs <<http://docs.sqlalchemy.org/en/latest/>>

Flask-SQLAlchemy Docs <<https://pythonhosted.org/Flask-SQLAlchemy/>>



SQL Alchemy Many-to-Many

Goals

- Make explicit joins while querying in SQLAlchemy
- Work with many-to-many relationships in SQLAlchemy

Navigating relationships

demo/models.py

```
def phone_dir_nav():
    """Phones of emps & depts."""

    emps = Employee.query.all()

    for emp in emps: # [<Emp>, <Emp>]
        if emp.dept is not None:
            print(
                emp.name,
                emp.dept.dept_code,
                emp.dept.phone,
            )
        else:
            print(emp.name, "--", "--")
```

Works – but runs several queries:

- one to get the employees
- one for each department

```
SELECT * FROM employees;

SELECT * FROM department
  WHERE dept_code = 'legal';

SELECT * FROM department
  WHERE dept_code = 'mktg';
```

Joining

Can also specify joins directly

- Can be more explicit about what you want to get
- Connect tables without defined relationships
- Can control inner/outer/cross joins

demo/models.py

```
def phone_dir_join():
    """Phones of emps & depts."""

    emps = db.session.query(
        Employee.name,
        Department.dept_name,
        Department.phone,
    ).outerjoin(Department).all()

    # is list of tuples of raw data
    # [(n, d, p), (n, d, p)]

    for name, dept, phone in emps:
        print(name, dept, phone)
```

Performs as one outer join, returning raw data & not instances:

```
SELECT name, dept_name, phone
FROM employees AS e
LEFT JOIN departments AS d
ON e.dept_code = d.dept_code;
```

demo/models.py

```
def phone_dir_join_instances():
    """Phones of emps & depts."""

    emps = db.session.query(
        Employee,
        Department,
    ).outerjoin(Department).all()

    # is list of tuples of instances
    # [(<E>, <D>), (<E>, <D>)]

    for emp, dept in emps:
        if dept:
            print(
                emp.name,
                dept.dept_name,
                dept.phone,
            )
        else:
            print(emp.name, "–", "-")
```

Performs as one outer join, returning instances:

```
SELECT *
FROM employees AS e
LEFT JOIN departments AS d
ON e.dept_code = d.dept_code;
```

demo/models.py

```
def phone_dir_join_innerjoin():
    """Phones of emps & depts."""

    emps = db.session.query(
        Employee,
        Department,
    ).join(Department).all()

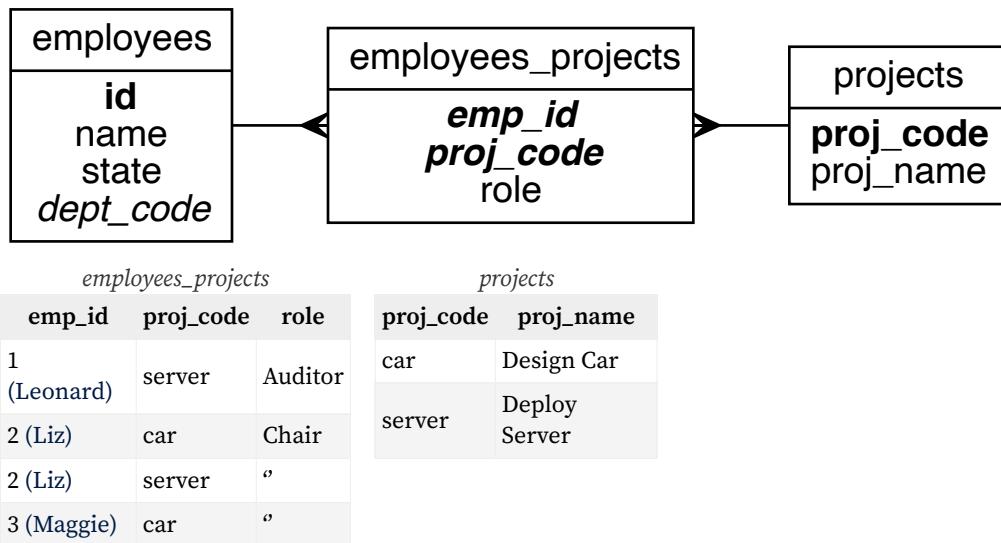
    # is list of tuples of instances
    # [(<E>, <D>), (<E>, <D>)]

    for emp, dept in emps:
        print(
            emp.name,
            dept.dept_name,
            dept.phone,
        )
```

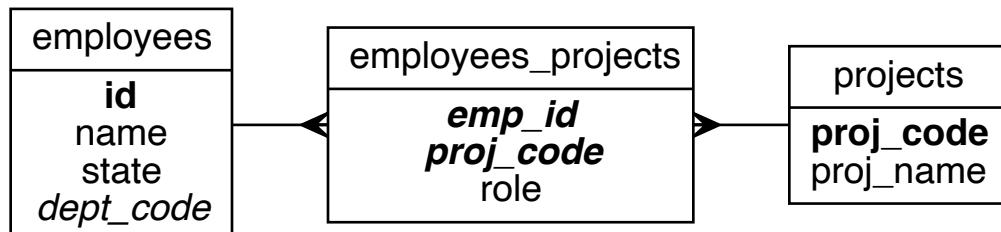
Performs as one inner join, returning raw data:

```
SELECT name, dept_name, phone
FROM employees AS e
JOIN departments AS d
ON e.dept_code = d.dept_code;
```

Many-to-Many relationships



Project



demo/models.py

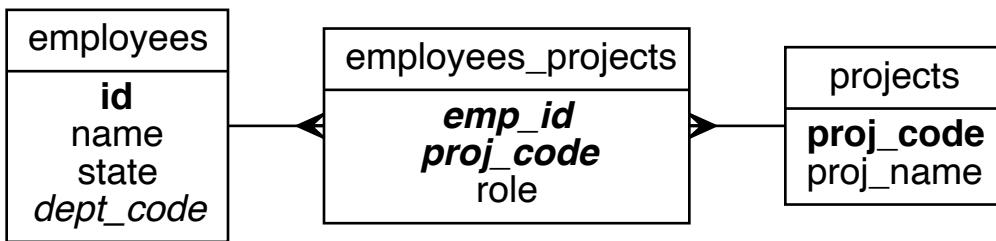
```
class Project(db.Model):
    """Project. Employees can be assigned to this."""

    __tablename__ = "projects"

    proj_code = db.Column(
        db.Text,
        primary_key=True)

    proj_name = db.Column(
        db.Text,
        nullable=False,
        unique=True)
```

EmployeeProject



demo/models.py

```
class EmployeeProject(db.Model):
    """Mapping of an employee to a project."""

    __tablename__ = "employees_projects"

    emp_id = db.Column(
        db.Integer,
        db.ForeignKey("employees.id"),
        primary_key=True)

    proj_code = db.Column(
        db.Text,
        db.ForeignKey("projects.proj_code"),
        primary_key=True)

    role = db.Column(
        db.Text,
        nullable=False,
        default='')
```

Relationships

demo/models.py

```
class Employee(db.Model):    # ...
    # direct navigation: emp -> employee_project & back
    assignments = db.relationship('EmployeeProject', backref='employee')
```

demo/models.py

```
class Project(db.Model):    # ...
    # direct navigation: proj -> employee_project & back
    assignments = db.relationship('EmployeeProject', backref='project')
```

```
>>> liz = Employee.query.get(2)
>>> liz.assignments
[<EmployeeProject 2, server>, <EmployeeProject 2, car>]

>>> car = Project.query.get('car')
>>> car.assignments
[<EmployeeProject 2, car>, <EmployeeProject 3, car>]
```

These “stop at” *EmployeeProject*; but can go on:

```
>>> liz.assignments
[<EmployeeProject 2, server>, <EmployeeProject 2, car>]

>>> liz.assignments[0].project
<Project server Deploy Server>
```

“Through” Relationships

demo/models.py

```
class Employee(db.Model):    # ...
    # direct navigation: emp -> project & back
    projects = db.relationship(
        'Project', secondary='employees_projects', backref='employees')
```

```
>>> liz.projects
<Project server Deploy Server>, <Project car Design Car>

>>> car.employees
[<Employee 2 Liz CA>, <Employee 3 Maggie DC>]
```

These go “through” *employees_projects* to get result

Fine (& sometimes useful) to have both:

demo/models.py

```
class Employee(db.Model):    # ...
    # direct navigation: emp -> employee_project & back
    assignments = db.relationship('EmployeeProject', backref='employee')

    # direct navigation: emp -> project & back
    projects = db.relationship(
        'Project', secondary='employees_projects', backref='employees')
```

Adding To Relationships

Can append to “through” relationship directly:

```
>>> nadine = Employee.query.get(4)
>>> nadine.projects.append(car)
>>> db.session.commit()
>>> nadine.assignments
[<EmployeeProject 4, car>]
```

Can append to middle table:

```
>>> nadine.assignments.append(
...     EmployeeProject(proj_code='server', role='Tester'))
>>> db.session.commit()
>>> nadine.projects
[<Project server Deploy Server>, <Project car Design Car>]
```

Can add a new middle record directly:

```
>>> m_server = EmployeeProject(emp_id=3, proj_code='server', role='Tester')

>>> db.session.add(m_server)    # need to do this now, though
>>> db.session.commit()
```

Useful if you only have keys, not a user or project



Flask Forms

You can make forms yourself!

- Write the HTML (including labels, etc)
- Write server-side validating code for each field
- Add logic for form for showing validation messages
- Add protection against security attacks

This is tedious.

WTForm

WTForm is a Python library providing:

- Validation
- HTML production
- Security

Flask-WTF

Flask-WTF is built on top of that, and adds integration with Flask (get data from request, etc)

Install

```
(venv) $ pip3 install flask-wtf
```

Basic example

Defining form class

```
demo/forms.py
```

```
from flask_wtf import FlaskForm
from wtforms import StringField, FloatField
```

```
demo/forms.py
```

```
class AddSnackForm(FlaskForm):
    """Form for adding snacks."""

    name = StringField("Snack Name")
    price = FloatField("Price in USD")
```

Form route handler

demo/app.py

```
from forms import AddSnackForm

@app.route("/add", methods=["GET", "POST"])
def add_snack():
    """Snack add form; handle adding."""

    form = AddSnackForm()

    if form.validate_on_submit():
        name = form.name.data
        price = form.price.data
        # do stuff with data/insert to db

        flash(f"Added {name} at {price}")
        return redirect("/add")

    else:
        return render_template(
            "snack_add_form.html", form=form)
```

This validates submitted form or passes instance of form to template.

Add form template

demo/templates/snack_add_form.html

```
<form id="snack-add-form" method="POST">
    {{ form.hidden_tag() }} <!--add type=hidden form fields -->

    {% for field in form
        if field.widget.input_type != 'hidden' %}

        <p>
            {{ field.label }}
            {{ field }}

            {% for error in field.errors %}
                {{ error }}
            {% endfor %}
        </p>

    {% endfor %}

    <button type="submit">Submit</button>
</form>
```

Models vs forms

- SQLAlchemy provides **model**: class for logical object
- WTForm provides **form class**

- A single model may have different forms
 - Not all fields on add form might appear on edit form
 - Different validation might apply on add/edit
 - Different kinds of users (staff *v* manager) can edit different fields

You'll often take the form data and create/edit an SQLAlchemy object.

Field types

BooleanField

Normally appears as a checkbox

DateField / DateTimeField

Date or Date & Time

IntegerField / FloatField

Numeric types

StringField / TextAreaField

Single line of text / larger text area

Selection from choices

RadioField

Series of radio buttons from *choices*

SelectField

Drop-down menu from *choices*

SelectMultipleField

Multi-select box from *choices*

```
weather = SelectField('Weather',
    choices=[('rain', 'Rain'), ('nice', 'Nice Weather')]
)
```

To convert result to integer:

```
priority = SelectField('Priority Code',
    choices=[(1, 'High'), (2, 'Low')],
    coerce=int
)
```

Can set dynamic choices:

forms.py

```
class AddFriendForm(FlaskForm):
    """Form to pick a friend."""

    friend = SelectField("Friend", coerce=int)
```

app.py

```
@app.get("/get-friend")
def handle_friend_form():
    """Handle the add-friend form."""

    form = AddFriendForm()

    # get current list of users, like:
    # [(1, "Joel"), (2, "Elie")]
    users = [(u.id, u.name) for u in User.query.all()]

    # dynamically set friend choices
    form.friend.choices = users
```

NOTE Tuples and WTForms

You may recall that we previously mentioned in the SQLAlchemy Associations lecture that you can return tuples from a query by using the following code:

```
>>> db.session.query(Employee.id, Employee.name).all()
[(1, 'Leonard'), (2, 'Liz'), (3, 'Maggie'), (4, 'Nadine')]
```

and you may be wondering why we've purposely constructed tuples for our AddFriendForm choices above, if SQLAlchemy can just return tuples for us automatically. We simplified things a bit in the Associations lecture—the data structure returned by SQLAlchemy is in fact a Row. Rows look exactly like tuples, and mostly behave like them, but not always. One of those times is when dealing with WTForms: WTForms inspects the incoming data structure to see if it's either a list or a tuple, and won't accept a SQLAlchemy Row. So, we need to explicitly convert our data to be in tuple form before passing it to WTForms.

Validation

WTForm provides “validators”:

demo/forms.py

```
from wtforms.validators import InputRequired, Optional, Email
```

demo/forms.py

```
class UserForm(FlaskForm):
    """Form for adding/editing friend."""

    name = StringField(
        "Name",
        validators=[InputRequired()])

    email = StringField(
        "Email Address",
        validators=[Optional(), Email()])
```

See <https://wtforms.readthedocs.io/en/2.3.x/validators/#built-inValidators>
<<https://wtforms.readthedocs.io/en/2.3.x/validators/#built-inValidators>>

Update Forms

demo/app.py

```
@app.route("/users/<int:uid>/edit", methods=["GET", "POST"])
def edit_user(uid):
    """Show user edit form and handle edit."""

    user = User.query.get_or_404(uid)
    form = UserForm(obj=user)

    if form.validate_on_submit():
        user.name = form.name.data
        user.email = form.email.data
        db.session.commit()
        flash(f"User {uid} updated!")
        return redirect(f"/users/{uid}/edit")

    else:
        return render_template("user_form.html", form=form)
```

Passing `obj=data-obj` provides form with defaults from object

CSRF Security

Cross-Site Request Forgery

A form on any site can submit to any other site!

on a webpage at evilhacker.com

```
<form action="http://yourbank.com/transfer" method="POST">
    <input type="hidden" name="from" value="your-acct">
    <input type="hidden" name="to" value="my-acct">
    <input type="hidden" name="amt" value="$1,000,000">
    <button type=submit>I Love Kittens!</button>
</form>
```

Therefore, most sites use a “CSRF Token”:

- This is generated by the server when a form is shown
- It is included in the HTML of the form
- It is checked by the server on form submission

Flask-WTF uses CSRF out-of-the-box:

- All forms include a hidden CSRF field
- The `validate_on_submit` method checks for this

Hidden CSRF field

demo/templates/snack_add_form.html

```
<form id="snack-add-form" method="POST">
    {{ form.hidden_tag() }} <!--add type=hidden form fields -->

    {% for field in form
        if field.widget.input_type != 'hidden' %}

        <p>
            {{ field.label }}
            {{ field }}

            {% for error in field.errors %}
                {{ error }}
            {% endfor %}
        </p>

    {% endfor %}

    <button type="submit">Submit</button>
</form>
```

Testing

For tests to work, need to disable CSRF checking in tests:

demo/tests.py

```
app.config['WTF_CSRF_ENABLED'] = False
```

demo/tests.py

```
class SnackViewsTestCase(TestCase):
    """Tests for views for Snacks."""

    def test_snack_add_form(self):
        with app.test_client() as client:
            resp = client.get("/add")
            html = resp.get_data(as_text=True)

            self.assertEqual(resp.status_code, 200)
            self.assertIn('<form id="snack-add-form"'', html)

    def test_snack_add(self):
        with app.test_client() as client:
            d = {"name": "Test2", "price": 2}
            resp = client.post("/add", data=d, follow_redirects=True)
            html = resp.get_data(as_text=True)

            self.assertEqual(resp.status_code, 200)
            self.assertIn("Added Test2 at 2", html)
```

Best practices

- Make distinct add/edit forms, if sensible
- Add lots of form validation, if appropriate
- All non-GET routes return *redirect* (not *render_template*) on success

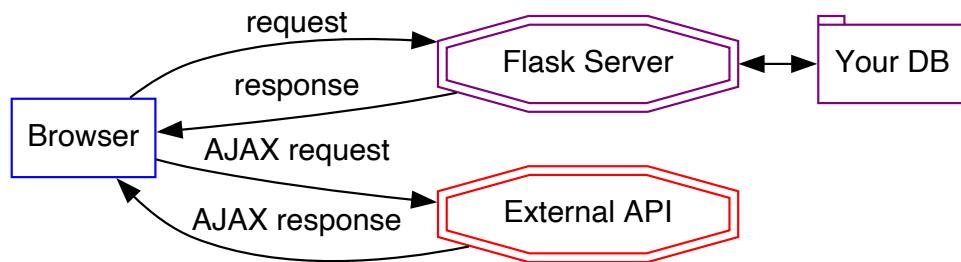
Flask with External Web APIs

API Requests

Two ways to talk with APIs:

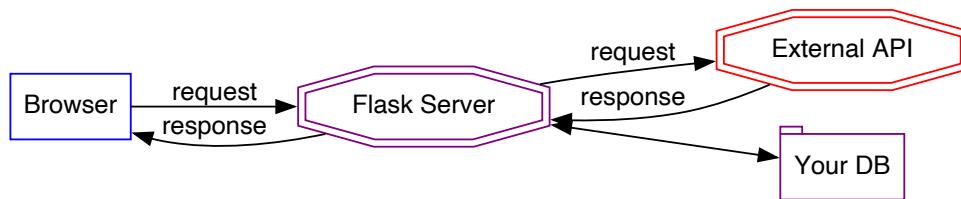
- Client-side requests (via AJAX)
- Server-side requests

Why Use Client-Side Requests?



- You can do easily using AJAX libraries
- Don't have to involve Flask in the API
- Can be faster: browser could talk directly to, say, Google Maps

Why Use Server-Side Requests?



- Same-Origin Policy may prevent browser requests
- Easier for server to store/process the data
 - eg have Flask request restaurants and store in database
- Need password to access API
 - If API uses password & we make request in browser JS, people could learn password from reading JS

iTunes API

```
$ curl -i  
> 'https://itunes.apple.com/search?term=billy+bragg&limit=3'  
{  
  "resultCount":5,  
  "results": [  
    {"wrapperType": "track", "kind": "song", "artistId": "163251",  
    ...
```

Returns JSON responses

Python Requests

```
(venv) $ pip3 install requests
```

GET Requests

```
requests.get(url, params)  
  
import requests  
  
resp = requests.get(  
    "https://itunes.apple.com/search",  
    params={"term": "billy bragg", "limit": 3}  
)  
  
print(resp.json())
```

POST Requests

```
requests.post(url, data, json)  
  
data  
    Dictionary of data to send in traditional web form format  
  
json  
    Dictionary of data to send as a JSON string  
  
Most modern APIs expect to receive JSON, not traditional web form format.
```

Setting HTTP Headers

Setting up an Authorization header in a post request. Authorization is a standard header for Oauth

```
token = "some-string..."  
requests.post(  
    "http://some-api.com/searchWithAuth",  
    headers={"Authorization": f"Bearer {token}"})
```

Responses

Both `.get()` and `.post()` return a Response instance

`.text`
Text of response

`.status_code`
Numeric status code (200, 404, etc)

`.json()`
Convert JSON response text to Python dictionary

API Keys/Secrets

Many APIs require “keys” and “secrets”
(similar to a “username” and “password”)

Why Do They Need API Keys?

- The API provides access to confidential data or sensitive methods
 - Only you should be able to send tweets from your Twitter account
- The API costs money to use
 - They need to know who to charge
- They want to limit abuse
 - Google Maps is free, but they want to keep you from abusing it

Where Do You Get API Keys?

Typically: you register on their site.

The process is different for every site.

Example: [YouTube API Key <https://console.developers.google.com/apis/credentials/>](https://console.developers.google.com/apis/credentials/)

How Do You Use API Keys?

It varies by different APIs

For example, if this API needed a secret key sent with requests, they might expect as a URL parameter:

```
requests.get("http://some-api.com/search",
    params={"key": "some-key",
        "isbn": "4675436632"})
```

Or, they might need complex encoding – varies by API!

Read the API docs!

Keeping Your Secrets

What's the potential problem?

app.py

```
from flask import Flask

API_SECRET_KEY = "this-is-secret"

app = Flask(__name__)

...
```

You'll want to store this file in Git – and probably GitHub

You don't want the world to learn your API key!

Strategy:

- get the key from an *environmental variable*
- store those variables in a small, secret file

Don't check that file into Git!

Example

```
(venv) $ pip install python-dotenv
```

.env

```
API_SECRET_KEY=this-is-secret
```

app.py

```
import os
from flask import Flask

API_SECRET_KEY = os.environ['API_SECRET_KEY']

app = Flask(__name__)

...
```

```
.gitignore
```

```
.env
```

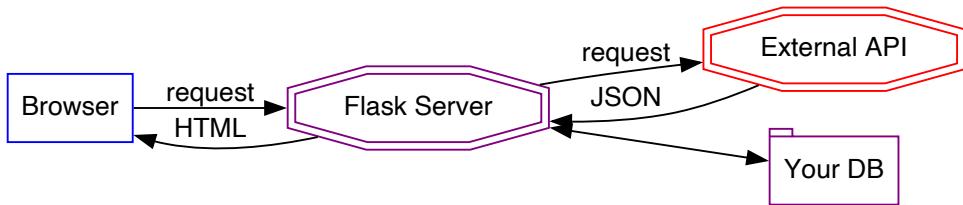
Make sure it *never* gets into your Git!

```
$ git status  
# Should NOT show up here at all  
  
$ git add .  
  
$ git status  
# Should NOT show up here at all  
  
$ git commit ...
```

External APIs and Flask

How External APIs Get Used in Flask

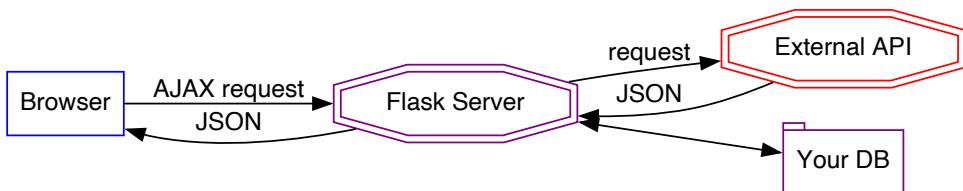
Sometimes Flask gets JSON data and it returns HTML:



```
app.py
```

```
@app.get("/book-info")  
def show_book_info():  
    """Return page about book."""  
  
    isbn = request.args["isbn"]  
  
    resp = requests.get("http://some-book-api.com/search",  
                        params={"isbn": isbn, "key": API_SECRET_KEY})  
  
    book_data = resp.json()  
  
    # using the APIs JSON data, render full HTML page  
    return render_template("book_info.html", book=book_data)
```

Sometimes Flask gets JSON data, sends JSON data to front end:



app.py

```
@app.get("/book-data")
def show_book_info():
    """Return info about book."""

    isbn = request.args["isbn"]

    resp = requests.get("http://some-book-api.com/search",
        params={"isbn": isbn, "key": API_SECRET_KEY})

    book_data = resp.json()

    # using the APIs JSON data, return that to browser

    return jsonify(book_data)
```

This is helpful if you can't make request info directly from browser — because of Same-Origin-Policy or need to keep key/secret out of browser

CORS

The developers of an *API server* can allow *cross-origin resource sharing*, as GitHub does, so their API can be used via AJAX.

Do this by enabling “CORS headers” sent as headers in the response. To enable this in Flask, use [Flask-Cors <https://flask-cors.readthedocs.io/en/latest/>](https://flask-cors.readthedocs.io/en/latest/).

Getting Around the Same Origin Policy

You can't!

If server developers didn't enable CORS, you can't use their API via AJAX (only pages served from the same origin are allowed to)

But: the SOP/CORS only applies to *AJAX calls* not *any request*.

So things like curl and Insomnia can reach their server, because those aren't using AJAX—they're just making a standard, non-AJAX request.

So, from your server-side code, you can get to the Twitter API, because server-side code is *not* AJAX!

API Libraries

Some popular APIs have specialized libraries (*sometimes known as SDKs*) written for a specific programming language that can help out.

For example, there is a Python library for calling the Twitter API:

[Python-Twitter <https://github.com/bear/python-twitter>](https://github.com/bear/python-twitter)



REST and JSON APIs

Goals

- Review GET vs POST
- Review other HTTP verbs (PUT, PATCH, DELETE)
- Describe what REST is
- Build and Test JSON APIs

Reviewing HTTP verbs

GET and POST

GET

- Remains in history, can be cached/
bookmarked
- Data sent in URL, in query string
- Repeatable

POST

- Doesn't remain in history, is not cached/
bookmarked
- Data sent in body of the request
- Not repeatable

When to use *GET* or *POST*?

- Searching / Filtering? *GET*
- Sending an email? *POST*
- Updating a user? *POST* ?

PUT / PATCH / DELETE

PUT

Update entire resource

PATCH

Update *part of* resource (*patch it up*)

DELETE

Delete resource

Requesting with methods

HTTP Verb	Links	Forms	AJAX	Server-side
GET	✓	✓	✓	✓
POST	✗	✓	✓	✓
PUT / PATCH	✗	✗	✓	✓
DELETE	✗	✗	✓	✓

Safety & idempotence

A **safe** operation is one that does not change the data requested.

An **idempotent** operation can be performed many times (with same data) with the result of all calls being the same, as if it was done once.

- Idempotence refers to side-effects, not all-effects or responses.
- Example: In arithmetic, calculating absolute value

Which methods are safe / idempotent?

HTTP Verb	Safe?	Idempotent?
GET	✓	✓
POST	✗	✗
PUT / PATCH	✗	✓
DELETE	✗	✓

Why do we care about this?

- Better describe the routes that we create
- Build standards around how we define routes
- Core part of the REST standard!

Introduction to REST

Imagine you're a developer

- Hopefully this should not be imagination!
- Your task: create route for an API that will update a user!
 - `POST /users/update` ?
 - `POST /users/change` ?

- `PATCH /users/[id]?`
- With this much flexibility, it's very helpful to standardize!

REST

- Architectural style defining constraints for creating web services
 - Includes things like: client-server model, statelessness, and caching
- APIs that adhere to these constraints are called *RESTful APIs*

RESTful APIs

- Usually have base url
 - eg `http://api.site.com/` or `http://site.com/api/`
- Have a *resource* after the base url
 - eg `http://api.com/books` or `http://site.com/api/books`
- Use standard HTTP verbs (GET, POST, PUT/PATCH, DELETE)
- Structure routes in a standardized way (*RESTful routing*)

Resource

- An object with type, associated data, relationships to other resources
- A set of methods that operate on it
- Analogous to instance/methods in OO
 - HTTP verbs describe methods on resource
 - `DELETE /cats/fluffy` is same idea as `fluffy.delete()`

Not every route in a RESTful API will necessarily be around resources. For example, you may have routes to initially authenticate with the API that aren't using a resource in the URL.

RESTful routes

RESTful routes for a resource called *snacks*:

HTTP Verb	Route	Meaning
GET	<code>/snacks</code>	Get all snacks
GET	<code>/snacks/[id]</code>	Get snack
POST	<code>/snacks</code>	Create snack
PUT / PATCH	<code>/snacks/[id]</code>	Update snack
DELETE	<code>/snacks/[id]</code>	Delete snack

RESTful route responses

Not entirely standardized — but these are common:

GET /snacks

Returns 200 OK, with JSON describing *snacks*

GET /snacks/[id]

Returns 200 OK, with JSON describing single *snack*

POST /snacks

Returns 201 CREATED, with JSON describing new *snack*

PUT or PATCH /snacks/[id]

Returns 200 OK, with JSON describing updated *snack*

DELETE

Returns 200 OK, with JSON describing success

HTTP Verb	Route	Meaning	Status	Response JSON
GET	/snacks	Get all	200	{"snacks": [{"id": 1, "name": "Cheetos", "cals": 100}, {"id": 2, "name": "M&Ms", "cals": 150}, {"id": 3, "name": "Pretzels", "cals": 200}, {"id": 4, "name": "Ruffles", "cals": 250}, {"id": 5, "name": "Snickers", "cals": 300}]}]
GET	/snacks/[id]	Get	200	{"snack": {"id": 1, "name": "Cheetos", "cals": 100}}
POST	/snacks	Create	201	{"snack": {"id": 6, "name": "New Snack", "cals": 350}}
PUT / PATCH	/snacks/[id]	Update	200	{"snack": {"id": 1, "name": "Cheetos", "cals": 120}}
DELETE	/snacks/[id]	Delete	200	{"deleted": "snack-id"}

Examples of RESTful routing:

- Stripe <<https://stripe.com/docs/api?lang=curl#charges>>
- Github <<https://developer.github.com/v3/repos/>>
- Yelp <<https://www.yelp.com/developers/documentation/v3/event>>
- Spotify <<https://developer.spotify.com/documentation/web-api/reference/playlists/>>

Nested routes

HTTP Verb	Route	Response
GET	/businesses	Get info about all businesses
GET	/businesses/[biz-id]	Get info about business
POST	/businesses	Create business
PUT / PATCH	/businesses/[biz-id]	Update business
DELETE	/businesses/[biz-id]	Delete business

HTTP Verb	Route	Response
GET	/businesses/[biz-id]/reviews	Get all reviews for business
GET	/businesses/[biz-id]/reviews/[rev-id]	Get review for business
POST	/businesses/[biz-id]/reviews	Create review for business
PUT / PATCH	/businesses/[biz-id]/reviews/[rev-id]	Update review for business
DELETE	/businesses/[biz-id]/reviews/[rev-id]	Delete review for business

RESTful APIs with Flask

- Can still use Flask and Flask-SQLAlchemy
- Will respond with JSON, not HTML
 - Won’t typically use Jinja to make JSON, just `jsonify` in route
 - Can’t redirect — return JSON of answer

Flask jsonify

```
jsonify(thing)
    Returns JSON of thing (usually dict, but could be list)

jsonify(name="Jane", age=21)
    Returns JSON like {"name": "Jane", "age": 21}
```

- JSON can only represent dictionaries, lists, and primitive types
 - Cannot represent things like SQLAlchemy model instances
- Python can’t just “turn your objects into JSON”
 - Requires a process called *serialization*

Serialization

You can turn your instances into dictionaries or lists:

demo/models.py

```
class Dessert(db.Model): ...
    def serialize(self):
        """Serialize to dictionary."""

        return {
            "id": self.id,
            "name": self.name,
            "calories": self.calories,
        }
```

demo/app.py

```
@app.get("/desserts")
def list_all_desserts():
    """Return JSON {'desserts': [{id, name, calories}, ...]}"""

    desserts = Dessert.query.all()
    serialized = [d.serialize() for d in desserts]

    return jsonify(desserts=serialized)
```

demo/app.py

```
@app.get("/desserts/<dessert_id>")
def list_single_dessert(dessert_id):
    """Return JSON {'dessert': {id, name, calories}}"""

    dessert = Dessert.query.get_or_404(dessert_id)
    serialized = dessert.serialize()

    return jsonify(dessert=serialized)
```

Sending data to a Flask JSON API

- For Insomnia, choose JSON as the request type.
- For cURL, set the *Content-Type* header:

```
$ curl localhost:5000/api/desserts \
> -H "Content-Type: application/json" \
> -d '{"name": "chocolate bar", "calories": 200}'
```

(Makes a POST to /api/desserts, passing in that JSON data)

- For AJAX using Axios, sending JSON is the default

Receiving data in a Flask JSON API

If request is made with *Content-Type: application/json*

- it won't be in *request.args* or *request.form*
- will be inside of *request.json!*

demo/app.py

```
@app.post("/desserts")
def create_dessert():
    """Create dessert from posted JSON data & return it.

    Returns JSON {'dessert': {id, name, calories}}
    """

    name = request.json["name"]
    calories = request.json["calories"]

    new_dessert = Dessert(name=name, calories=calories)

    db.session.add(new_dessert)
    db.session.commit()

    serialized = new_dessert.serialize()

    # Return w/status code 201 --- return tuple (json, status)
    return (jsonify(dessert=serialized), 201)
```

Testing our API

- We will be testing the JSON response, not HTML
 - Send data via a `json` argument (not `data`)
 - Look at `response.json`, not `response.data`
- This makes things even easier! We're just testing data, not presentation
- Can experiment before/while writing tests with Insomnia or curl

`demo/tests.py`

```
def test_all_desserts(self):
    with app.test_client() as client:
        resp = client.get("/desserts")
        self.assertEqual(resp.status_code, 200)

        self.assertEqual(
            resp.json,
            {'desserts': [
                {
                    'id': self.dessert_id,
                    'name': 'TestCake',
                    'calories': 10
                }
            ]})
```

`demo/tests.py`

```
def test_all_desserts_nibble(self):
    with app.test_client() as client:
        resp = client.get("/desserts")
        self.assertEqual(resp.status_code, 200)

        self.assertEqual(
            resp.json['desserts'][0]['name'],
            'TestCake')

        self.assertEqual(
            resp.json['desserts'][0]['calories'],
            10)
```

`demo/tests.py`

```
def test_create_dessert(self):
    with app.test_client() as client:
        resp = client.post(
            "/desserts", json={
                "name": "TestCake2",
                "calories": 20,
            })

        self.assertEqual(resp.status_code, 201)
        self.assertIsInstance(resp.json['dessert']['id'], int)

        # don't know what ID it will be, so pull out of resp and use in test
        id = resp.json['dessert']['id']
        self.assertEqual(
            resp.json,
            {"dessert": {'id': id, 'name': 'TestCake2', 'calories': 20}})

        self.assertEqual(Dessert.query.count(), 2)
```

TIP Making a copy of resp.json()

Another way to test this would be by simply removing the id from `resp.json`. However, Flask can not mutate `resp.json` so if we want to remove things from it, we first need to make a copy:

`demo/tests.py`

```
def test_create_dessert_with_delete(self):
    with app.test_client() as client:
        resp = client.post(
            "/desserts", json={
                "name": "TestCake2",
                "calories": 20,
            })
    self.assertEqual(resp.status_code, 201)

    # don't know what ID it will be, so test then remove
    self.assertIsInstance(resp.json['dessert']['id'], int)
    data = resp.json.copy()
    del data['dessert']['id']

    self.assertEqual(
        data,
        {"dessert": {'name': 'TestCake2', 'calories': 20}})

    self.assertEqual(Dessert.query.count(), 2)
```

Wrap up

- RESTful APIs have standards around routes & methods
- These are used for *API* applications, not HTML-returning applications
- Great resources for reviewing and learning more:
 - [7 Rules for REST API URL Design <https://blog.restcase.com/7-rules-for-rest-api-uri-design/>](https://blog.restcase.com/7-rules-for-rest-api-uri-design/)
 - [Awesome Reference Card <https://blog.octo.com/wp-content/uploads/2014/12/OCTO-Refcard_API_Design_EN_3.0.pdf>](https://blog.octo.com/wp-content/uploads/2014/12/OCTO-Refcard_API_Design_EN_3.0.pdf)
 - Interesting points about dashes vs underscores in URLs <https://writing.fletom.com/dashes_vs_underscores_in_URLs>



Hashing and Login

Goals

- Define authentication and authorization
- Define hashing
- Learn about Bcrypt
- Implement authentication/authorization in Flask

Registering and logging in

Authentication & authorization

Authentication

Has user presented valid credentials (eg, username/password)?

Authorization

Is the user allowed to perform this task?

- Sometimes, there's no authentication and everyone is authorized
 - eg, a public site with no user accounts or security
- On many sites, you authenticate and some actions are authorized
 - eg, New York Times: you must log in to prove you're a subscriber
 - ... but only editors get to add/change articles
- You log in with a username/password
- The site *authenticates* those to see if they're valid
- It remembers that you're valid and logged in (often via session)
- When you visit “protected routes”, it checks if you're *authorized*

User class

demo/bad_password/models.py

```
class BadUser(db.Model):
    """Site user."""

    __tablename__ = "bad_users"

    id = db.Column(
        db.Integer,
        primary_key=True,
        autoincrement=True)

    username = db.Column(
        db.Text,
        nullable=False,
        unique=True)

    password = db.Column(
        db.Text,
        nullable=False)
```

demo/bad_password/app.py

```
@app.route("/register", methods=["GET", "POST"])
def register():
    """Register user: produce form & handle form submission."""

    form = RegisterForm()

    if form.validate_on_submit():
        name = form.username.data
        pwd = form.password.data

        user = BadUser(username=name, password=pwd)
        db.session.add(user)
        db.session.commit()

        # on successful login, redirect to secret page
        return redirect("/secret")

    else:
        return render_template("register.html", form=form)
```

```
demo/bad_password/app.py
```

```
@app.route("/login", methods=["GET", "POST"])
def login():
    """Produce login form or handle login."""

    form = LoginForm()

    if form.validate_on_submit():
        name = form.username.data
        pwd = form.password.data

        user = BadUser.query.filter_by(username=name).one_or_none()

        if user and user.password == pwd:
            # on successful login, redirect to secret page
            return redirect("/secret")

        else:
            # re-render the login page with an error
            form.username.errors = ["Bad name/password"]

    return render_template("login.html", form=form)
```

```
SELECT * FROM bad_users;
```

username	password
rita	squid-13
roger	meeples4ever

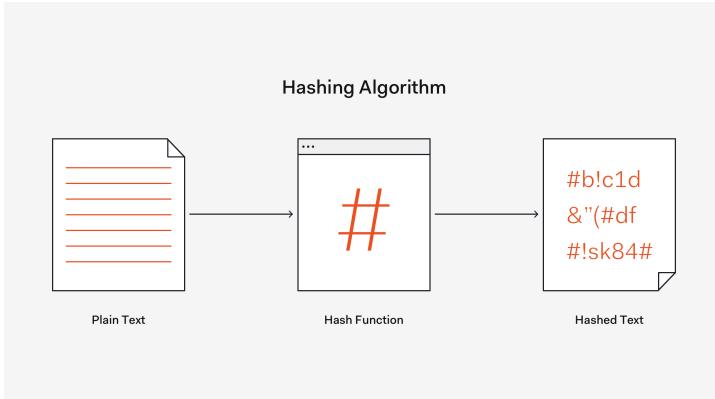
Ut oh.

Plaintext passwords

- Access to database allows access to all passwords!
- People use same passwords for multiple sites
- Don't ever do this! [Useful discussion <https://stackoverflow.com/questions/2283937/>](https://stackoverflow.com/questions/2283937/) about this

Hashing

Hashing performs a one-way transformation on a password.



“One-way” means it is effectively impossible to reverse.

“One-way encryption”

`demo/bad_hash.py`

```
def awful_hash(phrase):
    """Truly terrible hash:
       simply shifts each letter (a->b, etc).

    >>> awful_hash('yay')
    'z bz'
    """

    return ''.join(next_char(c) for c in phrase)
```

But is that really one-way?

One-Way encryption

`demo/bad_hash.py`

```
def slightly_better_hash(phrase):
    """Better hash: returns every other letter, shifted, max 4.

    >>> slightly_better_hash('penguin1')
    'qovo'

    Since this is "lossy", multiple inputs return same output:

    >>> slightly_better_hash('penguin1~pretzel7')
    'qovo'

    >>> slightly_better_hash('p?nguinZ')
    'qovo'
    """

    return ''.join(next_char(c) for c in phrase[0:8:2])
```

- Now is one-way (non-reversible)
- Same input always equal same output

Python has this kind of hash built-in:

```
>>> hash('penguin1')
6678229702981429425
```

(Python's built in `hash` seeds itself randomly on startup, so the same input only returns the same output for any individual Python process. As such, it's not suitable for storing in a database, even if it were designed to be cryptographically secure.)

Salts

(Salts are an interesting idea, but not critical to understand).

Salt: a random string introduced before hashing.

password	salt	hashed result
penguin1	xab17	qovoyc8 xab17
penguin1	meeps	qovonft meeps

Salt is usually concatenated to the password, then hashed.

`demo/bad_hash.py`

```
def salting_hash(phrase, salt=None):
    """Adds random salt; returns "salt|hash(phrase+salt)

    >>> salting_hash('hey', salt='abc')
    'izbd|abc'

    >>> salting_hash('hey', salt='def')
    'izeg|def'
    """

if salt is None:
    salt = str(randint(1000, 9999))

hashed = slightly_better_hash(f"{phrase}|{salt}")
return f"{hashed}|{salt}"
```

The same password will generate a different hash with a different salt.

NOTE Storing Salt

You may have noticed that the salt is clearly visible in the hashed result.

This is totally fine; the application needs the salt value in order to compare passwords properly. Even if an attacker gained access to the database and saw all the salts, they would still have to reverse the hashing algorithms to get the original password, which is extremely difficult and slow.

You can read more about [storing salts](https://security.stackexchange.com/questions/17421/how-to-store-salt) [<https://security.stackexchange.com/questions/17421/how-to-store-salt>](https://security.stackexchange.com/questions/17421/how-to-store-salt).

Cryptographic hash

- Non-reversible
- Change in input changes output unpredictably

password	hashed result
penguin1	dsfdsfj33gw
penguin2	kj34kjf28z

Popular algorithms

Cryptographic Hashes:

- MD5
 - SHA (*family*)
- (Fast, non-reversible, output very different)

Password Hashes:

- Argon2
 - Bcrypt
 - Scrypt
- (Same but slow and hard to optimize)

HINT Using Bcrypt by hand

```
>>> import bcrypt # pip install bcrypt
>>> salt = bcrypt.gensalt()
>>> salt
b'$2b$12$uYNRTDE7RrMvwDcF9f1Yyu'
>>> bcrypt.hashpw(b'secret', salt)
b'$2b$12$uYNRTDE7RrMvwDcF9f1Yyvuu48PzANrWy88Iz3z1tRTfdXi6DlNW'
```

Bcrypt

There's a nice Flask integration for Bcrypt, *flask-bcrypt*:

```
>>> from flask_bcrypt import Bcrypt
>>> bcrypt = Bcrypt()
>>> hash = bcrypt.generate_password_hash("secret").decode('utf8')
>>> hash
'$2b$12$s.tjeALK2I7rfI2gV27me.mkZu5IQd1Y1EBAXsbTvNExIEQcID/te'
>>> bcrypt.check_password_hash(hash, "secret")
True
```

NOTE What's this utf-8?

Bcrypt's *generate_password_hash* method returns a low-level type of string called a *byte string*. Before using it or trying to store it in your database, you should convert it to the normal kind of Python string, a *unicode string*. That's what `.decode("utf8")` does in this code.

Work factor

- Bcrypt algorithm is designed to be slow

- But computers get faster all the time!
- So, you can specify how many *rounds* of encryption it should use
 - Higher will make the algorithm slower
 - This makes it harder for hackers to compute hashes
 - This makes it harder for your server to compute hashes

```
>>> hash = bcrypt.generate_password_hash("secret", 15).decode('utf8')
>>> hash
'$2b$15$xyzalk2i7rfi2gv27me.mkzu5iqd1y1ebaxsbtvnexieqcid/te'
```

Fortunately, you don't need to do this yourself — *flask-bcrypt* bumps up the default ever year or so!

Flask password hashing

Class methods

It's good to move logic out of views

Let's make convenient class methods for registering & validating

Registering

demo/good_password/models.py

```
class User(db.Model): # ...
    @classmethod
    def register(cls, username, pwd):
        """Register user w/hashed password & return user."""

        hashed = bcrypt.generate_password_hash(pwd).decode('utf8')

        # return instance of user w/username and hashed pwd
        return cls(username=username, password=hashed)
```

Authenticating

demo/good_password/models.py

```
class User(db.Model): # ...
    @classmethod
    def authenticate(cls, username, pwd):
        """Validate that user exists & password is correct.

        Return user if valid; else return False.
        """
        u = cls.query.filter_by(username=username).one_or_none()

        if u and bcrypt.check_password_hash(u.password, pwd):
            # return user instance
            return u
        else:
            return False
```

Using class methods

```
>>> roger = User.register("roger", "cupcakes")
>>> db.session.add(roger)
>>> db.session.commit()
```

```
>>> User.authenticate("roger", "cupcakes")
<User 3>
```

```
SELECT * FROM users;
```

username	password
rita	\$2b\$12\$KD6YjzB6jyDUYxS3E/QDMeaLosFsnG/G6UVv6Ls3rWolypPXmU4LO
roger	\$2b\$12\$/GIp9nJDuoEinr4b1lbUKOXKfTANlABT47jJhFDX.jIhHft9taePi

Encrypted! 

User sessions

Remembering logged-in users

When they sign up or authenticate, store their *user_id* in the session:

```
demo/good_password/app.py
```

```
@app.route("/login", methods=["GET", "POST"])
def login():
    """Produce login form or handle login."""

    form = LoginForm()

    if form.validate_on_submit():
        name = form.username.data
        pwd = form.password.data

        # authenticate will return a user or False
        user = User.authenticate(name, pwd)

        if user:
            session["user_id"] = user.id # keep logged in
            return redirect("/secret")

        else:
            form.username.errors = ["Bad name/password"]

    return render_template("login.html", form=form)
```

Keeping user logged in

Anywhere: you can check if `user_id` is in session:

```
demo/good_password/templates/index.html
```

```
{% if 'user_id' in session %}
    <li><a href="/logout">Logout</a></li>
    <li><a href="/secret">Secret</a></li>

    <form action="/logout" method="POST">
        {{form.hidden_tag()}}
        <button type="submit" class="btn btn-primary">Logout</button>
    </form>
{% endif %}
```

Ensuring users are authorized

On any “protected” or “secret” route...

```
demo/good_password/app.py
```

```
@app.get("/secret")
def secret():
    """Example hidden page for logged-in users only."""

    if "user_id" not in session:
        flash("You must be logged in to view!")
        return redirect("/")

    # alternatively, can return HTTP Unauthorized status:
    #
    # from werkzeug.exceptions import Unauthorized
    # raise Unauthorized()

else:
    return render_template("secret.html")
```

Logging out

Just remove `user_id` from the session!

```
demo/good_password/app.py
```

```
@app.post("/logout")
def logout():
    """Logs user out and redirects to homepage."""

form = CSRFProtectForm()

if form.validate_on_submit():
    # Remove "user_id" if present, but no errors if it wasn't
    session.pop("user_id", None)

return redirect("/")
```

Since this is a POST, we need to make sure we're protected against CSRF!

```
demo/good_password/forms.py
```

```
class CSRFProtectForm(FlaskForm):
    """Form just for CSRF Protection"""
```

Our form does not need any fields, we just need it for CSRF protection.

Make sure that the HTML for logging out is contained in a form and that you are using the `{}
form.hidden_tag() {}` method



Python Wrap-Up

About Python, redux

Python is ...

- **high-level**: you think at a relatively high-level
- **dynamic**: running script can create its own functions/classes
- **dynamically-typed**: same variable can be used for int/string/etc
- **strongly-typed**: “a” + 3 doesn’t eval to “a3”
- **compiled**

Python is compiled

```
def add(x, y, double=False):
    # do the adding
    result = x + y
    return result * 2 if double else result
```

gets “compiled” into “bytecode”:

```
4      0 LOAD_FAST              0 (x)
      2 LOAD_FAST              1 (y)
      4 BINARY_ADD
      6 STORE_FAST             3 (result)

6      8 LOAD_FAST              2 (double)
10     10 POP_JUMP_IF_FALSE    20
12     12 LOAD_FAST             3 (result)
14     14 LOAD_CONST            1 (2)
16     16 BINARY_MULTIPLY
18     18 RETURN_VALUE
>>   20 LOAD_FAST             3 (result)
22     22 RETURN_VALUE
```

You don’t do this compilation separately.

It happens when you first run/import Python file.

Previously-compiled version is stored in `__pycache__/add.pyc`

You don’t need those file in Git — they get created when needed

Python can have type hints

```
def add(x: int, y: int) -> int:  
    """Add x and y and return results."""  
  
    return x + y
```

- Editors can use this to help find errors
- Can produce prettier help/API documentation

Python can be lazy

this works great...

```
def find_liked_num(nums):  
    """Prompt user until they like a number."""  
  
    for num in nums:  
        if input(f"Do you like {num}? ") == 'y':  
            return num
```

works great for this...

```
find_liked_num([1, 3, 4, 8])
```

If we wanted to do that for “all even numbers” ...

```
find_liked_num([2, 4, 6, 8, ...])
```

we need a new function and new logic

```
def find_liked_even_num():  
    """Prompt user until they like an even number."""  
  
    num = 0  
    while True:  
        if input(f"Do you like {num}? ") == 'y':  
            return num  
        num += 2
```

- We can’t use the *find_liked_nums* function we already have
- We need a new function with special logic
 - Which is no longer about finding *any* liked number
 - It’s about finding a *liked even* number — had to specify that

Laziness through *yield*

we can do this ...

```
def evens(start):
    """Yield even numbers starting at start."""

    while True:
        yield start
        start = start + 2
```

and get a little machine you can think of as a magic list

```
all_even_nums = evens(start=8) #[8, 10, 12, ...]
```

then we can do this...

```
find_liked_num(all_even_nums)
```

yield is like “return this value now, and remember where it left off”

Laziness is good

It's nice to be able to loop over data ...

- even if it's infinite (like all even numbers)
- or it's just too huge to hold in memory
- or it's expensive to pre-calculate when you might only need some

A lot of big-data stuff relies on this

There are even lazy list comprehensions: *generator expressions*

Operator overloading

In both JS and Python, some operators (like `+`) mean different things, depending on the types of objects being acted on:

JavaScript

```
3 + 5 // 8
"hello " + "Whiskey" // "hello Whiskey"
```

Python

```
3 + 5 # 8
"hello " + "Whiskey" # "hello Whiskey"
```

In Python, you can “overload” an operator in a custom class: that operator can mean something different, and you can control that

Case-Insensitive strings

demo/cistr.py

```
class CIString(str):
    """Subclass of string that is case-insensitive.

    >>> CIString("apple") == CIString("Apple")
    True

    >>> CIString("apple") < CIString("Banana")
    True
    """

    def __eq__(self, other):
        """Is self == other?"""
        return self.lower() == other.lower()

    def __lt__(self, other):
        """Is self < other?"""
        return self.lower() < other.lower()

    def __le__(self, other):
        """Is self <= other?"""
        return self.lower() <= other.lower()
```

Python libraries

Python standard library

Lots of useful data structures and features:

- queues and stacks
- binary search trees
- statistics
- complex numbers, fractions, cool math stuff
- functional programming helpers

Beautiful Soup

A lot of sites have APIs that return data.

Many don't, and you need to "scrape" HTML to get data.

Beautiful Soup is a terrific library for this.

Common data science libraries

Numpy

Super-fast linear algebra and matrix math

Pandas

Data slicing/grouping/querying

SciKit-Learn

Common machine learning algorithms

Good place to start <<http://www.scipy-lectures.org>>

Jupyter

Jupyter <<http://jupyter.org>> is “interactive computing”

- Like IPython in a web page
- Can mix in documentation, drawings, code snippets
- Often used to play with data or share analyses
- Can publish on the web
- Can even interactively edit as a group!

And it's not just for Python :)

Zen Of Python

Beautiful is better than ugly. Readability counts.

Explicit is better than implicit.

Simple is better than complex.

Special cases aren't special enough to break the rules.

Errors should never pass silently.

In the face of ambiguity, refuse the temptation to guess.

If the implementation is hard to explain, it's a bad idea.

TIP Seeing this

This was written by Tim Peters, one of the core contributors to the Python project.

If you ever need a helpful reminder of this:

>>> `import this`



Flask Wrap-Up

Flask features

A scalable, powerful, general-purpose web application framework.

What we covered

- Routes
- Jinja templates
- Flask-SQLAlchemy
- Flask Testing
- Cookies & Sessions
- JSON and flask
- Flask-WTForms

We left a lot out — intentionally

What we didn't cover

url_for

you did this once...

```
@app.get("/users/<int:id>")
def user_profile(id): ...
```

you did this a dozen places...

```
<a href="/users/{{ user.id }}>See user</a>
```

and this in lots of places

```
def some_other_view():
    ...
    return redirect(f"/users/{user.id}")
```

What if you wanted to change that URL?

- Perhaps to /profiles/[user-id]?
- Perhaps to /warbler/users/[user-ud]?

you still do this once...

```
@app.get("/users/<int:id>")
def user_profile(id): ...
```

but now you don't need to hardcode URL

```
<a href="{{ url_for('user_profile', id=user.id) }}>go</a>
```

and this in lots of places

```
from flask import url_for

def some_other_view():
    ...
    redirect_url = url_for('user_profile', id=user.id)
    return redirect(redirect_url)
```

Blueprints

Build “applications” in Flask:

- Each app can have own models, forms, tests, views
- Can re-use an app in many sites
 - Many sites could use “blogly” app
- Useful for larger/more complex sites

Signals

“When [this thing] happens, do [this other] thing.”

(eg send an email when a user registers, no matter how they register)

Lots of Jinja stuff

Lots of additional features in Jinja:

- sharing parts of templates/repeated code
- formatting of numbers, dates, lists in the template
- caching parts of templates (“this part only changes every 5 minutes”)
- and more!

Popular add-ons

WTForms & SQLA

you did this a lot

```
def edit_pet(pet_id):
    pet = Pet.query.get(pet_id)
    form = EditPetForm(obj=pet)

    if form.validate_on_submit():
        pet.name = form.name.data
        pet.species = form.species.data
        pet.color = form.color.data
        pet.age = form.age.data
        pet.weight = form.weight.data
        pet.num_legs = form.num_legs.data
    ...

```

you can do this

```
def edit_pet(pet_id):
    pet = Pet.query.get(pet_id)
    form = EditPetForm(obj=pet)

    if form.validate_on_submit():
        form.populate_obj(pet)
```

WTForms-Alchemy

Can generate WTForms from SQLAlchemy model:

forms.py

```
from flask_wtf import FlaskForm
from wtforms_alchemy import model_form_factory
from models import db, Pet, Owner

BaseModelForm = model_form_factory(FlaskForm)

class ModelForm(BaseModelForm):
    @classmethod
    def get_session(self):
        return db.session

class PetForm(ModelForm):
    class Meta:
        model = Pet

class OwnerForm(ModelForm):
    class Meta:
        model = Owner
```

Flask-Login

Product that provides common parts of user/passwords/login/logout

Similar to what you built, but out-of-box.

Flask-Mail

Can send email from Flask!

Flask-Admin

Can get decent CRUD admin views from SQLAlchemy models:

	User	Post Title	Date
<input type="checkbox"/>	harry	de Finibus Bonorum et Malorum - Part I	2017-01-07 21:45:07.296042
<input type="checkbox"/>	amelia	de Finibus Bonorum et Malorum - Part I	2017-08-14 21:45:07.296528
<input type="checkbox"/>	oliver	de Finibus Bonorum et Malorum - Part II	2017-11-14 21:45:07.296790
<input type="checkbox"/>	jack	de Finibus Bonorum et Malorum - Part II	2018-10-08 21:45:07.297036
<input type="checkbox"/>	isabella	de Finibus Bonorum et Malorum - Part II	2018-07-17 21:45:07.297279
<input type="checkbox"/>	charlie	de Finibus Bonorum et Malorum - Part I	2018-06-04 21:45:07.297535

Flask-Restless

Get CRUD API endpoints from SQLAlchemy models:

```
from flask.restless import APIManager

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Unicode)
    birth_date = db.Column(db.Date)

class Computer(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Unicode)
    vendor = db.Column(db.Unicode)
    purchase_time = db.Column(db.DateTime)
    owner_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    owner = db.relationship('Person')

# Create the Flask-Restless API manager.
manager = APIManager(app, flask_sqlalchemy_db=db)

# API endpoints, available at /api/<tablename>
manager.create_api(User, methods=['GET', 'POST', 'DELETE'])
manager.create_api(Computer, methods=['GET'])
```

Will you use Flask?

Maybe.

It's popular and used by many real companies, large and small.
It's also a great choice for personal projects, code challenges, etc.

Flask versus ...

Flask v Node-Express

Pretty similar, actually.
Both work at same “level of concepts”, and share lots of ideas.
You can even use Jinja to make templates with Express!

Flask v Django

Django is a popular, larger, more featureful Python Framework.
It's a *higher level* and *more opinionated*

Flask model.py

```
class Pet(db.Model):
    id = ...
    name = ...
    color = ...
    owner_id = ...

    owner = db.relationship("Owner", backref="pets")
```

Django model.py

```
class Pet(models.Model):
    name = ...
    color = ...
    owner = models.ForeignKey("Owner")

    # assumes "id" of auto-incrementing int
    # defines relationship & make "owner_id" column
```

Flask app.py

```
@app.route("/pets/<int:id>/edit", methods=["GET", "POST"])
def edit_pet(id):
    """Show pet edit form / handle edit."""

    pet = Pet.query.get(id)
    form = PetEditForm(obj)      # need to make form!

    if form.validate_on_submit():
        pet.name = form.name.data
        pet.color = form.color.data
        db.session.commit()
        redirect(f"/pets/{id}")

    return render_template("pet_edit.html", form=form)
```

Django views.py

```
class PetEditView(generic.UpdateView):
    """Show pet edit form / handle edit."""

    model = Pet
```

So, is Django “better”?

Nope.

If you like Django’s patterns & they fit your use cases, you can build an app faster by following those patterns.

But:

- they take a longer amount of time to learn
- if things break, it can be harder to understand
- if you want to change how things work, it can be harder

Flask is like a really nice bicycle:

It’s great for easy trips, can scale up to long journeys, isn’t too opinionated about where you use it, and it’s relatively easy to understand and fix.



Deployment with Render

ElephantSQL

- Host for PostgreSQL databases
- Have excellent commercial options, but also a free, small instance

Getting a database

1. Create account at [ElephantSQL <https://www.elephantsql.com>](https://www.elephantsql.com) using GitHub
2. Create a “Tiny Turtle” (free) instance
3. Select region: *US-West-1 (even if others are closer to you)*.
 - If you get an error selecting *US-West-1*, pick *US-East-1*
4. Confirm and create
5. Click on your new instance and copy the URL

Seeding your new database

```
$ pg_dump -O warbler | psql (url you copied here)
```

This dumps your existing *warbler* database and loads it in your new database.

Checking your database

```
$ psql (url you copied here)
```

Render

- A service for serving web applications from the cloud
- Similar to Salesforce’s Heroku product, but has a free tier

Installing gunicorn

When we deploy an application in production, we will always want to use a server that is production ready and not meant for just development.

The server we will be using is gunicorn so let’s make sure we:

```
(venv) $ pip install gunicorn
```

Ensuring a correct deployment

- Render needs to know our dependencies!
- Make sure you `pip freeze > requirements.txt`

Setting up your app

1. Create an account at [Render <https://render.com>](https://render.com) using GitHub
2. From dashboard, create a new instance of “Web service”
3. Connect to your repository
4. Give it a name (*this must be globally unique*)
5. Make sure the start command is `gunicorn app:app`
6. Choose advanced, and enter environmental variables:
 - *DATABASE_URL*: URL from ElephantSQL (change `postgres:` → `postgresql:`)
 - *SECRET_KEY*: anything you want (*to be secure: long and random*)
 - *PYTHON_VERSION*: `3.11.2`
7. Choose “Create Web Service”

Debugging your app

From the dashboard for your app, you can view the logs

Updating your app

When you push to your GitHub repo, it will automatically redeploy your site.

You can turn that off under *Settings* → *Auto-Deploy*, and then can do manual deploys.



Deployment with Heroku

Heroku

- a platform as a service, runs on Amazon Web Services
- easier and faster to deploy, but far less customization

Installing Heroku

- [Sign up for an account on Heroku <https://signup.heroku.com/>](https://signup.heroku.com/)

For Mac users, Install the Heroku CLI with the command below

```
$ brew install heroku/brew/heroku
```

For linux/wsl users to install the CLI on Ubuntu (or in WSL under Ubuntu)

```
$ curl https://cli-assets.heroku.com/install-ubuntu.sh | sh
```

Installing gunicorn

When we deploy an application in production, we will always want to use a server that is production ready and not meant for just development.

The server we will be using is gunicorn so let's make sure we:

```
(venv) $ pip install gunicorn
```

Ensuring a correct requirements.txt

- Heroku needs to know our dependencies!
- Make sure you `pip freeze > requirements.txt`

Adding a Procfile

- When we push code to Heroku, we need to tell Heroku what command to run to start the server.
- This command must be placed in a file called *Procfile*.
- Make sure this filename does not have any extension and begins with capital P.

```
$ echo "web: gunicorn app:app" > Procfile
```

Adding a runtime.txt

- To make sure you are using a certain version of Python on Heroku, add a file called *runtime.txt* and specify the version of Python you want to use.

```
$ echo "python-3.9.13" > runtime.txt
```

Creating your Heroku app

- Login to your heroku account
- Create an application and make sure you have a correct remote.
- Add and commit your recent changes.
- Push your code to the new remote.

```
$ heroku login  
$ heroku create NAME_OF_APP  
$ git remote -v      # make sure you see heroku  
$ git status         # make sure you see your Procfile and runtime.txt  
$ git add .          # make sure you are only adding what you want to commit!  
$ git commit -m "preparing to deploy to heroku"  
$ git push heroku main
```

Environment Variables

Since we're on a different server, we need different environment variables values:

```
$ heroku config:set SECRET_KEY=never tell FLASK_DEBUG=False  
$ heroku config    # see all your environment variables
```

Adding a Postgres Database

In order to use a production database, we need Heroku to make one:

```
$ heroku addons:create heroku-postgresql:hobby-dev  
$ heroku config    # you should see DATABASE_URL
```

Making sure you connect to the correct database

Now that we have a postgres database, we need to make sure that we are connecting to the correct database when in production!

We have provided this already in the Warbler starter code, but want you to be aware of this for future deployments

app.py

```
import os

database_url = os.environ['DATABASE_URL']

# fix incorrect database URIs currently returned by Heroku's pg setup
database_url = database_url.replace('postgres://', 'postgresql://')

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = database_url
```

Open your Heroku app

Now that we have a DATABASE_URL environment variable, we should be able to open our app!

```
$ heroku open
```

Connecting to *psql*

```
$ heroku pg:psql
```

Running a SQL file on Heroku

```
$ heroku pg:psql -f data.sql
```

Running commands on your production server

```
$ heroku run python seed.py
```

Debugging a Heroku application

It's **never** going to work perfectly the first time. Make sure you look at the server logs to debug!

To see what went wrong, check out the server logs:

```
$ heroku logs
```

Heroku hints

- Make sure you've added and committed before pushing to production
- Any time you want to make a change to your production application you always need to add, commit and push

- If things break **ALWAYS** go to *heroku logs* and see what's breaking

NOTE Further Reading

There are a number of helpful guides on the Heroku Dev Center that walk you step-by-step through deploying applications in the technology of your choice. Guides for Python projects can be found [here <https://devcenter.heroku.com/categories/python-support>](https://devcenter.heroku.com/categories/python-support).



JavaScript Asynchronicity

Goals

- Examine callbacks in context of asynchronicity
- Understand role of promises
- Understand role of *async / await*
- See common patterns and how to solve with *async / await*

Callbacks

A callback is a function passed to another function, for it to call.

They are used for many things in JS.

Functional Programming Patterns

```
let paidInvoices = invoices.filter(  
  inv => inv.owed <= 0  
)
```

Useful for separating processing patterns from business logic.

Event-Driven Programming

```
$("#myForm").on("submit", sendDataToServer);
```

Register a function to be called when an event happens.

Asynchronous Code

```
setTimeout(callMeInASec, 1000);  
  
ajaxLibrary.get("/page", callMeWhenResponseArrives);
```

Call those callbacks when asynchronous operation completes.

NOTE Some AJAX Libraries Use Callbacks

While axios doesn't allow callbacks as a control-flow possibility for AJAX, other AJAX libraries, like jQuery, do.

Callbacks Will Always Be Useful

First two cases will always be done with callbacks:

- Functional programming patterns
- Event-driven programming

Let's talk about third case, for handling asynchronicity

Callbacks for Asynchronicity

Why Not This?

```
// pause for a second
stopHere(1000);
// now this code runs

// get via AJAX
var response = ajaxLibrary.get("/page");
// now we have response
```

- JS is *single-threaded*, only one bit of code can run at once
- If JS *actually* stopped there, it couldn't do other things
 - Respond to events (clicks in browser, etc)
 - Repaint DOM changes in browser

So, by having a callback function for “once-async-thing-is-done”, JS can finish running your code as quickly as possible.

This way it can get to those other waiting tasks ASAP.

NOTE Is JavaScript Single Threaded?

Largely, yes. JavaScript has “worker threads” that can do some background tasks, but these are limited in their scope. Unlike many other programming languages, JavaScript cannot run multiple top-level threads.

Callback Patterns

Sequential callbacks can lead to hard-to-understand code:

```
ajaxLib.get("/step-1", function f2(resp) {
  ajaxLib.get("/step-2", {resp.body}, function f3(resp) {
    ajaxLib.get("/step-3", {resp.body}, function done(resp) {
      console.log("got final answer", resp.body);
    });
  });
});
```

This is often called “callback hell”

You can flatten that by using non-anonymous functions:

```
ajaxLib.get("/step-1", doStep2);

function doStep2(resp) {
  ajaxLib.get("/step-2", {resp.body}, doStep3);
}

function doStep3(resp) {
  ajaxLib.get("/step-3", {resp.body}, giveAnswer);
}

function giveAnswer(resp) {
  console.log("got final answer", resp.body);
}
```

Each function needs to know what to do next; this makes writing independent functions hard.

It can be particularly hard to handle errors well for things like this.

Promises

Promises provide an alternate way to think about asynchronicity.

A promise is **one-time guarantee of future value**.

demo/pokemon.js

```
const url = `${BASE_URL}/1`;
const p = axios({ url });
console.log("first", p); // Promise {<pending>}
```

- Promises in JavaScript are objects
- They are native to the language as of ES2015
- A promise can be in one of three states:
 - *Pending* - It doesn't yet have a value
 - *Resolved* - It has successfully obtained a value
 - *Rejected* - It failed to obtain a value for some reason
- The only way to access the resolved or rejected value is to chain a method on the end of the promise (or await it)

.then and *.catch*

- Promises provide a *.then* and a *.catch*, which both accept callbacks.
- The callback to *.then* will run if the promise is resolved, and has access to the promise's resolved value.
- The callback to *.catch* will run if the promise is rejected, and typically has access to some reason behind the rejection.

NOTE Thenables

When reading about promises, you'll often see a related term, called a **thenable**. A thenable is simply any object or function that has a *then* method defined on it.

By this definition, all promises are thenables, but not all thenables are promises! There are many more specifications that a promise needs to satisfy.

Here's a simple example of a thenable that isn't a promise:

```
const notAPromise = {
  fruit: "apple",
  veggie: "carrot",
  then: () => {
    console.log("I'm a random object with a then method.");
  }
};

notAPromise.then();
// "I'm just a random object with a then method."
```

demo/pokemon.js

```
const validUrl = `${BASE_URL}/1`;
const futureResolvedPromise = axios({ url: validUrl });

futureResolvedPromise
  .then(console.log)
  .catch(console.warn);
// keeps going ...
// ...
// ...

// NFW to get that answer

// promise to be rejected

const invalidUrl = `http://nope.nope`;
const futureRejectedPromise = axios.get(invalidUrl);

futureRejectedPromise
  .then(console.log)
  .catch(console.warn);

// promise chaining
axios({ url: `${BASE_URL}/1` })
  .then(function f1(r1) {
    console.log(`#1: ${r1.data.name}`);
    return axios({ url: `${BASE_URL}/2` });
})
  .then(function f2(r2) {
    console.log(`#2: ${r2.data.name}`);
    return axios({ url: `${BASE_URL}/3` });
})
  .then(function f3(r3) {
    console.log(`#3: ${r3.data.name}`);
  })
  .catch(function (err) {
    console.error(err);
```

demo/pokemon.js

```
const invalidUrl = `http://nope.nope`;
const futureRejectedPromise = axios.get(invalidUrl);

futureRejectedPromise
  .then(console.log)
  .catch(console.warn);

// promise chaining
axios({ url: `${BASE_URL}/1` })
  .then(function f1(r1) {
    console.log(`#1: ${r1.data.name}`);
    return axios({ url: `${BASE_URL}/2` });
})
  .then(function f2(r2) {
    console.log(`#2: ${r2.data.name}`);
    return axios({ url: `${BASE_URL}/3` });
})
  .then(function f3(r3) {
    console.log(`#3: ${r3.data.name}`);
  })
  .catch(function (err) {
    console.error(err);
```

Promise Chaining

- When calling `.then` on a promise, can return *new* promise in callback!
 - Can chain asynchronous operations together with `.then` calls
- Only need *one* `.catch` at the end—don’t have to catch every promise

demo/pokemon.js

```
axios({ url: `${BASE_URL}/1` })
  .then(function f1(r1) {
    console.log(`#1: ${r1.data.name}`);
    return axios({ url: `${BASE_URL}/2` });
})
  .then(function f2(r2) {
    console.log(`#2: ${r2.data.name}`);
    return axios({ url: `${BASE_URL}/3` });
})
  .then(function f3(r3) {
    console.log(`#3: ${r3.data.name}`);
  })
  .catch(function (err) {
    console.error(err);
});
```

Benefits of Promises Over Callbacks

- Easier to write good functions
 - Each step doesn’t have to be tied directly to next step
 - With promises, `.then` method can just return value for next without having to itself know what comes next

- Error handling is much easier

Async / Await

async / await are language keywords for working with promises.

async

- You can declare any function in JavaScript as *async*
- *async* functions always return promises!
- In *async* function, you write code that looks synchronous
 - But it doesn't block JavaScript

await

- Inside an *async* function, we can use *await*
- *await* pauses execution
- Can *await* any promise (eg other *async* functions!)
- *await* waits for promise to resolve & evaluates to its resolved value
- It then resumes execution
- Think of the *await* keyword like a pause button

demo/pokemon.js

```
async function getPokemonAwait() {
  const r1 = await axios({ url: `${BASE_URL}/1/` });
  // ...
  console.log(`#1: ${r1.data.name}`);

  const r2 = await axios({ url: `${BASE_URL}/2/` });
  console.log(`#2: ${r2.data.name}`);

  const r3 = await axios({ url: `${BASE_URL}/3/` });
  console.log(`#3: ${r3.data.name}`);
}
```

Error Handling

demo/pokemon.js

```
async function getPokemonAwaitCatch() {
  try {
    const r1 = await axios({ url: `${BASE_URL}/1/` });
    console.log(`#1: ${r1.data.name}`);

    const r2 = await axios({ url: `${BASE_URL}/2/` });
    console.log(`#2: ${r2.data.name}`);

    const r3 = await axios({ url: `${BASE_URL}/3/` });
    console.log(`#3: ${r3.data.name}`);
  } catch (err) {
    console.warn("Try again later!");
  }
}
```

Comparing *.then/.catch* and *async/await*

- Under the hood, they do the same thing
- *async/await* are the modern improvement
 - Code can be written more naturally
- There are a few cases where it's easy to deal with promises directly

Asynchronous Patterns

Many Calls, Do Thing On Return & Don't Block

Need to make several AJAX calls and do things *as they return*:

Promises

Async/Await

```
let results = [];

axios({ url: "/1" })
  .then(function f1(r1) {
    doThing(r1);
  });

axios({ url: "/2" })
  .then(function f2(r2) {
    doThing(r2);
  });

axios({ url: "/3" })
  .then(function f3(r3) {
    doThing(r3);
  });

// rest runs while that's happening
```

```

async function getAndDo1() {
  let r = await axios({ url: "/1" });
  doThing(r);
}

async function getAndDo2() {
  let r = await axios({ url: "/2" });
  doThing(r);
}

async function getAndDo3() {
  let r = await axios({ url: "/3" });
  doThing(r);
}

// don't await the calling of these
getAndDo1();
getAndDo2();
getAndDo3();

// rest runs while that's happening

```

Many Calls, in Sequence

Need to make AJAX calls one-at-a-time, in order:

Promises

```

const results = [];

axios({ url: "/1" })
  .then(function f1 (r1) {
    results.push(r1.data);
    return axios({ url: "/2" });
  })
  .then(function f2 (r2) {
    results.push(r2.data);
    return axios({ url: "/3" });
  })
  .then(function (r3) f3 {
    results.push(r3.data);
    // here is final list of results
  });

```

Async/Await

```

const r1 = await axios({ url: '/1' });
const r2 = await axios({ url: '/2' });
const r3 = await axios({ url: '/3' });

let results = [r1, r2, r3];

console.log(results);

```

- *Promise.all* accepts an array of promises and returns a *new* promise
- New promise will resolve when every promise in array resolves, and will be rejected if any promise in array is rejected
- *Promise.allSettled* accepts an array of promises and returns a *new* promise
- The promise resolves after all of the given promises have either fulfilled or rejected, with an array of objects that each describes the outcome of each promise.

Promise.allSettled

```
const GITHUB_BASE_URL = 'https://api.github.com';

let elieP = axios({url:`${GITHUB_BASE_URL}/users/elie`});

let joelP = axios({url:`${GITHUB_BASE_URL}/users/joelburton`});

let doesNotExistP = axios({url:`${GITHUB_BASE_URL}/user/dsdsdsds`});

let results = await Promise.allSettled([elieP, joelP, doesNotExistP]);

console.log(results);

/*
  {status: 'fulfilled', value: {...}}
  {status: 'fulfilled', value: {...}}
  {status: 'rejected', reason: Request failed with status code 404...}
*/
```

TIP Promise.all - many calls, in any order, fail fast

Make several calls, but they don't depend on each other. If one fails, the whole thing fails

```
const p1 = axios({ url: '/1' });
const p2 = axios({ url: '/2' });
const p3 = axios({ url: '/3' });

const resultsPromise = await Promise.all(
  [p1, p2, p3]);
```

Many Calls, First One Wins

Can get answer from any call; stop after any responds:

Promises

```
const p1 = axios({ url: '/1' });
const p2 = axios({ url: '/2' });
const p3 = axios({ url: '/3' });

const answerPromise = Promise.race(
  [p1, p2, p3]);

// can `await` or `.then` this promise
```

Async/Await

```
// tricky to do with async / await
```

- *Promise.race* accepts an array of promises and returns a *new* promise
- This new promise will resolve or reject as soon as *one* promise in the array resolves or rejects

NOTE Advanced: Building Own Promises

- You can use *Promise* with the *new* keyword to make your own promises
- Unfortunately, the syntax here takes some getting used to
- *Promise* accepts a single function (call it *fn*) as an argument
 - *fn* accepts two functions as arguments, *resolve* and *reject*
 - Pass *resolve* a value for the promise to resolve to that value
 - Pass *reject* a value for the promise to reject to that value

Example: setTimeout as a Promise

Async callbacks can be rewritten as promises:

demo/promiseTimeout.js

```
function wait(ms) {
  let p = new Promise(
    function (resolve, reject) {
      setTimeout(function () {
        // cause promise to resolve:
        // any await-er will resume
        resolve();

        // we have no error conditions -- if we did, we
        // would call reject() to reject this
      }, ms);
    }
  );
  return p;
}

async function demo() {
  console.log("hi");
  await wait(1000);
  console.log("there");
}
```

Wrap Up

- Callbacks will always be useful!
 - But they're not the best way to handle asynchronicity
- Promises improve things
 - But they're much easier with *async / await* for most things
- Prefer using *async / await* over promises
 - Watch out for stuff like:

```
var resp = await axios({ url: "/1" })
  .then(resp => console.log());
```

- What is this awaiting?! What is *resp*?

resp will be the result of *await* for the entire line — the axios call is resolved by the arrow function, which does console log the error, but returns a promise of undefined — which is then awaited and becomes the resolution of *undefined* which is assigned to *resp*.

Phew.

In other words: don't mix *await* and *.then*. It almost certainly isn't useful. Just *await* things.



Intro to Node.js

Node.js

- A JavaScript environment that runs server-side
 - It uses the Chrome V8 engine, but doesn't require/use Chrome
- Can be used to build any kind of server-side JS
 - Including building web applications!
 - or as a general-purpose scripting language

Why Node.js?

- The entire stack (frontend/backend) can be JavaScript
- There is an extensive set of add-on libraries via *npm*
 - Many can also be used in client-side JS
- Widely used

Starting Node

(in directory with a file `myScript.js`)

```
$ node myScript.js  
Hi there!
```

or, for an interactive REPL:

```
$ node  
>
```

(`.exit` or `Control-D` to return to the normal terminal)

NPM

- Massive registry of add-on libraries
- Command line tool, *npm*, comes with Node
- Easy dependency management for a project
- Register with npm to publish open-source or proprietary packages

Starting a Project with NPM

```
$ cd my-project  
$ npm init
```

- Creates `package.json` with metadata & dependencies
- In a hurry? Want the defaults? `npm init --yes` (or `npm init -y`)

Example `package.json`

```
{  
  "name": "node-files",  
  "version": "1.0.0",  
  "description": "Exercise to create file listing functions.",  
  "main": "",  
  "scripts": {  
    "start": "node index.js"  
  },  
  "author": "Whiskey the Dog",  
  "license": "GPL-v3",  
  "dependencies": {  
    "axios": "0.x"  
  }  
}
```

Store your `package.json` in Git

Installing Packages

In the root project directory:

```
$ npm install axios
```

- Adds latest version of axios to `dependencies` object in `package.json`
- Installs the package in a local `node_modules` folder

node_modules

A directory containing all dependencies in the root directory of your project.

Always add `node_modules` to `.gitignore`

(It's just a collection of dependencies that can be reinstalled)

Reinstalling Packages

When you clone an existing project from GitHub or delete your `node_modules`, here's how you get those dependencies:

```
$ npm install
```

- `npm install` without arguments uses *dependencies* object in `package.json`
- Similar to `pip install -r requirements.txt` in Python

NPM Summary

- Install dependencies to project's with `npm install <package_name>`
 - This downloads and adds package to `node_modules` directory
 - Dependencies are noted in `package.json` and `package-lock.json`
 - `package-lock.json` locks in the exact versions as installed. Do not modify this file, you will work with the `package.json`
 - Always commit `package.json`, but never commit `node_modules`!
 - Use `npm install` on a fresh repository to get a new `node_modules`

TIP `package-lock.json` and versioning

You might also notice a file being created called `package-lock.json` when you run `npm install` or install modules. You do not ever need to modify this file, it is helpful for knowing exact versions of all dependencies for consistent development.

- `package.json` contains metadata & dependency *requirements*
- For `npm install axios`, this lists dependency as `"axios": "0.x"`
- Meaning “use axios version 0.18.0 or greater” (because of ^ symbol)
- `package-lock.json` contains *exact* versions actually installed
- `npm install` looks at `package-lock.json` first, before `package.json`
 - It will therefore install the exact version
 - That's good to make sure production uses same version as development
 - But it means you'll never get bug-fixes/new versions of the library

Node *process* Library

Node provides a global object, `process`, for managing the current script.

Using `process`, we can:

- Access environmental variables
- See the command-line arguments passed to the script
- Kill the script

See `process` documentation <<https://nodejs.org/api/process.html>> for a full list of functions.

process.env

```
process.env.SECRET_KEY
```

Get value of environmental variables from shell

process.env is an object and its keys are the names of environment variables.

```
$ export SECRET_INFO=abc123
$ node
> process.env.SECRET_INFO
'abc123'
```

process.argv

```
process.argv[index]
```

process.argv is array of args given to shell to start this program

process.argv Example

demo/basics/showArgs.js

```
const argv = process.argv;

for (let i = 0; i < argv.length; i += 1) {
  console.log(i, argv[i]);
}
```

```
$ node showArgs.js hello world
0 '/path/to/node'
1 '/path/to/showArgs.js'
2 'hello'
3 'world'
```

process.exit

```
process.exit(exit_code)
```

Exit the program immediately and return an exit code to the shell.

By convention, 0 is “no error”; other code (1, 2, etc.) are script errors.

The Module System

Modules are the way to share code across different files in a Node project.

You might hear this system referred to as *CommonJS Modules*.

There aren't `<script>` tags in the Node ecosystem, so you have to include other files by exporting/importing explicitly.

Importing a Project File

All imports use the `require` keyword.

To import a local project file, specify a *relative path* to that file:

`demo/modules/other.js`

```
const usefulStuff = require("./usefulStuff");

const results = usefulStuff.add(2, 3);

console.log(results);
```

- This usually means `./` for current directory or `../` for parent directory
- You don't need to include the file extension for `.js` and `.json` files.

Importing a Library

To import a built-in module or NPM package, use the name:

`demo/modules/google.js`

```
const axios = require("axios");

async function getGoogle() {
  const resp = await axios.get("https://google.com/");
  console.log(resp.data.slice(0, 80), "...");
}

getGoogle();
```

Node will look in its included core modules and `node_modules`.

Destructuring Imports

When importing an object, you can destructure into variables:

`demo/modules/other2.js`

```
const { add, User } = require("./usefulStuff");

const results = add(2, 3);

console.log(results);
```

Exporting from a File

Use built-in `module.exports` to make things importable by other files:

`demo/modules/usefulStuff.js`

```
const MY_GLOBAL = 42;

function add(x, y) {
  return x + y;
}

class User {
  constructor(name, username) {
    this.name = name;
    this.username = username;
  }
}

const notNeededElsewhere = "nope"; // I don't get exported!

// export an object
module.exports = {
  MY_GLOBAL: MY_GLOBAL,
  add: add,
  User: User,
};
```

module.exports

Normally `module.exports` is an object; this can export multiple things.

But you can actually set it to whatever you want:

in one file

```
module.exports = function() {
  console.log('hello');
}
```

in another file

```
const sayHello = require('./example');

sayHello();
// hello
```

Modules Summary

- Export things in file with `module.exports` (*usually an object*)
- Import exports of local files with relative path: `require("./other")`
- Import exports of installed libraries with name: `require("axios")`

TIP Node Callbacks (an older pattern)

Many older Node library functions utilize asynchronous callbacks by default. It's unlikely that you may encounter these, but if you do here is how we used to handle them before Promises and async/await

For example, to find the IP address for a hostname:

```
const dns = require("dns");
dns.lookup("rithmschool.com", cbFunction);
```

That does an asynchronous lookup, and calls our callback when finished.

Node.js callbacks usually conform to an “error-first” pattern.

- Callback’s first parameter will be *error*.
 - Node will supply error object (if something bad happened)
 - If no error happened, this will be null
- Then follow the other parameters, if there are any:

```
dns.lookup("rithmschool.com", function (err, hostname) {
  if (err) {
    // handle error
  } else {
    console.log(hostname);
  }
});
```

Good explanation of [error-first callback pattern <https://nodejs.org/api/errors.html#error_first_callbacks>](https://nodejs.org/api/errors.html#error_first_callbacks).

Handling Errors

In the browser, there are different things to do with errors:

- Show some “an error happened” message in the DOM
- Pop up an alert box
- Log to the console

In Node, you will often do one of these:

- Log the error to the console
- Exit the program with `process.exit(1)`

TIP `util.promisify()`

Sick of callbacks? For an advanced reading, consider [promisifying your callbacks <https://nodejs.org/dist/latest-v8.x/docs/api/util.html#util_util_promisify_original>](https://nodejs.org/dist/latest-v8.x/docs/api/util.html#util_util_promisify_original).

Just don’t look at *callbackify*. We don’t talk about *callbackify*. 😊

File System Module

fs

fs.promises module is built-in and provides interface to file system.

- You'll use it often read and write files.
- The underlying functions are asynchronous and return promises

```
const fsP = require('fs/promises')
```

Reading Files

```
fsP.readFile(path, encoding)
```

- *path*: path to file (relative to working directory)
- *encoding*: how to interpret file
 - for text files, this is almost always “utf8”
 - for binary files (like an image), omit this argument

demo/files/read.js

```
const fsP = require("fs/promises");

async function readMyFile() {
  try {
    let contents = await fsP.readFile("myFile.txt", "utf8");
    console.log("file contents", contents);
  } catch (err) {
    process.exit(1);
  }
}
```

Writing Files

```
fsP.writeFile(path, data, encoding)
```

- *path*: path to file (relative to working directory)
- *data*: data to output to file (typically a string)
- *encoding*: how to write file
 - for text files, this is almost always “utf8”
 - for binary files (like an image), omit this argument

demo/files/write.js

```
const fsP = require("fs/promises");

const content = "THIS WILL GO IN THE FILE!";

async function writeOutput() {
  try {
    await fsP.writeFile("./files/output.txt", content, "utf8");
  } catch (err) {
    console.error(err);
    process.exit(1);
  }
  console.log("Successfully wrote to file!");
}
```

Callback Versions

The original API for the *fs* module used callbacks, not promises.

That makes the code trickier to write and understand, but you will see lots of older code using *fs*, not *fs/promises*

File System Summary

- *fs* is a built-in module
- The default methods are asynchronous reading and writing
- Stringify file contents when writing with encoding (normally “utf8”)

Node vs Browser JS

- Most programmatic behavior is exactly the same (yay V8!)
- The “global object” isn’t *window*, it’s *global*
 - This is where global vars go, where *setTimeout* is, etc
- Node doesn’t have *document* & DOM methods
- Node provides access to filesystem & can start server processes
- Many NPM libraries are “isomorphic” (can be used in web JS or Node)

Advice: Experiment in the Console

Open a Node console while programming in Node.

This is convenient for quick experimentation in the console.

Looking Ahead

- File I/O Exercise
- Writing our first servers with Express.js



Testing with Jest

Goals

- Learn to run unit tests using Jest
- Learn more about test matchers and expectations
- Learn to use the browser debugger with Node applications

An Introduction to Jest

Jest

- Jest is an open-source testing library
- It's built on top of Jasmine
- Easy to test in environments that aren't browser-based
- Also very popular for testing React apps
- More information: jestjs.io <<https://jestjs.io>>

Installing Jest

```
$ npm i --global jest
```

This installs jest globally so you can use it anywhere

Organizing Tests

- Test files should be named `{NAME_OF_FILE}.test.js`
 - You can place in the same directory as the JS file it tests
 - Or, you can organize all tests into a folder called `__tests__`
- If you have a `package.json`, you don't need additional configuration.
 - If not, create `jest.config.js` file
 - (*it can be empty, it just needs to exist*).
- Run all tests using the command `jest`
 - You can run an individual test using `jest {NAME_OF_FILE}`

Test

Write tests inside of *test* function callbacks:

```
demo/add.js
```

```
function add(x, y) {  
  return x + y;  
}  
  
module.exports = { add };
```

```
demo/add.test.js
```

```
const { add } = require("./add");  
  
test("add should return sum", function () {  
  let sum = add(2, 3);  
  expect(sum).toEqual(5);  
});
```

Describe

To group together related tests, wrap these in *describe* callback:

```
demo/add.js
```

```
function add(x, y) {  
  return x + y;  
}  
  
module.exports = { add };
```

```
demo/add.test.js
```

```
describe("add function", function () {  
  
  test("return sum", function () {  
    let sum = add(2, 3);  
    expect(sum).toEqual(5);  
  });  
  
  test("return sum w/neg numbers", function () {  
    let sum = add(-2, 3);  
    expect(sum).toEqual(1);  
  });  
});
```

Expectations

```
expect(1 + 1).toEqual(2);
```

Expectations should go inside of *test* function callbacks

A function can have several expectations — but be thoughtful about keeping tests small and simple.

Matchers

.toEqual(obj)

Has the same value (eg, different objects with same values match)

.toBe(obj)

Is the same object (eg, different objects with same values do not)

.toContain(sought)

Does iterable contain this item?

.not.

Add before matcher to invert (eg `expect("hi").not.toEqual("bye")`)

[https://jestjs.io/docs/en/using-matchers <https://jestjs.io/docs/en/using-matchers>](https://jestjs.io/docs/en/using-matchers)

Matchers in action

Let's compare `toBe` and `toEqual`

`matchers.test.js`

```
describe("matchers", function () {
  test("toBe and toEqual are different", function () {
    let nums = [1, 2, 3];
    let newNums = nums.slice();

    expect(nums).not.toBe(newNums); // not the same reference!
    expect(nums).toEqual(newNums); // same values so we use toEqual
  });
});
```

Our advice: prefer `.toEqual()` for most things so that when you do use `.toBe()`, it's obvious that you're really testing for the same object ref.

Expecting Anything

Sometimes, you're not sure what part of an object will be.

Use `expect.any(type)` and it will match any of that type.

`demo/any.js`

```
const TOYS = ["doll", "top", "iPad"];

function getRandomToy() {
  let idx = Math.floor(
    Math.random() * TOYS.length);
  return {
    toy: {
      name: TOYS[idx],
      price: 34.99,
    },
  };
}

module.exports = { getRandomToy };
```

`demo/any.test.js`

```
const { getRandomToy } = require("./any");

test("random toy", function () {
  let toy = getRandomToy();
  expect(toy).toEqual({
    toy: {
      name: expect.any(String),
      price: 34.99,
    },
  });
});
```

Before / After

Demo: Shopping Cart Totals

We have a shopping cart system with function to test:

```
getCartTotal(cart, discount=0)
```

demo/cart.test.js

```
describe("getCartTotal", function () {
  test("total w/o discount", function () {
    const cart = [
      { id: "le croix", price: 4, qty: 3 },
      { id: "pretzels", price: 8, qty: 10 },
    ];

    const total = getCartTotal(cart);
    expect(total).toEqual(92);
  });

  test("total w/discount", function () {
    const cart = [
      { id: "le croix", price: 4, qty: 3 },
      { id: "pretzels", price: 8, qty: 10 },
    ];

    const total = getCartTotal(cart, 0.5);
    expect(total).toEqual(46);
  });
});
```

We can make data in tests:

demo/cart.test.js

```
describe("getCartTotal", function () {
  test("total w/o discount", function () {
    const cart = [
      { id: "le croix", price: 4, qty: 3 },
      { id: "pretzels", price: 8, qty: 10 },
    ];

    const total = getCartTotal(cart);
    expect(total).toEqual(92);
  });

  test("total w/discount", function () {
    const cart = [
      { id: "le croix", price: 4, qty: 3 },
      { id: "pretzels", price: 8, qty: 10 },
    ];

    const total = getCartTotal(cart, 0.5);
    expect(total).toEqual(46);
  });
});
```

Can factor out common setup:

demo/cart.test.js

```
describe("getCartTotal", function () {
  // will hold the cart for the tests
  let cart;

  beforeEach(function () {
    cart = [
      { id: "le croix", price: 4, qty: 3 },
      { id: "pretzels", price: 8, qty: 10 },
    ];
  });

  test("total w/o discount", function () {
    const total = getCartTotal(cart);
    expect(total).toEqual(92);
  });

  test("total w/discount", function () {
    const total = getCartTotal(cart, 0.5);
    expect(total).toEqual(46);
  });
});
```

Before / After

Jest gives us *hooks* we can tap into so we don't repeat common setup/teardown:

- For one-time setup/teardown:
 - *beforeAll*: run once before all tests start
 - *afterAll*: run once after all tests finish
- For frequent setup/teardown:

- `beforeEach`: run before each test starts
- `afterEach`: run after each test finishes

```
describe("my set of tests", function () {
  beforeAll(function() {
    console.log("Run once before all tests");
  });

  beforeEach(function() {
    console.log("Run before each test");
  });

  afterEach(function() {
    console.log("Run after each test");
  });

  afterAll(function() {
    console.log("Run once after all tests");
  });
});
```

Can put these directly in file (outside of functions) to run before/after any/all tests in entire file.

Command Line Options

Jest comes with a few command line options to make testing easier

- `--watch` (useful for detecting changes in a specific file)
- `--watchAll` (useful for detecting changes in any file)

Jest also comes in with a built in report for coverage!

Coverage

If you run your tests with `--coverage` flag, Jest will:

- Output a report in the terminal, showing:
 - Statement coverage: has each statement been executed?
 - Branch coverage: has each branch of control structures been executed?
 - Function coverage: has each function been executed?
 - *Don't worry too much about these; the differences are often minor*
- Generate a report you can view in the browser

Guidelines around coverage

- 100% coverage is an ambitious goal — and not always worth the effort
- Always good: maintain good coverage, especially of bug-magnet parts
 - A good rule of thumb is not to go below 90% coverage

- Use coverage to figure out what has been tested and what hasn't

Debugging Node

Oh No! A Bug!

demo/sumEvens.js

```
// what's wrong with this code??

function testSumEvens(...nums) {
  let sum = 0;

  for (let num in nums) {
    if (num % 2 === 0) {
      sum += num;
    }
  }

  return sum;
}

console.log(testSumEvens(1,2,3,4,5,6));
```

Always Have the Console

```
console.log(msg, msg, ...)
```

Always available; always helpful.

Chrome Dev Tools Debugger

Can also use the Chrome Dev Tools debugger

Start up Node with `--inspect-brk` flag:

```
$ node --inspect-brk sumEvens.js
Debugger listening on ws://127.0.0.1:9229/a98973...
For help, see: https://nodejs.org/en/docs/inspector
```

Open `chrome://inspect` to pull it up in the Chrome Debugger! 🎉🎊🎉



Intro to Express.js

Goals

- Learn what Express.js is
- Compare Express in Node to Flask in Python
- Respond to HTTP requests with route handlers
- Extract request data from the URL or the body
- Handle errors in Express
- Debug Express applications

Express.js

What is Express.js?

- Minimalist web framework
- Conceptually very similar to Flask
- Most popular in the Node ecosystem

Express can be used to build:

- traditional (HTML-returning) applications **or**
- JSON-returning APIs

We built many HTML-returning applications in Flask, so we're going to mostly build APIs in Express.

A Server In A Few Lines

```
const express = require("express");

const app = express();

app.listen(3000, function () {
  console.log("App started at http://localhost:3000/");
});
```

- App doesn't do anything except respond 404s, but server is running!
- *app.listen* takes a port and a callback.
 - Binds the server to port & listens for requests there.
 - Calls callback once server has started up.

How Does Express Work?

Apps

Express apps are like Flask apps:

app.js

```
const express = require("express");
const app = express();
/** Make dogs bark. */
app.get("/dogs", function (req, res) {
  return res.send("Dogs go brk brk");
});
```

app.py

```
from flask import Flask
app = Flask(__name__)
@app.get('/dogs')
def bark():
    """Make dogs bark."""
    return "Dogs go brk brk"
```

Both respond with text (or HTML) to requests at /dogs

Route Handler Callbacks

```
app.get("/dogs", function (req, res) {
  return res.send("Dogs go brk brk");
});
```

- `app.get('/dogs')` listens for a GET request to /dogs endpoint
- Every route has a callback:
 - `req`: request object (query string, url params, form data)
 - `res`: response object (methods for returning html, text, json)
 - It is conventional to name these parameters `req` and `res`
 - Express builds `req/res` for every request and passes to callback
- `res.send("...")` issues response of text or HTML

The Request-Response Cycle

When you start the server, Express runs through the file and registers all the event listeners before the server starts.

When a request is made, Express invokes **first matching route handler** until a response is issued via a method on `res` object.

This is called the *request-response cycle* for Express.

```

app.get("/dogs", function (req, res) {
  return res.send("Dogs go brk brk");
});

// this will never get matched
app.get("/dogs", function (req, res) {
  return res.send("but what about these dogs??");
});

```

First route handler gets matched because it was registered first.

Second handler never matched because a response is issued in the previous handler, thus concluding the request cycle.

Route Methods

Routing by path and HTTP method:

- `app.get(path, callback)`
- `app.post(path, callback)`
- `app.put(path, callback)`
- `app.patch(path, callback)`
- `app.delete(path, callback)`
- `app.all(path, callback)` (all methods)

First Express app

```

$ mkdir first-express-app
$ cd first-express-app
$ npm init -y
$ npm install express@next

```

IMPORTANT Make sure you install `express@next`

We're using the very latest version of Express, which is not yet out of beta. It has some important and helpful differences from the most recent non-beta version (version 4). By asking for "express@next", you'll get the latest version, even if it's still in beta.

An Express app should contain at least two files:

`demo/first-app/app.js`

```
/** Simple demo Express app. */

const express = require("express");
const app = express();

// process JSON body => req.body
app.use(express.json());

// process traditional form data => req.body
app.use(express.urlencoded());

/** Homepage renders simple message. */

app.get("/", function (req, res) {
  return res.send("Hello World!");
});

module.exports = app;
```

`demo/first-app/server.js`

```
const app = require("./app");

app.listen(3000, function () {
  console.log(
    "Started http://localhost:3000/"
);
});
```

Start app with `node server.js`

URL Parameters

Specify parameters in the route by prefixing them with a colon `:`.

`demo/first-app/app.js`

```
/** Show info on instructor. */

app.get("/staff/:fname", function (req, res) {
  return res.send(`This instructor is ${req.params.fname}`);
});
```

- All of our parameters become keys in an object found at `req.params`
- The values are *always* strings

Other Useful Request Properties

- Query string (*GET* requests): `request.query`
- HTTP headers: `request.headers`
- What about the body of the request? POST, PUT, etc

Parsing the Body

Tell Express to parse request bodies for either form data or JSON:

demo/first-app/app.js

```
// process JSON body => req.body
app.use(express.json());

// process traditional form data => req.body
app.use(express.urlencoded());
```

TIP body-parser

A recent update to Express added the method `express.json()`. Previously you had to utilize an add-on library called [body-parser](https://expressjs.com/en/resources/middleware/body-parser.html) `<https://expressjs.com/en/resources/middleware/body-parser.html>` to do this. So if you see body-parser in existing Express code, it does the same thing.

Access form or JSON data with `req.body`:

demo/first-app/app.js

```
/** Add a new instructor. */

app.post("/api/staff", function (req, res) {
  if (req.body === undefined) throw new BadRequestError();
  // ... Do some database operation here...
  // ... then return something as JSON ...
  return res.json({
    fname: req.body.fname,
  });
});
```

ATTENTION req.body, Error Handling, and Optional Chaining

`req.body` defaults to undefined as of the most recent update to Express. Due to this, it is a good idea to check that `req.body` exists and throw a `BadRequestError` if not, as seen above.

There are some cases where you may prefer to assign a variable to `undefined` if there is no request body, rather than throwing an error. This is a great time to use optional chaining:

```
fname = req.body?.fname
```

Optional chaining will check if `req.body` exists before trying to access `.fname` on it. If it does not exist, it will simply evaluate to `undefined` rather than throwing an error for trying to access a property on `undefined`.

Returning JSON

It's straightforward:

demo/first-app/app.js

```
/** Show JSON on instructor */

app.get("/api/staff/:fname", function (req, res) {
  return res.json({ fname: req.params.fname });
});
```

Status Codes

To issue status codes with the responses, call the `res.status(code)` method first, and then chain `.json()` to finish the response.

`demo/first-app/app.js`

```
/** Sample of returning status code */

app.get("/whoops", function (req, res) {
  return res
    .status(404)
    .json({ oops: "Nothing here!" });
```

Error Handling in Express

Validation and Errors

`demo/first-app/app.js`

```
/** Sample of validating / error handling */

app.get("/users/:id", function (req, res) {
  const user = USERS.find(u =>
    u.id === req.params.id
  );

  if (!user) {
    return res.status(404).json({ err: "No such user" });
  }

  return res.json({ user });
});
```

This will work to start, but we can do better!

Throwing errors

An easy way is to `throw` an error!

`demo/first-app/app.js`

```
/** Throwing Error */

app.get("/users2/:id", function (req, res) {
  const user = USERS.find(u =>
    u.id === req.params.id
  );

  if (!user) {
    throw new Error("No such user");
  }

  return res.json({ user });
});
```

Result for /users2/missing

```
{
  "error": {
    "message": "No such user",
    "status": 500
  }
}
```

We'd like that to be a 404, not 500

Using HTTP-specific errors

demo/first-app/app.js

```
/** Throwing NotFoundError */

app.get("/users/:id", function (req, res) {
  const user = USERS.find(u =>
    u.id === req.params.id
  );

  if (!user) {
    throw new NotFoundError("No such user");
  }

  return res.json({ user });
});
```

Result for /users3/missing

```
{
  "error": {
    "message": "No such user",
    "status": 404
  }
}
```



Copy `expressError.js` to project and import `NotFoundError` from it:

```
const { NotFoundError } = require("./expressError");
```

Error classes available:

- `NotFoundError` (404)
- `UnauthorizedError` (401)
- `BadRequestError` (400)
- `ForbiddenError` (403)
- `ExpressError`: create with a message and a status code; used for any other status code to return.

NOTE Our HTTP-specific error classes

These are classes that Rithm wrote and we use throughout our examples and sites. Since Express doesn't include HTTP-specific errors, most users end up creating classes like these for convenience.

It may be interesting to look at the code for these; these all inherit from the base `ExpressError` class, which is a nice OO pattern.

Handling general 404s

To return a 404 when no route matches, put this **below** all other routes:

```
/** 404 handler: matches unmatched routes. */
app.use(function (req, res) {
  throw new NotFoundError();
});
```

Then, since no other route matched, this will match and throw a 404.

Providing errors as useful JSON

Normally, Express returns a plain HTML error page for errors.

To return JSON (with useful info for developers), add a “global error handler”.

Put this **below all routes**, including below the 404 route:

demo/first-app/app.js

```
/** Error handler: logs stacktrace and returns JSON error message. */
app.use(function (err, req, res, next) {
  const status = err.status || 500;
  const message = err.message;
  if (process.env.NODE_ENV !== "test") console.error(status, err.stack);
  return res.status(status).json({ error: { message, status } });
});
```

IMPORTANT Make sure that takes four parameters!

An interesting quirk of Express is that it only treats a function like this as the global error handler if it takes exactly four parameters: (*err, req, res, next*). So, a global error handler must have that as the exact signature.

Later, we’ll cover what the *next* parameter is used for.

An entire Express app

app.js

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

const { NotFoundError } = require("./expressError");

app.use(express.json());                                     // process JSON data
app.use(express.urlencoded());                                // process trad form data

// ... your routes go here ...

app.use(function (req, res) {                               // handle site-wide 404s
  throw new NotFoundError();
});

app.use(function (err, req, res, next) {                   // global err handler
  const status = err.status || 500;
  const message = err.message;
  if (process.env.NODE_ENV !== "test") console.error(status, err.stack);
  return res.status(status).json({ error: { message, status } });
});

module.exports = app;                                      // don't forget this!
```

Debugging Express

- `node --inspect-brk server.js` to start at first line of code

- This is useful for adding breakpoints in browser with *debugger* keyword

```
$ node --inspect-brk server.js
Debugger listening on ws://127.0.0.1:9229/a98973...
For help, see: https://nodejs.org/en/docs/inspector
```

Open *chrome://inspect* to pull it up in the Chrome Debugger! 🎉🎂🎉



Express Middleware and Routers

Goals

- Learn how Express uses (and is built of!) *middleware*
- Learn to write your own middleware
- Learn how to break a large *app.js* into *routers*
- Write integration tests with Supertest

Middleware

Express apps are made up of functions that do **one** of these things:

- Send a response to the browser with `res.send()` or `res.json()`
 - These tend to be your “routes”, like `/user/:id`
- Throw an error to the global error handler
 - Any function that throws an error will end up at the global handler
- Do neither of those things – proceed to the “next” function
 - These functions are called *middleware*
- Print to terminal information about all incoming requests
- Examine request and make JSON in it available
- Examine request to decide if a user is logged in and throw an error if not

Registering functions

app.js

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

// Get a middleware function and register it for all routes
app.use(express.json());

// Log all requests
app.use(myLoggingFunction);

// Log all requests (any HTTP method) when path **starts with** "/cats"
app.use("/cats", myLoggingFunction);

// Handle GET request when path equals "/cats"
app.get("/cats", myViewFn);

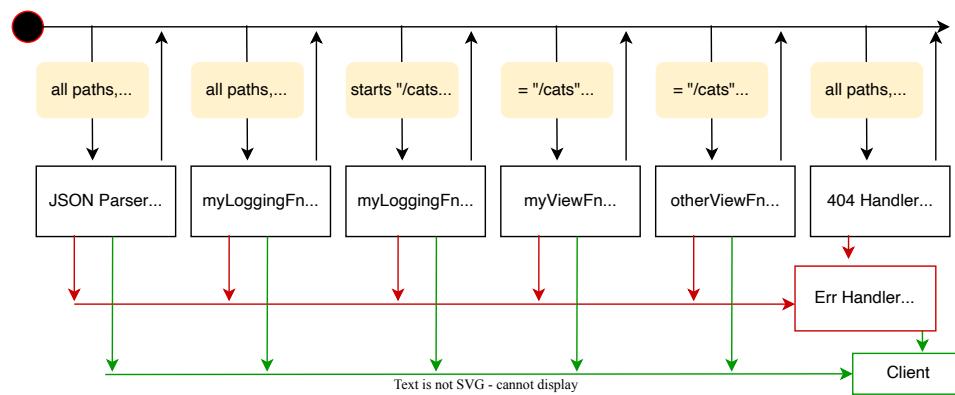
// Handle POST request when path equals "/cats"
app.post("/cats", otherViewFn);

// ... 404 and global err handler ...
```

Each of these ensure a request goes to the “next” thing in the chain of matching registration, ending up (hopefully!) in one of your routes.

The chains

```
app.use(express.json());
app.use(myLoggingFunction);
app.use("/cats", myLoggingFunction);
app.get("/cats", myViewFn);
app.post("/cats", otherViewFn);
app.use(our404Handler);
app.use(globalErrorHandler);
```



- ✖ If an error thrown: goes directly to global error handler
- ✓ If `res.json()` or `res.send()` called, goes directly to client
- ▶ If `next()` function is called, goes to next in chain

WARNING Each routed function must do one of these things!

If you have a function that doesn't any of these things, Express will never return *anything* to the client and may eventually print a cryptic error message to the terminal.

The next function

demo/routing/middleware.js

```
/** Logger: prints log message and goes to next. */

function logger(req, res, next) {
  console.log(`Sending ${req.method} request to ${req.path}.`);
  return next();
}
```

- Express functions can take a third argument (a callback function), *next*
- To tell Express to go to next item in chain, call it!
- This is how a function becomes middleware

NOTE Supplying the *next* function

Only certain functions (middleware) need to list *next* in their function parameters, because only middleware will call that function. A normal view function won't call *next()* — it will call *res.send()* or *res.json()*.

Express will always call a function with all three parameters (*req*, *res*, and *next*), but you don't need to put *next* in your function signature — JavaScript never complains if it passes arguments not in a function signature.

Therefore, for a non-middleware function, both of these will work:

```
app.get("/cats", function(req, res) {
  res.send("meow!");
}

app.get("/cats", function(req, res, next) {
  // doesn't ever use `next` anyway
  res.send("meow!");
})
```

Authorization middleware

demo/routing/middleware.js

```
/** Check that name param must be Elie or raise Unauth. */

function onlyAllowElie(req, res, next) {
  if (req.params.name === "Elie") {
    return next();
  } else {
    throw new UnauthorizedError("Unauthorized");
  }
}
```

We could register that globally with:

```
app.use(onlyAllowElie);
```

But this check might only be useful for one route. Let's do that instead.

Registering middleware for a route

A registration can be of form `app.get(path, fn-1, fn-2, ...)`:

`demo/routing/app.js`

```
/** Greet user, only if it is Elie. */

app.get(
  "/hello/:name",
  // first function -- middleware function
  onlyAllowElie,
  // second function -- our view function
  function (req, res) {
    return res.json({ "greeting": `Hello, ${req.params.name}` });
  }
);
```

Using external middleware

- Instead of our simple logger, we'll a common one, `morgan`
- When using external middleware, we follow a simple process:
 - install it - `npm install morgan`
 - require it - `const morgan = require("morgan");`
 - use it - `app.use(morgan('dev'));`
- You can then see each request logged in the console – route requested, HTTP verb, and more.

Summarizing Middleware

- We've already been using built in middleware like `express.json()`
- Middleware are functions that can intercept the request/response cycle
- When using external middleware: install, `require`, and `use`.

NOTE The router is a piece of middleware!

It's not important to use the router, but it's interesting to realize that a router is an instance of the `Router` class which is callable and takes the usual `req, res, next` function. So the router is a piece of middleware – adding everything routed into the chain of things that might be called as part of servicing a request.

An older use for next

In modern Express, the `next()` function is used just to call the next item in the chain of matching functions, as described above.

In older versions, it had another purpose: it was also how you could signal that an error happened so that you could jump directly to the global error handler.

This was done by calling `next()` but **providing an argument to that call**. When `next()` is called without an argument, it goes to the next item in the chain. When it is called with an argument, it would go to the next item in the chain.

For example:

```
app.get("/donuts", function(req, res, next) {
  // calling next with arg will go to error handler!
  return next(new Error("Donuts not found! Oh no!"));
}
```

This will still work in Express 5, but it's easier just to throw an error rather than fiddling around with `next()` like that:

```
app.get("/donuts", function(req, res) {
  throw new Error("Donuts not found! Oh no!");
}
```

(it would also work if `next` was supplied as the third argument to the latter example; it's just not needed, since it's not called in that version).

You may still see `next()` being used like this in real-world codebases you come across.

Routers

- Placing all routes in `app.js` gets messy quickly!
- Routes can be elsewhere and used in `app.js` !

A Router Outside of `app.js`

`demo/routing/userRoutes.js`

```
const express = require("express");

const db = require("./fakeDb");
const router = new express.Router();

/** GET /users: get list of users */
router.get("/", function (req, res) {
  return res.json(db.User.all());
});

/** DELETE /users/[id]: delete user, return {message: Deleted} */
router.delete("/:id", function (req, res) {
  db.User.delete(req.params.id);
  return res.json({ message: "Deleted" });
});

module.exports = router;
```

Using Our Router in *app.js*

We apply the router to all */users* routes with *app.use*:

demo/routing/app.js

```
// apply a prefix to every route in userRoutes
app.use("/users", userRoutes);
```

Benefits of the Express Router

- We can make our *app.js* file smaller and more readable.
- We can separate different RESTful resources into their own files:
 - */users* has its own router inside *userRoutes.js*
 - */pets* has its own router inside *petRoutes.js*

Integration Tests in Express: Setup

- Making sure the parts work together
- Essential to have along with unit tests!
- More involved than unit tests

Integration Tests with Supertest

- A library for testing Express applications
- Our tool for integration testing
- Like Flask's test client: can make requests against app in tests
- [Supertest Docs <https://github.com/visionmedia/supertest>](https://github.com/visionmedia/supertest)

Installing Supertest

```
$ npm i --save-dev supertest
```

Integration Tests in Express: An Example

The application we are going to be building

- A simple API for CRUD on cats!
- We're going to use a fake database for this example

- We'll abstract that into a file called *fakeDb.js*

demo/testing/fakeDb.js

```
class Cat {
  static deleteAll()
  static all()
  static get(name)
  static add(cat)
  static delete(name)
  static updateName(name, newName)
}
```

What our test setup looks like

demo/testing/routes/cats-routes.test.js

```
const request = require("supertest");
const app = require("../app");
let db = require("../fakeDb");

let pickles = { name: "Pickles" };

beforeEach(function() {
  db.Cat.add(pickles);
});

afterEach(function() {
  db.Cat.deleteAll();
});
```

What should I test?

- Getting all cats
- Getting a single cat
 - What finding successfully looks like
 - What happens when it is not found
- Deleting a cat
 - What deleting successfully looks like
 - What happens when it is not found
- Adding a cat
 - What creating successfully looks like
 - What happens when you create a duplicate cat
 - What happens when you are missing required data

Testing Reading

demo/testing/routes/cats-routes.test.js

```
/** GET /cats - returns '{cats: [cat, ...]}' */

describe("GET /cats", function() {
  it("Gets a list of cats", async function() {
    const resp = await request(app).get('/cats');

    expect(resp.body).toEqual({ cats: [pickles] });
  });
});
```

Testing Creating

demo/testing/routes/cats-routes.test.js

```
/** POST /cats - create cat from data; return '{cat: cat}' */

describe("POST /cats", function() {
  it("Creates a new cat", async function() {
    const resp = await request(app)
      .post('/cats')
      .send({
        name: "Ezra"
      });
    expect(resp.statusCode).toEqual(201);
    expect(resp.body).toEqual({
      cat: { name: "Ezra" }
    });
  });
});
```

Testing Updating

demo/testing/routes/cats-routes.test.js

```
/** PATCH /cats/:name - update cat; return '{cat: cat}' */

describe("PATCH /cats/:name", function() {
  it("Updates a single cat", async function() {
    const resp = await request(app)
      .patch(`/cats/${pickles.name}`)
      .send({
        name: "Troll"
      });
    expect(resp.body).toEqual({
      cat: { name: "Troll" }
    });
  });

  it("Responds with 404 if name invalid", async function() {
    const resp = await request(app).patch('/cats/not-here');
    expect(resp.statusCode).toEqual(404);
  });
});
```

Testing Deleting

demo/testing/routes/cats-routes.test.js

```
/** DELETE /cats/[name] - delete cat,
 *  return `{message: "Cat deleted"}` */

describe("DELETE /cats/:name", function() {
  it("Deletes a single a cat", async function() {
    const resp = await request(app)
      .delete(`/cats/${pickles.name}`);
    expect(resp.body).toEqual({ message: "Deleted" });
    expect(db.Cat.all().length).toEqual(0);
  });
});
```

Debugging your tests

- You can always `console.log` inside of your test files
- If you'd like to use the chrome dev tools, write the following:
 - `node --inspect-brk $(which jest) --runInBand NAME_OF_FILE`

Coming Up

- Adding PostgreSQL to Express
- Testing using a Database



PostgreSQL with Node

Goals

- Use *pg* to connect and execute SQL queries
- Explain what SQL injection is and how to prevent it with *pg*
- Examine CRUD on a single resource with Express and *pg*

Introduction

The Node SQL Ecosystem

There are different ways to talk to the database:

- Use an ORM (what we did with Flask)
- Write the SQL yourself (what we will be doing in Express!)
- You can [read more about this <https://www.rithmschool.com/blog/different-approaches-express>](https://www.rithmschool.com/blog/different-approaches-express) from Joel

PostgreSQL with Node

Scaffolding

```
demo/simple/app.js
/** Express app for pg-intro-demo */

const express = require("express");
const { NotFoundError } = require("./expressError");

const app = express();

// Parse request bodies for JSON
app.use(express.json());

const uRoutes = require("./routes/users");
app.use("/users", uRoutes);

// ... 404, global err handler, etc.
```

pg

- Similar to *psycopg2* with python

- Connects to database and executes SQL

```
$ npm install pg
```

It's common to abstract this logic to another file, so let's create a file `db.js` :

`demo/simple/db.js`

```
/** Database setup for users. */
const { Client } = require("pg");

const DB_URI = process.env.NODE_ENV === "test" // 1
  ? "postgresql://simple_users_test"
  : "postgresql://simple_users";

let db = new Client({
  connectionString: DB_URI
});

db.connect() // 2

module.exports = db; // 3
```

1. Specify a database
 - Using environmental var
 - Don't trash real DB during testing
2. Connect to DB
3. Export DB to other files

Queries

`demo/simple/routes/users.js`

```
const db = require("../db");
```

(results)

```
[nothing!]
```

`demo/simple/routes/users.js`

```
/** Get users: [user, user, user] */

router.get("/all", function (req, res, next) {
  const results = db.query(
    `SELECT id, name, type
     FROM users`);
  console.log("results=", results);
  const users = results.rows;

  console.log("users=", users);
  return res.json({ users });
});
```

What's the bug?

DB queries are asynchronous! We have to wait for the query to finish before!

Fixing with async/await

demo/simple/routes/users.js

```
/** (Fixed) Get users: [user, user, user] */

router.get("/", async function (req, res, next) {
  const results = await db.query(
    `SELECT id, name, type
     FROM users`);

  console.log("results=", results);

  const users = results.rows;

  console.log("rows=", users);

  return res.json({ users });
});
```

(results)

```
{
  users: [
    {
      "id": 1,
      "name": "Juanita",
      "type": "admin"
    },
    {
      "id": 2,
      "name": "Jenny",
      "type": "staff"
    },
    {
      "id": 3,
      "name": "Jeff",
      "type": "user"
    }
  ]
}
```

demo/simple/routes/users.js

```
/** Search by user type. */

router.get("/search",
  async function (req, res, next) {
    const type = req.query.type;

    const results = await db.query(
      `SELECT id, name, type
       FROM users
      WHERE type = '${type}'`);

    const users = results.rows;

    return res.json({ users });
});
```

(for 'staff')

```
{
  users: [
    {
      "id": 2,
      "name": "Jenny",
      "type": "staff"
    }
  ]
}
```

But there's a problem...

SQL Injection

A technique where an attacker tries to execute undesirable SQL statements on your database.

It's a common attack, and it's easy to be vulnerable if you aren't careful!

If the search type is `"staff"`, everything works fine.

But what if the search type is `"bwah-hah'; DELETE FROM users; --"`?

```
SELECT id, type, name
FROM users
WHERE type='bwah-hah'; DELETE FROM users; --'
```

Ut oh.

Parameterized Queries

- To prevent against SQL injection, we need to sanitize our inputs
- ORMs typically do this for you automatically
- We can sanitize our inputs by using **parameterized queries**

NOTE Prepared Statements

It's not the most important part to understand, but if you're curious how the `pg` module does this, it uses a feature called "prepared statements".

Prepared statements are a database tool used to template and optimize queries you plan on running frequently. You've seen prepared statements already when we worked with SQLAlchemy in Flask, though we didn't specifically call them out as such.

You don't need to worry about the details, but because of the way that prepared statements work on the database level, they naturally protect against SQL injection. If you're curious about the details, check out [this <https://en.wikipedia.org/wiki/Prepared_statement>](https://en.wikipedia.org/wiki/Prepared_statement) article on Wikipedia.

Here's the same approach, but safe from SQL injection.

`demo/simple/routes/users.js`

```
/** (Fixed) Search by user type. */

router.get("/good-search",
  async function (req, res, next) {
    const type = req.query.type;

    const results = await db.query(
      `SELECT id, name, type
       FROM users
       WHERE type = $1`, [type]);
    const users = results.rows;
    return res.json({ users });
  }
)
```

(results for 'staff')

```
{
  users: [
    {
      "id": 2,
      "name": "Jenny",
      "type": "staff"
    }
  ]
}
```

Parameterized Queries Overview

- In your SQL statement, represent variables like `$1`, `$2`, `$3`, etc.
 - You can have as many variables as you want
- For the second argument to `db.query`, pass an array of values
 - `$1` will evaluate to the first array element, `$2` to the second, etc.
- **Note:** the variable naming is 1-indexed!

More CRUD Actions

Example: Create

`demo/simple/routes/users.js`

```
/** Create new user, return user */

router.post("/", async function (req, res, next) {
  console.log("*** POST / req.body:", req.body);
  if (!req.body) throw new BadRequestError();

  const { name, type } = req.body;
  const result = await db.query(
    `INSERT INTO users (name, type)
      VALUES ($1, $2)
      RETURNING id, name, type`,
    [name, type],
  );
  const user = result.rows[0];
  return res.status(201).json({ user });
});
```

NOTE Status Code 201

Note that we use HTTP status code 201 (`CREATED`) here, not 200 (`OK`).

Some APIs return 200 in cases like this, but the REST convention suggests that 201 is a better choice, so that's what we're using.

RETURNING clause

In SQL, for INSERT/UPDATE/DELETE, you can have a *RETURNING* clause.

This is to *return data* that was inserted, updated, or deleted:

```
INSERT INTO users (name, type) VALUES (...) RETURNING id, name;
```

```
INSERT INTO users (name, type) VALUES (...) RETURNING *;
```

DANGER Never use `*` in RETURNING clause

It's a terrible idea to use `SELECT *` or `RETURNING *` in the SQL used in applications. That returns all columns and, if new sensitive columns were added after the code was written, it would risk returning that sensitive data.

It's far better to explicitly list the columns that should be selected or returned.

Example: Update

demo/simple/routes/users.js

```
/** Update user, returning user */

router.patch("/:id", async function (req, res, next) {
  if (req.body === undefined) throw new BadRequestError();
  const { name, type } = req.body;

  const result = await db.query(
    `UPDATE users
      SET name=$1,
          type=$2
      WHERE id = $3
      RETURNING id, name, type`,
    [name, type, req.params.id],
  );
  const user = result.rows[0];
  return res.json({ user });
});
```

Example: Delete

demo/simple/routes/users.js

```
/** Delete user, returning {message: "Deleted"} */

router.delete("/:id", async function (req, res, next) {
  await db.query(
    "DELETE FROM users WHERE id = $1",
    [req.params.id],
  );
  return res.json({ message: "Deleted" });
});
```

Committing

With SQLAlchemy, you had to commit after all changes — because SQLAlchemy put all work into a db transaction.

That isn't the case with *pg* — so you don't need to explicitly commit (each INSERT/UPDATE/DELETE commits automatically)

Testing our Database

Adding test database

We're going to need a different database for testing, so let's configure that!

demo/cats/db.js

```
/** Database setup for users. */

const { Client } = require("pg");

const DB_URI = process.env.NODE_ENV === "test"
  ? "postgresql://cats_test"
  : "postgresql://cats";

let db = new Client({
  connectionString: DB_URI
});

db.connect();

module.exports = db;
```

Running Tests

Create test database first, otherwise running tests will mysteriously hang.

```
$ createdb cats_test
$ psql cats_test -f cats.sql
```

Once you have database, run your tests as usual:

```
$ jest -i
```

TIP Always include the `-i`

The `-i` flag tells Jest to run your tests “in-line” (that is, only running one test file at a time, rather than running several test files simultaneously).

If you let it run multiple tests files at the same time, some tests will fail because the database will suddenly change in the middle of a test.

Test Setup/Teardown

Setup at beginning:

demo/cats/routes/cats.test.js

```
let testCat;

beforeEach(async function () {
  await db.query("DELETE FROM cats");
  let result = await db.query(`
    INSERT INTO cats (name)
    VALUES ('TestCat')
    RETURNING id, name`);
  testCat = result.rows[0];
});
```

Testing CRUD Actions

Our Restful JSON API

What routes do we need for a RESTful JSON API with CRUD on cats? (ZOMG so many acronyms.)

HTTP Verb	Route	Response
GET	/cats	Display all cats
GET	/cats/:id	Display a cat
POST	/cats	Create a cat
PUT / PATCH	/cats/:id	Update a cat
DELETE	/cats/:id	Delete a cat

Testing Read

demo/cats/routes/cats.test.js

```
/** GET /cats - returns '{cats: [cat, ...]}' */

describe("GET /cats", function () {
  test("Gets list", async function () {
    const resp = await request(app).get('/cats');
    expect(resp.body).toEqual({
      cats: [testCat],
    });
  });
});
```

demo/cats/routes/cats.test.js

```
/** GET /cats/[id] - return data about one cat: '{cat: cat}' */

describe("GET /cats/:id", function () {
  test("Gets single cat", async function () {
    const resp = await request(app).get(`/cats/${testCat.id}`);
    expect(resp.body).toEqual({ cat: testCat });
  });

  test("404 if not found", async function () {
    const resp = await request(app).get('/cats/0');
    expect(resp.statusCode).toEqual(404);
  });
});
```

Testing Create

demo/cats/routes/cats.test.js

```
/** POST /cats - create cat from data; return '{cat: cat}' */

describe("POST /cats", function () {
  test("Create new cat", async function () {
    const resp = await request(app)
      .post(`/cats`)
      .send({ name: "Ezra" });
    expect(resp.statusCode).toEqual(201);
    expect(resp.body).toEqual({
      cat: { id: expect.any(Number), name: "Ezra" },
    });
  });

  test("400 if empty request body", async function () {
    const resp = await request(app)
      .post(`/cats`)
      .send();
    expect(resp.statusCode).toEqual(400);
  });
});
```

Testing Update

demo/cats/routes/cats.test.js

```
/** PATCH /cats/:id - update cat; return '{cat: cat}' */

describe("PATCH /cats/:id", function () {
  test("Update single cat", async function () {
    const resp = await request(app)
      .patch(`/cats/${testCat.id}`)
      .send({ name: "Troll" });
    expect(resp.statusCode).toEqual(200);
    expect(resp.body).toEqual({
      cat: { id: testCat.id, name: "Troll" },
    });
  });

  test("404 if not found", async function () {
    const resp = await request(app)
      .patch(`/cats/0`)
      .send({name: "Troll"});
    expect(resp.statusCode).toEqual(404);
  });

  test("400 if empty request body", async function () {
    const resp = await request(app)
      .patch(`/cats/${testCat.id}`)
      .send();
    expect(resp.statusCode).toEqual(400);
  });
});
```

Testing Delete

demo/cats/routes/cats.test.js

```
/** DELETE /cats/[id] - delete cat,
 *  return `{message: "Cat deleted"}' */

describe("DELETE /cats/:id", function () {
  test("Delete single cat", async function () {
    const resp = await request(app)
      .delete(`/cats/${testCat.id}`);
    expect(resp.statusCode).toEqual(200);
    expect(resp.body).toEqual({ message: "Cat deleted" });
  });
});
```

Looking Ahead

- Associations with pg
- Building our own lightweight ORM!

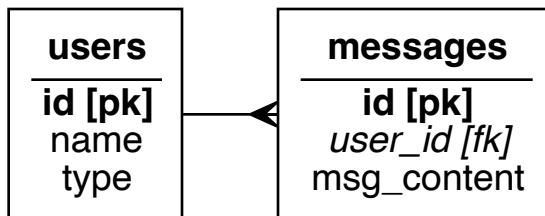


Express Postgres Relationships

Goals

- Work with 1:M relationships in pg
- Work with M:M relationships in pg
- Handle missing data by sending 404s

One to Many Relationships



```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE,
    type TEXT NOT NULL
);

CREATE TABLE messages (
    id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL REFERENCES users,
    msg_content TEXT NOT NULL
);
```

We want our API to include:

GET /users/1

Return detail of user *and* list of message:

```
{
  user: {
    id: 1,
    name: "Juanita",
    type: "admin",
    messages: [
      {id: 1, msg_content: 'msg #1'},
      {id: 2, msg_content: 'msg #2'}
    ]
  }
}
```

GET /users/[id] With Messages

demo/routes/users.js

```
/** Get User: => {
 *   user: {
 *     id,
 *     name,
 *     type,
 *     messages: [ {id, msg_content} ]
 *   }
 * } */

router.get("/:id",
  async function (req, res, next) {
    const id = req.params.id;
    const uResults = await db.query(`SELECT id, name, type
      FROM users
      WHERE id = $1`, [id]);
    const user = uResults.rows[0];

    const mResults = await db.query(`SELECT id, msg_content
      FROM messages
      WHERE user_id=$1`, [id]);
    const messages = mResults.rows;

    user.messages = messages;
    return res.json({ user });
  });
}
```

(results)

```
{
  user: {
    id: 1,
    name: "Juanita",
    type: "admin",
    messages: [
      {id: 1, msg_content: 'm#1'},
      {id: 2, msg_content: 'm#2'}
    ]
  }
}
```

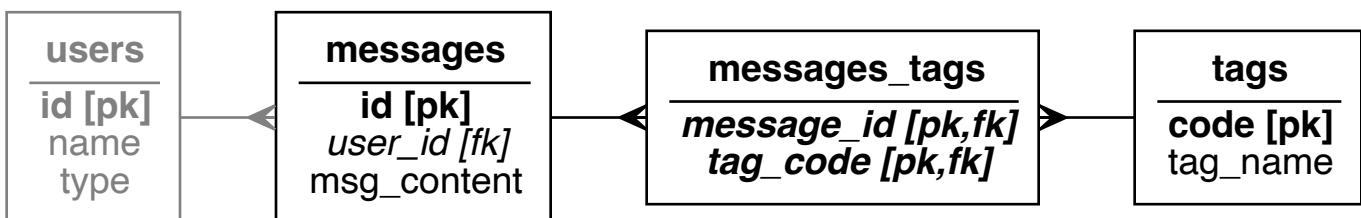
We just add property on user and populate with messages!

NOTE Optimizing this code

For ease of readability, we've awaited two database queries sequentially in the above code example. We could have just as easily run these queries in parallel by wrapping them in a `Promise.all`, since the message query doesn't depend on the result of the user query.

You might also be wondering why we don't use a join, and simply make one request to the database. What would be some advantages to using this approach? What might some disadvantages be?

Many to Many Relationships

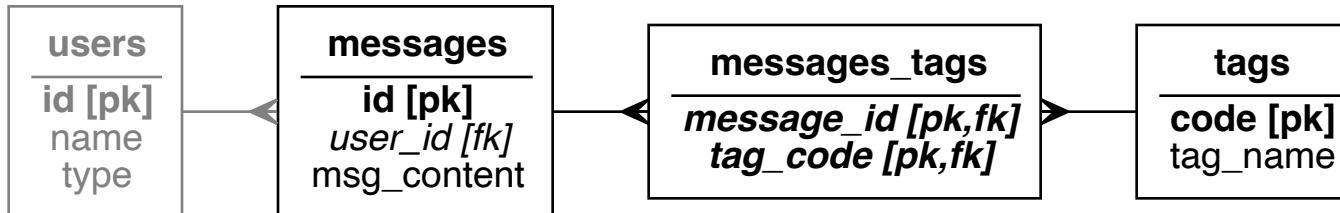


```

CREATE TABLE tags (
    code TEXT PRIMARY KEY,
    tag_name TEXT NOT NULL UNIQUE
);

CREATE TABLE messages_tags (
    message_id INTEGER NOT NULL REFERENCES messages,
    tag_code TEXT NOT NULL REFERENCES tags,
    PRIMARY KEY(message_id, tag_code)
);

```



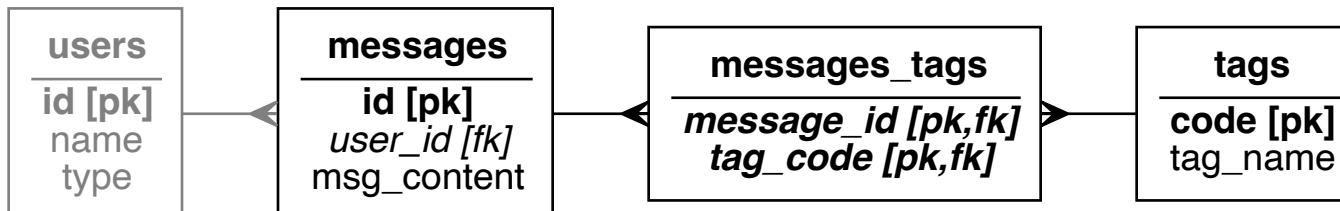
We want our API to include:

`GET /messages/1`

Return info about message *and* associated tag names:

```
{
  message: {
    id: 1,
    msg_content: "msg #1",
    tags: ["Python", "JavaScript"]
  }
}
```

What about these queries?



```

SELECT id, msg_content
FROM messages
WHERE id = 1;

SELECT tag_name
FROM messages_tags AS mt
JOIN tags AS t ON mt.tag_code = t.code
WHERE mt.message_id = 1;

```

Restructuring This Data

we get from database

```
messages = [
  {id: 1, msg_content: "msg #1"}
]

tags = [
  {tag_name: "Python"},
  {tag_name: "JavaScript"},  
]
```

(we want our API to return)

```
{
  message: {
    id: 1,
    msg_content: "msg #1",
    tags: ["Python", "JavaScript"]
  }
}
```

demo/routes/messages.js

```
/** Get message: {message: {id, msg_content, tags: [name, name]} } */

router.get("/:id", async function (req, res, next) {
  const id = req.params.id;
  const mResults = await db.query(
    `SELECT id, msg_content
      FROM messages
      WHERE id = $1`, [id]);
  const message = mResults.rows[0];

  const tResults = await db.query(
    `SELECT tag_name
      FROM messages_tags AS mt
      JOIN tags AS t ON mt.tag_code = t.code
      WHERE mt.message_id = $1
      ORDER BY tag_name`, [id]);
  message.tags = tResults.rows.map(r => r.tag_name);

  return res.json({ message });
});
```

NOTE Manipulating data

When it comes to handling these many-to-many relationships, you'll find that you often need to manipulate arrays of objects in JavaScript. There are many helper libraries with utilities that can assist with this process (such as [lodash](https://lodash.com/) <<https://lodash.com/>>), but for now, we'll focus on writing all of the business logic ourselves.

Could We Write This in One Query?

Yes! But it might not be better.

If we wrote this as one query joining all three tables, we'd get a result set repeating the message id and message text. If the message text isn't very long, or there aren't many tags for that message, there might be a small improvement to be made with one query rather than two.

However, if the message text was a page-letter note, and there were a dozen tags on it, that repeated message text would take longer (and more memory!) for the database to send that any savings from writing this as one query.

So: to really know which would be better, you need to think about your data.

Handling Missing Resources

We want:

PATCH /messages/[id]

Given {msg_content}, updates DB & return {message: {id, user_id, msg_content}}

demo/routes/messages.js

```
/** Update message:
 * {msg_content} => {message: {id, user_id, msg_content}} */

router.patch("/:id", async function (req, res, next) {
  if (req.body === undefined) throw new BadRequestError();
  const id = req.params.id;
  const results = await db.query(
    `UPDATE messages
      SET msg_content=$1
      WHERE id = $2
      RETURNING id, user_id, msg_content`,
    [req.body.msg_content, id]);
  const message = results.rows[0];

  return res.json({ message });
});
```

Just returns *undefined* if not found!

demo/routes/messages.js

```
/** Update message #2:
 * {msg_content} => {message: {id, user_id, msg_content}} */

router.patch("/v2/:id", async function (req, res, next) {
  if (req.body === undefined) throw new BadRequestError();
  const id = req.params.id;
  const results = await db.query(
    `UPDATE messages
      SET msg_content=$1
      WHERE id = $2
      RETURNING id, user_id, msg_content`,
    [req.body.msg_content, id]);
  const message = results.rows[0];

  if (!message) throw new NotFoundError(`Not found: ${id}`);
  return res.json({ message });
});
```

NOTE Don't Forget about the debugger!

Remember, if this code starts to become too hard to track, you can use the *debugger* to pause code execution and see what's going on!



Database OO Design Patterns

Goals

- Refactor our Express apps to separate view logic (*routing*) from model logic (*data*)
- Compare different OO designs for interfacing with our database
- Borrow useful ideas from ORMs to build our own model layers

Current Design

routes

```
/** get all cats: [{id, name, age}, ...] */

router.get("/meh", async function (req, res, next) {
  const result = await db.query("SELECT id, name, age FROM cats");
  const cats = result.rows;
  return res.json(cats);
});
```

It's *ok*, but it's better to get SQL out of routes

Why No SQL In Routes?

- You tend to have lots of routes
 - So lots of copy-and-paste of similar SQL
- It's nice to centralize validation, schema, etc
- Separation of concerns: routes should be about web-stuff

Object Orientation Review

Why do we use Object Orientation?

To help organize our code!

Often makes it easier to manage complex software requirements

Abstraction

OO can offer abstraction (*to hide implementation details when not needed*)

- Not everyone should have to understand everything

- Only one person has to worry about SQL, validation, etc

Encapsulation

OO can offer **encapsulation** (*to group functionality into logical pieces*)

- To get in a “capsule”
 - Everything related to cat data/functionality lives in *Cat*

Inheritance

OO can offer **inheritance** (*ability to call methods/get properties defined on ancestors*)

- One class subclasses another, inheriting properties and methods
 - A *ColoredTriangle* is a kind of *Triangle*; it should have sides a and b, and can use *Triangle* methods for getting the area and hypotenuse
 - These are *inherited*; we only need to define what is different about *ColoredTriangle*

Polymorphism

Meh...

```
class Dog {
  bark() { return "woof!" }
}

class Cat {
  meow() { return "meow!" }
}

class Snake {
  hiss() { return "Sssssss...." }
}

// make all animals make noise
for (const pet of pets) {
  if (pet instanceof Dog) pet.bark();
  if (pet instanceof Cat) pet.meow();
  if (pet instanceof Snake) pet.hiss();
}
```

OO can offer **polymorphism** (*making classes interchangeable when similar*)

- The ability to make similar things work similarly
 - We could have other kinds of animals with same API
 - eg dogs and cats could both have a *speak()* method, even though it works differently (“Meow” vs “Woof”)

Yay, polymorphism!

```
class Dog {
  speak() { return "woof!" }
}

class Cat {
  speak() { return "meow!" }
}

class Snake {
  speak() { return "Sssssss...." }
}

// make all animals make noise --- MUCH BETTER
for (const pet of pets) {
  pet.speak();
}
```

Simple OO Model

- We can make a single class for “all cat-related functions”
- It *won’t* hold data
- You won’t ever instantiate it!
- All the methods are static (called on *Cat*)
- Benefit: help organization, gets SQL out of routes

Getting All Cats

Cat model

```
/** get all cats: returns [{id, name, age}, ...] */

static async getAll() {
  const result = await db.query(
    `SELECT id, name, age
     FROM cats
     ORDER BY id`);
  return result.rows;
}
```

(that’s a method inside class *Cat*)

routes

```
/** (fixed) get all cats: [{id, name, age}] */

router.get("/", async function (req, res, next) {
  const cats = await Cat.getAll();
  return res.json(cats);
});
```

Getting A Cat

Cat model

```
/** get cat by id: returns {id, name, age} */

static async getById(id) {
  const result = await db.query(
    `SELECT id, name, age
     FROM cats
     WHERE id = $1`, [id]);
  const cat = result.rows[0];

  if (!cat) throw new NotFoundError(`No such cat: ${id}`);
  return result.rows[0];
}
```

routes

```
/** get cat by id: {id, name, age} */

router.get("/:id", async function (req, res, next) {
  const cat = await Cat.getById(req.params.id);
  return res.json(cat);
});
```

Creating a Cat

Cat model

```
/** create a cat: returns {id, name, age} */

static async create(name, age) {
  const result = await db.query(
    `INSERT INTO cats (name, age)
     VALUES
     ($1, $2)
     RETURNING id, name, age`, [name, age]);
  const cat = result.rows[0];

  return cat;
}
```

routes

```
/** create cat from {name, age}: return {name, age} */

router.post("/", async function (req, res, next) {
  if (req.body === undefined) throw new BadRequestError();
  const cat = await Cat.create(req.body.name, req.body.age);
  return res.json(cat);
});
```

Deleting a Cat

Cat model

```
/** delete cat with given id */

static async remove(id) {
  const result = await db.query(
    `DELETE
     FROM cats
     WHERE id = $1
     RETURNING id`, [id]);
  const cat = result.rows[0];

  if (!cat) throw new NotFoundError(`No such cat: ${id}`);
}
```

routes

```
/** delete cat from {id}; returns "deleted" */

router.delete("/:id", async function (req, res, next) {
  await Cat.remove(req.params.id);
  return res.json("deleted");
});
```

Aging a Cat

What if we want to something special?

Like, age a cat by one year?

Cat model

```
/** age cat by 1 year, return new age */

static async makeOlder(id) {
  const result = await db.query(
    `UPDATE cats
     SET age=age + 1
     WHERE id = $1
     RETURNING age`, [id]);
  const cat = result.rows[0];

  if (!cat) throw new NotFoundError(`No such cat: ${id}`);
  return cat.age;
}
```

routes

```
/** age cat: returns new age */

router.post("/:id/age", async function (req, res, next) {
  const newAge = await Cat.makeOlder(req.params.id);
  return res.json(newAge);
});
```

Meh. Annoying to have to make special function.

We could make a special “update-data” function.

Smarter OO Model

- We can make a more traditional OO class
- You *will* instantiate it — once per dog!
- It will hold data specific to each dog
- It has static methods
 - To get all dogs, get a particular dog
- It has regular methods
- It's like a mini-ORM

Simple vs. Smarter

using a simple oo model

```
// will not hold data
// you never create an instance
// all methods are static

// POJO with data about fluffy
const fluffyData = await Cat.getById(id);

await Cat.remove(id);

await Cat.makeOlder();
```

using a smarter oo model

```
// will hold data
// you will create an instance
// has both static and regular methods

// instance of the Dog class
const fido = await Dog.getById(id);

fido.age += 1;
await fido.save();

await fido.remove();
```

Dogs

We'll make a “smarter model” for dogs.

Dog model

```
constructor({ id, name, age }) {
  this.id = id;
  this.name = name;
  this.age = age;
}
```

Getting All Dogs

Dog model

```
/** get all dogs: returns [dog, ...] */

static async getAll() {
  const result = await db.query(
    `SELECT id, name, age
     FROM dogs`);
  return result.rows.map(dog => new Dog(dog));
}
```

routes

```
/** get all dogs: [{id, name, age}, ...] */

router.get("/", async function (req, res, next) {
  const dogs = await Dog.getAll();
  return res.json(dogs);
});
```

We get Dog instances, but Express can turn them into JSON

Getting A Dog

Dog model

```
/** get dog by id: returns dog */

static async getById(id) {
  const result = await db.query(
    `SELECT id, name, age
     FROM dogs
     WHERE id = $1`, [id]);
  const dog = result.rows[0];

  if (!dog) throw newNotFoundError(`No such dog: ${id}`);
  return new Dog(dog);
}
```

routes

```
/** get dog by id: {id, name, age} */

router.get("/:id", async function (req, res, next) {
  const dog = await Dog.getById(req.params.id);
  return res.json(dog);
});
```

Creating a Dog

Dog model

```
/** create a dog: returns dog */

static async create(name, age) {
  const result = await db.query(
    `INSERT INTO dogs (name, age)
      VALUES
      ($1, $2)
      RETURNING id, name, age`, [name, age]);
  const dog = result.rows[0];

  return new Dog(dog);
}
```

routes

```
/** create dog from {name, age}: return id */

router.post("/", async function (req, res, next) {
  if (req.body === undefined) throw new BadRequestError();
  const dog = await Dog.create(req.body.name, req.body.age);
  return res.json(dog);
});
```

Deleting a Dog

Dog model

```
/** delete dog */

async remove() {
  await db.query(`DELETE FROM dogs WHERE id = $1`, [this.id]);
}
```

routes

```
/** delete dog from {id}; returns "deleted" */

router.delete("/:id", async function (req, res, next) {
  const dog = await Dog.getById(req.params.id);
  await dog.remove();
  return res.json("deleted");
});
```

Notice: it's just a method that acts on current dog!

Aging a Dog

Now, we don't need special functionality to age a dog

We can just age on instance and `.save()` it!

Dog model

```
async save() {
  await db.query(
    `UPDATE dogs
     SET name=$1,
        age=$2
    WHERE id = $3`, [this.name, this.age, this.id]);
}
```

routes

```
/** age dog: returns new age */

router.post("/:id/age", async function (req, res, next) {
  let dog = await Dog.getById(req.params.id);
  dog.age += 1;
  await dog.save();
  return res.json(dog.age);
});
```

Simple vs. Smarter

using a simple oo model

```
// will not hold data
// you never create an instance
// all methods are static

// POJO with data about fluffy
const fluffyData = await Cat.getById(id);

await Cat.makeOlder(id);

await Cat.remove(id);
```

using a smarter oo model

```
// will hold data
// you will create an instance
// has both static and regular methods

// instance of the Dog class
const fido = await Dog.getById(id);

fido.age += 1;
await fido.save();

await fido.remove();
```

Which Is Better?

- “Simple class” (*no data, only static methods*)
 - Can be easier to write class
 - Fewer SQL queries may fire (compare delete between *Cat* and *Dog*)
 - Doing more interesting things can be trickier
- “Smarter class” (*data, real methods*)
 - Real attributes can be handy!
 - Can do things like `fluffy.speak()` rather than `Cat.speak(id)`

Are There ORMs For JavaScript?

Yes!

There's a nice one called [Sequelize <http://docs.sequelizejs.com>](http://docs.sequelizejs.com)

Not as popular as ORMs in other languages, though.



Hashing and JWTs with Node

Goals

- Hash passwords with Bcrypt
- Using JSON web tokens for API authentication

Password Hashing with Bcrypt

Similar to Flask, but with asynchronous API.

To use, install library:

```
$ npm install bcrypt
```

Import bcrypt:

```
const bcrypt = require("bcrypt");
```

```
bcrypt.hash(password-to-hash, work-factor)
```

Hash password, using work factor (12 is a good choice).

Returns *promise* – resolve to get hashed password.

```
bcrypt.compare(password, hashed-password)
```

Check if password is valid.

Returns *promise* – resolve to get boolean.

Hashing Password

demo/auth-api/routes/auth.js

```
/** Register user.
 *  {username, password} => {username} */

router.post("/register", async function (req, res, next) {
  if (req.body === undefined) throw new BadRequestError();
  const { username, password } = req.body;
  const hashedPassword = await bcrypt.hash(
    password, BCRYPT_WORK_FACTOR);
  const result = await db.query(
    `INSERT INTO users (username, password)
      VALUES
      ($1, $2)
      RETURNING username`,
    [username, hashedPassword]);

  return res.json(result.rows[0]);
});
```

Logging in

- Try to find user first
 - If exists, compare hashed password to hash of login password
- `bcrypt.compare()` resolves to boolean—if *true*, passwords match!

`demo/auth-api/routes/auth.js`

```
/** Login: returns {message} on success. */

router.post("/login-1", async function (req, res, next) {
  if (req.body === undefined) throw new BadRequestError();
  const { username, password } = req.body;
  const result = await db.query(
    `SELECT password
     FROM users
     WHERE username = $1`,
    [username]);
  const user = result.rows[0];

  if (user) {
    if (await bcrypt.compare(password, user.password) === true) {
      return res.json({ message: "Logged in!" });
    }
  }
  throw new UnauthorizedError("Invalid user/password");
});
```

JSON Web Tokens

Authentication Via Cookies

- Make request with username/password to login route
- Server authenticates & puts user info session
 - Session is encoded & signed
- Session info is sent back to browser in cookie
- Session info is automatically resent with every request via cookie
- This works great for traditional web apps & is straightforward to do
- What if
 - We didn't want to send auth info with certain requests?
 - We wanted to share authentication info across multiple APIs / hostnames?
- We'll use a more API-server friendly process!

Authentication Via Tokens

For our Express API apps, we'll handle authentication differently:

- Make request with username/password to AJAX login route

- Server authenticates & returns token in JSON
 - Token is encoded & signed with open standard, “JSON Web Token”
- Front-end JavaScript receives token & stores (*in var or localStorage*)
- For future requests that need it, JS can send token in request
 - Server gets token from request & validates token

JSON Web Tokens

[Homepage of JSON Web Tokens <https://jwt.io/>](https://jwt.io/)

JWTs are an open standard and are implemented across technology stacks, making it simple to share tokens across different services.

JWTs can store any arbitrary “payload” of info, which are “signed” using a secret key, so they can be validated later (similar to Flask’s session).

The JWT token itself is a string comprising three parts:

- **Header:** metadata about token (*signing algorithm used & type of token*)
- **Payload:** data to be stored in token (*typically an object*)
 - Often, this will store things like the user ID
 - This is *encoded*, not *encrypted* — don’t put secret info here!
- **Signature:** version of header & payload, signed with secret key
 - Uses algorithm specified in header (*we’ll use default, “HMAC-SHA256”*)

NOTE JWTs Versus Flask sessions

JWTs do the same process as a Flask session: encode the payload and sign it using a secret key. Flask’s built-in session uses a Flask-specific encoding and signing algorithm, but there are add-on products for Flask to use JWTs as the encoding/signing scheme for sessions.

The bigger difference is in how this is transmitted: Flask’s standard sessions are transmitted via cookies, so they are passed automatically between the server and the browser. The JWT standard isn’t involved itself with how tokens are sent — this is up to the application developer. We’ll be doing so by sending these in the request manually, and retrieving them manually from the request in the server.

None of this is inherently specific to Flask or Express — there are cookie-based authentication add-ons for Express, and there are JWT libraries for Python, so Flask could emit JWTs for API-based server.

For more information, here’s a good [discussion of JWTs versus server-side sessions <https://stackoverflow.com/questions/43452896/authentication-jwt-usage-vs-session>](https://stackoverflow.com/questions/43452896/authentication-jwt-usage-vs-session).

Using JWTs

Install JSON web token:

```
$ npm install jsonwebtoken
```

Creating Tokens

```
jwt.sign(payload, secret-key, jwt-options)
```

payload: object to store as payload of token

secret-key: secret string used to “sign” token

jwt-options is optional object of settings for making the token

This returns the token (a string)

```
const jwt = require("jsonwebtoken");

const SECRET_KEY = "oh-so-secret";
const JWT_OPTIONS = { expiresIn: 60 * 60 }; // 1 hour

let payload = {username: "jane"};
let token = jwt.sign(payload, SECRET_KEY, JWT_OPTIONS);
```

Decoding / Verifying Tokens

```
jwt.decode(token)
```

Return the payload from the token (works *without secret key*. Remember, the tokens are signed, not enciphered!)

```
jwt.verify(token, secret-key)
```

Verify token signature and return payload is valid. If not, raise error.

```
jwt.decode(token);           // {username: "jane"}

jwt.verify(token, SECRET_KEY); // {username: "jane"}

jwt.verify(token, "WRONG");  // error!
```

Using JWTs in Express

Login

`demo/auth-api/routes/auth.js`

```
/** (Fixed) Login: returns JWT on success. */

router.post("/login", async function (req, res, next) {
  if (req.body === undefined) throw new BadRequestError();
  const { username, password } = req.body;
  const result = await db.query(
    "SELECT password FROM users WHERE username = $1",
    [username]);
  const user = result.rows[0];

  if (user) {
    if (await bcrypt.compare(password, user.password) === true) {
      const token = jwt.sign({ username }, SECRET_KEY);
      return res.json({ token });
    }
  }
  throw new UnauthorizedError("Invalid user/password");
});
```



Express Middleware and Authorization

Goals

- Use middleware to simplify route security
- Test routes that require authentication and authorization
- Set up a configuration file for environment variables

Protected Routes

After client receives token, they should send with every future request that needs authentication.

For our demo, we'll look in `req.query` or `req.body` for a token called `_token`

Front End JS

```
// get token from login route
let resp = await axios.post(
  "/login", {username: "jane", password: "secret"});
let token = resp.data;

// use that taken for future requests
await axios.get("/secret", {params: {_token: token}});
await axios.post("/other", {_token: token});
```

Verifying a token

demo/auth-api/routes/auth.js

```
/** Secret-1 route than only users can access */

router.get("/secret-1", async function (req, res, next) {
  // try to get the token out of the query string
  const tokenFromQueryString = req.query?._token;

  // verify this was a token signed with OUR secret key
  // (jwt.verify raises error if not)
  jwt.verify(tokenFromQueryString, SECRET_KEY);

  return res.json({ message: "Made it!" });
});
```

That works, but can we refactor this?

- We don't want to repeat this one every route
- How can we intercept the request and verify the token?
- **Middleware!**

Middleware

Authentication vs Authorization

It's best to *separate these security concerns*:

- **Authentication:** are you who you claim to be?
- **Authorization:** are you allowed to do this thing?

NOTE You don't always do something in both parts

There are cases where you don't need any specific authorization: I might let anyone into my party, but if you show me a passport at my door, I'll make sure it's really yours. If you don't show me a passport (or perhaps if its obviously a forgery), I *might* still let you in — but I won't trust that I know who you are. That's *authentication*: have you proven who you really are?

Obviously, though, I might only let into a party if they were on the guest list. That “are you on the guest list?” is *authorization* [are you allowed do to this]. However, in this case, I'd also need to authenticate you — otherwise, you could say you're my friend Jane, and if I don't verify that, my security is terrible.

There are cases, though, where there might be authorization without authentication. Perhaps no one is allowed to come to my otherwise-open party after midnight. If I just want to turn away everyone, I might not even care to look at your passport. I don't need to know who you are; no one is allowed to do this. That would be authorization without authentication.

Keeping these ideas separate in code is very helpful, as it lets you reason about the realities of security clearly and with well-understood terminology.

Authentication Middleware

`demo/auth-api/middleware/auth.js`

```
/** Auth JWT token, add auth'd user (if any) to res. */

function authenticateJWT(req, res, next) {
  try {
    const tokenFromRequest = req.query?._token || req.body?._token;
    const payload = jwt.verify(tokenFromRequest, SECRET_KEY);
    res.locals.user = payload;
    return next();
  } catch (err) {
    // error in this middleware isn't error -- continue on
    return next();
  }
}
```

(Stores user data on `res.locals` for later requests)

TIP What HTTP status code should you return?

There are bunch of status codes APIs might return, but two important ones here are *401 UNAUTHORIZED* and *403 FORBIDDEN*.

- **401:** you have not presented valid credentials that would help the site establish that you should be able to do these things (perhaps you didn’t supply a valid password, or perhaps you didn’t even attempt to log in.)
- **403:** it’s not your fault or related to your password or other credentials. Instead, this is disallowed for everyone.

Using our earlier examples: if you aren’t on the guest list for my party, I should return a *401 UNAUTHORIZED* to you. If you try to arrive after midnight (which is globally disallowed), I should return a *403 FORBIDDEN* to you.

These aren’t the only status codes, of course — if someone has simply failed to provide all the required data for a route, that’s just a *400 BAD REQUEST*, and if someone looks for something that doesn’t exist, that’s still a *404 NOT FOUND*.

Using Middleware on All Routes

demo/auth-api/app.js

```
const express = require("express");
const routes = require("./routes/auth");
const { NotFoundError } = require("./expressError");
const { authenticateJWT } = require("./middleware/auth");

const app = express();

app.use(express.json());

app.use(authenticateJWT);
```

Middleware runs on all routes defined after this line.

Authorization Middleware

demo/auth-api/middleware/auth.js

```
/** Require user or raise 401 */

function ensureLoggedIn(req, res, next) {
  const user = res.locals.user;
  if (user && user.username) {
    return next();
  }
  throw new UnauthorizedError();
}
```

We can have more specific authorization requirements.

Here’s a version that requires the username be “admin”:

demo/auth-api/middleware/auth.js

```
/** Require admin user or raise 401 */

function ensureAdmin(req, res, next) {
  const user = res.locals.user;
  if (user && user.username === "admin") {
    return next();
  }
  throw new UnauthorizedError();
}
```

NOTE Better version

This is a simple check, useful for just understanding the idea that you might have more specialized authorization middleware.

A more realistic implementation of “make they’re admins” might use an *is_admin* field in the user table, and the JWT would include that field along with the username. Then, this middleware would check whether that value is in the JWT.

Using Middleware on Specific Routes

You typically need flexibility in route authorization; not just “all”.

Some routes, like */register* and */login*, need to be open

demo/auth-api/routes/auth.js

```
/** Secret: must be logged in */

router.get("/secret",
  ensureLoggedIn,
  function (req, res, next) {
    return res.json(
      { message: "Made it!" });
});
```

demo/auth-api/routes/auth.js

```
/** Secret-admin: only admins */

router.get("/secret-admin",
  ensureAdmin,
  async function (req, res, next) {
    return res.json(
      { message: "Made it!" });
});
```

You can add as many middleware functions as you want

Testing Auth

- Before each request, create test users and service tokens for them
- Store these tokens in global variables that can be used across tests

Before hook

demo/auth-api/routes/auth.test.js

```
let testUserToken;
let testAdminToken;

beforeEach(async function () {
  await db.query("DELETE FROM users");
  const hashedPwd = await bcrypt.hash("secret", BCRYPT_WORK_FACTOR);
  await db.query(
    `INSERT INTO users VALUES
      ('test', $1)`, [hashedPwd]);
  await db.query(
    `INSERT INTO users VALUES
      ('admin', $1)`, [hashedPwd]);

  // we'll need tokens for future requests
  const testUser = { username: "test" };
  const testAdmin = { username: "admin" };
  testUserToken = jwt.sign(testUser, SECRET_KEY);
  testAdminToken = jwt.sign(testAdmin, SECRET_KEY);
});
```

Protected Routes

demo/auth-api/routes/auth.test.js

```
describe("GET /secret success", function () {
  test("returns 'Made it'", async function () {
    const response = await request(app)
      .get('/secret')
      .query({ _token: testUserToken });
    expect(response.statusCode).toEqual(200);
    expect(response.body).toEqual({ message: "Made it!" });
  });
});
```

demo/auth-api/routes/auth.test.js

```
describe("GET /secret failure", function () {
  test("returns 401 when logged out", async function () {
    const response = await request(app)
      .get('/secret'); // no token being sent!
    expect(response.statusCode).toEqual(401);
  });

  test("returns 401 with invalid token", async function () {
    const response = await request(app)
      .get('/secret')
      .query({ _token: "garbage" }); // invalid token!
    expect(response.statusCode).toEqual(401);
  });
});
```

Common Configuration

- As application scales, variables like `SECRET_KEY` used all over.
 - Don't redefine in every file — tedious and bug-prone!
- Create a file, `config.js`, at app route, and export vars from there

`demo/auth-api/config.js`

```
/** Common settings for auth-api app. */

const DB_URI = process.env.NODE_ENV === "test"
  ? "postgresql://auth_api_test"
  : "postgresql://auth_api";

const SECRET_KEY = process.env.SECRET_KEY || "secret";

const BCRYPT_WORK_FACTOR = 12;

module.exports = {
  DB_URI,
  SECRET_KEY,
  BCRYPT_WORK_FACTOR,
};
```

Including ENV variables

Your `config.js` file is the **ideal** place to import your environment variables!

You can do this using `process.env.NAME_OF_VARIABLE`

To add and load environment variables, we will be using a module called `dotenv`

Using dotenv

First make sure to install the module, `npm install dotenv`

- Create a `.env` file in the root directory of your project.
- Add environment-specific variables on new lines in the form of NAME=VALUE
- In your `config.js`, load your variables using `require('dotenv').config()`
- Make sure you add `.env` to your `.gitignore` file!



Validation with JSONSchema

Goals

- Understand how and why to validate APIs
- Validate our JSON APIs using jsonschema

Data Validation with Schemas

Server-side Data Validation

An API server lacking adequate validation can result in:

- incomplete data
- poor experience for API users
- security bugs

Why JSON Schema?

There are three main reasons for using a schema validation system:

- So API users fail fast, before bad data gets to your db.
- To prevent unexpected/insecure information from entering db.
- To get a validation system that is easy to setup and maintain.

Rolling Your Own Validation Doesn't Always Scale

Let's assume you have a ice cream `/order` endpoint, and a JSON payload to add a new order looks like this:

```
{  
  "order": {  
    "flavor": "cardamom",  
    "numScoops": 2,  
    "cone": true,  
    "cost": 3.50  
  }  
}
```

Your request

Your /order POST request handler might look like this:

demo/routes/orders.js

```
router.post("/", function (req, res, next) {
  const order = req.body?.order;
  if (!order) throw new BadRequestError("Order data is required");

  // ... snip ...

  return res.json({ordered: true});
});
```

Light validation here — checking if *any* *req.body.order* is given

More validation

- We want to make sure *flavor* is a string
- We want to make sure *numScoops* is 1, 2, or 3
- We want to make sure *cone* is true or false
- We want to make sure *cost* is a floating-point number

If we rolled our own validation this way, every request handler would have lots of conditional logic checking for edge cases.

JSON Schema Basics

We can describe the schema of valid JSON:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "http://our.company.com/order.schema.json",
  "title": "Order",
  "description": "Order for an ice cream.",
  "type": "object",
  "properties": {
    "flavor": {"type": "string"},
    "numScoops": {
      "type": "integer",
      "minimum": 1,
      "maximum": 3
    },
    "cone": {"type": "boolean"},
    "cost": {"type": "number"}
  },
  "additionalProperties": false,
  "required": [
    "flavor",
    "numScoops",
    "cone",
    "cost"
  ]
}
```

The schema is always an object, and must contain these keys:

(part of schema JSON)

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "http://our.company.com/order.schema.json",
}
```

Our schema should get a title and description, and we're describing an object:

(part of schema JSON)

```
{
  "title": "Order",
  "description": "Order for an ice cream.",
  "type": "object",
}
```

We should have a *properties* key of valid property descriptions:

(part of schema JSON)

```
{
  "properties": {
    "flavor": {"type": "string"},
    "numScoops": {
      "type": "integer",
      "minimum": 1,
      "maximum": 3
    },
    "cone": {"type": "boolean"},
    "cost": {"type": "number"}
  }
}
```

We should indicate which are required, and if non-listed props are allowed:

(part of schema JSON)

```
{  
  "additionalProperties": false,  
  "required": [  
    "flavor",  
    "numScoops",  
    "cone",  
    "cost"  
  ]  
}
```

Complete Schema

demo/schemas/orderSchema.json

```
{  
  "$schema": "http://json-schema.org/draft-07/schema#",  
  "$id": "http://our.company.com/order.schema.json",  
  "title": "Order",  
  "description": "Order for an ice cream.",  
  "type": "object",  
  "properties": {  
    "flavor": {"type": "string"},  
    "numScoops": {  
      "type": "integer",  
      "minimum": 1,  
      "maximum": 3  
    },  
    "cone": {"type": "boolean"},  
    "cost": {"type": "number"}  
  },  
  "additionalProperties": false,  
  "required": [  
    "flavor",  
    "numScoops",  
    "cone",  
    "cost"  
  ]  
}
```

Validating our Schema

- Install and import the library `jsonschema`
- Call `.validate(userInput, mySchema, {required: true})`
- The validator checks if user input is valid against schema.
- If invalid, you respond with errors. Otherwise continue.

ATTENTION {required: true}

undefined is not a value that JSON recognizes. For some reason, the developers who created jsonschema decided that the validator should default to returning *true* if you attempt to validate something that is undefined. This is a great example of a bad design choice.

Passing in this third argument, `{required: true}` will change this default behavior, and throw an error if the data you are trying to validate has a value of *undefined*.

`demo/routes/orders.js`

```
const jsonschema = require("jsonschema");
const orderSchema = require("../schemas/orderSchema.json");
```

`demo/routes/orders.js`

```
/** POST /with-validation:
 *   { order: { flavor, numScoops, cone, cost } } => { ordered: true }
 */

router.post("/with-validation", function (req, res, next) {
  const result = jsonschema.validate(
    req.body?.order, orderSchema, {required: true});
  if (!result.valid) {
    // pass validation errors to error handler
    // (the "stack" key is generally the most useful)
    const errs = result.errors.map(err => err.stack);
    throw new BadRequestError(errs);
  }

  // insert into db ...
  return res.json({ordered: true});
});
```

NOTE Error handling

In our current example, the error we send back to the client if the schema validation fails is just a single string with all of the individual failure messages concatenated together. While that might be unfriendly if we were building a web application, APIs usually return terse explanations meant to be interpreted by other developers.

Learning More

You can check that an item is in a list of allowed choices and you can have arrays (or other objects) nested in a schema:

(part of schema JSON)

```
{  
  "properties": {  
    "flavor": {  
      "type": "string",  
      "enum": ["mint", "vanilla", "chocolate", "cardamom"]  
    },  
    "numScoops": {  
      "type": "integer",  
      "minimum": 1,  
      "maximum": 3  
    },  
    "cone": {"type": "boolean"},  
    "cost": {"type": "number"},  
    "toppings": {  
      "type": "array",  
      "items": [  
        { "type": "string" }  
      ]  
    }  
  },  
}
```

There are even more parts of JSON schemas:

- validating that a value is in a fixed list
 - we could use this to check that flavor is only one of our flavors
- validating using *regular expressions* for very specific text patterns
- reusing parts of a schema in other schemas

We won't need these here, but you can always [learn more <https://json-schema.org/learn/getting-started-step-by-step.html>](https://json-schema.org/learn/getting-started-step-by-step.html)

NOTE Making schemas programmatically from data

There are even websites that you can provide sample valid data to and they will produce a schema. You can try this out at [jsonschema.net <https://jsonschema.net>](https://jsonschema.net)

This can be helpful when you a long schema with lots of boilerplate fields. However, while here, we want you to make your schemas by hand — it's important for you to learn what kinds of validations will be required, and it will make you more comfortable if you switch later to autogenerated schemas, since you'll have a good sense of where to tweak them.



Express Testing: TDD & Mocks

Goals

- Revisit essential concepts with testing
- Introduce Test-Driven Development
- Learn about mocking functions
- Learn to mock axios calls directly

Good Testing Practices

- Make sure you write tests!
- Don't forget to test pessimistic cases and edge cases
- Examine your coverage and decide where you need more testing
- Make sure in your readme you specify how to run the tests!

Test the Side-Effect

```
/** POST /cats - create cat from data; return `{cat: cat}` */

describe("POST /cats", function () {
  test("Creates a new cat", async function () {
    const response = await request(app)
      .post('/cats')
      .send({
        name: "Ezra"
      });
    expect(response.statusCode).toBe(201);
    expect(response.body).toEqual({
      cat: { name: "Ezra" }
    });
  });
});
```

- We're not testing if we actually created anything!
- How should we test this? What do we test?

One Option: Validate via Database

```
/** POST /cats - create cat from data; return `{cat: cat}` */

describe("POST /cats", function () {
  test("Creates a new cat", async function () {
    const response = await request(app)
      .post('/cats')
      .send({
        name: "Ezra"
      });
    expect(response.statusCode).toBe(201);
    expect(response.body).toEqual({
      cat: { name: "Ezra" }
    });

    const catsRes = await db.query("SELECT name FROM cats;");
    expect(catsRes.rows).toEqual([{ name:"Ezra" }]);
  });
});
```

Another Option: Validate via API

```
/** POST /cats - create cat from data; return `{cat: cat}` */

describe("POST /cats", function () {
  test("Creates a new cat", async function () {
    // same, but instead of checking database ...

    const getCatsResp = await request(app).get('/cats/ezra')
    expect(response.body).toEqual({ cat: { name:"Ezra" } });
  });
});
```

Which Is Better?

- Opinions vary, but we recommend the latter strategy
 - It’s helpful to “stay in one lane” with testing
 - Your data models should have their own independent unit tests
- More important: be consistent with which approach you use

Test Driven Development

- Write tests **first** - they will fail!
- Only write the code necessary to get the tests to pass
- Focus on completing the task/user story at hand
- As you write more code, keep running tests and make sure they are passing

Red, Green, Refactor

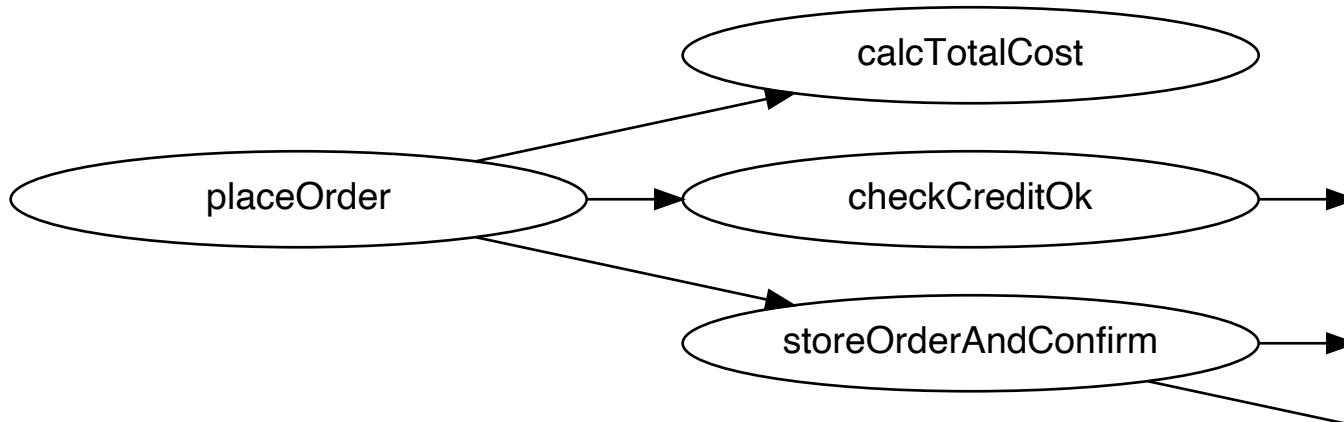
- Your tests fail (**red**)
- You write the code to get the tests to pass (**green**)
- You refactor!

Benefits of TDD

- Writing tests first helps you understand what a function needs to do
- Writing tests first can prevent you from making bugs in the first place
- Writing tests first leads to higher-quality tests
 - If you write the tests after the code, you often write tests that pass based on your memory of the implementation—rather than the requirements

Mocking

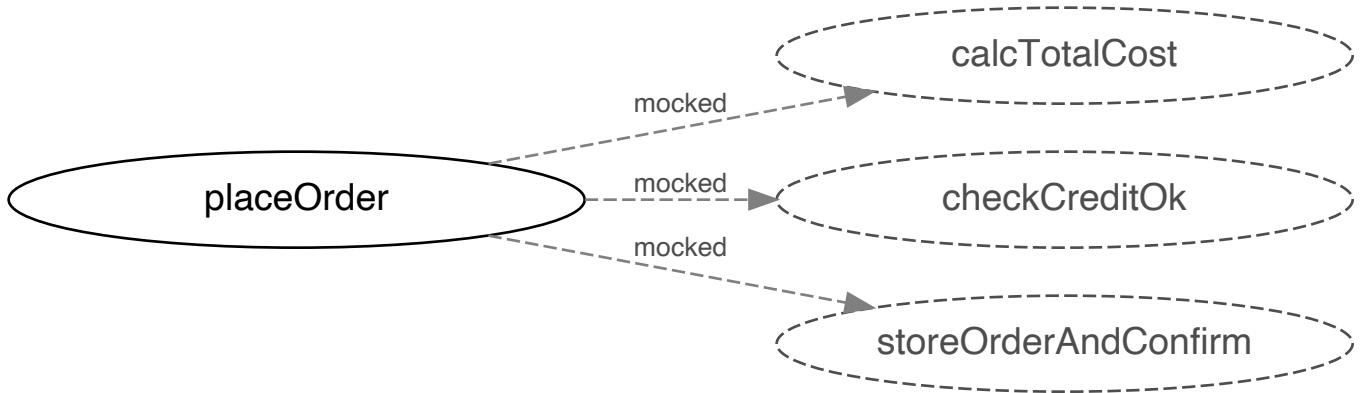
Imagine the function *placeOrder* and what functions it calls:



It would be very hard to write a function for *placeOrder* without worrying about all of those other functions!

However, there may be important logic in *placeOrder* to test, and it might be useful to test that independently of other stuff

We can *mock* those other functions — when *placeOrder* calls them, the *mocked functions* do nothing but return a reasonable fake result.



Why Mock?

- When a function calls other functions and it becomes hard to test
- When a function relies on data that comes from other servers
- When a function relies on something random or hard to predict

What kinds of things can you mock?

- AJAX requests
- Reading/Writing to files
- *Pure* functions like `Math.random`

A pure function relies only on its input, and its results are never determined by state held elsewhere or a random choice.

Mocking a Function

`demo/mocks/casino.js`

```

const { d6 } = require("./dice");

/** You win if you roll 7! */

function lucky7Game() {
  return d6() + d6();
}

module.exports = { lucky7Game };

```

`demo/mocks/casino.test.js`

```

// Since we're mocking something in
// "./dice", must do this before
// loading anything else

const dice = require("./dice");
dice.d6 = jest.fn();

// Now we can do our normal imports:

const { lucky7Game } = require("./casino");

test("always roll 4", function () {
  dice.d6
    .mockReturnValue(4);
  expect(lucky7Game()).toEqual(8);
});

```

```
demo/mocks/dice.js
```

```
/** Roll 6-sided die */

function d6() {
  console.log("d6 ran");
  return Math.floor((Math.random()*6)+1);
}

module.exports = { d6 };
```

You can provide individual values for each invocation:

```
demo/mocks/casino.test.js
```

```
test("lucky 7!", function () {
  dice.d6
    .mockReturnValueOnce(3)
    .mockReturnValueOnce(4);
  expect(lucky7Game()).toEqual(7);
});
```

You can see if the function was called as many times as you expect:

```
demo/mocks/casino.test.js
```

```
test("make sure rolled 2d6", function () {
  dice.d6.mockClear();
  lucky7Game();
  expect(dice.d6).toHaveBeenCalledTimes(2);
});
```

Mocking AJAX Calls

If AJAX call is in a function, you can mock that function, as seen.

It can be helpful to use a library specifically for mocking axios, though — then you can test your API-calling function while still not calling a server.

We use [axios-mock-adapter](https://github.com/ctimmerm/axios-mock-adapter) <<https://github.com/ctimmerm/axios-mock-adapter>> for this.

```
demo/mocks/nums.js
```

```
const BASE_URL = "http://numbersapi.com";
const axios = require("axios");

/** Fact about num from numbersapi */

async function getFact(num) {
  const url = `${BASE_URL}/${num}`;
  const resp = await axios.get(url);
  return resp.data.fact;
}

module.exports = { getFact, BASE_URL };
```

```
demo/mocks/nums.test.js
```

```
const AxiosMockAdapter = require(
  "axios-mock-adapter");
const axios = require("axios");
const axiosMock = new AxiosMockAdapter(axios);

const { getFact, BASE_URL } = require("./nums");

test("fact about 7", async function () {
  axiosMock.onGet(`${BASE_URL}/7`)
    .reply(200, { fact: "7 is lucky" });

  const res = await getFact(7);
  expect(res).toEqual("7 is lucky");
});
```

There are additional features, like:

- simulating slow servers,
- network errors,
- responding with cookies,
- and more!

Wrap Up

- Remember to test!
 - Including pessimistic tests
 - And examine your coverage to see important gaps
- Writing tests first (TDD) can be very helpful
- Mocking functions can be useful
 - Test a function alone, without calling others
 - Bypassing random or impure functions
 - Faking AJAX requests and responses



Deployment with Heroku

Heroku

- a platform as a service, runs on Amazon Web Services
- easier and faster to deploy, but far less customization

Ensuring the correct dependencies

- Heroku needs to know our dependencies!
- Make sure your `package.json` is up to date!

Adding a Procfile

- When we push code to Heroku, we need to tell Heroku what command to run to start the server.
- This command must be placed in a file called `Procfile` in the root directory for your application.
- Make sure filename does not have any extension and begins with capital P.

```
$ echo "web: node server.js" > Procfile
```

Creating your Heroku app

- Login to your heroku account
- Create an application and make sure you have a correct remote.
- Push your code to the new remote and make sure you have a worker.
- Open your heroku app!

```
$ heroku login
$ heroku create NAME_OF_APP
$ git remote -v          # make sure you see heroku
$ git push heroku main   # make sure you've added & committed!
$ heroku open
```

Debugging a Heroku application

It's **never** going to work perfectly the first time. Make sure you look at the server logs to debug!

To see what went wrong, check out the server logs:

```
$ heroku logs --tail
```

Environment Variables

Since we're on a different server, we need different environment variables values:

```
$ heroku config:set SECRET_KEY=my-secret NODE_ENV=production  
$ heroku config:set PGSSLMODE=no-verify  
$ heroku config # see all your environment variables
```

```
// use secret key in production or default to our dev one  
const SECRET_KEY = process.env.SECRET_KEY || 'i-have-a-secret'
```

Adding a Postgres Database

In order to use a production database, we need Heroku to make one:

```
$ heroku addons:create heroku-postgresql:hobby-dev  
$ heroku config # you should see DATABASE_URL
```

Making sure you connect to the correct database

Now that we have a postgres database, we need to make sure that we are connecting to the correct database when in production!

```
/*  
 * Use dev database, testing database,  
 * or via env var, production database  
 * heroku gives us one called DATABASE_URL  
 */  
  
const DB_URI = (process.env.NODE_ENV === "test")  
  ? "jobly_test"  
  : process.env.DATABASE_URL || "jobly";
```

Making sure you connect to the correct port

Heroku will set a port for you in production so make sure you do not hard code the port you are connecting to.

In a `config.js`:

```
const PORT = process.env.PORT || 3000;
```

In your `server.js`

```
app.listen(process.env.PORT, function() {
  console.log("Server starting!");
})
```

Connecting to *psql*

```
$ heroku pg:psql
```

Running a SQL file on Heroku

```
$ heroku pg:psql -f jobly-schema.sql
$ heroku pg:psql -f jobly-seed.sql
```

Heroku hints

- Make sure you've added and committed before pushing to production
- If things break **ALWAYS** go to `heroku logs --t` to see what's breaking

TIP **Further Reading**

There are a number of helpful guides on the Heroku Dev Center that walk you step-by-step through deploying applications in the technology of your choice.



Deployment with Render

ElephantSQL

- Host for PostgreSQL databases
- Have excellent commercial options, but also a free, small instance

Getting a database

1. Create account at [ElephantSQL <https://www.elephantsql.com>](https://www.elephantsql.com) using GitHub
2. Create a “Tiny Turtle” (free) instance
3. Select an Amazon region: *US-West-1 (even if others are closer to you)*
4. Confirm and create
5. Click on your new instance and copy the URL

Seeding your new database

```
$ pg_dump -O jobly | psql (url you copied here)
```

This dumps your existing *jobly* database and loads it in your new database.

Checking your database

```
$ psql (url you copied here)
```

Render

- A service for serving web applications from the cloud
- Similar to Salesforce’s Heroku product, but has a free tier

Setting up your app

1. Create an account at [Render <https://render.com>](https://render.com) using GitHub
2. Add your dashboard, create a new instance of “Web service”
3. Connect to your repository
4. Give it a name, which must be globally unique

5. Choose advanced, and enter environmental variables:
 - *DATABASE_URL*: URL from ElephantSQL
 - *SECRET_KEY*: anything you want
6. Choose “Create Web Service”

Debugging your app

From the dashboard for your app, you can view the logs

Updating your app

When you push to your GitHub repo, it will automatically redeploy your site.

You can turn that off under *Settings* → *Auto-Deploy*, and then can do manual deploys.



Node/Express Wrap Up

Express

Serving Static Files

Can serve static HTML, CSS, images, etc:

```
// serve static files (kept in static directory)

app.use("/js", express.static('static/js'));
app.use("/css", express.static('static/css'));
```

Templating HTML

Pug <<https://pugjs.org/api/getting-started.html>> is a popular template system

Unlike Jinja/Nunjucks, you don't write HTML – you write simpler text:

```
doctype html
html(lang="en")
  head
    title= pageTitle
  body
    h1 Pug - node template engine
    #container.col
      if youAreUsingPug
        p You are amazing
      else
        p Get on it!
```

Common Security Fixes

Helmet <<https://www.npmjs.com/package/helmet>>

Provides tools for dealing with CSRF and other concerns.

Authentication/Login

Passport.js <<http://passportjs.org/>>

Provides common pattern for authentication.

Also provides login via Facebook, Twitter, etc.

Dealing with Cookies

```
const cookieParser = require('cookie-parser');

app.use(cookieParser());

app.get('/', function(req, res, next) {
  console.log('Cookies: ', req.cookies);
})
```

Can also sign cookies, to make tamper-free cookies.

Other Node Web Frameworks

Koa2

Koa2 <<https://github.com/koajs/koa>>

- Written by original author of Node
- A bit more modern & opinionated
- Not as popular as Express — perhaps one day?

Sails

Sails <<https://sailsjs.com/>>

- Larger, more opinionated framework
- Similar to Django or Ruby on Rails
- Includes ORM, *Waterline*

Node

Popular Library: date-fns

date-fns <<https://date-fns.org/>>

Convenient functions for date manipulation & conversion.

Provides “humanized” dates, like “a few minutes ago”, “yesterday”.

```

format(new Date(), "'Today is a' eeee")
//=> "Today is a Thursday"

formatDistance(subDays(new Date(), 3), new Date(), { addSuffix: true })
//=> "3 days ago"

formatRelative(subDays(new Date(), 3), new Date())
//=> "last Friday at 7:26 p.m."

```

Popular Library: Validator.js

[Validator.js <https://github.com/chriso/validator.js>](https://github.com/chriso/validator.js)

Popular library of string validators:

- is all uppercase?
- is email?
- is URL?
- and so on

Popular Library: Lodash

[Lodash <https://lodash.com/>](https://lodash.com/)

Useful set of small utility functions for common actions on arrays, objects, functions.

Grouping, filtering, transforming, and more!

npm Scripts

`package.json` can define scripts to run:

```
{
  "scripts" :
  {
    "test": "jest",
    "debug": "nodemon --inspect server.js",
  }
}
```

Can then run like `npm test` and `npm run debug`

PostgreSQL & Node/Express

Query Builders

There is software like `knex` that are “in-between” raw SQL and ORMs.

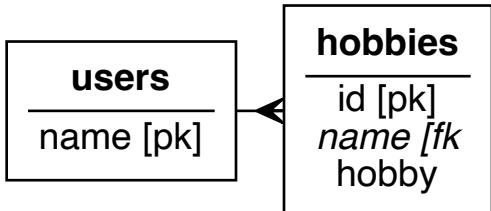
They can take some of the finickiness out of building flexible SQL:

```
// data = {'name': 'New Name', 'description': 'New Description'}
```

```
result = await knex('companies').update(data)
```

You can [read more about it <https://www.rithmschool.com/blog/different-approaches-express>](https://www.rithmschool.com/blog/different-approaches-express) from Joel.

Querying Relationships



```
CREATE TABLE users (name VARCHAR(25) PRIMARY KEY);
```

```
CREATE TABLE hobbies (
    id SERIAL PRIMARY KEY,
    user_name VARCHAR(25) REFERENCES users,
    hobby VARCHAR(25));
```

```
INSERT INTO users VALUES ('elie'), ('matt');
```

```
INSERT INTO hobbies (user_name, hobby) VALUES
    ('elie', 'dancing'),
    ('elie', 'javascript'),
    ('matt', 'math'),
    ('matt', 'cooking');
```

If we want `{name, hobbies: [hobby, ...]}` ...

- You could write a query and make the nested JSON in JS
- Or you could tell PostgreSQL to do it!

```
SELECT name, json_agg(hobby) AS hobbies
FROM users AS u
JOIN hobbies AS h ON (u.name = h.user_name)
GROUP BY name;
```

name	hobbies
elie	["dancing", "javascript"]
matt	["math", "cooking"]

Websockets

- We've used Node/Express to deal with HTTP requests

- It can also serve HTTPS
 - Though, typically, that's handled elsewhere by DevOps
- It can also serve “websocket” protocol
- HTTP is a pretty wordy, heavy protocol
 - So many things in headers!
- HTTP is stateless
 - Ask for answer, get answer, hang up connection
- Websockets are tiny and stateful — they stay connected!
 - They're often used for “tell the browser something has changed”

In Client

```
const ws = new WebSocket(`ws://localhost:3000/chat`);

ws.onopen = function(evt) {
  // called when browser connects to server
};

ws.onmessage = function(evt) {
  // called when browser receives a "message"
  console.log("got", evt.data);
};

ws.onclose = function(evt) {
  // called when server closes connection
};
```

to send a message to server

```
ws.send("this is a message from browser");
```

In Server

Library **express-ws** makes Websockets act like other routes

app.js

```
const wsExpress = require("express-ws")(app);

app.ws("/chat", function (ws, req, next) {
  ws.on("open", function () {
    // called when connection is opened
  });

  ws.on('message', function (data) {
    // called when message is received from browser
  });

  ws.on('close', function () {
    // called when browser closes connection
  });
});
```

to send a message to client

```
ws.send("this is a message from server");
```

Goodbye, Node?

Nope

This is the end of our time with backend JS.

But we'll see that React apps are often made using Node — to setup project, run tests, run dev server, etc.



ReactJS

Goals

- What is React?
- Components
- JSX & Babel
- Properties
- Styling React

Front End Frameworks

- Larger JS libraries
- Provide “blueprint” for apps
- “Opinionated”
 - “This is how you should design a JS app”
- Often: provide for code re-use
- Often: provide templating of HTML (like Jinja)

Popular Front End Frameworks

- Angular
- Ember
- Vue
- React
- Svelte

There are many others, but these are among the most popular.

There are differences between them but they largely share lots of common ideas and after learning one framework, you’ll be in a better position to learn about others.

Is JQuery a “framework”?

Most people would consider it a “library”, but not a “framework”. Often, people distinguish between these based on questions like:

- is it large and complex? (frameworks often are)
- does it push you toward a particular way of thinking about the app design? (frameworks almost always do)
- is it “in charge”, calling out to your code as needed? (often, frameworks do this).

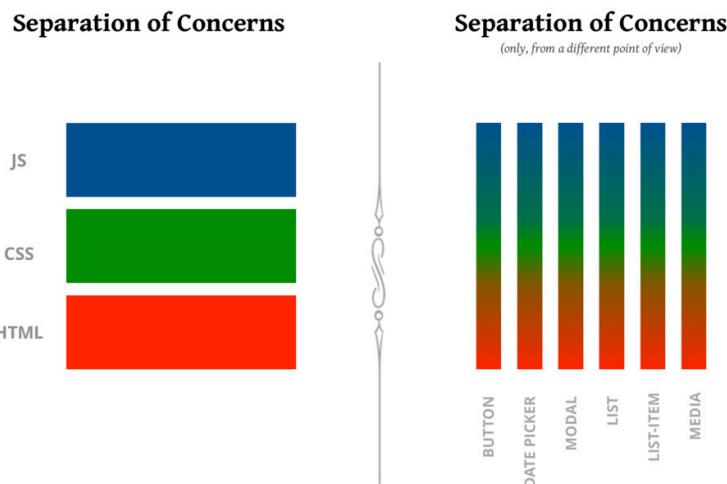
React

Popular, powerful front-end framework.

Developed by and sponsored by Facebook.

- Make it easy to make reusable “view components”
 - These “encapsulate” logic and HTML into a single function
- Often make it easier to build modular applications

Goal



Components

- The building blocks of React
- Pieces of UI & view logic
- Functions that know how to render themselves into HTML

(a bit like this)

```
function Cat() {  
  const name = "Fluffy";  
  
  return `<p>Meow! I'm ${name}!</p>`;  
}
```

Demo: Hello

demo/hello/index.html

```
<!DOCTYPE html>
<html>
<head><title>Hello 1</title></head>
<body>

<div id="root"> <!-- component will go in this div --> </div>

<script src=
  "https://unpkg.com/react/umd/react.development.js"></script>
<script src=
  "https://unpkg.com/react-dom/umd/react-dom.development.js">
</script>

<script src="https://unpkg.com/babel-standalone"></script>

<script src="index.js" type="text/jsx"></script>

</body>
</html>
```

Component are written as functions:

demo/hello/index.js

```
function Hello() {
  return <p>Hi Rithm!</p>;
}
```

Component is added to HTML with `root.render()`:

demo/hello/index.js

```
const root = ReactDOM.createRoot(
  document.getElementById("root")
)

root.render(<Hello />);
```

That *renders* the `Hello` component and puts the resulting HTML in `div#root`.

TIP A warning you may see in the Chrome Console

If you happen to see the following in the Chrome Console:

Warning: You are importing `createRoot` from “react-dom” which is not supported. You should instead import it from “react-dom/client”.

No need to worry - this will not impact anything with your application and will be addressed in an upcoming lecture when we use Create React App.

JSX

demo/hello/index.js

```
function Hello() {
  return <p>Hi Rithm!</p>;
}

const root = ReactDOM.createRoot(
  document.getElementById("root")
)

root.render(<Hello />);
```

What's this thing in our JavaScript?

JSX is like DOM structures embedded in JavaScript:

```
if (score > 100) {  
  return <b>You win!</b>; // <-- JSX  
}
```

You can also “re-embed” JS expressions in JSX:

```
if (score > 100) {  
  return <b>You win, { playerName }</b>;  
}
```

(evaluates JavaScript expression `playerName`)

- Files with JSX aren't legal JavaScript
 - They have to be “transpiled” to JavaScript
- You can do this with [Babel <https://babeljs.io>](https://babeljs.io)

Transpiling JSX in Browser

- Easy for getting started – nothing to install!
- Load *Babel* standalone library:

```
<script src="https://unpkg.com/babel-standalone"></script>
```

- Mark JSX files with `type="text/jsx"`:

```
<script src="index.js" type="text/jsx"></script>
```

NOTE Use Babel on Command Line

While it's convenient to transpile JSX into JavaScript directly in the browser like this, it's not suitable for real-world deployment: it takes a second to do the conversion, leading to a poor experience for users.

Better for deployment is to convert JSX to JavaScript once, via the command line, and then save and use the converted JS directly.

To do this:

1. You need to install [npm <http://npmjs.com>](http://npmjs.com)
2. Then use *npm* to install Babel and settings for React:

```
$ npm install @babel/core @babel/cli @babel/preset-react
```

3. To convert a file:

```
$ node_modules/@babel/cli/bin/babel.js --presets @babel/react  
`file.jsx > file.js
```

Serving Demo

For security reasons, Babel won't work with `file://` scripts

Run files under a simple static server:

```
$ python3 -m http.server
```

Then can visit at `http://localhost:8000/yourfile.html`

JSX Rules

Elements in JSX must either:

- Have an explicit closing tag: ` ... `
- Be explicitly self-closed: `<input name="msg" />`
 - Cannot leave off that `/` or will get syntax error

JSX must have a single top element:

```
const out = <b>Hi</b>;
```

They cannot have multiple top elements:

```
const out = <b>Hi</b> <i>!!!</i>;
```

You can always wrap those in a top-level element:

```
const out = <div> <b>Hi</b> <i>!!!</i> </div>;
```

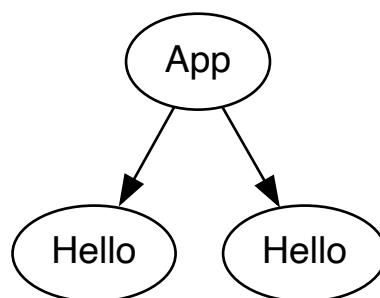
Applications

Real world applications are made up of many components.

It is very conventional to name the “top” component *App*.

This renders the other components:

```
function App() {
  return (
    <div>
      <h1>Greetings!</h1>
      <Hello />
      <Hello />
    </div>
  );
}
```



- This way, readers of code know where to start
- *App* is the only thing rendered in `index.js`

Files

Our demo had *Hello* in same `index.js` as placement code:

`demo/hello/index.js`

```
function Hello() {
  return <p>Hi Rithm!</p>;
}

const root = ReactDOM.createRoot(
  document.getElementById("root")
)

root.render(<Hello />);
```

Better: one file each plus `index.js` at root:

`index.js`

```
ReactDOM.render(<App />,
  document.getElementById("root"));
```

`App.js`

```
function App() {
  return <div> <Hello /><Hello /> </div>
}
```

`Hello.js`

```
function Hello() {
  return <p>Hi Rithm!</p>;
}
```

Order of Script Tags

`index.html`

```
<script src=
  "http://unpkg.com/react/umd/react.development.js"></script>
<script src=
  "http://unpkg.com/react-dom/umd/react-dom.development.js">
</script>

<script src="http://unpkg.com/babel-standalone"></script>

<script src="Hello.js" type="text/jsx"></script>
<script src="App.js" type="text/jsx"></script>
<script src="index.js" type="text/jsx"></script>
```

Make sure any components you need in a file are loaded by a previous `script` tag.

(*We'll learn about a better way to manage imports soon.*)

Props

A useful component is a reusable one.

This often means making it configurable or customizable.

`Hello.js`

```
function Hello() {
  return <p>Hi Rithm!</p>;
}
```

It would be better if we could *configure* our greeting.

Our greeting will be *Hi _____ - _____*.

Let's make two "properties":

to

Who we are greeting

author

Who our greeting is from

Demo: Hello Props

demo/hello-props/App.js

```
function App() {
  return <Hello to="you" author="me" />;
}
```

Set properties on element; get using *props.propName*, (*where props is first argument to component function*)

demo/hello-props/Hello.js

```
function Hello(props) {
  return (
    <div>
      <p>Secret Message: </p>
      <p>
        Hi {props.to} - {props.author}
      </p>
    </div>
  );
}
```

Reusing Component

You can use a component many times:

App.js

```
function App() {
  return (
    <div>
      <Hello to="Kay" author="Kim" />
      <Hello to="you" author="me" />
    </div>
  );
}
```

Properties Requirements

- Properties are for *configuring* your component
- Properties are immutable
- Properties can be strings:

```
<User name="Jane" title="CEO" />
```

- For other types, embed JS expression using the curly braces:

```
<User name="Jane" salary={ 100_000 }
      hobbies={ ["bridge", "reading", "tea"] } />
```

Destructuring Props

You can write this:

App.js

```
function Hello(props) {
  return <b>Hi { props.to } - { props.author }!</b>;
}
```

But it's often nicer to *destructure* the props in the signature:

App.js

```
function Hello({ to, author }) {
  return <b>Hi { to } - { author }!</b>;
}
```

Default Properties

By doing the destructuring, you can supply default arguments:

App.js

```
function Hello({ to, author="Joel" }) {
  return <b>Hi { to } - { author }!</b>;
}
```

Another way to configure default properties

There's another way to configure a component to have default properties, using *.defaultProperties*.

```
function Hello(props) {
  return (
    <p> Hi {props.to} - {props.author} </p>
  );
}

Hello.defaultProps = {
  to: "Bob",
  author: "Jenny",
};
```

This is a bit more verbose and clunky, but it can be useful when you can't destructure your props easily because the property names aren't valid JavaScript identifier names.

Conditionals in JSX

A function component can return either:

- a **single valid DOM object** (`return <div>...</div>`)
- an **array of DOM objects** (but don't do this yet!)
- `null` to show nothing (`undefined` is not ok!)

You can put whatever logic you want in your function for this:

```
function Lottery({ winner }) {  
  if (winner)  
    return <b>You win!</b>;  
  else  
    return <b>You lose!</b>;  
}
```

Ternary

It's very common to use ternary operators:

```
function Lottery({ winner }) {  
  return (  
    <b>You {winner ? "win" : "lose"}!</b>  
  );  
}
```

Demo: Slots!

demo/slots/Machine.js

```
function Machine({ s1, s2, s3 }) {  
  const winner = s1 === s2 && s2 === s3;  
  
  return (  
    <div className="Machine">  
      <b>{s1}</b> <b>{s2}</b> <b>{s3}</b>  
      <p>You {winner ? "win!" : "lose!"}</p>  
    </div>  
  );  
}
```

demo/slots/App.js

```
function App() {  
  return (  
    <div>  
      <h1>Casino!</h1>  
      <Machine s1="🎰" s2="🎰" s3="🎰" />  
    </div>  
  );  
}
```

Looping in JSX

It's common to use `array.map(fn)` to output loops in JSX:

```
<Messages msgs={[
  {id: 1, text: "Greetings!"},
  {id: 2, text: "Goodbye!"},
]} />
```

```
function Messages({ msgs }) {
  return (
    <ul>
      {msgs.map(m => <li>{m.text}</li>) }
    </ul>
  );
}
```

Demo: Friends!

`demo/friends/Friend.js`

```
function Friend({ hobbies, name }) {
  return (
    <div>
      <h1>{name}</h1>
      <ul>
        {hobbies.map(h => <li>{h}</li>) }
      </ul>
    </div>
  );
}
```

`demo/friends/App.js`

```
function App() {
  return (
    <div>
      <Friend name="Jess" hobbies={['Tea', 'Python']} />
      <Friend name="Jake" hobbies={['Chess', 'Darts']} />
    </div>
  );
}
```

NOTE Warnings about key props

If you look in the console, you'll see that React is mad at you for not adding something called a "key" prop when you map over an array and render components. You don't need to worry about this for now; later on, we'll talk more about what's happening here.

Styling React

You can add CSS classes in JSX.

However: since `class` is a reserved keyword in JS, spell it `className` in JSX:

```
function Message() {  
  return <div className="Message">Emergency!</div>;  
}
```

```
function Message({ urgencyColor }) {  
  return <div className={urgencyColor}>Emergency!</div>;  
}
```

You can inline CSS styles, but now `style` takes a JS object:

```
function Box({ favoriteColor, otherColor, message }) {  
  const myStyles = {  
    color: favoriteColor,  
    backgroundColor: otherColor,  
  };  
  
  return <b style={myStyles}>{message}</b>;  
}
```

Debugging React

Install React Developer Tools <<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadoplbjbjfkapdkoienihi>>



React: Modules and CRA

Goals

- Understand what **Create-React-App** is and how to use it
- Use ES2015 modules to share code across files
- Compare default vs. non-default exports
- Using assets (images and CSS) in components

Create React App

React is a front-end library — you don't need server-side stuff.

You *can* get `react.js` and `react-dom.js` from a CDN.

You *can* transpile JSX in the browser at runtime.

But there's a better way!

Create-React-App is a utility script that:

- Creates a skeleton React project
- Sets it up so that JS files are run through Babel automatically
- Lets us use super-modern JavaScript features/idioms
- Makes testing & deployment much easier

npx

To scaffold a project with Create React App, we'll use **npx**.

npx will download Create React App and execute it.

You can think of **npx** as being an alternative to installing packages globally.

Example

```
$ npx create-react-app my-app-name
```

Skeleton

This provides a nice starter skeleton:

	<code>README.md</code>	<code>README</code> , can edit or delete
	<code>package-lock.json</code>	Lock file, don't edit directly
	<code>package.json</code>	Can edit, as usual
	<code>public</code>	Rarely need to edit these
	<code>favicon.ico</code>	
	<code>index.html</code>	Main HTML page of site
	<code>logo192.png</code>	
	<code>logo512.png</code>	
	<code>manifest.json</code>	
	<code>robots.txt</code>	
	<code>src</code>	Where React stuff goes
	<code>App.css</code>	CSS for example component
	<code>App.js</code>	Example component
	<code>App.test.js</code>	Tests for App component
	<code>index.css</code>	Site-wide CSS
	<code>index.js</code>	Start of JS
	<code>logo.svg</code>	React logo
	<code>reportWebVitals.js</code>	(Ignore this for now)
	<code>setupTests.js</code>	Starter test configuration

Starting Your App

\$ npm start

IMPORTANT Remove Boilerplate Code

CRA will create a basic, boilerplate app for you – you should go ahead and remove the things you don't need:

- Change the App component so it returns an empty div.
- Delete the logo.svg file.
- Delete the favicon.ico file (or replace with your own!)
- Remove the `<React.StrictMode>` tags around the `<App/>` component in index.js. React Strict mode will render *all of your components twice*, to test that they will render the same way when given them same input. This can be confusing to beginners who are still learning how React works, and when each component is rendered.

Webpack

CRA is built on top of Webpack, a JS utility that:

- Enables module importing/exporting
 - Packages up all CSS/images/Javascript into a single file for browser
 - Dramatically reduces # of HTTP requests for performance
- Hot reloading: when you change a source file, automatically reloads
 - Is very clever and tries to only reload relevant files
- Enables easy testing & deployment

NOTE The Webpack Rabbit Hole

Webpack is a powerful tool, and configuring it can be quite complicated. Create React App abstracts away that configuration from you, which is great when you're first learning. It's not worth your time right now to learn too much about webpack other than the high-level bullet points we've outlined. If you're curious, you can always go to the [Webpack website <https://webpack.js.org/>](https://webpack.js.org/), but be warned: Webpack is a rabbit hole it's easy to go down and isn't terribly important at this stage in your learning.

Modules

CRA uses ES2015 “modules”

This is a newer, standardized version of Node's `require()`

You use this to export/import classes/data/functions between JS files

Sample Component

`demo/my-app-name/src/App.js`

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>Edit <code>src/App.js</code> and save to reload.</p>
        <a
          className="App-link"
          href="https://reactjs.org"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

Importing “Default” Export

`demo/import-export/mystuff.js`

```
function myFunc() {
  console.log("Hi");
}

export default myFunc;
```

```
demo/import-export/index.js
```

```
// Must start with dot --- "mystuff" would be a npm module!
import myFunc from './mystuff';
```

Importing Non-Default Named Things

```
demo/import-export/mythings.js
```

```
function otherFunc() {
  console.log("Hey");
}

const luckyNumber = 13;

export { otherFunc, luckyNumber };
```

```
demo/import-export/index.js
```

```
import { otherFunc, luckyNumber} from "./mythings";
```

Importing Both

```
demo/import-export/both.js
```

```
function mainFunc() {
  console.log("Ok");
}

const msg = "Awesome!";

export default mainFunc;
export { msg };
```

```
demo/import-export/index.js
```

```
import mainFunc, { msg } from "./both";
```

To Default or Not?

- Conventionally, default exports are used when there's a “most likely” thing to export.
- For example, in a React component file, it's common to have the component be the default export.
- You never **need** a default export, but it can be helpful to indicate most important thing in a file.

Resources

Export <<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/export>>

Import <<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import>>

CRA and Components

Good style:

- Each React component goes in separate file
 - `src/Car.js` for *Car* component
 - `src/House.js` for *House* component
- Define your function component, then export it as the default
- Skeleton assumes top object is *App* in `App.js`
 - Best to keep this

Assets and CRA

To include images and CSS, you can import them in JS files!

`demo/my-app-name/src/App.js`

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>Edit <code>src/App.js</code> and save to reload.</p>
        <a
          className="App-link"
          href="https://reactjs.org"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

CSS

- Make a CSS file for each React component
 - `House.css` for *House* component
- Import it at the top of `House.js`
 - Create-React-App will automatically load that CSS

- Conventional to add `className="House"` onto *House* div

- And use that as prefix for sub-items to style:

```
<div className="House">
  <p className="House-title">{ props.title }</p>
  <p className="House-address">{ props.addr }</p>
</div>
```

Images

- Store images in *src/* folder with the components
- Load them where needed and use imported name where path should go:

Animal.js

```
import puppy from './puppy.jpg';

function Animal() {
  return (
    <div>
      <img src={puppy} alt="Cute puppy!" />
    </div>
  );
}
```

Building for Deployment

`npm run build` makes `build/` folder of static files

You can serve from a web server.



Events and React State

Goals

- Attach event handlers to components in React
- Define React state
- Initialize and change state with *useState*
- Write event handlers to change component state

Events in React

DOM vs. React

- Traditional DOM manipulation: in-line event handlers

```
<button onclick="activateLasers()">  
  Activate Lasers  
</button>
```

- More familiar: add an event listener

```
<button id="laserButton">  
  Activate Lasers  
</button>
```

```
button = document.getElementById('laserButton')  
button.addEventListener("onclick", activateLasers)
```

Example

```
<button onclick="activateLasers()">  
  Activate Lasers  
</button>
```

This is slightly different in React:

```
<button onClick={activateLasers}>  
  Activate Lasers  
</button>
```

- React events are *camelCase*, rather than *lowercase*.
- With JSX you pass a function as event handler, rather than a string.

Event Attributes

Any event you can listen for in JS, you can listen for in React.

- Mouse events: `onClick`, `onMouseOver`, etc
- Form events: `onSubmit`, etc
- Keyboard events: `onKeyDown`, `onKeyUp`, `onKeyPress`
- [Full list <https://reactjs.org/docs/events.html#supported-events>](https://reactjs.org/docs/events.html#supported-events)

An example in a component

`demo/click-me/src/GoodClick.js`

```
import React from "react";

function handleClick(evt) {
  console.log("GoodClick clicked!");
}

function GoodClick() {
  return (
    <button onClick={handleClick}>
      GoodClick
    </button>
  );
}

export default GoodClick;
```

Functions vs. Invocations

Remember: event listeners expect to receive *functions* as values.

Don't invoke your function when you pass it!



`demo/click-me/src/BrokenClick.js`

```
function handleClick(evt) {
  console.log("BrokenClick clicked!");
}

function BrokenClick() {
  return (
    <button onClick={handleClick()}>
      BrokenClick
    </button>
  );
}
```



`demo/click-me/src/GoodClick.js`

```
function handleClick(evt) {
  console.log("GoodClick clicked!");
}

function GoodClick() {
  return (
    <button onClick={handleClick}>
      GoodClick
    </button>
  );
}
```

React State

Core React Concepts

Component

Building block of React; combines logic and presentation.

Prop

Data passed to a component (*or found via defaults*). Immutable; component cannot change its own props.

State

Data specific to an instance of a component; **can change!**

What common things belong in state

- A variable that decides whether to hide or show something
- Data fetched from an API
- Form data
- For a piece of information, ask: *will this ever change?*
 - If so, it should be somewhere in state!

State

In React, state is created using `useState`

```
const [mood, setMood] = useState("happy");
```

- This returns an array of two items:
 - The value of that state (this can change!)
 - A function to change it
- You must call `useState` in the component, not outside it
- Best: keep `useState` calls at the top of your component

WARNING You can't call `useState` in loops or if-blocks

It's important that React knows exactly how many times `useState` is called, and that it is the same number of times every time a component function is called. Therefore, it is not valid to call `useState` inside of a `for` loop, or inside an `if` body.

Naming conventions

```
const [mood, setMood] = useState("happy");
```

- The name of the hook is called `useState`.

- You can call the two returned values whatever you’d like.
- **Very conventional:** paired names, like `foo` and `setFoo`.

Initial State

Pass an argument to `useState` to be the **initial value** of the state:

```
import React, { useState } from "react";

function Person() {
  const [mood, setMood] = useState("happy");

  return (
    <div> Your mood is {mood} </div>
  );
}
```

While our component runs every time it renders the initial value is ignored after the first time it runs (*otherwise it would overwrite mood with “happy” each time*)

Complex Initial State

If the initial state is a time-intensive calculation, you can pass a callback function instead. React will only call this the first time:

```
import React, { useState } from "react";

function MyGame() {
  const [board, setBoard] = useState(makeEmptyBoard);

  function makeEmptyBoard() { return [ /* ... */ ] }

  return(
    <div> ... </div>
  );
}
```

No Initial State

If you don’t pass anything in to `useState`, the **initial value** of the state will be `undefined`.

```

import React, { useState } from "react";

function Person() {
  const [mostRecentMeal, setMostRecentMeal] = useState();

  return(
    <div> You ate {
      (mostRecentMeal !== undefined)
        ? mostRecentMeal
        : "nothing"
      } today!
    </div>
  );
}

```

Changing State

- We'll do this using our *setMood* function!
- Whatever we pass to this function will be set as the new value of *mood* during the next re-render

Like this...

```

import React, { useState } from "react";

function Person() {
  const [mood, setMood] = useState("happy");

  function getExcited() {
    setMood('excited');
  }

  return (
    <div>
      <div> Your mood is {mood} </div>
      <i onClick={getExcited}> Change! </i>
    </div>
  );
}

```

- *setMood* is called
- The component re-renders
- In this re-render, *mood* will be set to *excited*
- Remember: the initial state is only used the first time a component renders

In-lining a Callback

Can pass a function like this...

```
import React, { useState } from "react";

function Person() {
  const [mood, setMood] = useState("happy");

  function getExcited() {
    setMood('excited');
  }

  return (
    <div>
      <div> Your mood is {mood} </div>
      <i onClick={getExcited}> Change! </i>
    </div>
  );
}
```

Or by inlining a callback...

```
import React, { useState } from "react";

function Person() {
  const [mood, setMood] = useState("happy");

  return (
    <div>
      <div> Your mood is {mood} </div>
      <i onClick={() => setMood('excited')}>
        Change!
      </i>
    </div>
  );
}
```

(There's nothing special about using the arrow function syntax in the second case; you could also use a more traditional function expression — all that really matters is that you pass a *function*, not *call* the function you want for the handler)

HINT Deciding between a named function or inlining function

Very simple functions (like `setMood` in the example) might be a good candidate for inlining, but if your function isn't a single expression, or if its purpose would be clearer if it had a name, make it a traditional, named function.

Click Rando

Let's see another example!

`demo/click-me/src/random.js`

```
/** Get random integer [0..max). */
function getRandom(max) {
  return Math.floor(Math.random() * max);
}

export { getRandom };
```

`demo/click-me/src/ClickRando.js`

```
import React, { useState } from "react";
import { getRandom } from "./random";

/** A random number that changes. */
function ClickRando({ maxNum: max }) {
  const [num, setNum] = useState(getRandom(max));

  console.log("ClickRando", num);

  return (
    <i onClick={() => setNum(getRandom(max))}>
      Click Rando: {num}
    </i>
  );
}

export default ClickRando;
```

Multiple Pieces of State

You can call `useState` multiple times if a component needs multiple pieces of state.

demo/click-me/src/Complex.js

```
/** An example of a component with state/props/children. */

function Complex({ maxNum, initialMsg }) {
  const [pushed, setPushed] = useState(false);
  const [num, setNum] = useState(getRandom(maxNum));

  console.log("Complex", pushed, num);

  function handleClick(evt) {
    setPushed(true);
    setNum(getRandom(maxNum));
  }

  return (
    <button className="btn" onClick={handleClick}>
      <b>{pushed ? `Number: ${num}` : initialMsg}</b>
    </button>
  );
}
```

State vs Props

What belongs in state and what belongs in props?

If the data will ever change, it needs to be in state!

Example: Let's build a game!

- If we want to build a game with a board, we might want a component called *GameBoard*.
 - current score: props or state?
 - width of board: props or state?
 - names of the players: props or state?

Documenting Components

Components should be documented for:

- the props they take
- their state
- what components render them/what they render

```
/** Order entering system before it ships.
 *
 * Props:
 * - orderId
 * - price (before tax)
 * - salespersonId
 *
 * State:
 * - isConfirmed: true/false
 *
 * Customer -> Order -> OrderItem
 */
function Order({ orderId, price, salespersonId }) {
  // ...
}
```

Wrap Up

- More on state
- More on events
- Passing functions that change state
- Testing!

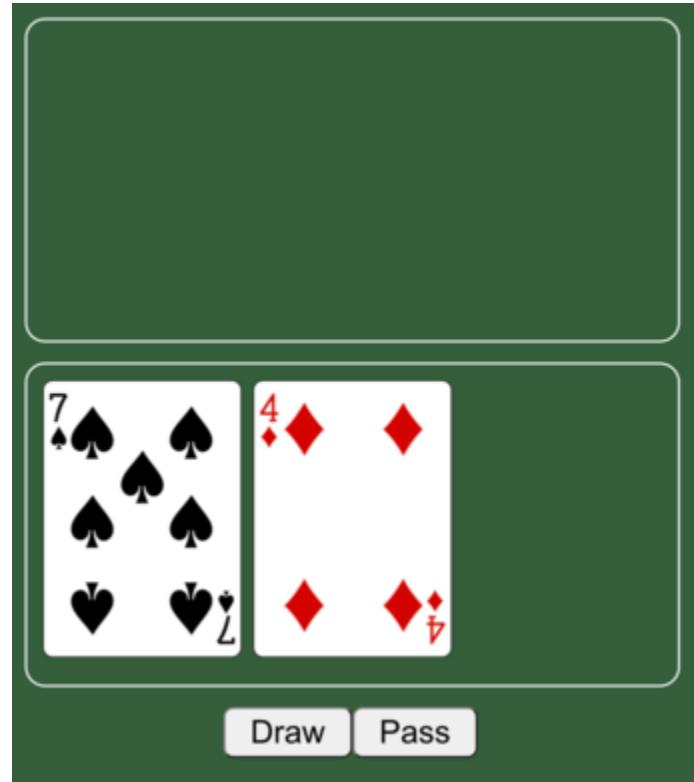
React Testing

Goals

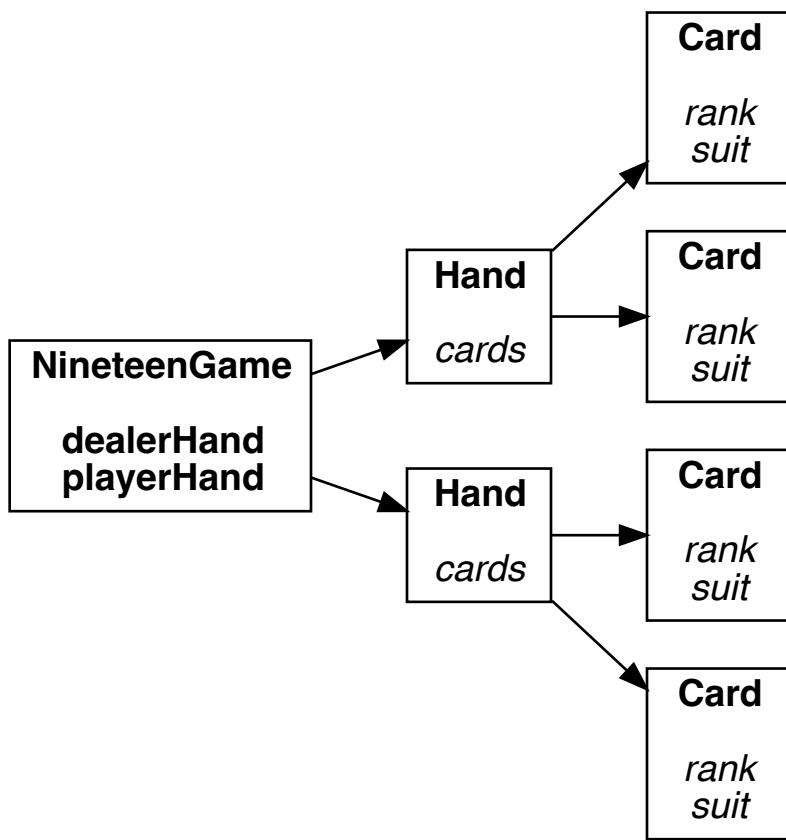
- Meet the hot new gambling game **Nineteen**
- Learn to integrate existing testing techniques into React
- Learn how to test components
- Learn how to test interactions

Nineteen

- Player is dealt two cards
- Player can draw one more card or pass
- Dealer then draws two cards and passes
- Scoring:
 - Hand score is total of points
 - But only counts last digit!
 - Dealer wins all ties



Component Hierarchy



Card and Hand

demo/nineteen/src/Hand.js

```
/** Show a hand of cards.
 *
 * Props:
 * - cards: [ { rank, suit }, ... ]
 */

function Hand({cards}) {
  return (
    <div className="Hand">
      {cards.map(c =>
        <Card
          rank={c.rank}
          suit={c.suit}/>)}
    </div>
  );
}
```

demo/nineteen/src/Card.js

```
/** Show a single card.
 *
 * Props:
 * - rank: 1-9,0,J,Q,K,A
 * - suit: C,D,H,S
 */

function Card({ rank, suit }) {
  const id = rank + suit;
  return (
    <img
      className="Card"
      src={`${IMG_BASE}/${id}.png`}
      alt={id}
    />
  );
}
```

Game

`demo/nineteen/src/NineteenGame.js`

```
function NineteenGame() {
  const [playerHand, setPlayerHand] = useState(getPair);
  const [dealerHand, setDealerHand] = useState([]);
  /* Player draws a card and then dealer plays. */
  function takeCard() {
    setPlayerHand(h => [...h, getRandomCard()]);
    playDealer();
  }
  /* Dealer draws a pair and then hand is scored. */
  function playDealer() {
    setDealerHand(getPair());
  }
  const result = dealerHand.length > 0 ? score(playerHand, dealerHand) : null;

  return (
    <main>
      <Hand cards={dealerHand} />
      <Hand cards={playerHand} />
      {result === null && <p>
        <button id="draw" onClick={takeCard}>Draw</button>
        <button id="pass" onClick={playDealer}>Pass</button>
      </p>}
      {result !== null && <p>{result}</p>}
    </main>
  );
}
```

Testing Nineteen

How can we start testing an application like this?

Start by designing it well!

Unit Testing

Part of designing a React app for testing is moving logical parts outside of components, so they can be unit tested.

For Nineteen, all the logic around drawing cards and scoring is in a non-React set of functions, so they can be easily tested or re-used.

Nineteen Game Logic

demo/nineteen/src/nineteen.js

```
/** Return a random card: { rank, suit } */
function getRandomCard() {
  return { rank: choice(RANKS), suit: choice(SUITS) };
}

/** Return random pair: [ { rank, suit }, { rank, suit } ] */
function getPair() {
  return [getRandomCard(), getRandomCard()];
}

/** Score a hand: score is the last digit of rank sums */
function scoreHand(hand) {
  let sum = 0;

  for (let card of hand) {
    sum += RANK_TO_VALUE[card.rank] || +card.rank;
  }

  return sum % 10;
}

/** Score game: player loses ties to dealer. */
function score(playerHand, dealerHand) {
  return scoreHand(playerHand) > scoreHand(dealerHand)
    ? "You win!"
    : "You lose!";
}
```

Testing a Function

demo/nineteen/src/nineteen.test.js

```
describe("scoreHand", function () {
  test("scores 2 cards", function () {
    const hand = [{ rank: "A", suit: "S" }, { rank: "9", suit: "C" }];
    expect(scoreHand(hand)).toEqual(0);
  });

  test("scores 3 cards", function () {
    const hand = [
      { rank: "A", suit: "S" },
      { rank: "9", suit: "C" },
      { rank: "7", suit: "D" },
    ];
    expect(scoreHand(hand)).toEqual(7);
  });
});
```

Mocking a Function

demo/nineteen/src/nineteen.test.js

```
import * as random from './random';

random.choice = jest.fn();

test("getPair", function () {
  random.choice
    .mockReturnValueOnce("A")
    .mockReturnValueOnce("S")
    .mockReturnValueOnce("9")
    .mockReturnValueOnce("C");
  const pair = getPair();
  expect(pair).toEqual([
    { rank: "A", suit: "S" },
    { rank: "9", suit: "C" },
  ]);
});
```

We can test our function without relying on *random.choice()*

Run these Tests

```
$ npm test src/nineteen.test.js

PASS  src/nineteen.test.js
  ✓ getRandomCard (2ms)
  ✓ getPair (1ms)
  scoreHand
    ✓ scores 2 cards
    ✓ scores 3 cards (1ms)
  score
    ✓ player win
    ✓ player lose: lower count
    ✓ player lose: ties (1ms)

Test Suites: 1 passed, 1 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        1.315s
Ran all test suites matching /src/nineteen.test.js/i.
```

Jest is set up to *watch* — it will re-run tests as you save code.

```
Active Filters: filename /src/nineteen.test.js/
  > Press c to clear filters.

Watch Usage
  > Press a to run all tests.
  > Press f to run only failed tests.
  > Press o to only run tests related to changed files.
  > Press q to quit watch mode.
  > Press p to filter by a filename regex pattern.
  > Press t to filter by a test name regex pattern.
  > Press Enter to trigger a test run.
```

Designing Your App

There's nothing React-specific about this — but remember: small, discrete functions with good separation of concerns will always make testing much easier and more pleasant!

React Testing Library

- React *can* use any testing framework
- *create-react-app* ships with *jest* and *react-testing-library*
- *npm test* is set up to find & run jest tests
 - Files like `*.test*`, `*.spec*`, or in `__tests__` folder
 - We prefer `{MyComponent}.test.js` in same folder as code to test
- *npm test* runs all tests; can optionally specify a file

React Testing Library <<https://github.com/testing-library/react-testing-library>>

“The more your tests resemble the way your software is used, the more confidence they can give you.

—React Testing Library guiding principle

Tests are about how users interact with apps, not the implementations of those apps.

Testing Components

Testing Card Component

Here's the simplest possible test: does it render without crashing?

`demo/nineteen/src/Card.test.js`

```
import React from "react";
import { render } from "@testing-library/react";
import Card from "./Card";

it("renders without crashing", function () {
  // this is a low-value test, but better than nothing
  render(<Card rank="A" suit="C" />);
});
```

This type of *does it even run?* tests are sometimes called *smoke tests*. They don't add a lot of value, but they're a basic place to start. If you're pressed for time, they're better than testing nothing.

Testing Using the DOM

The `render` method returns many things, of which we'll use two:

- `container`: an DOM component of a `div` that holds what you rendered
- `debug(elem)`: a method that returns debugging information about `elem`

Full list <<https://testing-library.com/docs/react-testing-library/api/#render-result>>

```
it("has the correct alt text & src", function () {
  const { container, debug } = render(<Card rank='A' suit='C' />);
  const img = container.querySelector("img");
  debug(img);

  expect(img.getAttribute("alt")).toEqual("AC");
  expect(img.getAttribute("src")).toContain("AC.png");
});
```

We can use standard DOM methods like `querySelector` and `getAttribute`!

Testing a Snapshot

Once you've decided that a component works (either via tests or in the browser), it can be useful to ensure there are no presentational regressions — that the component renders in the same way it did when it worked.

You can take a `snapshot` to catch regressions.

```
it("matches snapshot", function () {
  const { container } = render(<Card rank='A' suit='C' />);
  expect(container).toMatchSnapshot();
});
```

The first time you run this test, it will output:

```
Snapshot Summary
> 1 snapshot written from 1 test suite.
```

Catching Regressions

Now, if you edit that component such that it renders anything different (even by a single character), you will be able to detect that change.

```
> 5 snapshots failed. Inspect your code changes
or press u to update them.
```

If the change is something you want, press `s` to update the snapshot so that the new rendering is what is expected on future test runs.

Jest stores these in `__snapshots__/` (*they're surprisingly readable*)

You should keep this folder in Git — so others can test for regressions.

Testing Interactions

Testing the Full Game

If we know the game renders properly at the start, we can prevent regressions with a snapshot test of that. We'll need to mock the function that makes choices, so we can get a chosen hand:

```
demo/nineteen/src/NineteenGame.test.js
```

```
random.choice = jest.fn();
```

Then our snapshot test is easy to write:

```
demo/nineteen/src/NineteenGame.test.js
```

```
it("matches initial player hand: 2C 3C", function () {
  _feedChoice("2C 3C");
  const { container } = render(<NineteenGame />);
  expect(container).toMatchSnapshot();
});
```

Testing Interactions

To test our full game, we'll need to test events: we need to click the “pass” or “draw” buttons and have the game be scored.

```
demo/nineteen/src/NineteenGame.test.js
```

```
it("matches winning game: 2C 3C 4C > 5C 6C", function () {
  _feedChoice("2C 3C 4C 5C 6C");
  const { container } = render(<NineteenGame />);
  fireEvent.click(container.querySelector("#draw"));
  expect(container).toMatchSnapshot();
});
```

Now we can be certain that exact hand/play is rendered the same.

Don't Just Use Snapshot Tests

It's often easy to write snapshot tests, that it's tempting to only do that.

A risk is that they only test if things *keep working* and you may not have established yet that things ever did work properly for all cases.

Another risk is that if you're actively building your app, you'll often tweak the rendering in tiny way, and break those tests, and will update the snapshots casually. Make sure you have real tests telling you if it works.

A Specific Test for Nineteen

Here's a good test to make sure the right number of cards are rendered, and the result text is rendered properly:

```
demo/nineteen/src/NineteenGame.test.js
```

```
it("deals to player on draw", function () {
  _feedChoice("2C 3C 4C 5C 6C");
  const { container, debug } = render(<NineteenGame />);

  expect(container.querySelectorAll(".Card").length).toEqual(2);
  fireEvent.click(container.querySelector("#draw"));

  // now player has 3 cards (=9) and dealer has 2 cards (=11)
  expect(container.querySelectorAll(".Card").length).toEqual(5);

  debug(container);
  expect(container.querySelector("p")).toContainHTML("You win!");
});
```

APIs

The Basics

```
import React from "react";
import { render, fireEvent } from "@testing-library/react";

test("my test", function () {
  const { container, debug } = render(
    <MyComponent with="these" as="my-props" />
  );
});
```

Additional Matchers

Create React App configures us to be able to use additional matches than the ones that normally come with Jest:

.toHaveClass()

Check whether an element has a certain class

.toBeInTheDocument()

Check whether an element is in the document

.toContainHTML()

Check whether the element contains a certain HTML string

.toBeEmpty()

Check whether the element has any content

Full list <<https://github.com/testing-library/jest-dom#custom-matchers>>

Testing Events

React Testing Library provides a *fireEvent* method that you can use to mimic user interaction with your app.

```
.fireEvent.click(HTMLElement)
  Fire a click event

.fireEvent.submit(HTMLElement)
  Fire a submit event

.fireEvent.input(HTMLElement)
  Fire an input event
```

Documentation <<https://testing-library.com/docs/dom-testing-library/api-events>>

Debugging Testing

Debugging from render

`render` provides a `debug` method that will `console.log` a component's DOM structure.

```
const { debug, container } = render(<MyComponent />)
debug(container); // see the structure of the component

fireEvent.click(container.querySelector("#my-button"));
debug(container); // how has the structure changed?
```

Debugging Tests

If you want to set break points, edit your package JSON to include:

`package.json`

```
"scripts": {
  // ... keep other things and add this
  "test:debug": "react-scripts --inspect-brk test --runInBand"
}
```

Add `debugger` line in test or component you want to test

And now `npm run test:debug` will run tests where you can use Chrome debugger!

Visit `chrome://inspect` to debug in Chrome.

Wrap Up

- Design your apps for testability
 - Break things into small, discrete functions
 - Unit test things that can be kept outside of component
- Test every component, even if it's a “smoke test”
- Write snapshot tests to notice regressions — but don't be too cavalier that those alone prevent bugs
- Don't forget the `debug()` method in tests
- Get comfortable using the debugger with Jest



React State Patterns

Goals

- Learn why mutating React state variables doesn’t work
- Learn how to update state that depends on the previous state
- Understand *key* prop that React asks for when mapping over data

State Review

Change state using the setter function returned by `.useState()`.

```
const [data, setData] = useState(initialState);
```

- During initial render, returned state (*data*) is equal to *initialState*.
- `setData(newVal)` enqueues state change & re-rendering
- Convention: name second arg `setX` where *X* is name of first arg

Normally, variables disappear when their scope ends — but state variables are preserved by React.

Mutable State

It’s fine to have *mutable variables* as your state in React.

Let’s look at a game with a piece of state that is mutable.

Lucky 7 Game

demo/casino/src/Lucky7.js

```
import { useState } from "react";

/** Gets random integer: [1..6]. */

function d6() {
  return Math.floor(Math.random() * 6) + 1;
}

/** Get two rolls => [num, num]. */

function getRolls() {
  console.log("rolling dice!");
  return [d6(), d6()];
}
```

demo/casino/src/Lucky7.js

```
/** Lucky7 Game: roll 2d6 and win with 7! */

function Lucky7() {
  const [dice, setDice] = useState(getRolls());
  const won = dice[0] + dice[1] === 7;

  function roll() { setDice(getRolls()); }

  return (
    <section>
      <div>{ dice[0] }</div>
      <div>{ dice[1] }</div>
      <p>You {won ? "Win" : "Lose"}!</p>
      <button onClick={roll}>
        Roll Again!
      </button>
    </section>
  );
}

export default Lucky7;
```

That works fine — the state is a mutable array, but when it changes, we're just reassigning a set of dice rolls. Everything works!

Easy 7 Game

Our customers complained that Lucky 7 is too hard! We'll make the game easier — you can re-roll just the first die.

The new dice roll will be *[new-die-roll, whatever-was-here]*.

Let's do that by trying to mutate state.

demo/casino/src/Easy7.js

```
/** Easy7 Game: roll 2d6 and win with 7! */

function Easy7() {
  const [dice, setDice] = useState(getRolls());
  const won = dice[0] + dice[1] === 7;

  console.log("Easy7", dice);

  function roll() { dice[0] = d6(); }

  return (
    <section>
      <div>{ dice[0] }</div>
      <div>{ dice[1] }</div>
      <p>You {won ? "Win" : "Lose"}!</p>
      <button onClick={roll}>
        Re-roll first!
      </button>
    </section>
  );
}
```

- It never changes!
- React doesn't care about the mutation
- Even if we try convince it

```
function roll() {
  dice[0] = d6();
  setDice(dice);
}
```

- Why?

Changing State Requires a New Identity

- React only actually changes the state if it has a new *identity*
 - If it's a *new Array*, *new Object*, etc.
- So: let's make a *new* set of dice

```
function roll() {
  const newDice = [d6(), dice[1]];
  setDice(newDice);
}
```

- It works!
- But there's one more improvement to make here...

Depending on Previous State

Sometimes your new state depends on the value of your previous state.

demo/casino/src/Counter.js

```
function Counter() {
  const [num, setNum] = useState(0);

  function up1() { setNum(num + 1); }
  function up2() { setNum(num + 1); setNum(num + 1); }

  return (
    <div>
      <h3>Count: {num}</h3>
      <button onClick={up1}>Up By 1</button>
      <button onClick={up2}>Up By 2</button>
    </div>
  );
}
```

What's the problem here?

```
function up1() {
  setNum(num + 1);
}

function up2() {
  setNum(num + 1);
  setNum(num + 1);
}
```

- *up1* queues state change $0 \rightarrow 1$ (ok!)
- *up2*:
 - queues state change $0 \rightarrow 1$ (ok!)
 - queues state change $0 \rightarrow 1$ (grr!)

If your new state depends on the previous state, you should use the *callback pattern* for *useState*. Setter function returned by *useState* can accept a callback function.

Callback is called when all *already requested* state changes have finished.

It is passed the state as an argument & should return new state.

A better approach:

[demo/casino/src/BetterCounter.js](#)

```
function BetterCounter() {
  const [num, setNum] = useState(0);

  function up1() { setNum(n => n + 1); }
  function up2() {
    setNum(n => n + 1);
    setNum(n => n + 1);
  }

  return (
    <div>
      <h3>Count: {num}</h3>
      <button onClick={up1}>Up By 1</button>
      <button onClick={up2}>Up By 2</button>
    </div>
  );
}
```

TIP An even better fix

Of course, for this example, there's an even better fix:

[demo/casino/src/BestCounter.js](#)

```
function BestCounter() {
  const [num, setNum] = useState(0);

  function up1() { setNum(n => n + 1); }
  function up2() { setNum(n => n + 2); }

  return (
    <div>
      <h3>Count: {num}</h3>
      <button onClick={up1}>Up By 1</button>
      <button onClick={up2}>Up By 2</button>
    </div>
  );
}
```

Given that now there's only one place where the state is being set, it would also work if that was just a simple (non-callback) use of `setNum`, like `setNum(num + 2)`.

However, if this code changed and another set to the state happened, the code might break again. It's always best to follow this rule: **if the new state depends on the old state, use the functional pattern.**

Our Fix for Easy 7

Now, we can fix Easy 7 properly:

```
function roll() {
  // since this relies on current state, use cb
  setDice(curr => [d6(), curr[1]]);
}
```

Functional Patterns for State

Zero 7 Game

Now, we have a new game: you roll 3 dice and can “zero” out the first die.

So $[2, 3, 5]$ would turn into $[0, 3, 5]$.

Since our new state depends on the old state, we'll use the callback form!

demo/casino/src/Zero7.js

```
function Zero7() {
  const [dice, setDice] = useState(getRolls);
  const won = sum(dice) === 7;

  function zeroFirst() {
    setDice(curr => {
      curr[0] = 0;
      return [...curr];
    });
  }

  return (
    <section>
      <div>{ dice[0] }</div>
      <div>{ dice[1] }</div>
      <div>{ dice[2] }</div>
      <p>You {won ? "Win" : "Lose"}!</p>
      <button onClick={zeroFirst}>
        Zero First
      </button>
    </section>
  );
}
```

We're using the callback form for `setDice` since we depend on previous state.

It works!

But we can find a nicer way to do this.

Functional Patterns

Remember `.map(cb)` and `.filter(cb)`?

`arr.map((item, idx) => ...)` `arr.filter((item, idx) => ...)`

Better Zero First

```
function zeroFirst() {
  setDice(curr => curr.map(
    (val, i) => i === 0 ? 0 : val));
}
```

Another useful map

```
function zeroSixes() {
  setDice(curr => curr.map(
    val => val === 6 ? 0 : val));
}
```

A useful filter

```
function removeEvens() {
  setDice(curr => curr.filter(
    val => val % 2 !== 0));
}
```

These are very useful for changing mutable things without mutation.

Nested State

These ideas can also be used for changing parts of nested objects.

Imagine: we want to re-roll the second die — but now, a die is an object. It has a color (which doesn't need to change now) and a value (which does).

```
[  
  { color: "pink", val: 2 },  
  { color: "blue", val: 6 }, // <-- re-roll yet keep color  
  { color: "tan", val: 5 },  
]
```

```
[  
  { color: "pink", val: 2 },  
  { color: "blue", val: 6 }, // <-- re-roll yet keep color  
  { color: "tan", val: 5 },  
]
```

```
function reroll(pos) {  
  setDice(curr =>  
    curr.map( (die, i) =>  
      i === pos  
        ? { ...die, val: d6() }  
        : die  
    ));  
}
```

- We get a new array of dice (a new identity — so setDice works!)
- Dice that don't change don't need a new identity
- But the die that is being re-rolled gets a object via spread

It can take a while for this to feel natural.

Practice this skill — this is a very useful way to transform objects!

Lists and Keys

Add 7 Game

We have another new game! Keep adding dice to your rolls, aiming for 7.

We know how to manage our changing state without mutation.

But now, the number of dice are variable.

`demo/casino/src/Add7.js`

```
function Add7() {
  const [dice, setDice] = useState([]);
  const won = sum(dice) === 7;

  function addDie() {
    setDice(curr => [...curr, d6()]);
  }

  return (
    <section>
      {dice.map(val => <div>{ val }</div>) }
      <p>You {won ? "Win" : "Lose"}!</p>
      <button onClick={ addDie }>
        Add Die
      </button>
    </section>
  );
}
```

We use the callback form to `setDice`.

It works!

But there's scary warning in the console:

Warning: Each child in a list
should have a unique "key" prop.

Check the render method of 'Add7'.

What's that about? Is it a real problem?

Keys

- Keys help React identify which items are changed/added/removed.
- Keys should be given to elements in an array to provide a *stable identity*.

If there's a natural key, use it:

```
books.map( b =>
  <Book key={ b.isbn } title={ b.title } /> );

```

If not, you can use the loop index:

```
dice.map( (val, i) =>
  <Die key={ i } num={ val } />
);

```

There can be problems using the index,
though!

Delete 7 Game

We have another game! Now, you start with 6 dice.

When you click a button, you get rid of the first remaining die.

`demo/casino/src/Delete7.js`

```
function Delete7() {
  const [dice, setDice] = useState(getRolls);
  const won = sum(dice) === 7;

  function removeFirst() {
    setDice(curr =>
      curr.filter((d, i) => i !== 0));
  }

  return (
    <section>
      {dice.map((val, i) =>
        <div key={i}>{ val }</div> )}
      <p>You {won ? "Win" : "Lose"}!</p>
      <button onClick={removeFirst}>
        disabled={dice.length === 0}>
          Remove First
        </button>
    </section>
  );
}
```

- Handling state-depending-on state!
- Using a nice idiom with `filter`
- Have a `key` for loop
- But this still might break!
- The keys aren't *stable*
- This can cause strange errors
 - Performance problems
 - Inaccurate rendering

Making stable identities

At times like this, you'll need to make a stable identity for our die:

We can make a simple counter:

```
class Counter {
  i = 0;
  next = () => this.i++;
}

const counter = new Counter();

// now, counter.next() will return
// a new ID each time.
```

Can use a library:

```
import { v4 as uuid } from "uuid";

// now uuid() will return a new ID
// each time its called
```

You have to determine the unique identity of the die when you create the die, not when you render the die!

Nope

```
dice.map(val=>
  <Die key={ uuid() } num={ val } />
);
```

Yep

```
function addDie() {
  setDice(curr => [...curr,
    {id: uuid(), val: d6()}
  ]);
}

// later, in return ...

dice.map(d =>
  <Die key={ d.id } num={ d.val } />
);
```

Otherwise, they change every time.

NOTE [More details on Keys](#)

Here is some further reading on keys, if you're interested:

[Key props and rendering in React](https://reactjs.org/docs/reconciliation.html) <<https://reactjs.org/docs/reconciliation.html>>

[Index as a key is an anti-pattern](https://medium.com/@robinpokorny/index-as-a-key-is-an-anti-pattern-e0349aece318) <<https://medium.com/@robinpokorny/index-as-a-key-is-an-anti-pattern-e0349aece318>>

Wrap Up

- Don't mutate state!
- Instead, create a new identity (new Array/Object).
- Use the callback form of state setters if new state depends on old.
- Learn and practice the functional idioms for transforming state.
- Things rendered in an array require a *key* prop
 - Use a natural key if you have one
 - Can use the loop index *if* they're stable
 - If no natural key, and loop indexes aren't stable, make a key

Demos

Lucky 7

Used mutable state, but was replaced on click. Works!

Easy 7

Tried to mutate state and never had a “new identity”.

Zero 7

Replacing clunky patterns with *map/filter*.

Add 7

Renders a dynamic list, which requires a *key* prop. The indexes are stable, so can use them for the key.

Delete 7

Now items are being deleted, so the keys are not stable. Will have subtle bugs until a stable key is used.



React Component Design

Goals

- Understand strategies for “component decomposition”
- Practice designing a React app!
- Learn to how pass functions between components

Component hierarchies

Lucky 7 Game

`demo/dice-game/src/Lucky7.js`

```
import { useState } from "react";
import { getRolls, sum } from "./utils";
import "./Lucky7.css";

/** Luck Game: roll 2d6 and win with 7! */

function Lucky7() {
  const [dice, setDice] = useState(getRolls(2));
  const won = sum(dice) === 7;

  function roll() { setDice(getRolls(2)); }

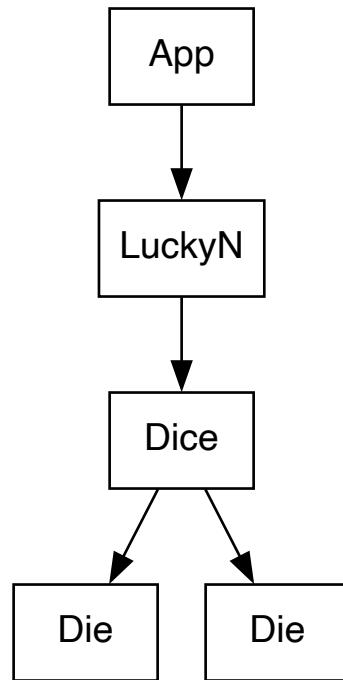
  return (
    <main className="Lucky7">
      <h1>Lucky7 {won && "You won!"}</h1>
      <section className="Lucky7-dice">
        <div className="Lucky7-die">{dice[0]}</div>
        <div className="Lucky7-die">{dice[1]}</div>
      </section>
      <button onClick={roll}>
        Roll Again!
      </button>
    </main>
  );
}
```

- This component is doing a lot!
- It's also inflexible
 - It always rolls 2 dice
 - It always wins on 7
- There aren't reusable parts
 - Showing a set of dice
 - Show an individual die
- Let's do better!

Requirements

A set components for:

- Play a dice game with *numDice* number of dice
- Shows a win message when total equals *goal*
- A “roll again” button that re-rolls all the dice

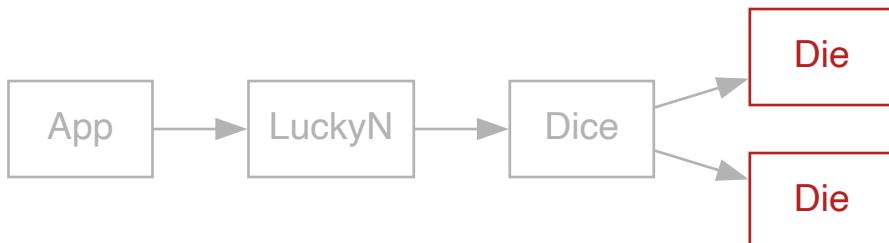


△ Our component hierarchy

What props do we need? What state do we need? And where?

Die Component

- Props
 - *val*: number
- State
 - none!
- Events
 - none!



```
/** Single die. */

function Die({ val }) {
  return (
    <div className="Die">
      {val}
    </div>
  );
}
```

Dice Component

- Props

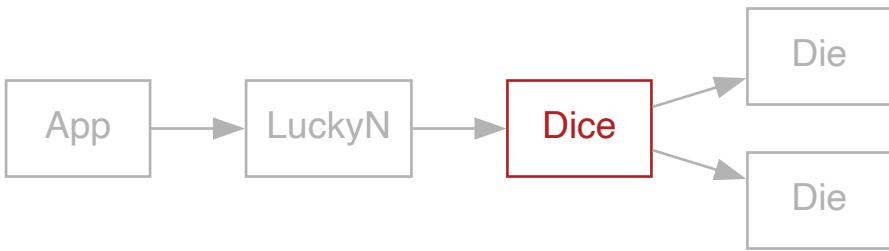
- `dice: [n, ...]`

- State

- none!

- Events

- none!



```
/** Shows a set of dice. */

function Dice({ dice }) {
  return (
    <section className="Dice">
      {dice.map((v, i) =>
        <Die key={i} val={v} />)}
    </section>
  );
}
```

LuckyN Component

- Props

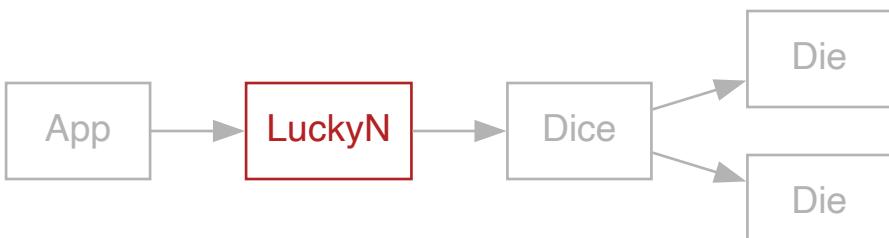
- `numDice`
- `goal`

- State

- `dice: [n, ...]`

- Events

- `roll()`



```
/** Luck Game: roll numDice & d6 and win with goal! */

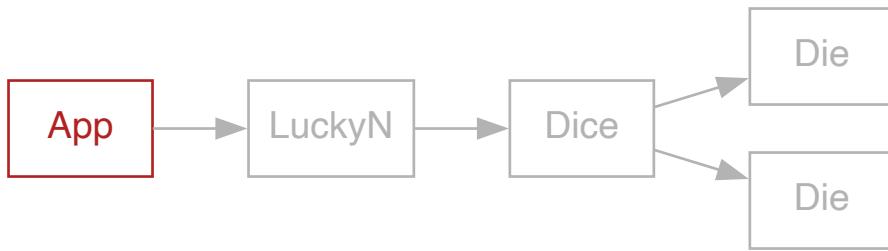
function LuckyN({ numDice, goal }) {
  const [dice, setDice] = useState(getRolls(numDice));
  const won = sum(dice) === goal;

  function roll() { setDice(getRolls(numDice)); }

  return (
    <main className="LuckyN">
      <h1>Lucky{goal} {won && "You won!"}</h1>
      <Dice dice={dice} />
      <button onClick={roll}>
        Roll Again!
      </button>
    </main>
  );
}
```

App Component

- Props
 - none!
- State
 - none!
- Events
 - none!



demo/dice-game/src/App.js

```
/** Site with dice games. */

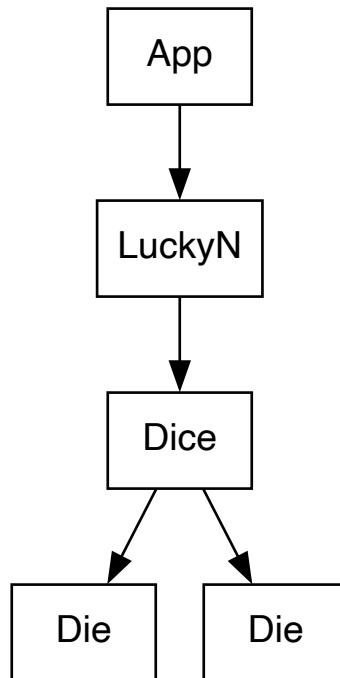
function App() {
  return (
    <div className="App">
      <Lucky7 />
    </div>
  );
}
```

State design principle

| Lift state as high as needed – but no higher.

Where should the dice-rolls state live?

- ✗ *App*: don't need it, so shouldn't lift it
- ✓ *LuckyN*: this is the game itself!
- ✗ *Dice*: should just be about showing a hand
- ✗ *Die*: need to know roll total; not just for one



Decoupling logic from presentation

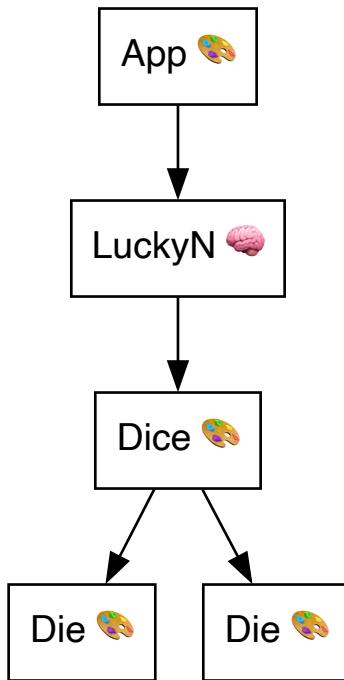
Generally, aim to have components be one of two types:

Presentational 🎨

Doesn't have state; is primarily about appearance/UI.

Logical 🧠

Has state or related logic; isn't about a specific UI.



Getting logic out of views

```
demo/dice-game/src/utils.js
```

```
/** Gets random integer: [1..6]. */

function d6() {
  return Math.floor(Math.random() * 6) + 1;
}

/** Get n rolls => [num, ...]. */

function getRolls(n) {
  return Array.from({length: n}, () => d6());
}

/** Get sum of nums. */

function sum(nums) {
  return nums.reduce((prev, cur) => prev + cur, 0);
}

export { d6, getRolls, sum };
```

Not everything needs to be done in your component function:

- common logic can be shared
- even if only used once, this can still be very useful:
 - can be unit tested
 - can be mocked

LuckyN

```
/** Luck Game: roll numDice & d6 and win with goal! */

function LuckyN({ numDice, goal }) {
  const [dice, setDice] = useState(getRolls(numDice));
  const won = sum(dice) === goal;

  function roll() { setDice(getRolls(numDice)); }

  return (
    <main className="LuckyN">
      <h1>Lucky{goal} {won && "You won!"}</h1>
      <Dice dice={dice} />
      <button onClick={roll}>
        Roll Again!
      </button>
    </main>
  );
}
```

- *LuckyN* takes *numDice* and *goal*
- What if we wanted a more different game?
- One where you had to roll less than a number?
- Right now, logic for winning is in this function

Passing a function as a prop

In JavaScript, functions are *first-class objects* – you can pass them around!

We can pass a *winCheck(dice)* function to our *LuckyN* game!

```
/** Possible game strategy: sum of dice < 4. */
function lessThan4(dice) {
  return sum(dice) < 4;
}

<LuckyN numDice={3} winCheck={lessThan4} />
```

```
/** Luck Game: roll numDice & d6 and win with goal! */

function LuckyN({ numDice, winCheck }) {
  const [dice, setDice] = useState(getRolls(numDice));
  const won = winCheck(dice);

  // ...
}
```

NOTE That could also be inlined

Given how simple the logic for that particular win condition is, this could even be inlined in the rendering of the function:

```
<LuckyN
  numDice={3}
  winCheck={dice => sum(dice) < 4} />
```

However, that would make it much harder to unit test the condition or mock it, so this is probably not a good idea.

This is a very powerful idea – now, our *LuckyN* component is about the *view* our our app, and it can take configurable logic.

Decomposing components

```
/** Luck Game: roll numDice & d6 and win with goal! */

function LuckyN({ numDice, goal }) {
  const [dice, setDice] = useState(getRolls(numDice));
  const won = sum(dice) === goal;

  function roll() { setDice(getRolls(numDice)); }

  return (
    <main className="LuckyN">
      <h1>Lucky{goal} {won && "You won!"}</h1>
      <Dice dice={dice} />
      <button onClick={roll}>
        Roll Again!
      </button>
    </main>
  );
}
```

- Let's make this button fancier
 - But *LuckyN* is about logic/state 🧠, not UI 🎨
- Also: we might like to re-use this button elsewhere
- Let's move it to its own component!

```
/** Luck Game: roll numDice & d6 and win with goal! */

function LuckyNButton({ numDice, title, winCheck }) {
  const [dice, setDice] = useState(getRolls(numDice));
  const won = winCheck(dice);

  function roll() { setDice(getRolls(numDice)); }

  return (
    <main className="LuckyN">
      <h1>{title} {won && "You won!"}</h1>
      <Dice dice={dice} />
      <ReRollButton roll={roll} />
    </main>
  );
}
```

```
/** Button to re-roll dice. */

function ReRollButton({ roll }) {
  return (
    <button
      className="ReRollButton"
      onClick={roll}>
      Roll Again!
    </button>
  );
}
```



Passing functions to child

- It's common to pass a function to a child, for them to execute as needed.
- Our *dice* state is in the game, so button can't change it directly.
- Our *roll()* function changes the state, so game with re-render.

Making button more reusable

Even better — we can make our button even more generic!

```
/** Luck Game: roll numDice & d6 and win with goal! */

function LuckyNButton2({ numDice, title, winCheck }) {
  const [dice, setDice] = useState(getRolls(numDice));
  const won = winCheck(dice);

  function roll() { setDice(getRolls(numDice)); }

  return (
    <main className="LuckyN">
      <h1>{title} {won && "You won!"}</h1>
      <Dice dice={dice} />
      <Button label="Roll again!" click={roll} />
    </main>
  );
}
```

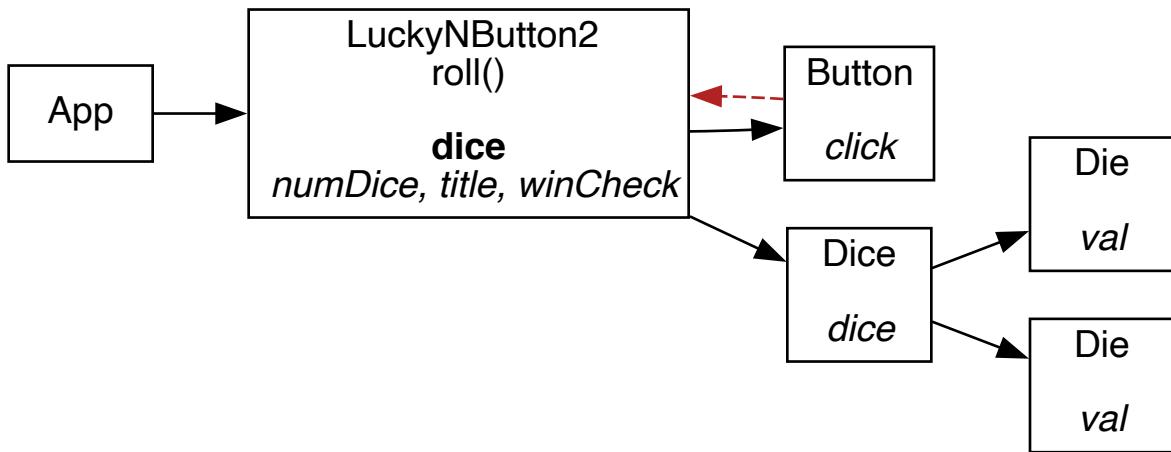
```
/** Button to re-roll dice. */

function Button({ click, label }) {
  return (
    <button
      className="Button"
      onClick={click}>
      {label}
    </button>
  );
}
```



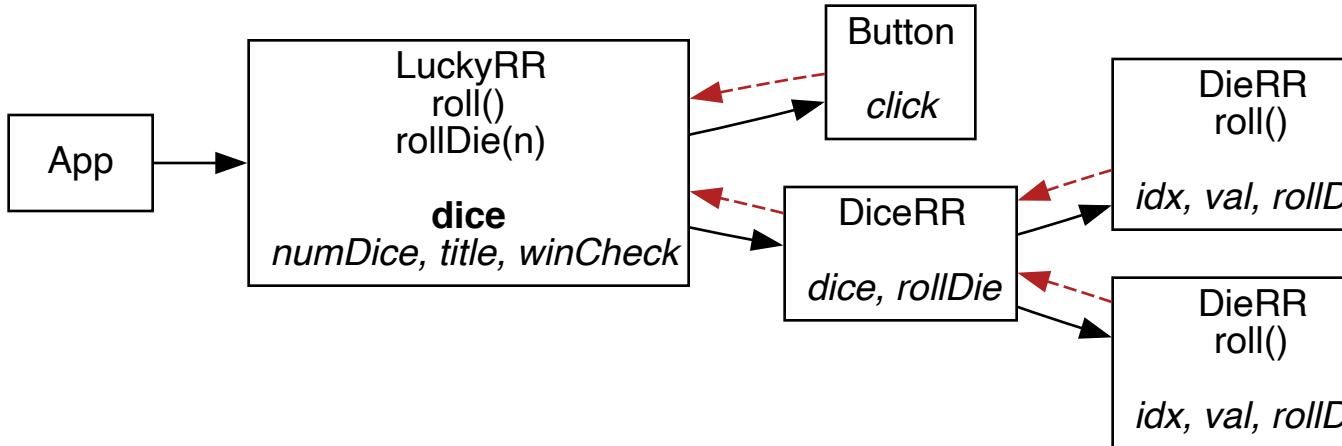
Function passing

New feature: re-rolling individual dice



How could we add a feature to re-roll an individual die when it's clicked on?

We can't add it to *LuckyNButton2* or *Die* without changes!



- Add `rollDie(n)` to game component
- Pass it to *Die* through *Dice*
- Each *Die* needs to know its “position” from *Dice* loop
- On click, *Die* will call `roll()` which calls `rollDie(myIndex)`

```
/** Luck Game: roll numDice & d6 and win with goal! */

function LuckyRR({ numDice, title, winCheck }) {
  const [dice, setDice] = useState(getRolls(numDice));
  const won = winCheck(dice);

  function roll() { setDice(getRolls(numDice)); }
  function rollDie(n) {
    setDice(dd => dd.map((v, idx) =>
      idx === n ? d6() : v));
  }

  return (
    <main className="LuckyN">
      <h1>{title} {won && "You won!"}</h1>
      <DiceRR dice={dice} rollDie={rollDie}/>
      <Button label="Roll again!" click={roll} />
    </main>
  );
}
```

```
/** Shows a set of dice. */

function DiceRR({ dice, rollDie }) {
  return (
    <section className="Dice">
      {dice.map((v, idx) =>
        <DieRR idx={idx}
          key={idx}
          rollDie={rollDie}
          val={v} />)
    </section>
  );
}
```

```
/** Single die. */

function DieRR({ idx, rollDie, val }) {
  function roll() { rollDie(idx); }

  return (
    <div className="Die"
      onClick={roll}>{val}</div>
  );
}
```

How data flows

This shows a common pattern in React:

- A parent component defines a function
- The function is passed as a prop to a child component
- The child component invokes the prop function
- The parent function is called, usually setting new state
- The parent component is re-rendered along with its children

Alternative strategy: pass “custom function” to Die

Right now, `DiceRR` passes `rollDie()` through `Dice` to each `Die`. `Dice` has to tell each die their index:

LuckyRR

```
<DiceRR dice={dice} rollDie={rollDie}/>
```

DiceRR

```
{dice.map((v, idx) =>
  <DieRR idx={idx} key={idx} rollDie={rollDie} val={v} />
)}
```

DieRR

```
function DieRR({ idx, rollDie, val }) {
  function roll() { rollDie(idx); }

  return <div className="Die" onClick={roll}>{val}</div>;
}
```

We could instead pass a *custom function* to each child:

LuckyRR

```
<DiceRR dice={dice} rollDie={rollDie}/> // unchanged
```

DiceRR

```
{dice.map((v, idx) =>
  // making bespoke `rollMe` for each Die
  <DieRR idx={idx} rollMe={idx => rollDie(idx)} val={v} />
)}
```

DieRR

```
function DieRR({ rollMe, val }) {
  return <div className="Die" onClick={rollMe}>{val}</div>;
}
```

Which is better?

- Using arrow functions in the parent simplifies the child
- However, there are performance optimization considerations
- Prefer passing `rollDie` to `Die` over a custom arrow function for now

Learning design

Designing a React application is a new skill will take practice to master.

Components

Generally, components should be small & do one thing.

This often makes them more reusable.

Example: component that displays a todo w/task could be used in lots of “lists”.

Presentational Components

Often, small components are simple & don't have state:

```
function Todo(props) {  
  return <div className="Todo">{ props.task }</div>;  
}
```

This can be used like:

```
function ListOfTodos() { // ... lots missing  
  return (  
    <div className="ListOfTodos">  
      todos.map(t => <Todo task={t} />  
    </div>  
  );  
}
```

Components like *Todo* are called “presentational” or “dumb” [*in a good way!*]

Don't Store Derived Info

If one thing can be calculated from another, don't store both:

```
function TaskList() {  
  const [todos, setTodos] = useState(["wash car", "wash cat"]);  
  const [numTodos, setNumTodos] = useState(2);  
  
  return (  
    <div>  
      You have {numTodos} tasks ...  
    </div>  
  );  
}
```

Yuck! Just calculate the number of todos as needed!

Coming Up

- Learning to “route” in React applications
- Learning how to handle “side effects” in a component
- Learning other ways to pass data around

Using arrow functions for components

Components are just functions. We can write them with arrow syntax if we choose.

If the component only renders, you can make use of an arrow function's implicit return.

what we've been doing:

```
return (
  <div className="Die">
    {val}
  </div>
);
// end
```

a different approach:

```
const Die = ({ value }) => (
  <div className="Die">
    {value}
  </div>
);
```

Should you use arrow functions for components? We recommend ✗ no — but, as always, follow the style of existing code.

You should understand the syntax so you recognize it in some documentation, code examples, etc.

Whichever you choose: be consistent.



React Forms

[Download Demo Code <..../react-forms-demo.zip>](#)

Goals

- Build forms with React
- Understand what controlled components are

Forms

- HTML form elements work differently than other DOM elements in React
 - Form elements naturally keep some internal state.
 - For example, this form in plain HTML accepts a single name:

```
<form>
  Full Name:
  <input name="fullname" />
  <button>Add!</button>
</form>
```

Thinking About State

```
<form>
  Full Name:
  <input name="fullname" />
  <button>Add!</button>
</form>
```

- It's convenient to have a JS function that
 - handles the submission of the form *and*
 - has access to the data the user entered.
- The technique to get this is *controlled components*.

Controlled Components

- In HTML, form elements such as `<input>`, `<textarea>`, and `<select>` typically maintain their own state and update it based on user input.
- In React, mutable state is kept in the `state` of components, and updated with the setter function returned from `useState()`.
- How do we use React to control form input state?

One Source of Truth

- We make React state be the “single source of truth”
- React controls:
 - What is *shown* (the value of the component)
 - What happens when user types (*this gets kept in state*)
- Input elements controlled in this way are called *controlled components*.

How the Controlled Form Works

- Since *handleChange* runs on every keystroke and updates React state, the displayed value will update as the user types.
- Since form element’s value is set from state, its display will always match, making React state the source of truth.
- With a controlled component, every form element change has a related handler function, making it easy to modify or validate user input.

handleChange Methods

Here are the methods that update state based on input.

```
function SimpleNameForm() {
  const [firstName, setFirstName] = useState("");
  const [lastName, setLastName] = useState("");

  function handleChangeFirstName(evt) {
    const firstName = evt.target.value;
    setFirstName(firstName);
  }

  function handleChangeLastName(evt) {
    const lastName = evt.target.value;
    setLastName(lastName);
  }

  // more code follows ...
}
```

Accessibility: Labeling Inputs

Good accessibility practice puts *<label>* elements in forms:

```
<form>
  <label for="fullname-input">Full Name:</label>
  <input id="fullname-input" name="fullname" />
  <button>Add!</button>
</form>
```

- Our `<label>` elements have an important attribute, `for`
- If a `for` attribute matches the id of an input, clicking on that label changes focus to the input.
- But there's a problem here!

htmlFor instead

- `for` is a reserved word in JavaScript, just like `class`
- Just as we replaced `class` with `className`, we replace `for` with `htmlFor`
- You will get console warnings if you forget this

Handling Multiple Inputs

How we've done this so far

```
function SimpleNameForm() {
  // multiple pieces of state
  const [firstName, setFirstName] = useState("");
  const [lastName, setLastName] = useState("");

  // multiple handleChange functions
  function handleChangeFirstName(evt) {
    const firstName = evt.target.value;
    setFirstName(firstName);
  }

  function handleChangeLastName(evt) {
    const lastName = evt.target.value;
    setLastName(lastName);
  }
}
```

To handle multiple controlled inputs:

- Instead of making a separate `onChange` handler for every single input, we can make one generic function for multiple inputs!
- Add HTML `name` attribute to each JSX input element
- Then the handler function can determine the key in state to update based on `event.target.name`.

Instead, what if we have one piece of state - an object!

```
{firstName: "", lastName: ""}
```

And we figure out what fields changed like this

```
const fieldName = evt.target.name;
```

```
const [formData, setFormData] = useState({
  firstName: "",
  lastName: ""
});

function handleChange(evt) {
  const fieldName = evt.target.name;
  const value = evt.target.value;

  setFormData(currData => {
    currData[fieldName] = value;
    return {...currData};
  });
}
```

Using this method, keys in state must match input `name` attributes.

Computed Property Names

Fancy technique

one way

```
let fieldThatChanged = "firstName";
let newState = {...oldState};

newState[fieldThatChanged] = newValue;
```

another way

```
let fieldThatChanged = "firstName";

let newState = {
  ...oldState,
  [fieldThatChanged]: newValue
};
```

ES2015 Review

```
let fieldThatChanged = "firstName";

let newState = {
  ...oldState,
  [fieldThatChanged]: newValue
};
```

- ES2015 introduced a few object enhancements...
- Including: object literals can have property names dynamically determined by expressions.
- The feature is called *computed property names*.

Using computed property names

demo/name-form-demo/src/NameForm.js

```
const [formData, setFormData] = useState({
  firstName: "",
  lastName: ""
});

function handleChange(evt) {
  const { name, value } = evt.target;
  setFormData(fData => ({
    ...fData,
    [name]: value,
  }));
}
```

Remember, keys in state must match input `name` attributes.

Shopping List Example

- Parent: `ShoppingList` (manages a list of shopping items)

- Child: *NewListItemForm* (form to add a new item to the list)

```
function ShoppingList() {

  /** Add new item object to cart. */
  function addItem(item) {
    let newItem = { ...item, id: uuid() };
    setItems(items => [...items, newItem]);
  }
}
```

```
function NewListItemForm({ addItem }) {

  /** Send {name, quantity} to parent
   * & clear form. */
  function handleSubmit(evt) {
    evt.preventDefault();
    addItem(formData);
    setFormData(initialState);
  }
}
```

Passing Data Up to a Parent Component

- In React we generally have downward data flow: “Smart” parent components with simpler child components.
- But it’s common for form components to manage their own state...
- But smarter parent component usually has a *doSomethingOnSubmit* method to update its state after the form submission...
- So: parent passes its *doSomethingOnSubmit* method as a prop to child.
- Child component calls this method, updating the parent’s state.
- Child is “dumber” — all it knows is to invoke that function with its data.

Keys and UUIDs

- Using iteration index as *key* prop is a poor idea when list changes
- No natural unique key? Use a library to create a *uuid*
- Universally unique identifier (UUID) is a way to uniquely identify info
- Install it using `npm install uuid`

Using the UUID Module

demo/shopping-list/src/ShoppingList.js

```
import { v4 as uuid } from 'uuid';
/** Add new item object to cart. */
function addItem(item) {
  let newItem = { ...item, id: uuid() };
  setItems(items => [...items, newItem]);
}
```

demo/shopping-list/src/ShoppingList.js

```
function renderItems() {
  return (
    <ul>
      {items.map(item => (
        <li key={item.id}>
          {item.name}: {item.qty}
        </li>
      ))}
    </ul>
  );
}
```

Other Concepts

Uncontrolled components

- If React is *not* in control of the form state, this is called an *uncontrolled component*.
- Some inputs and external libraries require it.

Validation

- Useful for UI
- **Not an alternative to server side validation**
- [Formik <https://jaredpalmer.com/formik/docs/overview>](https://jaredpalmer.com/formik/docs/overview)

Testing Forms

- To test typing in form inputs, we can use `fireEvent.change`
- When using this, we'll need to mock `evt.target.value`: this is how we'll tell React testing library what to place in the input
- For controlled components, state will then automatically update

`demo/shopping-list/src/ShoppingList.test.js`

```
it("can add a new item", function () {
  const { getLabelText, queryByText } = render(<ShoppingList />);

  // no items yet
  expect(queryByText("ice cream: 100")).not.toBeInTheDocument();

  const nameInput = getLabelText("Name:");
  const qtyInput = getLabelText("Qty:");
  const submitBtn = queryByText("Add a new item!");

  // fill out the form
  fireEvent.change(nameInput, { target: { value: "ice cream" } });
  fireEvent.change(qtyInput, { target: { value: 100 } });
  fireEvent.click(submitBtn);

  // item exists!
  expect(queryByText("ice cream: 100")).toBeInTheDocument();
});
```

Looking Ahead

- *useEffect*
- AJAX with React



Client-Side Routing with React Router

Goals

- Describe what client-side routing is and why it's useful
- Compare client-side routing to server-side routing
- Describe how router URL parameters work
- Learn how to test React Router components

Server-Side Routing

- “Server-side routing” is the traditional pattern
 - Clicking an `<a>` link causes browser to request page & replace entire DOM
 - Server decides what HTML to return based on URL requested

Client-Side Routing

Faking Client Side Routing

`demo/nonrouted/src/App.js`

```
function App() {
  const [page, setPage] = useState("home");

  function goToPage(newPage) {
    setPage(newPage);
  }

  function showPage() {
    if (page === "home") return <Home />;
    if (page === "eat") return <Eat />;
    if (page === "drink") return <Drink />;
  }

  return (
    <main>
      <nav>
        <a onClick={() => goToPage("drink")}>Drink</a>
        <a onClick={() => goToPage("eat")}>Eat</a>
        <a onClick={() => goToPage("home")}>Home</a>
      </nav>
      {showPage()}
    </main>
  );
}
```

That's okay

- It does let us show different “pages”
 - All in the front-end, without loading new pages from server
- But we don’t get
 - A different URL as we move around “pages”
 - The ability to use the back/forward browser buttons ← → 🎉
 - Any way to bookmark a “page” on the site 📚 📄 🍷

Real Client-Side Routing

React can give us real Client-Side Routing

Client-Side Routing: What?

- Client-side routing handles mapping between URL bar and page user sees via *browser* rather than via *server*.
- Sites that exclusively use this are *single-page applications*.
- JavaScript manipulates the URL bar with the History Web API

React Router

To get started with React Router, install *react-router-dom*.

```
$ npx create-react-app routed
$ cd routed
$ npm install react-router-dom
```

Including the Router

demo/routed/src/App.js

```
function App() {
  return (
    <div className="App">
      <BrowserRouter>
        <NavBar />
        <Routes>
          <Route path="/drink" element={<Drink/>} />
          <Route path="/eat" element={<Eat/>} />
          <Route path="/" element={<Home/>} />
        </Routes>
      </BrowserRouter>
    </div>
  );
}
```

Wrap the things that need routing with *<BrowserRouter>*

There are other routers besides *BrowserRouter* — don't worry about them.

Other types of routers

If you read through the React Router docs, you'll see examples of other types of routers. Here's a brief description of them:

- *HashRouter*: this router is designed for support with older browsers that may not have access to the full history API. In such cases, you can still get single-page type functionality by inserting an anchor (#) into the URL. However, this does not provide full backwards-compatibility: for this reason, the React Router documentation recommends *BrowserRouter* over *HashRouter* if possible.
- *MemoryRouter* This router mocks the history API by keeping a log of the browser history in memory. This can be helpful when writing tests, since tests are typically run outside of a browser environment.
- *NativeRouter* This router is designed for React Native applications.
- *StaticRouter* This is a router that never changes location. When would you ever use this? According to the docs, "This can be useful in server-side rendering scenarios when the user isn't actually clicking around, so the location never actually changes. Hence, the name: static. It's also useful in simple tests when you just need to plug in a location and make assertions on the render output."

Routes and Links

A Sample Application

App.js

```
import React from "react";
import { BrowserRouter, Route, Routes } from "react-router-dom";

import Home from "./Home";
import Eat from "./Eat";
import Drink from "./Drink";
import NavBar from "./NavBar";

function App() {
  return (
    <div className="App">
      <BrowserRouter>
        <NavBar />
        <Routes>
          <Route path="/drink" element={<Drink/>} />
          <Route path="/eat" element={<Eat/>} />
          <Route path="/" element={<Home/>} />
        </Routes>
      </BrowserRouter>
    </div>
  );
}

export default App;
```

Route Component

```
<Route path="/eat" element={<Eat />} />
```

- *Route* component acts as translation service between routes & components.
 - Tell it path to look for in URL, and what to render when it finds match.
- Props you can set on a *Route*:
 - *path*: path that must match
 - *element* : the component to render when the route matches

Routes Component

- *Routes* finds a matching *Route* and renders only that.
- Wrap all of your *<Route>* components with a *<Routes>* component

TIP Avoid nested routes for now

If you look in the React Router docs, you'll see that there are ways to nest your *<Route>* components inside of other *<Route>* components. This is a complicated design and is beyond the scope of what you will need to do in this course (and may not even be something you see at the company you work at).

Navigation Links

Link Component

- The *<Link>* component acts as a replacement for *<a>* tags.
- Instead of an *href* attribute, *<Link>* uses a *to* prop.
- Clicking on *<Link>* does *not* issue a GET request.
 - JS intercepts click and does client-side routing

```
<p>Go to <Link to="/drink">drinks</Link> page</p>
```

TIP *NavLink* Component

- <NavLink> is just like link, with one additional feature
 - If at page that link would go to, the <a> gets a CSS class of *active*
 - This lets you have CSS like this:

```
.MyNavBarClass a {  
  color: white;  
}  
  
.MyNavBarClass a.active {  
  color: black;  
}
```

- Very helpful for navigation menus

A Sample Navigation Bar

Nav.js

```
import React from "react";  
import { Link } from "react-router-dom";  
import "./NavBar.css";  
  
function NavBar() {  
  return (  
    <nav className="NavBar">  
      <Link to="/">  
        Home  
      </Link>  
      <Link to="/eat">  
        Eat  
      </Link>  
      <Link to="/drink">  
        Drink  
      </Link>  
    </nav>  
  );  
  
  export default NavBar;
```

URL Parameters

An Anti-Pattern

```
function App() {
  return (
    <App>
      <Routes>
        <Route path="/food/tacos" element={<Food name="tacos" />} />
        <Route path="/food/salad" element={<Food name="salad" />} />
        <Route path="/food/sushi" element={<Food name="sushi" />} />
      </Routes>
    </App>
  );
}
```



What's the Problem?

- Lots of duplication
- What if we want to add more foods?
- Solution: Let's use URL parameters!

A Better Way

```
import React from "react";
import Nav from "./Nav";
import { Route, BrowserRouter, Routes } from "react-router-dom";
import Food from "./Food";

function App() {
  return (
    <div className="App">
      <BrowserRouter>
        <Nav />
        <Routes>
          <Route path="/food/:name" element={<Food/>} />
        </Routes>
      </BrowserRouter>
    </div>
  );
}

export default App;
```

Like with Express, we indicate a URL parameter with a colon :

Accessing URL Parameters

The `useParams` hook stores info on the URL parameters.

```
<Route path="/food/:name" element={<Food/>} />
```

- given the above code and a url of `/food/sushi`
 - the `useParams` hook returns an object
 - the key of the object in this example will be `name`
 - the value will be `sushi`

For the route `/food/:name`

```
import { useParams } from "react-router-dom"

function Food() {
  const params = useParams(); // { name: ... }
  return (
    <div>
      <h1>You must like { params.name }</h1>
    </div>
  );
}
```

TIP Multiple URL Parameters

In that example, we only used one URL parameter.

It's possible to have multiple parameters in a single route.

For example, to have food and beverage pairings in route:

```
<Route path="/food/:foodName/drink/:drinkName">
  <Food />
</Route>
```

Here, `useParams()` will return an object with two keys: `foodName` and `drinkName`.

Redirects

Client-side Redirects

- With React Router we can mimic the behavior of server-side redirects.
- Useful after certain user actions (e.g. submitting a form)
- Also useful to not allow users to go somewhere (e.g. unauthorized pages)

How to Redirect

- In React Router, there are two ways to redirect:
 - With the `useNavigate` hook

- Useful for “you’re now done here, go here instead, if you go back - no worries!”
- With the *Navigate* component
 - Useful for “you shouldn’t have gotten here, go here instead, do not go back”

The *useNavigate* hook: An Example

Useful for “you’re now done here, go here instead, if you go back - no worries!”

```
function Contact() {
  const [email, setEmail] = useState("");
  const navigate = useNavigate();

  function handleChange(evt) {
    setEmail(evt.target.value);
  }

  function handleSubmit(evt) {
    evt.preventDefault();
    navigate("/");
  }

  return (
    <div>
      <h1>This is the contact page.</h1>
      <p>Enter email to get in touch</p>
      <form onSubmit={handleSubmit}>
        <input
          type="email"
          name="email"
          value={email}
          onChange={handleChange} />
        <button>Submit</button>
      </form>
    </div>
  );
}
```

The *Navigate* Component: An Example

Useful for “you shouldn’t have gotten here, go here instead, do not go back”

`demo/redirects/src/AdminPage.js`

```
function AdminPage() {  
  
  const { username } = useParams();  
  
  if (username !== "admin") {  
    return <Navigate to="/" />;  
  }  
  
  return (  
    <div>  
      <h1>Welcome back admin!</h1>  
      <Link to="/">Go home</Link>  
    </div>  
  );  
}
```

NOTE History API

- All client-side routing libraries use the browser’s history API
- History API allows us to manipulate browser history via JS
- Common API methods on `window.history`:
 - `.back`: go back one page in history
 - `.forward`: go forward one page in history
 - `.go`: go to an arbitrary page in history
 - `.pushState`: add new entry in history & update URL *without* reloading page.
 - `.replaceState` - without adding new entry in history, update URL *without* reloading page.

The function signatures for `back`, `forward`, and `go` are all relatively straightforward. The first two functions don’t take any parameters; the third accepts one parameter, indicating how far back or forward in the session history you’d like to travel.

On the other hand, If you read about the history API on MDN, you’ll see that the signature for `pushState` and `replaceState` are a little... weird. Both accept three parameters: a `state` object, a `title` and a `url`. Let’s describe these in reverse order:

- `url` is simply the new URL you want to put into the URL bar.
- `title` parameter is, strangely, ignored by every browser. You can supply a string here if you want, but it does not matter.
- `state` parameter is an object that you can potentially access later if the user navigates back to this point in the session history by clicking back or forward. Practically speaking, this isn’t something you need to worry about for now.

While these function signatures can definitely be confusing, most times you can ignore the first and second parameters: the most important is the third.

For more on the history API in general, check out [MDN on History API](https://developer.mozilla.org/en-US/docs/Web/API/History_API)
`<https://developer.mozilla.org/en-US/docs/Web/API/History_API>`

Including a 404

```
function Routes404() {
  return (
    <Routes>
      <Route path="/about" element={<About />} />
      <Route path="/contact" element={<Contact />} />
      <Route path="/blog/:slug" element={<Post/>} />
      <Route path="/blog" element={<BlogHome/>} />
      <Route path="/" element={<Home/>} />
      <Route path="*" element={<NotFound />} />
    </Routes>
  );
}
```

NOTE 404s with a *Navigate* component

Instead of a component we could also use a *<Navigate>* component!

```
function RoutesList() {
  return (
    <Routes>
      <Route path="/about" element={<About />} />
      <Route path="/contact" element={<Contact />} />
      <Route path="/users/:username" element={<AdminPage />} />
      <Route path="/blog/:slug" element={<Post />} />
      <Route path="/blog" element={<BlogHome />} />
      <Route path="/" element={<Home />} />
      <Route path="*" element={<Navigate to="/" />} />
    </Routes>
  );
}
```

Testing React Router

Testing Components with React Router

Components rendered by router are harder to test than regular components:

- Components may depend on router hooks we'll have to mock:
 - eg, *useParams* hook
- Components require knowledge of the parent router during the test

Testing React Router

Consider our *Nav* component:

demo/redirects/src/Nav.js

```
function Nav() {
  return (
    <ul>
      <li><Link to="/">Home</Link></li>
      <li><Link to="/about">About Us</Link></li>
      <li><Link to="/contact">Contact</Link></li>
      <li><Link to="/blog">Blog</Link></li>
      <li><Link to="/blargh">Broken Link</Link></li>
    </ul>
  );
}
// end
```

Router Context Errors

Problems arise when we `render()` the component:

```
it('renders without crashing', function() {
  render(<Nav />);
});
```

You will get something like:

To avoid `*Error: Uncaught [Error: useHref() may be used only in the context of a <Router> component.]*`, use a mock router, `MemoryRouter`, which is designed for tests:

MemoryRouter

To avoid `*Error: Uncaught [Error: useHref() may be used only in the context of a <Router> component.]*`, use a mock router, `MemoryRouter`, which is designed for tests:

```
import { MemoryRouter } from 'react-router-dom';
```

demo/redirects/src/Nav.test.js

```
// full render
it('mounts without crashing', function () {
  const { getByText } = render(
    <MemoryRouter>
      <Nav />
    </MemoryRouter>
  );

  const blogLink = getByText(/Blog/i);
  expect(blogLink).toBeInTheDocument();
});
```



Testing individual routes

To test your specific routes, use the `initialEntries` prop on `MemoryRouter` to “navigate” to a route

`demo/redirects/src/RoutesList.test.js`

```
it('renders the about page', function () {
  const { debug, getByText } = render(
    <MemoryRouter initialEntries={['/about']}>
      <RoutesList />
    </MemoryRouter>
  );

  const h1Text = getByText(/This is the about page./i);
  expect(h1Text).toBeInTheDocument();
});

it('renders the blog page', function () {
  const { debug, container } = render(
    <MemoryRouter initialEntries={['/blog']}>
      <RoutesList />
    </MemoryRouter>
  );

  const links = container.querySelectorAll("li a")
  expect(links).toHaveLength(3)
});
```

If you want to explicitly mock `useParams`, here is what that might look like. This can be helpful when unit testing an individual component instead of the entire `<Routes>` component with `initialEntries`.

```
// Make sure that this is OUTSIDE of your test or describe block
jest.mock('react-router-dom', () => ({
  ...jest.requireActual('react-router-dom'),
  useParams: () => ({ name: "burrito" }),
}))
```

Patterns in React Router

- There aren’t strong standards about how to best use React Router
- Here are patterns *we* follow for React Router
- Some of these have been mentioned already—here they are in one place

Consider a single `RoutesList.js` file

- Don’t spread `<Route>` components across multiple files
- You can put all `<Route>`s directly in your `App`
- When you have many, it may be overwhelming
 - Having a place for all routing info may be preferable
 - May be easier to debug
 - Make a file, `RoutesList.js`, with a `RoutesList` component

Avoid nested routes

- Components rendered by a *Route* can themselves render *Route* components
- An example of nested routing, and is generally confusing and error prone
- Unless you need it, don't nest your routes!
 - You'll often end up with spaghetti code and a headache

React Router Tips

React Router Tips (Overview)

- Favor *Route* child components over other options
- Keep routes up high in the component hierarchy with a *<Routes>* component
- Avoid nested routes
- Use the *<Navigate>* component and *useNavigate* for redirecting



React Effects

Goals

- Understand more about how rendering and state interact
- Learn how to have “side effects” in a React app
- See lots of examples of `useEffect`

Rendering and State

When your component calls a state setter function, when does the state change?

Let’s find out with `<RenderDemo/>`!

A recap:

- A state setter function is called
- Your code finishes running
- The component re-renders
- In this re-render, state will be set to the new value

Can we do something *after* a render or re-render?

We can! These are called *side-effects* or *effects*

Rendering and Effects

This is what *effects* are for – doing after (& unrelated to) a render

This is useful for different kinds of things:

- Changing parts of the DOM not covered by React
- Async operations, like AJAX requests **when a component is mounted**
- Doing things when a component is about to be *unmounted*

Effects are used for “side-effects”: doing things unrelated to render

useEffect

To use an effect, register it with `useEffect(fn)`:

register `mySideEffect`:

common to inline these:

```

import React, { useEffect } from 'react';

function MyComponent() {
  function myEffect() {
    // ... do something
  }

  useEffect(myEffect);
  // rest of component
}

```

```

import React, { useEffect } from 'react';

function MyComponent() {
  useEffect(function myEffect() {
    // ... do something
  });
  // rest of component
}

```

```

function MyComponent() {
  useEffect(function myEffect() {
    // ... do something
  });
  // rest of component
}

```

- my effect always runs *after* first render
- by default, my effect runs *after* all re-renders
- my effect can do whatever you need
 - if it sets state, another render will be triggered
 - and my effect will run *after* that re-render

2nd Argument to useEffect

- `useEffect(myCallbackFn);`
 - Runs *myCallbackFn* effect after *every* render
- `useEffect(myCallbackFn, [productId, userId])`
 - Runs *myCallbackFn* effect only if *productId* or *userId* vars changed
- `useEffect(myCallbackFn, [])`
 - Runs *myCallbackFn* effect only the first time (on *mount*)

NOTE **Changing Non-React DOM After State Change**

A less common use of effects is when you need to change a Non-React part of the DOM after a state change.

Below is an example where an effect is used to update the title of a browser tab in response to a state change.

`demo/effects/src/EffectExample.js`

```
import React, { useState, useEffect } from "react";

/** Demo of a simple effect: changes doc title */

function EffectExample() {
  const [num, setNum] = useState(0);

  useEffect(function setTitleOnRerender() {
    document.title = `WOW${"!".repeat(num)}`;
  }, [num]);

  function increment(evt) {
    setNum(n => n + 1);
  }

  return (
    <button onClick={increment}>Get More Excited!</button>
  );
}
```

Making AJAX requests when events occur

Example: Fetching a quote on click

You may remember seeing something like this in the Productiv exercise!

`demo/effects/src/QuoteFetch.js`

```
function QuoteFetcher() {

  const [quote, setQuote] = useState({text: "", author: ""});

  async function fetchAndSetQuote() {
    const response = await axios.get(RANDOM_QUOTE_URL);
    const randomQuote = response.data.quote;
    setQuote(randomQuote);
  }

  return (
    <div>
      <button onClick={fetchAndSetQuote}>Get Quote Using handler</button>
      <h1>{quote.text}</h1>
      <h2>{quote.author}</h2>
    </div>
  );
}
```

What happens if we want to fetch data when the component is first mounted?

Fetching Data on Initial Render

Use: Getting Data via AJAX on mount

- “When a component renders, fetch data from an API”
 - Data fetching is asynchronous, and may take a moment to complete
 - Want to show user something, like a loading message, while fetching
- To fetch correctly:
 - Have an effect that runs only once
 - Inside effect, when API calls is finished, will set state & re-render

`demo/effects/src/ProfileViewer.js`

```
function ProfileViewer() {
  const [profile, setProfile] = useState({
    data: null,
    isLoading: true
  });

  useEffect(function fetchUserWhenMounted() {
    async function fetchUser() {
      const userResult = await axios.get(URL);
      setProfile({
        data: userResult.data,
        isLoading: false
      });
    }
    fetchUser();
  }, [ ]);

  if (profile.isLoading) return <i>Loading...</i>;
}

return (
  <div>
    <b>{profile.data.name}</b>
  </div>
);
}
```

- Callback for useEffect cannot be async function itself
- We solve that by:
 - Defining async fn in it
 - Calling that async fn
- Change state with response

NOTE Handling Errors

This example is very simple, and doesn't do anything helpful if a network error happens.

A common pattern would be to have a piece of state for errors raised by the AJAX call, and this will be set if an error is thrown.

It's also common to have a specific piece of state to show that the component is currently loading data.

```
function ProfileViewer() {
  const [profile, setProfile] = useState({
    data: null,
    isLoading: true,
    errors: null,
  });

  useEffect(function fetchUserWhenMounted() {
    async function fetchUser() {
      try {
        const userResult = await axios.get(URL);
        setProfile({
          data: userResult.data,
          isLoading: false,
          errors: null,
        });
      } catch (err) {
        setProfile({
          data: null,
          isLoading: false,
          errors: err,
        });
      }
      fetchUser();
    }, []);
  });

  if (profile.isLoading) return <i>Loading...</i>
  else if (profile.error) return <b>Oh no! {error}</b>
  return ( <div>... stuff using data ...</div> )
}
```

Updating After Dependency Changes

Updating Data

Goal: Fetch data for user when form changes

Example: Text Search

demo/effects/src/ProfileViewerWithSearch.js

```
function ProfileViewerWithSearch() {
  const [username, setUsername] = useState("elie");
  const [profile, setProfile] = useState({data: null, isLoading: true});

  useEffect(function fetchUserOnUsernameChange() {
    async function fetchUser() {
      const userResult = await axios.get(`${BASE_URL}/${username}`);
      setProfile({data: userResult.data, isLoading: false});
    }
    fetchUser();
  }, [username]);

  function search(username) {
    setProfile({data: null, isLoading: true});
    setUsername(username);
  }

  if (profile.isLoading) return <i>Loading...</i>

  return (
    <div>
      <ProfileSearchForm search={search} />
      <b>{profile.data.name}</b>
    </div>
  );
}
```

Cleaning Up an Effect

Some effects will do something that needs to be cleaned up later:

- removing a registered event listener
- clearing `setInterval` or `setTimeout` usages
- disconnecting from a web socket

If you provide a clean up method, it will be run:

- Before the next call of that effect (if any)
- Before the component *unmounts* (is removed from DOM)

The function is provided the same state as the previous effect got when it was running. This allows you to “clean up” things that happened there.

To do this, we return a function from `useEffect`!

```
useEffect(function myEffect() {
  // runs on the first render and all times after
  // because we didn't pass in an array as a 2nd arg!
  console.log('Effect ran!');

  // if we return clean up function, it runs:
  // - before effect runs again
  // - also: before component unmounts
  return function cleanUp() { console.log('Cleanup phase!'); }
})
```

Using a cleanup

demo/effects/src/MoveBox.js

```
function MoveBox() {
  const [left, setLeft] = useState(0);

  useEffect(function registerKeyListener() {
    console.log("USE-EFFECT: adding event listener");
    window.addEventListener("keydown", handleKey);

    return function unmount() {
      console.log("UNMOUNTING: removing event listener");
      window.removeEventListener("keydown", handleKey);
    };
  }, []);

  /** Handle keypress & if space, move. */
  function handleKey(evt) {
    if (evt.key === " ") setLeft(v => v + 5);
  }

  return (
    <div className="innerbox" style={{marginLeft: left}}/>
  );
}
```

Clean up function has access to previous variables

Inside of your cleanup function, any variables have the values from when the effect that provided that cleanup function ran. So, this may not always be the up-to-date information held by your component. This is intended: this function might need the previous data, so it can clean up from that state.

Effects Wrapup

- Calling a state setter function triggers a re-render and you have the new value of state on that re-render
- Effects run after re-render to do stuff not related to rendering
- Effect can run:
 - After every render (*no dependency list given*)
 - After first render (*empty dependency list given*)
 - When any dependency in list changes
- Components cannot be async functions nor can *useEffect* callbacks
 - You **can** however create async functions inside of your components and use them as callbacks (event handlers)
 - But *useEffect* callbacks can *call* async functions



React Context

Goals

- Explain what context is
- Use the Context API to provide and consume context

Motivation

What is Context?

- Universal data across your application
- Data accessible across all components

When Is It Useful?

Useful for global themes, shared data:

- Passing a user's name around
- User currency preferences
- Light or dark mode theme selection

Creating Context

demo/user-context/src/userContext.js

```
import { createContext } from "react";
const userContext = createContext();
export default userContext;
```

This gives us a component:

- *<userContext.Provider>*: can provide a value to context

Provider

demo/user-context/src/JoblyApp.js

```
import { useState } from "react";
import UserPrefForm from "./UserPrefForm";
import JobDetail from "./JobDetail";
import CompanyDetail from "./CompanyDetail";
import userContext from "./userContext";

const DEFAULT_PREFS = {color: "dark", currency: "USD"};

function JoblyApp() {
  const [prefs, setPrefs] = useState(DEFAULT_PREFS);

  function updatePrefs(newPrefs) {
    setPrefs(newPrefs);
  }

  return (
    <userContext.Provider value={{user: null, prefs}}>
      <UserPrefForm submit={updatePrefs} currPrefs={prefs} />
      <JobDetail />
      <CompanyDetail />
    </userContext.Provider>
  );
}
```

Any component descending from the Provider can access the context value.

useContext

demo/user-context/src/JobDetail.js

```
import { useContext } from "react";
import userContext from "./userContext";

function JobDetail({ title, description, salary }) {
  const { prefs } = useContext(userContext);

  return (
    <div>
      <h2>Fake Job!</h2>
      <p>You should apply! It pays lots of {prefs.currency}.</p>
    </div>
  )
}
```

In order to get the value, we need the `useContext` hook.

- `useContext` looks for a matching context, and reads its value.
- When value inside of context changes, components accessing that context will re-render.
- Components that read from context with `useContext` are sometimes called *consumers* (as opposed to providers).

Storing User in Context

```
function JoblyApp() {
  // same as before but also...

  const [user, setUser] = useState(null);

  function login({username, password}) {
    // lots of logic here, but then ...
    const userFromAPI = {get_this_from_api};
    setUser(userFromAPI);
  }

  return (
    <userContext.Provider value={{user, prefs}}>
      {/* same, plus ... */}
      <LoginForm submit={login} />
    </userContext.Provider>
  );
}
```

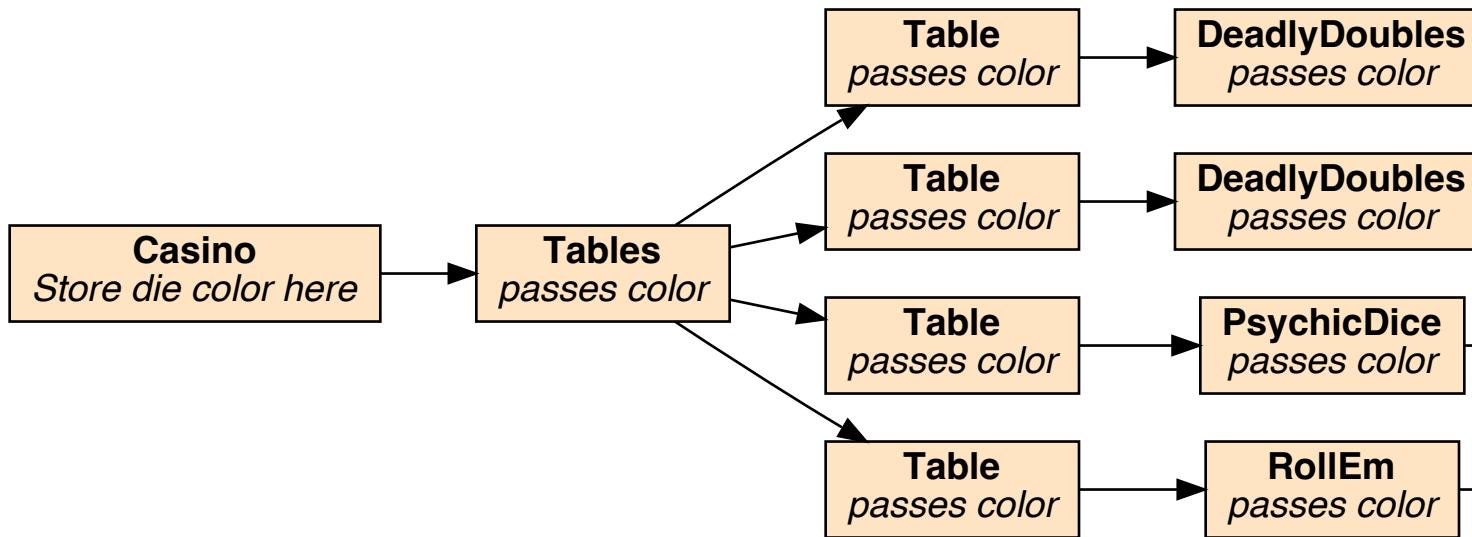
Demo Time

Deadly Doubles

A casino of different dice table games.

Try it out: <https://react-deadly-doubles.surge.sh/> <<https://react-deadly-doubles.surge.sh/>>

Without Using Context



Can you see the challenge?

All components between *Casino* and *Die* don't care about color for dice

Using Context



Now those components don't need to pass down props they don't care about

React Features

- A generic component, *Table*, which can render different games.
- React's context manager: *Casino* lets you choose a favorite color, and *Die* (several layers down) can access the color you chose.

Guidelines for When To Make a Component

- If I didn't, and inlined this in the parent component, would that make the parent state more complex?
 - Mixing together state in the games (what are values of the dice) with state in *Table* (how many wins/losses) would make things more complex
- Can I “not repeat myself”?
 - Having *Table* lets us reduce repetition in different game components.
- Might this component be usable elsewhere?
 - The *DiceSet* is useful in all the games
- Might I want to swap it out?
 - Having *Die* be a separate component makes it easier to replace it for *DieAlt*

Should I Use Context For Passing Everything?

No!

Passing the “real data” down as traditional props is better — it's clear, obvious, and how React is meant to be used.

Context is good for cases where lots of different components need incidental access to non-core-data, and is convenient for them to do it easily.



Introduction to TypeScript

Goals

- Understand what TypeScript is
- Understand benefits/drawbacks of typing
- See some basic examples of TS fundamentals
- Use interfaces to define a type pattern for an object

What is TypeScript?

- TypeScript is a *superset* of JavaScript
 - Any JavaScript program is *already* TypeScript
- Browsers don't use TypeScript directly; it needs to be transpiled
- Primarily provides optional static typing, classes and interfaces
- Helps editors to spot type errors while writing/reading code

```
let score = 0
// score is a number now

// TS won't allow type changes
score = "zero" // error
```

```
function rando() {
  return (Math.random() < 0.5)
    ? 42
    : "string";
}

let either = rando();
// can only use methods common
// to both strings and numbers
```

Static vs Dynamic Languages

Statically typed: Dynamically typed:

- | | |
|-----------|-----------------------------|
| ◦ Java | ◦ Ruby |
| ◦ Swift | ◦ Python (optionally typed) |
| ◦ C / C++ | ◦ JavaScript |

Benefits / Drawback of Typing

Benefits:

Drawbacks:

- Spot bugs early at compile time
- Code can be more verbose

- Predictability / readability
- Learning curve

How popular is TypeScript?

_images/ts-popularity.webp

△ StackOverflow survey in late 2021

Getting started

Compiling

```
$ npm install typescript
```

compile `hello.ts` to `hello.js`

```
$ npx tsc hello.ts
```

compile `hello.ts` to `out/hello.js`

```
$ npx tsc --outDir out hello.ts
```

Configuration file

TypeScript uses `tsconfig.json`, if it exists:

```
$ npx tsc --init
```

`tsconfig.json`

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  }
}
```

TIP Documentation on TS compiler

See Compiler options <<https://www.typescriptlang.org/docs/handbook/compiler-options.html>> and TSconfig options <<https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>>

HINT Can use Webpack / Create React App for this

When you build projects, you will not want to manually type `npx tsx` every time, thankfully there are tools like `Webpack` that will do this bundling and compiling for you.

When we write TypeScript with React, Create React App will automatically compile and handle this process for us.

TypeScript Playground

A great way to play with TypeScript:

<https://www.typescriptlang.org/play/> <<https://www.typescriptlang.org/play/>>

TypeScript basics

Variables

```
// learns that age is a number
let age = 42;

// or: provide type explicitly
let age: number;
```

Editor and compiler can detect errors

- Invalid methods

```
age.split("")
```

- Redefining to invalid value

```
age = "forty-two";
age = undefined;
```

Arrays

```
const sillyBabyNames = ["Carrot", "Chicken Nugget"];
sillyBabyNames.push("Kale");
```

This will learn this is an array of strings:

```
// will throw a type error
sillyBabyNames.push(1337);
```

Creating empty arrays:

```
const anyValue = [];

// could add anything to:
anyValue.push("string", 1337);

// can specify type for items directly
const onlyStrings: string[] = [];

// would throw an error
onlyStrings.push("string", 1337);
```

Objects

```
let movieData = {
  title: "The Godfather",
  year: 1972,
};
```

Will learn types and enforce:

```
movieData.year = 2022; // ok!
movieData.year = "nope"; // error!
```

Cannot add new properties, though:

```
movieData.director = "Coppola"; // error!
```

Records

An object type that can handle new keys using *Record<keyType, valueType>*:

```
const person: Record<string, string> = {};
person.firstName = "Enzo";
```

That angle-bracket syntax is using a “generic”.

TIP Map can also be used for this

The `Map` type in TypeScript/JavaScript is also an alternate way to handle arbitrary keys. Maps are often better choices because they can have non-string keys and can sort sensibly.

```
let letterToScore = new Map([
  ["A", 1],
  ["B", 3],
]);
```

Will learn type `<string, number>` for this Map:

```
letterToScore.set("C", 2);      // ok!
letterToScore.set("D", "oops"); // error!
```

Specifying type explicitly:

```
let letterToScore = new Map<string, number>();

letterToScore.set("C", 2);      // ok!
letterToScore.set("D", "oops"); // error!
```

Functions

```
function isSeven(n: number): string {
  if (n === 7) return "THAT IS INDEED 7!";
  return "NOPE, NOT SEVEN :(";
}
```

Return type is optional if can be inferred.

Cannot pass too many/two few/wrongly-typed arguments:

```
isSeven();           // error!
isSeven(1, 2);     // error!
isSeven("one");    // error!
```

If your function returns `undefined`, mark the return type as `void`.

```
function hi(): void {
  console.log("HI"); // doesn't return anything!
}
```

Classes

```
class Cat {  
    name: string;  
  
    constructor(name: string) {  
        this.name = name;  
    }  
  
    dance(dance: string = "tango") {  
        return `${this.name} dances the ${dance}`;  
    }  
}  
  
let fluffy = new Cat("Fluffy");
```

Interfaces

You can define your own types, called an *interface*.

```
interface UserInterface {  
    username: string;  
    age: number;  
}  
  
let jane: UserInterface = {  
    username: "jane",  
    age: 42,  
};
```

This will enforce requirements:

```
let bob: UserInterface = {  
    username: "bob",  
}; // error
```

```
interface UserInterface {  
    username: string;  
    age?: number;  
}  
  
let jane: UserInterface = {  
    username: "jane",  
};
```

TIP How should you name interfaces?

There are two different popular opinions here:

- Name it descriptively with the word “Interface” at the end, like *UserInterface*.
- Name it descriptively but with a leading “I”, like *IUser*.

Our lead instructors have different opinions here, and your company will probably have a preferred style. Remember, always match existing code style practices.

Interfaces and functions

```
function showUserInfo(user: UserInterface): void {
    console.log(` ${user.username} is ${user.age}`);
}
```

```
function makeRandomUser(name: string): UserInterface {
    return {
        username: name,
        age: Math.floor(Math.random() * 100),
    }
}
```

Can specify “compatible” part of interface:

```
interface StudentInterface {
    name: string;
    cohort: string;
}

interface InstructorInterface {
    name: string;
    salary: number;
}

function getUpperName(person: {name: string}): string {
    return person.name.toUpperCase();
}
```

NOTE Another way to make your own types

In addition to interfaces, there also exists a keyword called *Type* which you can use to create custom types.

```
// define the type
type Status = [number, string];

// use it
let response1: Status = [404, "NOT FOUND"];
let response2: Status = [200, "OK"];
```

This is similar to, but different from interfaces. It's not important to learn about this now, but if you end up working a lot with TypeScript, it will be helpful information.

You can read more about the differences here <https://react-typescript-cheatsheet.netlify.app/docs/basic/getting-started/basic_type_example/#useful-table-for-types-vs-interfaces>

Flexible types

Choice of types

Can provide multiple type possibilities:

```
let x: number | string;  
  
x = 42;  
x = "hello";
```

Can use only methods common to both strings and numbers.

Opting out with *any*

For the ultimate flexibility, can use type *any*:

```
let x: any;  
  
x = 42;  
x = "hello";
```

Be cautious about this — you lose the benefits of typing!

Forcing type info

Forcing a type

If you have more info about a value than TypeScript could, use *as*:

```
const h1 = document.querySelector("h1") as HTMLElement;
```

Forcing not null/undefined

A function might be typed to return a string or null (or undefined):

```
function myFunc(s: string): string | null {  
  if (s === "nope") return null;  
  return "ok!";  
}
```

If you know that it will return a non-null, non-undefined value, you can tell it that when you call it:

```
let a = myFunc("good");  
a.slice();           // error!  
  
let b = myFunc("good")!;    // note the "!"  
b.slice();
```

Checking types

```
function maybe(): string | void {
  if (Math.random() < 0.5) return "ok";
}

let x = maybe();

if (typeof x === "string") {
  // can use string methods here!
}
```

Async functions

Remember, *async* functions return promises.

```
async function myAsyncFunc(): number {
  return 42;
}
```

Instead, do this:

```
async function myAsyncFunc(): Promise<number> {
  return 42;
}
```

Next steps

- Start small!
- Migrating from JavaScript <<https://www.typescriptlang.org/docs/handbook/migrating-from-javascript.html>>
- Todo App in Typescript <<https://ts.chibicode.com/todo/>>
- You **really** do not need to know everything



TypeScript And React

Goals

- Review interfaces and learn when to use them
- Learn to use TypeScript with React
- Learn (the basics) of using TypeScript and Express

Interfaces

Interfaces can specify a custom type:

```
interface UserInterface {  
    name: string;  
    email: string;  
    age?: number;  
}  
  
function logIn(user: UserInterface) {  
    console.log(` ${user.name} ${user.age}`);  
}
```

But you could also do this:

```
function logIn(user: { name: string; email: string; age: number }) {  
    console.log(` ${user.name} ${user.age}`);  
}
```

Advantages of Interfaces

If this can be done with an inline declaration, why create an interface?

- Interfaces can be reused
- Interfaces can be extended

Reuse

Compare these:

```
function logIn(user: { name: string, email: string; age?: number }) { }  
function signUp(user: { name: string, email: string; age?: number }) { }
```

To these:

```
function logIn(user: UserInterface) { }
function signUp(user: UserInterface) { }
```

TIP How should you name interfaces?

There are two different popular opinions here:

- Name it descriptively with the word “Interface” at the end, like *UserInterface*.
- Name it descriptively but with a leading “I”, like *IUser*.

Our lead instructors have different opinions here, and your company will probably have a preferred style. Remember, always match existing code style practices.

Extensibility

Interfaces can inherit and build off of other interfaces.

Suppose some users are employees, and have a salary.

```
interface UserInterface {
  name: string;
  email: string;
  age?: number;
}

interface EmployeeInterface extends UserInterface {
  salary: number;
}
```

Definitions/Declarations files

- When using JavaScript code from TypeScript, it’s helpful to have type help.
- This information can be provided in `.d.ts` files.
 - Example: [React’s definition file <https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/react/v16/index.d.ts>](https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/react/v16/index.d.ts)
 - When you install any library you will always want to bring in the types as well
 - The project **DefinitelyTyped** has just about everything
 - Some libraries, like *axios*, already have types and don’t need a defs file

```
$ npm install @types/lodash --save-dev
```

```
import * as _ from "lodash";

function rollDice(): number {
  return _.random(1, 6);
}

console.log(rollDice())
```

Adding TypeScript to React

You can use the TypeScript template when making your app:

```
$ npx create-react-app todo-ts-demo --template typescript
```

Migrating to TypeScript

If your app didn't start with TypeScript, you can add it later.

```
$ npm install --save typescript @types/node @types/react  
$ npm install --save @types/react-dom @types/jest
```

Rename all of your files to `.ts` or `.tsx`

In `tsconfig.json` change *strict* option to *false*

IMPORTANT Make sure to add your own `tsconfig.json`

Create React App does not support generating a `tsconfig.json` file for migrating apps. In order to get the config file you'll need to create a fresh app with the typescript template and copy the `tsconfig.json` to your existing project.

Using with React

Props

Make props an interface — React will infer the return type

```
interface StudentPropsInterface {  
  name: string;  
  email: string;  
}  
  
function PersonInfo({ name, email }: StudentPropsInterface) {  
  console.log("email = ", email);  
}
```

State

With simple state (eg strings, numbers): React will infer the right type:

```
const [name, setName] = useState("");
```

If the state is complex, use a Generic:

```
const [user, setUser] = useState<UserInterface | null>(null);  
const [data, setData] = useState<{s: string}>({ s: "ok" });
```

Effects

Effect callbacks almost always return void

```
useEffect(function loadOnMount() : void {  
  // my effect ...  
})
```

NOTE Effect callback return a function for cleanup

There's an advanced feature where an effect callback can return a function that will be called when the component unmounts. If a callback returns a function, it will need to be typed with the function type.

Context

companyContext.ts

```
interface CompanyInterface {  
  name: string;  
  address: string;  
}  
  
const companyContext = React.createContext<CompanyInterface | null>(null);
```

MyComponent.ts

```
const sampleContext: CompanyInterface = {  
  name: "Rithm School",  
  address: "500 Sansome Street",  
};
```

Events

Typing events is tedious but straightforward:

```
function handleChange(evt: React.ChangeEvent<HTMLInputElement>) {  
  const { name, value } = evt.target;  
  setFormData(formData => ({  
    ...formData,  
    [name]: value,  
  }));  
}  
  
/** Submit form: call function from parent & clear inputs. */  
function handleSubmit(evt: React.FormEvent) {  
  evt.preventDefault();  
  // do stuff  
}
```

React-Typescript Cheatsheet: Forms and Events <https://react-typescript-cheatsheet.netlify.app/docs/basic/getting-started/forms_and_events>

TypeScript with Express

- Requires some manual configuration
 - There are boilerplate setups available online
- TypeScript uses *import* (not *require*) for modules

Trying out demo

```
$ npm install  
$ npm start
```

Cheatsheet

React-Typescript Cheatsheet <<https://react-typescript-cheatsheet.netlify.app/docs/basic/setup/>>



Cloud Storage & AWS S3

Cloud Storage

Databases are great for storing *data*, but not great for storing *files*:

- files typically will be served directly to browser; you shouldn't need to involve the db/web app to do so
- databases have complex transactions/replication requirements
- large files bloat databases, reducing query/index efficiency

For files, it's better to store them as files

Storing Files

If your site needs to store files, you could:

- Have a web form with a *file* input
- On submission of the form, you could write a file to disk, *and* store some metadata about the file (title, path, etc) in the database

resume_title	resume_path
User1 Resume	static/resumes/user1.pdf
User2 Resume	static/resumes/user2.pdf

```
<a href="{{ user.resume_path }}">{{ user.resume_title }}</a>
```

Scaling

- What if your site grows, and you have multiple application servers?
- You can build replication that copies the files across all servers.
- What are the security risks of writing to the same disk as your app?
- These are solvable, but there's a common good choice.

Object Storage

Different cloud services providers have created object storage — like Dropbox or Google Files, but for developers

- Can serve files for you, freeing up your server from doing that
- Can backup your files so even hardware failure won't lose them

- Can replicate and serve your files around the world
- Can offer extremely granular security
- Can have low, extremely predictable pricing

About AWS

Amazon is the largest provider of on-demand computing services.

- Great for huge sites that need massive scale, like Netflix
- Great for small sites that can't afford to build their own infrastructure
- Almost all services are charged by unit, making it easy to experiment

Free Tier

Many (but not all) AWS services fall under their “free tier”, where you can use a fully-working version of the service free for one year.

Some services (like SNS/SQS) even have a permanent free threshold: you can use them forever for free, assuming you use less than x amount.

<https://aws.amazon.com/free/> <<https://aws.amazon.com/free/>>

Creating an Account

Get an AWS account: <http://aws.amazon.com> <<http://aws.amazon.com>>.

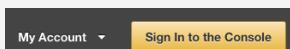
Creating an account

This process takes a few minutes. You'll need to provide Amazon with your contact information and a valid credit card number. The type of server used in this lecture is free for the first month and \$5 afterwards. Unlike some other subscription-based services, though, you really can cancel easily anytime, and you're only charged for the time you use.

Amazon will prompt you for the level of service contract you'd like for your account. Assuming you'd like to stay in the free tier, choose “Basic”.

Steps to create an AWS account

1. On the [Amazon Web Services homepage](https://aws.amazon.com/) <<https://aws.amazon.com/>>, click **Sign In to the Console** in the top right-hand corner of the screen.



2. Enter your email address and select **I am a new user**

Sign In or Create an AWS Account

What is your email (phone for mobile accounts)?

E-mail or mobile number:

I am a new user.

I am a returning user
and my password is:

[Sign in using our secure server](#)

3. Enter your login credentials on the next page and click **Create account**.

Login Credentials

Use the form below to create login credentials that can be used for AWS as well as Amazon.com.

My name is: Sarah Stringer

My e-mail address is:

Type it again:

note: this is the e-mail address that we will use to contact you about your account

Enter a new password:

Type it again:

[Create account](#)

4. Enter your contact information.

Company Account Personal Account

* Required Fields

Full Name*

Country*

Address*

City*

State / Province or Region*

Postal Code*

Phone Number*

Security Check



Please type the characters as shown above

5. Enter payment information. AWS accounts themselves are free, but higher levels of support and various AWS services are paid. You are required to enter a valid credit card number to create an account, but you may select to remain on the basic, free tier later in this process.

Please enter your payment information below. You will be able to try a broad set of AWS products for free via the Free Tier. We will only bill your credit or debit card for usage that is not covered by our Free Tier.

> Frequently Asked Questions

Credit/Debit Card Number

Expiration Date

Cardholder's Name

Use my contact address

Use a new address

[Continue](#)

6. AWS will confirm your identity through a phone call. Enter your phone number and click **Call me now**.

1. Provide a telephone number

Please enter your information below and click the "Call Me Now" button.

Security Check



Please type the characters as shown above

Country Code	Phone Number	Ext
United States (+1)	<input type="text"/>	<input type="text"/>

Call Me Now

You will receive a PIN on the screen. You should soon receive an automated call from AWS and you will be prompted to enter the PIN on your keypad during the call.

1. Provide a telephone number ✓

2. Call in progress
Please follow the instructions on the telephone and key in the following Personal Identification Number (PIN) on your telephone when prompted.

PIN: 6739

If you have not yet received a call at the number indicated above please wait. This page will automatically update with what you need to do next.

3. Identity verification complete

After you have successfully entered the PIN during the phone call, your screen should refresh and you will see **Identity verification complete**.

1. Provide a telephone number ✓

2. Call in progress ✓

3. Identity verification complete
Your identity has been verified successfully.

Continue to select your Support Plan

7. Choose your support plan. Assuming you'd like to stay in the free tier, choose "Basic".

Basic
Description: Customer Service for account and billing questions and access to the AWS Community Forums.
Price: Included

8. You will be redirected back to the AWS homepage, where you should see a **Complete Sign Up** button in the top right-hand corner of the screen. Click that button to finalize your account creation.

My Account ▾ **Complete Sign Up**

Amazon Credentials

Once you've done that, go to your account's Security Credentials, and create a new Access Key.

Make sure you save the access key ID and secret key — you're never shown the secret key a second time!

Using S3

Manage via Web Dashboard

<https://s3.console.aws.amazon.com/s3/home> <<https://s3.console.aws.amazon.com/s3/home>>

Manage via shell

<https://aws.amazon.com/cli/> <<https://aws.amazon.com/cli/>>

```
$ aws s3 ls s3://your-bucket  
$ aws s3 cp localfile s3://your-bucket/
```

Manage via Python

<https://aws.amazon.com/sdk-for-python/> <<https://aws.amazon.com/sdk-for-python/>>

```
s3 = boto3.client(  
    "s3",  
    "us-west-1",  
    aws_access_key_id="...",  
    aws_secret_access_key="...",  
)  
  
upload_file(file_name, bucket, object_name)
```

Manage via JavaScript

<https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/s3-examples.html>
<<https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/s3-examples.html>>

Getting Amazon to Serve Files

Check that your bucket security allows it!

Get a URL or “presigned URL” (time-expiring) via the API

Alternatives to AWS S3

Amazon is a problematic company

(Offered as a personal aside by Joel)

Amazon is ubiquitous in the technology landscape, but is a problematic company with practices that many not align with your values.

Some areas to consider:

- Their CEO is the wealthiest person *in the world*, yet is not involved in any meaningful contributions to supporting less affluent people.
- Despite their massive corporate wealth, their factory employees work in extremely poor conditions, denied job protections and union representation
- Their business practices combine elements of both a monopoly and a monopsony.
- The consolidation of computer power in a single component has considerable risks for innovation and electronic freedom

Of course, however, since they are the largest player in this scene, it is almost unavoidable that you'll work with them in your career. It's useful to have some experience with their services.

There are alternatives to AWS, however, and you can explore if you'd like and make your own judgements about using AWS.

/Joel steps off sandbox.

- [DigitalOcean](https://www.digitalocean.com) <<https://www.digitalocean.com>> offers S3-compatible storage and other services like S3
 - They have a different pricing model: \$5/month for a bucket with no other charges
- [Microsoft Azure](https://azure.microsoft.com) <<https://azure.microsoft.com>> offers similar services, and many other things like Amazon

Why do I need to know about cloud storage?

- 95% of serious web applications use a service like this, with the majority using AWS.
- It is very likely that you'll end up at a company utilizing cloud storage!

Your turn

- Using AWS S3 is a **requirement** for the CYOA sprint.
- We have intentionally omitted specific step-by-step instructions because we want you to do the research/setup on your own.
- Once you decide on the technologies you'll be using for your project, get S3 setup and working so that you're able to store files and serve them publicly.
- This will take longer than you think, so do not put this off to the very end!



React Wrapup

React Core Ideas

- Props
- State
- Effects
- Context
- Router

Class components

Another way to build components: as classes rather than functions

This is how most people built React components until 2021; some still do, and there's lots of code out there using this

```
function TaskList({ empId }) {  
  const [tasks, setTasks] = useState([]);  
  
  useEffect(function loadOnMount() {  
    async function load() {  
      const tasks = await api.get(empId);  
      setTasks(tasks);  
    }  
    load();  
  }, [empId]);  
  
  return (  
    <div>  
      {tasks.map(  
        task => <Task info={task} />)}  
    </div>  
  );  
}
```

```
class TaskList extends React.Component {  
  state = {tasks: []};  
  
  async getTasks() {  
    const tasks = await api.get(  
      this.props.empId);  
    setState({tasks});  
  }  
  
  componentDidMount() { getTasks() }  
  componentDidUpdate() { getTasks() }  
  
  render() {  
    return (  
      <div>  
        {this.state.tasks.map(  
          task => <Task info={task} />)}  
      </div>  
    );  
  }  
}
```

useRef

- *useRef* is a hook for holding onto a variable or DOM pointer
- It returns a mutable object (initially set to the default), which persists across renders
- Mutating the ref object does not trigger a re-render.
- This is often useful for:

- Accessing an underlying DOM element
- Setting up / clearing timers

Use: Manipulating a DOM Element

Need to access a DOM element to for a non-React API or to integrate some other JavaScript library?

This is a great time to use a ref!

`demo/effects/src/Video.js`

```
function Video({ src = URL }) {
  const vidRef = useRef(); /* 1 */
  const [speed, setSpeed] = useState(1);

  useEffect(function adjustPlaybackRate() {
    // vidRef.current is DOM Element; it has
    // .playbackRate that changes speed of video
    vidRef.current.playbackRate = speed; /* 3 */
  }, [vidRef, speed]);

  function slow(evt) { setSpeed(s => s / 2) }
  function fast(evt) { setSpeed(s => s * 2) }

  return (
    <div className="Video">
      <button onClick={slow}>slow</button>
      <button onClick={fast}>fast</button>
      Current speed: {speed}x
      <video autoPlay loop ref={vidRef}/* 2 */>
        <source src={src} />
      </video>
    </div>
  );
}
```

1. Create ref to fill in during render
2. Capture video element during render
3. Mutate element to change speed
 - Doesn't cause a re-render

Use: Timers

Another great time to use a ref is with timers like `setInterval`.

`setInterval` returns a timer ID, which we need to stop the timer from running.

We can store that ID in a ref, and then stop the timer when component unmounts.

demo/effects/src/TimerWithRef.js

```
function TimerWithRef() {
  const timerId = useRef(); /* 1 */
  let [count, setCount] = useState(0);

  useEffect(function setCounter() {
    console.log("setCounter");
    timerId.current = setInterval( /* 3 */
      () => setCount(c => c + 1),
      1000);

    return function cleanUpClearTimer() {
      console.log("cleanUpClearTimer");
      clearInterval(timerId.current); /* 4 */
    };
  }, [timerId]);

  return (
    <div>
      <h1>{count}</h1>
      timerId: {timerId.current} /* 2 */
    </div>
  );
}
```

1. Make ref to fill in later
2. Can use variable during render
3. Mutate to update value of ref
 - Does not trigger a rerender
4. During unmount, can use ref
 - Stops timer

WARNING Be Careful of Misuse

Since refs can expose DOM elements for us, it can be tempting to use them to access the DOM and make changes (toggle classes, set text, etc).

This is **not** how refs should be used. React should control the state of the DOM!

From the docs: <<https://reactjs.org/docs/refs-and-the-dom.html#dont-overuse-refs>>

Your first inclination may be to use refs to “make things happen” in your app. If this is the case, take a moment and think more critically about where state should be owned in the component hierarchy.

Refs Wrap Up

- `useRef` provides a mutable object that you can treat like a variable
 - Common: remember a DOM element so you can interact with it
 - Common: remember a non-state piece of information, like a timer ID

Reducers

For sites with complex state, can centralize state logic into a single complex state and a *reducer* function

What is a reducer?

- Function accepting two objects:
 - current state
 - { count: 1 }
 - action describing change to make
 - { type: "increment" }
- Pure functions
 - Don't mutate anything; rely only on args
- Use action to make & return a new state
 - reducer(st, action) // { count: 2 }

```
function reducer(state, action) {  
  if (action.type === "+") {  
    return { count: state.count + 1 };  
  }  
  if (action.type === "-") {  
    return { count: state.count - 1 };  
  }  
  
  return state;  
}
```

Reducer example: Counter

demo/counter/src/Counter.js

```
function reducer(state, action) { /* 1 */  
  if (action.type === "+") {  
    return { count: state.count + 1 };  
  }  
  if (action.type === "-") {  
    return { count: state.count - 1 };  
  }  
  return state;  
}  
  
function Counter() {  
  const [state, dispatch] = useReducer( /* 2 */  
    reducer, initState);  
  
  function dec() { dispatch({type: '-'}) } /* 3 */  
  function inc() { dispatch({type: '+'}) } /* 3 */  
  
  return (  
    <div>  
      Count: {state.count}  
      <button onClick={dec}>-</button>  
      <button onClick={inc}>+</button>  
    </div>  
  );  
}
```

1. Reducer function

- Takes curr state & action
- Returns new state
- Isn't called directly

2. Hook to make dispatch function

- Dispatches action to reducer

3. Dispatching an action

Patterns with useReducer

- Keep reducer in own file to unit test
- Can pass reducer around easily with *useContext*

```
demo/counter/src/rootReducer.js

const INITIAL = { count: 0 };

function reducer(state = INITIAL, action) {
  if (action.type === "+") {
    return {count: state.count + 1};
  }
  if (action.type === "-") {
    return {count: state.count - 1};
  }

  return state;
}
```

Benefits of reducer pattern

- Capture your state for a particular resource/element
- Easier to test functions that change your state
- Can use with *useState* if needed; doesn't have to be only approach

useState vs useReducer

- An independent piece of state to manage? *useState*
- State relies on another part of state to update? *useReducer*
- Complex nested data type in state? *useReducer*
- Tips:
 - Don't just use reducers because you have lots of state
 - Start with useState, move to useReducer when you need to

Effects and Functions: Function Dependencies

demo/use-callback-example/src/NumberFactNoCallback.js

```
function NumberFactNoCallback({  
  baseUrl = "http://numbersapi.com/", initialNum = 42  
}) {  
  const [num, setNum] = useState(initialNum);  
  const [fact, setFact] = useState("");  
  async function getFact(newNum) {  
    let response = await axios.get(` ${baseUrl}${newNum}?json`);  
    setNum(newNum);  
    setFact(response.data.text);  
  }  
  
  useEffect(() => { getFact(initialNum) }, [initialNum, getFact]);  
  
  return (  
    <div>  
      <NumberInput getFact={getFact} initialNum={initialNum} />  
      {fact ? <div><h3>{num}</h3><p>{fact}</p></div> : <p>Loading...</p>}  
    </div>  
  );  
}
```

⚠ ./src/NumberFactNoCallback.js webpackHotDevClient.js:138
Line 10:3: The 'getFact' function makes the dependencies of useEffect Hook (at line 16) change on every render. To fix this, wrap the 'getFact' definition into its own useCallback() Hook react-hooks/exhaustive-deps

useCallback

- *useCallback* is a built-in hook that accepts a function and an array of dependencies
- It returns a function that won't be re-declared on subsequent renders, as long as the dependencies don't change
- This allows you to add functions as dependencies to *useEffect* without hitting infinite render issues

demo/use-callback-example/src/NumberFactUseCallback.js

```
function NumberFactUseCallback({  
  baseUrl = "http://numbersapi.com/", initialNum = 42  
}) {  
  
  const [num, setNum] = useState(initialNum);  
  const [fact, setFact] = useState("");  
  
  const getFact = useCallback(async newNum => {  
    let response = await axios.get(` ${baseUrl}${newNum}?json`);  
    setNum(newNum);  
    setFact(response.data.text);  
  }, [baseUrl]);  
  
  useEffect(() => { getFact(initialNum); }, [initialNum, getFact]);  
  
  return (  
    <div>  
      <NumberInput getFact={getFact} initialNum={initialNum} />  
      {fact ? <div><h3>{num}</h3><p>{fact}</p></div> : <p>Loading...</p>}  
    </div>  
  );  
}
```

Here's a common scenario in React:

- You have a function you want to call inside of an effect that depends on props or state.
- Since it depends on props / state, it should be listed as a dependency.
- But if the function is defined inside of the component, this can cause infinite render loops!

useMemo

- `useMemo` is another built-in hook in React
- Like `useCallback`, but for remembering values other than functions
- Accepts a function returning a value and an array of dependencies
- React won't recompute the values if the dependencies stay the same
- Helpful for caching the results of expensive operations

`demo/more-hooks/src/NumberDivisors.js`

```
import React, { useMemo } from "react";
import { getDivisors } from "./helpers";

function NumberDivisors({ num }) {
  // don't recompute the divisors
  // if the number is unchanged
  let divisors = useMemo(() => getDivisors(num), [num])

  return (
    <div>
      Here are all the divisors of {num}!
      <ul>
        {divisors.map(divisor => (
          <li key={divisor}>{divisor}</li>
        )));
      </ul>
    </div>
  );
}
```

React.memo

- “Memoize” component rendering
- Performance improvement for component
 - Don’t re-render if props/state/context are same
 - Normally: re-renders when parent does
- `React.memo`: function wrapping a component

`demo/use-reducer-memes/src/Meme.js`

```
function Meme({ top, bottom, url }) {
  console.log("Meme rendered!")

  return (
    <div className="Meme container">
      <i className="top">{top}</i>
      <img src={url} />
      <i className="bottom">{bottom}</i>
    </div>
  );
}

export default React.memo(Meme);
```

Webpack

Create React App includes Webpack <<https://webpack.js.org/>>:

- Lets you use JS modules (*import / export*)
- Combines your JS into one file
- Can easily use NPM modules in your JS

You can use this in your non-CRA projects: [Webpack Getting Started <https://webpack.js.org/guides/getting-started/>](https://webpack.js.org/guides/getting-started/)

Babel

Babel transpiles JSX/ultra-modern JS into conventional JS

- You can [experiment with this online <https://babeljs.io/repl>](https://babeljs.io/repl)
- Or [install it a a command-line tools <https://babeljs.io/setup>](https://babeljs.io/setup)

Useful Add-Ons

PropTypes

Can document/verify that types of props are as expected:

```
$ npm install prop-types
```

```
import PropTypes from 'prop-types';

class Greeting extends Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

PropTypes docs <<https://reactjs.org/docs/typechecking-with-proptypes.html>>

Styled Components

Can make “CSS-wrapped components” from your components:

```

const Title = styled.h1`  

  font-size: 1.5em;  

  text-align: center;  

  color: palevioletred;  

`;  
  

const Wrapper = styled.section`  

  padding: 4em;  

  background: papayawhip;  

`;  
  

function Demo {  

  return (  

    <Wrapper>  

      <Title> Hello World! </Title>  

    </Wrapper>  

  );  

}

```

Getting Started With Styled Components <<https://www.styled-components.com/docs/basics>>

Redux

- An add-on library that is sometimes used with React
- Use the “reducer” pattern that *useReducer* does
 - But even larger and more featureful
- To use with React, use *react-redux*

Redux vs useReducer

- Redux has excellent developer tools
- Redux has additional middleware for complex state management
- A good place to start:
 - Use useState for apps with straightforward state
 - Use useState + useReducer + useContext for intermediate complexity
 - Use useState/useReducer + Redux for very complex state
 - <https://www.robinwieruch.de/redux-vs-usereducer> <<https://www.robinwieruch.de/redux-vs-usereducer>>

Other state-management libraries

- Extensions to Redux: **redux-saga** / **redux-thunk** / **reselect**
- Simplify changing state without mutating: **immer** or **immutable.js**
- Alternatives to redux: **MobX**, **Xstate**, **recoil**

React Wrap-Up

React pushes you toward:

- Separating your logic/presentational concerns
- Component composability/reusability
- Thinking about your state & how to render component in each state
- Testable, pure components

Does everyone use React?

No.

There are plenty of code bases in the world (including new ones) that use imperative approaches using vanilla JS or jQuery

But larger, opinionated frameworks are increasingly popular

React is the most popular, but Angular, Vue, and Svelte are also common

Other places for React

- Building backends using React ideas: **Next.js, Gatsby**
- Building mobile-native (iOS, Android) apps: **React Native**



Arrays and Linked Lists

Goals

- Describe what an “abstract data type” means
- Compare different types of arrays
- Define singly and doubly linked lists
- Compare performance characteristics of arrays and lists
- Implement linked lists in JavaScript

Lists

A *list* is an *abstract data type*

It describes a *set of requirements*, not an exact implementation.

- Keep multiple items
- Can insert or delete items at any position
- Can contain duplicates
- Preserves order of items

Arrays

Arrangement of items at equally-spaced, sequential addresses in memory

[3, 7, 2, 4, 1, 2]

In memory:

3	7	2	4	1	2				
---	---	---	---	---	---	--	--	--	--

Therefore, inserting or deleting an item requires moving everything after it.

Array Runtimes

- Retrieving by index
 - $O(1)$
- Finding
 - $O(n)$

- General insertion
 - $O(n)$
- General deletion
 - $O(n)$

Direct Arrays / Vectors

This kind of array is often called a *direct array* or *vector*

Direct arrays only work if items are same size:

- all numbers
- all same-length strings

Don't work well when items are varied sizes:

- different length strings
- subarrays or objects

They're not commonly used, but JavaScript provides these as [Typed Arrays](#)
[<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays)

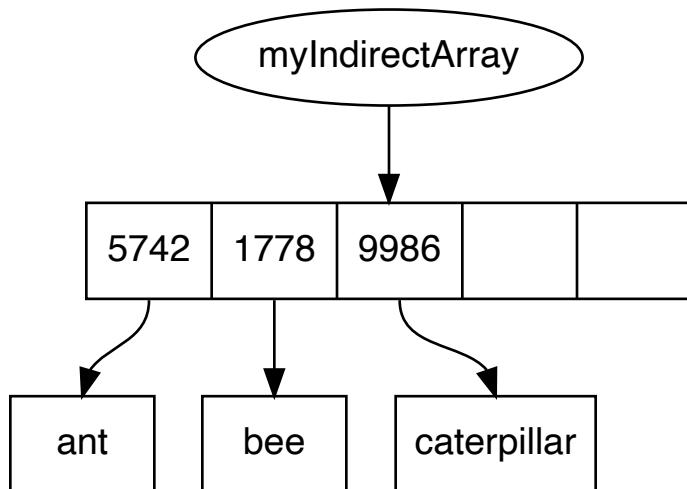
Indirect Arrays

In any *indirect array*, the array doesn't directly hold the value.

It holds the memory address of the real value.

This lets an array store different types of data, or different length data.

```
[ "ant", "bee", "caterpillar" ]
```



What Does JavaScript Use?

Indirect arrays — since you can store different-length things in them

NOTE JS Arrays: It's Complicated

JavaScript uses indirect arrays, but it's a bit more complicated. Some implementations have specialized or adaptive structures to handle edge cases like sparse arrays.

```
const items = [];
items[3] = "hello";
console.log(items) // [undefined, undefined, undefined, "hello"];
```

If you create a sparse array, JavaScript uses a different data structure that's better suited for this scenario.

Amortized Runtime

What happens when you run out of available spots?

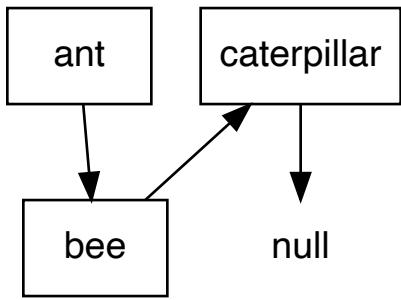
3	7	2		
---	---	---	--	--

3	7	2	1	5
---	---	---	---	---

3	7	2	1	5	9				
---	---	---	---	---	---	--	--	--	--

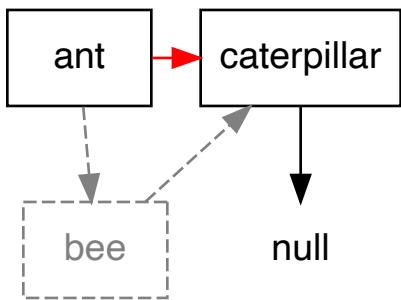
- Everything must be moved when you add an item to an array without any available spots
- Sooner or later adding something to the end of an array will be $O(n)$
- Everything gets moved and new empty spots are allocated
- Mathematically, this averages out to be $O(1)$ — this is amortized runtime

Linked Lists



Items aren't stored in contiguous memory; instead, each item references the next item in the sequence.

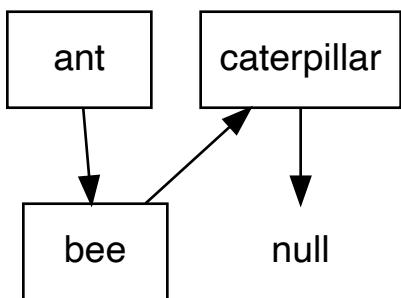
Can rearrange without having to move other items in memory.



This is a lot faster than having to move everything around in a big list.

A Node

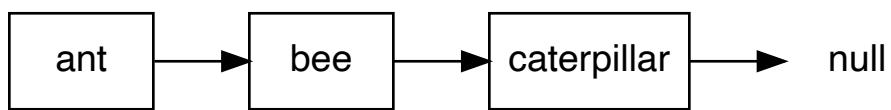
The basic unit of a linked list is a node.



ant, *bee*, and *caterpillar* are nodes.

A basic Node has
two attributes:

val
the information
the node
contains (could



```

be string, int,
instance, etc)

next
reference to
next node (for
last item, this is
null)

```

```

antNode;
// {val: "ant", next: beeNode}

beeNode;
// {val: "bee", next: caterpillarNode}

caterpillarNode;
// {val: "caterpillar", next: null}

```

The Node Class

demo/linkedlist.js

```

/** Node class for item in LL. */

class Node {
  val = null;
  next = null;

  constructor(val) {
    this.val = val;
  }
}

```

```

let antNode = new Node("ant");
let beeNode = new Node("bee");
let catNode = new Node("caterpillar");

antNode.next = beeNode;
beeNode.next = catNode;

```



```

antNode;
// {val: "ant", next: beeNode}

beeNode;
// {val: "bee", next: catNode}

catNode;
// {val: "caterpillar", next: null}

```

Smarter Node Class

Some people make a *Node* class which accepts optional *next* argument:

```

class Node {
  constructor(val, next=null) {
    this.val = val;
    this.next = next;
  }
}

```

Then you can add a chain of nodes:

```

let antNode = new Node("ant",
  new Node("bee",
    new Node("caterpillar")));

```

This ends up exactly the same, but can be harder to read at first.

LinkedList Class

A Linked List is just a bunch of nodes linked sequentially.

The only attribute it must have is a reference to its first node, called the *head*.

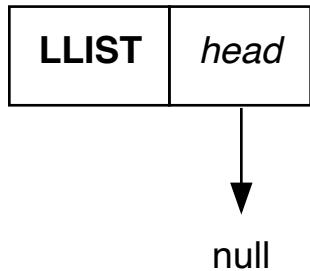
Since the list starts empty, the head is initially *null*.

```
class LinkedList {  
    head = null;  
}
```

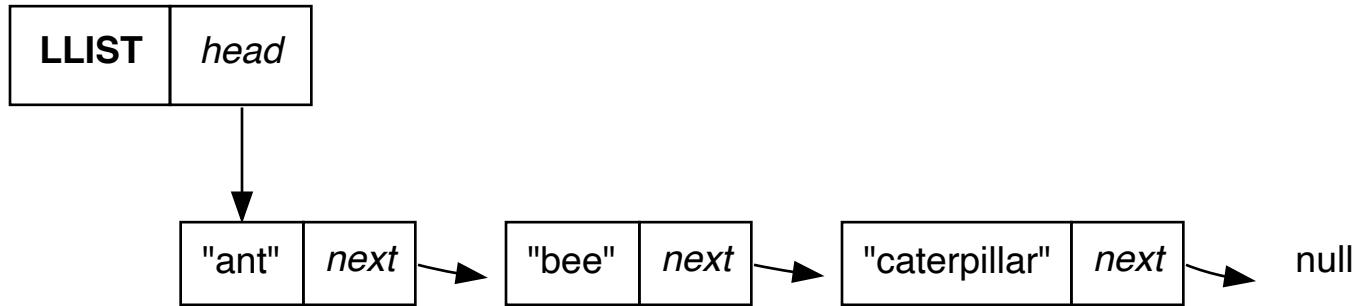
```
let insects = new LinkedList();
```

In Pictures...

An empty Linked List:



A Linked List with nodes in it:



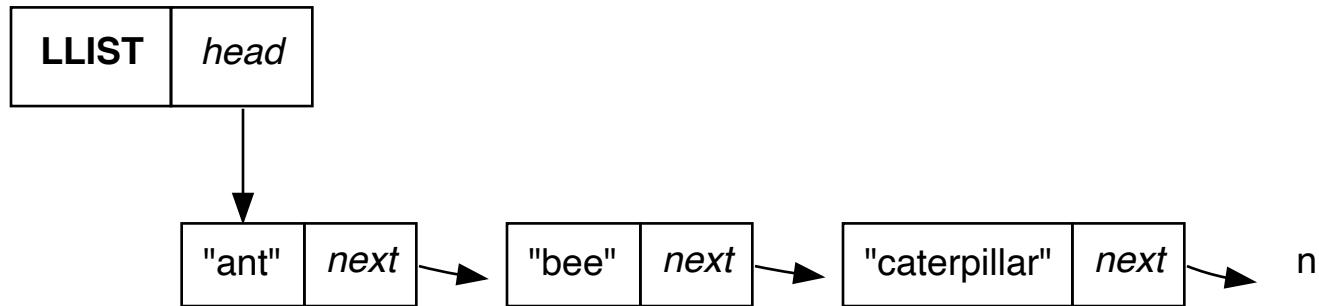
Things you might want to do

- Print each node
- Find a node by its data
- Append to end
- Insert at specific position
- Remove a node

Traversing

Assumption: we've already built list, leaving the actual construction for later.

We're just going to traverse the list and print it.



`demo/linkedlist.js`

```
/** print(): traverse & console.log each item. */

print() {
  let current = this.head;

  while (current !== null) {
    console.log(current.val);
    current = current.next;
  }
}
```

Searching

Like printing—but stop searching once we find what we're looking for.

`demo/linkedlist.js`

```
/** find(val): is val in list? */

find(val) {
  let current = this.head;

  while (current !== null) {
    if (current.val === val) return true;

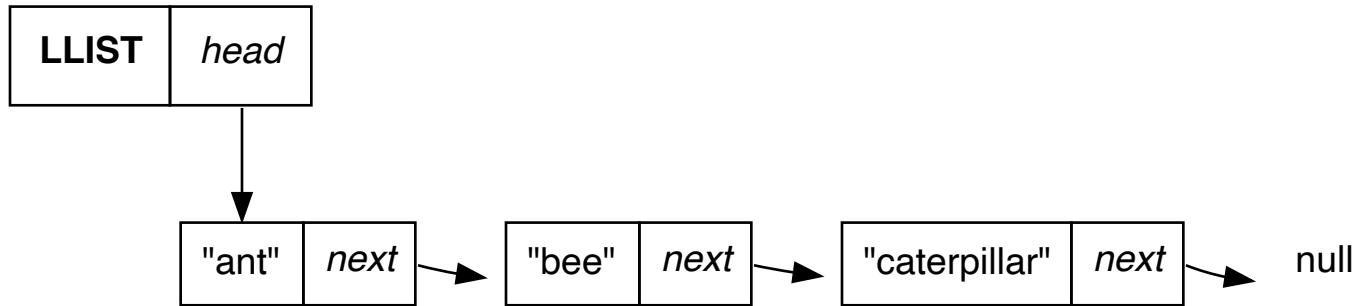
    current = current.next;
  }

  return false;
}
```

Appending/Removing Nodes

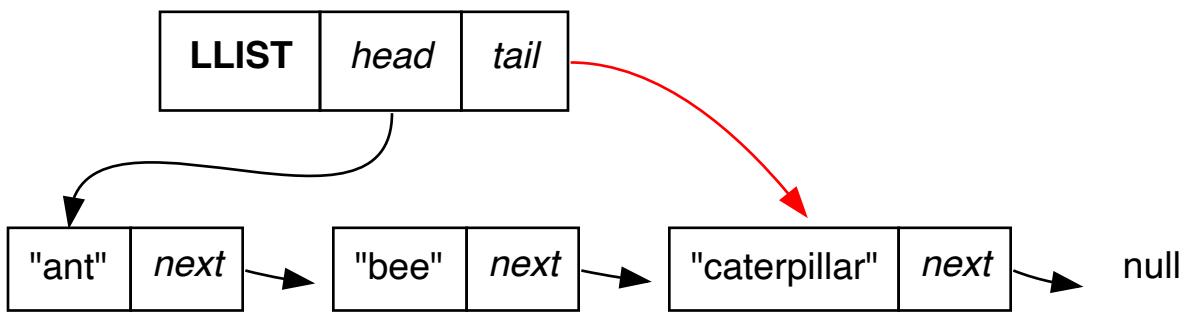
Append a Node

Q: How do we append a node to the end of a linked list?



A: Walk to the end and add it there.

(But wouldn't it be faster to append if we “know” the end?)

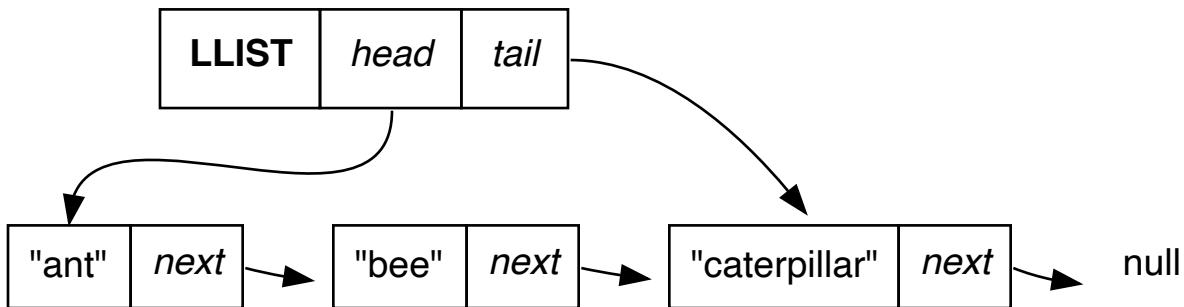


This way, appending is always $O(1)$

This becomes easier if we add a *tail* attribute onto our list. This way, we don't have to traverse the list every time we add a node.

We can do this with just *head*, but why if we can add a *tail*?

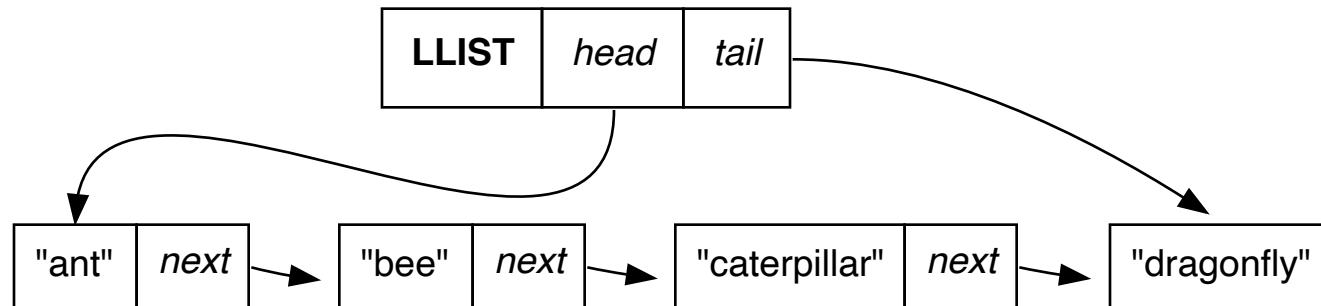
```
class LinkedList {
  constructor() {
    this.head = null;
    this.tail = null;
  }
}
```



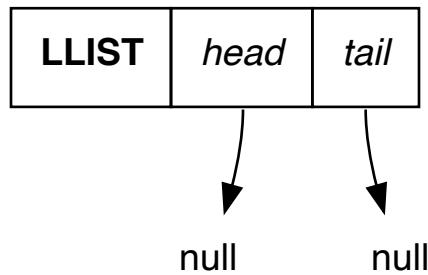
What do we need to do to add “dragonfly”?

- make new node *dragonfly*
- make *caterpillar.next* a reference to *dragonfly*
- make *list.tail* a reference to *dragonfly*

Success!



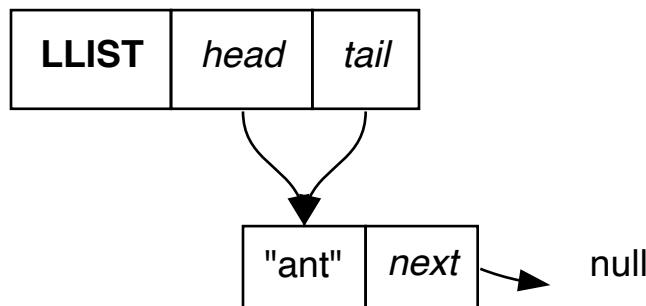
Don't forget to handle case of an empty list!



What do we need to do to add “ant”?

- make new node *ant*
- make *list.head* a reference to *ant*
- make *list.tail* a reference to *ant*

Success!



```
/** push(val): add node w/val to end of list. */

push(val) {
  let newNode = new Node(val);

  if (this.head === null) this.head = newNode;

  if (this.tail !== null) this.tail.next = newNode;

  this.tail = newNode;
}
```

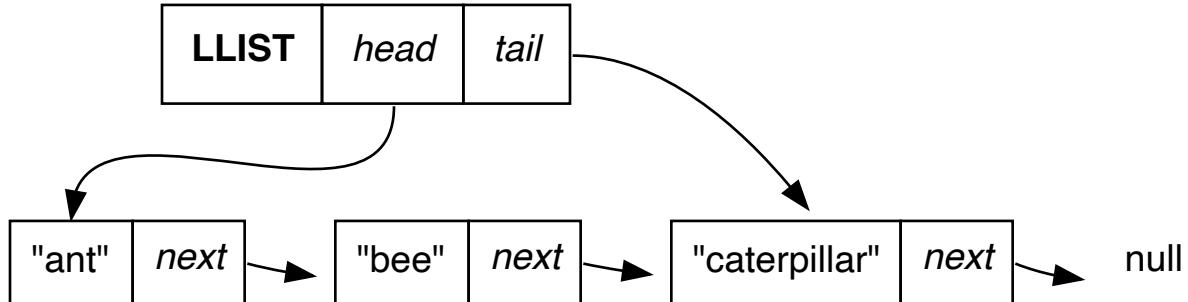
```
let insects = new LinkedList();

insects.push("ant");
insects.push("bee");
insects.push("caterpillar");
```

Remove a Node (by value)

What would you need to change to remove:

- the head (“ant”)
- the middle (“bee”)
- the tail (“caterpillar”)



All we are doing to “remove” a node from the list is redirecting the reference (or *next*) of a node to the one after the node we’re looking for.

There are many tricky ways of doing this.

We’re going to rely on a “daisy-chaining” effect and the fact that any given node’s *next* is just a node, which has its own *val* and *next*.

The code is a bit complex, since we need to handle:

- removing only item in linked list
 - Don’t forget to update head *and* tail to *null*
- removing first item
 - Don’t forget to update the head!
- removing an item in the middle

- removing the last item
 - Don’t forget to update the tail!

Runtime of Linked Lists

- | | |
|------------------------------------|--|
| ◦ Going to “next” item | ◦ Adding to start |
| ◦ $O(1)$ | ◦ $O(1)$ |
| ◦ Going to item by arbitrary index | ◦ Appending to end |
| ◦ $O(n)$ | ◦ $O(1)$ if know tail; $O(n)$ if don’t |
| ◦ Searching for value | ◦ Deleting at start |
| ◦ $O(n)$ | ◦ $O(1)$ |
| ◦ General insertion or deletion | |
| ◦ $O(n)$ | |

How do these compare to arrays?

Code Implementation

Can write with classic OO:

```
class Node { /* ... */ }

class LinkedList {
  constructor() {
    this.head = null;
    this.tail = null;
  }

  find(val) { /* ... */ }
}

let antNode = new Node("ant");
let insects = new LinkedList();

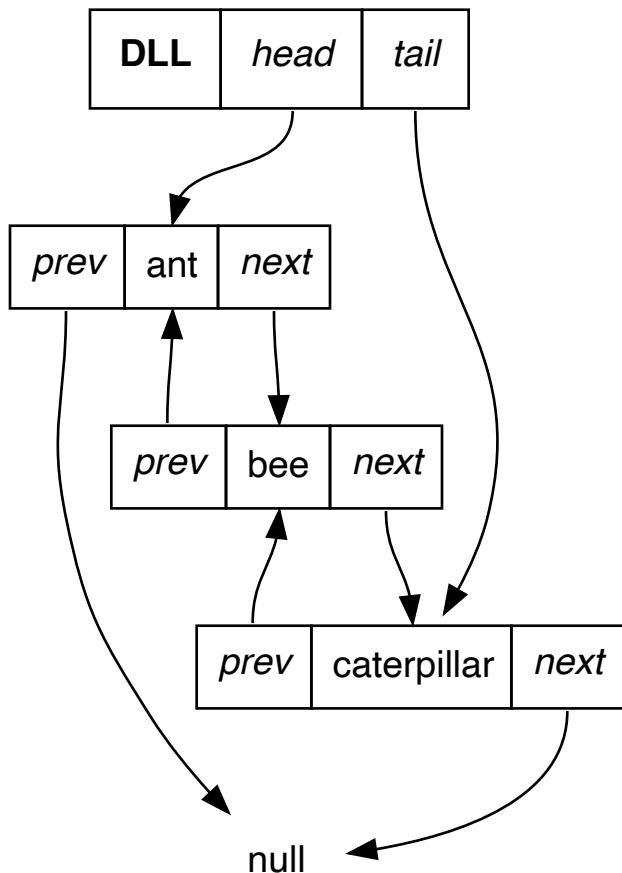
insects.find("ant");
```

NOTE Other Possibilities, too!

Less commonly, you may see implementations that use arrays or tuples to hold nodes, such that the linked list is series of nested arrays or tuples. These tend to be more common in languages without OO, and tend to be more complex to visualize or understand.

Doubly-Linked Lists

Sometimes, linked lists have *next* and a *prev* (the “previous node”)



n.b. While doubly-linked lists are relatively common and useful in actual programming, most interview questions are asking about a singly-linked list.

Resources

What's a Linked List, Anyway? [Base CS] <<https://medium.com/basecs/whats-a-linked-list-anyway-part-1-d8b7e6508b9d>>



Stacks and Queues

Goals

- Describe a queue data structure
- Describe a stack data structure
- Compare and contrast stacks / queues
- Implement stacks and queues in JavaScript

Lists ADT

Remember: an *abstract data type* defines requirements.

ADT for list:

- Keep multiple items
- Can insert or delete items at any position
- Can contain duplicates
- Preserves order of items

Where's the Bug?

movieTicketSales.js

```
// list, in order, of people who want tickets
let ticketBuyers = ["Elie", "Alissa", "Matt", "Michael"];

// ... lots of code

// sell tickets, in order
while (ticketBuyers.length) {
  buyer = ticketBuyers.pop();
  purchase(buyer);
}
```

- Is it right to sell tickets out of order?
- Of course: it's hard to see this bug 500 lines later

What's the Performance Problem?

movieTicketSales.js

```
// list, in order, of people who want tickets
let ticketBuyers = ["Elie", "Alissa", "Matt", "Michael"];

// sell tickets, in order
while (ticketBuyers.length) {
  buyer = ticketBuyers.shift();
  purchase(buyer);
}
```

- It's $O(n)$ to remove from start of array
 - Given that we're removing from the start, a LL would be better
- Of course: it's hard to know *how* a general list will be used

Constraints Are Useful

In both cases, we only need *some* of the requirements of the List ADT

- add new item (ticket buyer) to end
- remove first item (ticket buyer) from start

Knowing this, we could pick a better data structure!

If done well, we could prevent misuse (like buying out of order)

Let's meet two new *ADTs* for collections

Queues

Add at end, remove from beginning

Like a List, Except...

- Items are *only* added to a queue by **enqueueing** them at the *back*
- Items are *only* removed from a queue by **dequeueing** them at the *front*
- Thus, newer items are near back of queue, older items are near front
- **FIFO** for “First-in, first-out”

Typical methods

enqueue(item)

Add to end

dequeue()

Remove & return first item

peek()

Return first item, but don't remove

isEmpty()

Are there items in the queue?

We can't introduce a bug like we had with our ticket buying example, there are no methods for adding to the beginning or removing from the end!

Sometimes there are other common methods, like *.length()*

Sometimes *enqueue* and *dequeue* are called *push* and *pop*

Implementation

What's a good implementation for queues?

- Arrays?
- Linked Lists?
- Doubly Linked List?
- Objects?
- *Array*: no, dequeuing would be $O(n)$
- *Linked List*: yes, both enqueue & dequeue are $O(1)$ (*head is top*)
- *Doubly Linked List*: yes, both enqueue & dequeue are $O(1)$
- *Object*: no, dequeuing is $O(n)$ (*have to scan whole obj to find low key*)

Stacks

- A → “I want to order pizza for our party!”
 - B → In order to do that, I call the pizza place
 - C → They ask me how many I want
 - D → I put them on hold to ask my boss the budget
 - E → She gives amount in CAD, but pizza place takes USD
 - F → I look up USD→CAD conversion rates in my web browser
 - Now I can convert budget to CAD
 - Now I can tell pizza place my budget
 - ...

Like function calls — you return to “previous state” when you pop top task

Like a List, Except...

- Items are *only* added to a stack by **pushing** them onto the *top*

- Items are *only* removed from a stack by **popping** them off the *top*
- Thus, newer items are near top of stack, older items are near bottom
- **LIFO** for *Last-in, first-out*
- Examples: the function call stack, most laundry hampers

Typical methods

push(item)

Add to “top” of stack

pop()

Remove & return top item

peek()

Return (but don’t remove) top item

isEmpty()

Are there items in the stack?

Implementation

What’s a good implementation for stacks?

- Arrays?
- Linked Lists?
- Doubly Linked List?
- Objects?
- *Array*: **yes**, both push & pop are $O(1)$
- *Linked List*: **yes**, both push & pop are $O(1)$
 - Note: this requires some clever thinking w/r/t which end would be the top of the stack.
- *Doubly Linked List*: **yes**, both push & pop are $O(1)$
- *Object*: **no**, popping is $O(n)$ (*have to scan whole obj to find high key*)

Deques

An *ADT* for a “double-ended queue” – push, pop, shift & unshift

Less common than stack or queue

Use Case

A ticket buying application:

- Get in queue to buy ticket: added to end

- Buy ticket: removed from front
- VIP customers:
 - There are VIP customers and they don't go to end of line to buy tickets
 - VIP customers should be helped next and get added to front of the line

Typical Methods

Method names vary across implementations, but one set:

<i>appendleft()</i>	<i>peekleft()</i>
Add to beginning	Return (don't remove) beginning
<i>appendright()</i>	<i>peekright()</i>
Add to end	Return (don't remove) end
<i>popleft()</i>	<i>isEmpty()</i>
Remove & return from beginning	Are there items in the deque?
<i>popright()</i>	
Remove & return from end	

Implementation

What's a good implementation for deques?

- Arrays?
- Linked Lists?
- Doubly Linked List?
- Objects?
- *Array*: no, *appendleft* & *popleft* would be $O(n)$
- *Linked List*: no, *popright* would be $O(n)$
- *Doubly Linked List*: yes — everything is $O(1)$
- *Object*: no, *popleft* & *popright* would be $O(n)$

Priority Queue

An *ADT* for a collection:

- Add item (with priority)
- Remove highest-priority item

Typical Methods

add(pri, item)
Add item to queue

poll()
Remove & return top-priority item

peek()
Return (don't remove) top-priority item

isEmpty()
Are there items in queue?

Implementation

What's a good implementation for priority queues?

- Arrays?
- Linked Lists?
- Doubly Linked List?

Consider with two strategies:

- Keep unsorted, add to end, find top priority on poll
- Keep sorted, add at right place, top priority is first

Keep unsorted, add to end, find top priority on poll:

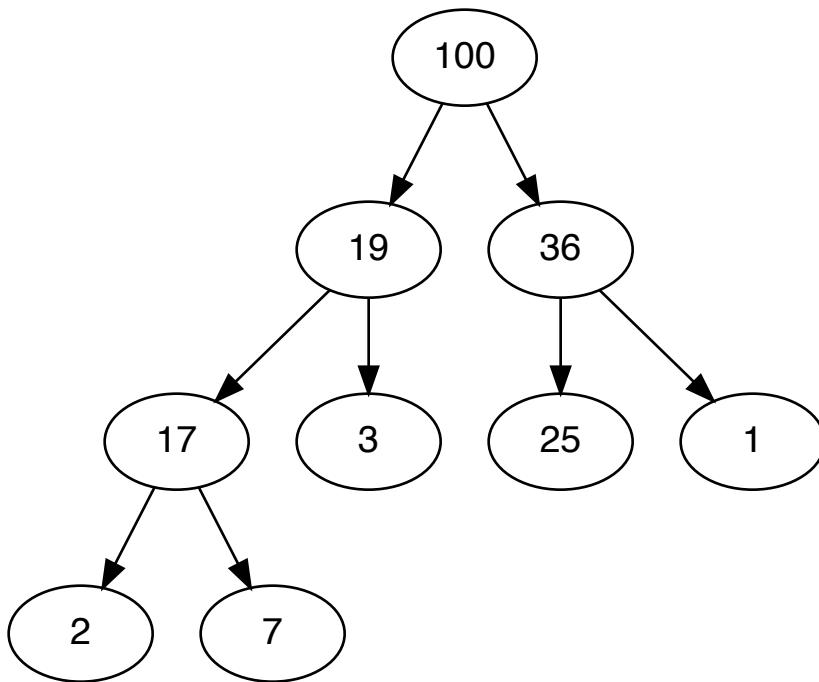
- *Array*: no, peek & poll would be $O(n)$
- *Linked List*: no, peek & poll would be $O(n)$
- *Doubly Linked List*: no, peek & poll would be $O(n)$

Keep sorted, add at right place, top priority is first:

- *Array*: no, add & poll would be $O(n)$
- *Linked List*: no, add would be $O(n)$
- *Doubly Linked List*: no, add would be $O(n)$

Heaps

Data structure optimized for priority queues: *heap*



Building Stacks/Queues

How should you build stacks and queues?

We want to use arrays and linked lists:

- But with the correct API (*eg, push/pop for stack*)
- Without having to mix the *interface* with the *implementation*

Stack: Delegation

- Arrays are a great implementation for a stack
- Create a *Stack* class that itself uses an array

```

class Stack {
  constructor() { this._array = []; }

  push(item) { this._array.push(item); }
  pop() { return this._array.pop(); }
  // ... also should have peek & isEmpty
}
  
```

- *Users* of this class don't have know that it "delegates" to an array
- Enforces the "right API" and makes it obvious that it is a stack
- A class that just delegates elsewhere is called a *facade*.

IMPORTANT Why `_array` for the name?

As usual, the leading-underscore-name suggests “this is a private implementation detail” and other programmers should know not to use it directly.

Queue: Delegation

- Linked lists are a good implementation for a queue:

```
class Queue {  
    constructor() { this._ll = new LinkedList(); }  
  
    enqueue(item) { this._ll.push(item); }  
    dequeue() { return this._ll.shift(); }  
    // ...  
}
```

- Sometimes, you might “cheat” and use an array (*which is often fine*):

```
class Queue {  
    constructor() { this._array = []; }  
    // ...  
}
```

- *Users* of this won’t need to do anything if your implementation changes

NOTE A less-appealing alternate approach

For a stack, you could subclass the built-in JavaScript `Array` class and prevent invalid methods by throwing errors for those not allowed:

```
class Stack extends Array {  
    shift() { throw new Error("Not allowed"); }  
    unshift() { throw new Error("Not allowed"); }  
}
```

Here, this doesn’t *delegate* to a real array — it *is* a real array, but it blocks inappropriate API methods with errors.

There are plenty of non-stack methods that arrays have, so you’d need to block `splice`, `slice`, and *all* other non-stack methods.

Considerations

- This implementation requires that you get everything exactly right, taking care to block every single method that should not be allowed.
- What if you forget to block a method?
- If a new JavaScript Array method is added to the class, you’ll need to remember to update your class to block the newly added method too. This approach could be more work to maintain long-term.

(Of course, this would be the same with a `Queue` class that extended either the JS `Array` class or your `LinkedList` class.)

Or use Array with good names

At the very-very-least, use a built-in-type — but make it clear that its a stack or queue in the name:

```
let buyersQueue = [];  
let processesStack = [];
```

This is not great — if you want to move to a better implementation later, you might have to change code in several places.

Resources

Stacks and Overflows <<https://medium.com/basecs/stacks-and-overflows-dbcf7854dc67>>

To Queue or Not To Queue <<https://medium.com/basecs/to-queue-or-not-to-queue-2653bcde5b04>>

Learning to Love Heaps <<https://medium.com/basecs/learning-to-love-heaps-cef2b273a238>>

Rithm School Lecture on Heaps <<https://rithm-students-assets.s3.us-west-1.amazonaws.com/r19/lectures/dsa-pqueues/handout/index.html>>



Recursion

Having a function call itself

Also: a very powerful programming technique

Also: a popular interview question topic

The Tiniest Review

Functions Calling Functions

```
function a() {  
    console.log("hello");  
    b();  
    console.log("coding");  
}  
  
function b() {  
    console.log("world");  
    c();  
    console.log("love");  
}  
  
function c() {  
    console.log("i");  
}
```

c() "i" ↓ undef

b() "world" ↑ "love" ↓ undef

a() "hello" ↑ "coding" ↓ undef

→ “hello world i love coding”

Remember, when you call a function, you “freeze” where you are until that function returns, and then continue where you left off.

So *a* prints `hello`, calls *b* which prints `world`, calls *c* which prints `i` and returns back to *b* which then prints `love` which then returns back to *a* which prints `coding`.

Loops and Recursion

Any loop can be written instead with recursion

Any recursion can be written instead with a loop

... but often, one way is easier for a problem

Count to 3

Using a *while* loop:

Using recursion:

```

function count() {
  let n = 1;

  while (n <= 3) {
    console.log(n);
    n += 1;
  }
}

count();

```

```

function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
}

count();

```

Call Frames / Stack

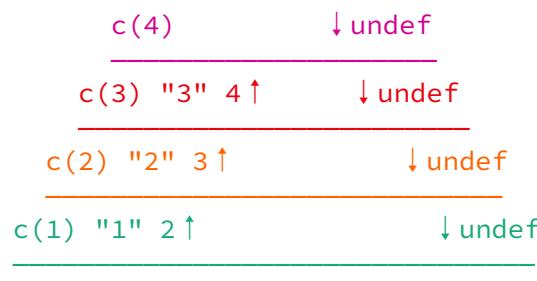
```

function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
}

count();

```



More Counting

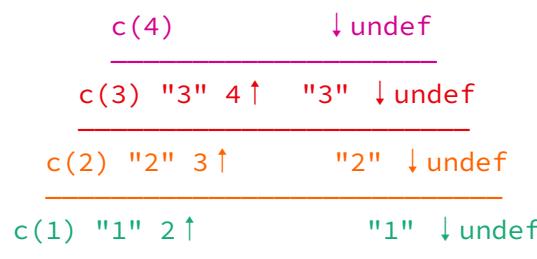
```

function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
  console.log(n);
}

count();

```



Loops versus Recursion

Using a *while* loop:

Using recursion:

```

function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
}

count();

```

```

function count() {
  let n = 1;

  while (n <= 3) {
    console.log(n);
    n += 1;
  }
}

count();

```

Which do you prefer?

Requirements

Base Case

```

function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
}

```

- Every recursive function needs a *base case*
 - How do we know when we’re done?

Often a base case is a “degenerate case”.

- concat([1, 2, 3]) →
- “1” + concat([2, 3]) →
- “1” + “2” + concat([3]) →
- “1” + “2” + “3” + concat([]) ← **degenerate: empty array**

Degenerate Cases

A “degenerate case” is one that is so reduced that it’s fundamentally different from the others and would need to be treated differently.

Consider counting up to 3 recursively:

```

function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
}

```

Here, our base case is “when we hit 3, don’t keep recursing”. This is a base case, but it’s not “degenerate” — we *could* keep counting up after 3; there’s nothing preventing us from doing so besides our goal to stop.

Compare this with finding the length of a list recursively:

```

function lenlist(nums) {
  if (nums[0] === undefined) return 0

  return 1 + lenlist(nums.slice(1));
}

```

Here, our base case is “the length of an empty list is 0, so return that and don’t recurse”. This base is “degenerate” — there’s no possible way for us to find the length of a list with -1 items in it! It wouldn’t even be possible for us to keep recursing; this base case is a hard limit on what’s possible.

Not all recursive problems have a degenerate base case, but thinking about if one is possible is often helpful in figuring what your base case is and how the recursion should work.

No Base Case

```

function count(n=1) {
  console.log(n);
  count(n + 1);
}

count();

```

Stack Overflow!

Explicit vs. Hidden Base Cases

```

function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
}

```

```

function count(n=1) {
  if (n <= 3) {
    console.log(n);
    count(n + 1);
  }
}

```

Which do you prefer?

Progress

```

function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
}

```

Returning Data

Finding Sum of List

“Return sum of list using recursion”

- What’s our base case?
 - An empty list has sum = 0!

```
function sum(nums) {  
    if (nums.length === 0) return 0;  
  
    return nums[0] + sum(nums.slice(1));  
}  
  
sum([1, 2, 4, 5]);
```

The diagram illustrates the recursive steps for the function call `sum([1, 2, 4, 5])`. It shows four rows of calculations:

- Row 1: $s([])$ $\downarrow 0$
- Row 2: $s([5])$ $[5] \uparrow$ $\downarrow 5$
- Row 3: $s([4, 5])$ $[5] \uparrow$ $\downarrow 9$
- Row 4: $s([2, 4, 5])$ $[4, 5] \uparrow$ $\downarrow 11$
- Row 5: $s([1, 2, 4, 5])$ $[2, 4, 5] \uparrow$ $\downarrow 12$

Arrows indicate the progression from the base case to the final result.

List Doubler

The Problem

“For every number in array, print the value, doubled”

```
data = [ 1, 2, 3 ] // => 2 4 6
```

```
function doubler(nums) {  
    for (let n of nums) {  
        console.log(n * 2);  
    }  
}
```

The Challenge

- Some items can be lists themselves
- We want to “flatten” them and still print doubled

```
data = [ 1, [2, 3], 4 ] // => 2 4 6 8
```

```

function doubler(nums) {
  for (let n of nums) {
    if (Array.isArray(n)) {
      for (let o of n) console.log(o * 2);
    } else {
      console.log(n * 2);
    }
  }
}

```

Oh No!

Some of *those* items can be lists!

```
data = [ 1, [2, [3], 4], 5 ] // => 2 4 6 8 10
```

```

function doubler(nums) {
  for (let n of nums) {
    if (Array.isArray(n)) {
      for (let o of n) {
        if (Array.isArray(o)) {
          for (let p of o) console.log(p * 2);
        } else {
          console.log(o * 2);
        }
      }
    } else {
      console.log(n * 2);
    }
  }
}

```

Arbitrary Depth with Loop

```

let data = [ 1, [2, [3], 4], 5 ] // => 2 4 6 8 10

function doubler(nums) {
  let stack = nums.reverse();

  while (stack.length > 0) {
    let n = stack.pop();
    if (Array.isArray(n)) {
      // If array, add it to stack, reversed
      for (let inner of n.reverse()) {
        stack.push(inner);
      }
    } else {
      console.log(n * 2);
    }
  }
}

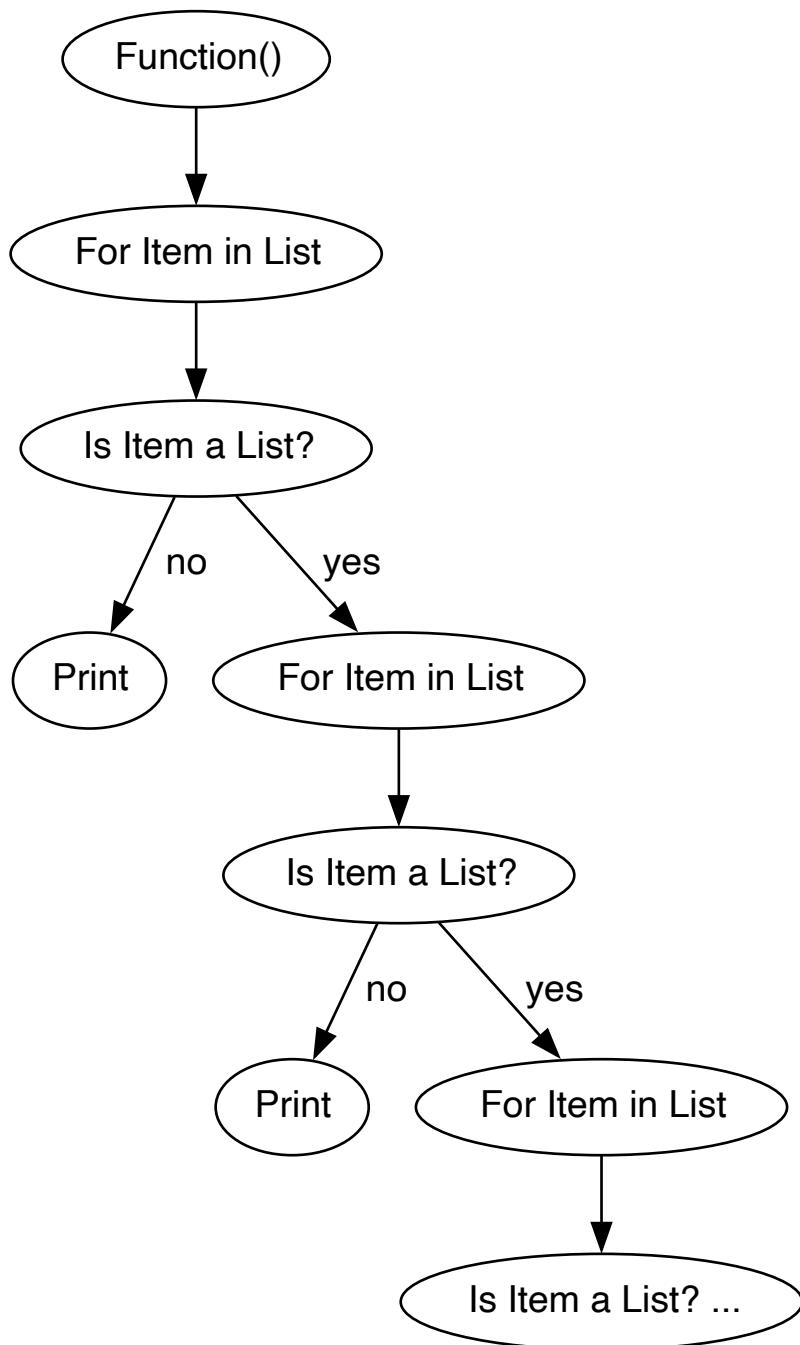
```

It works, but it's pretty hairy!

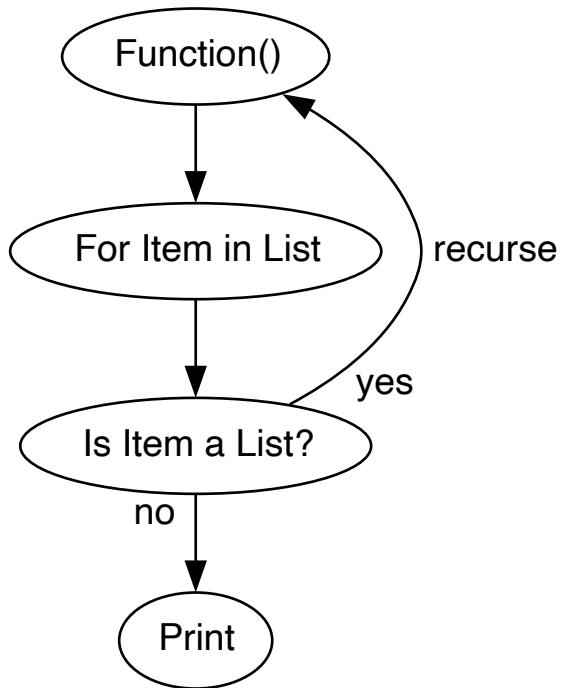
This solution uses a data structure called a “stack”, adding new work to the end and popping them off the end.

This code may be worth study, even though this problem is more easily solved with recursion.

Non-Recursively



Recursively

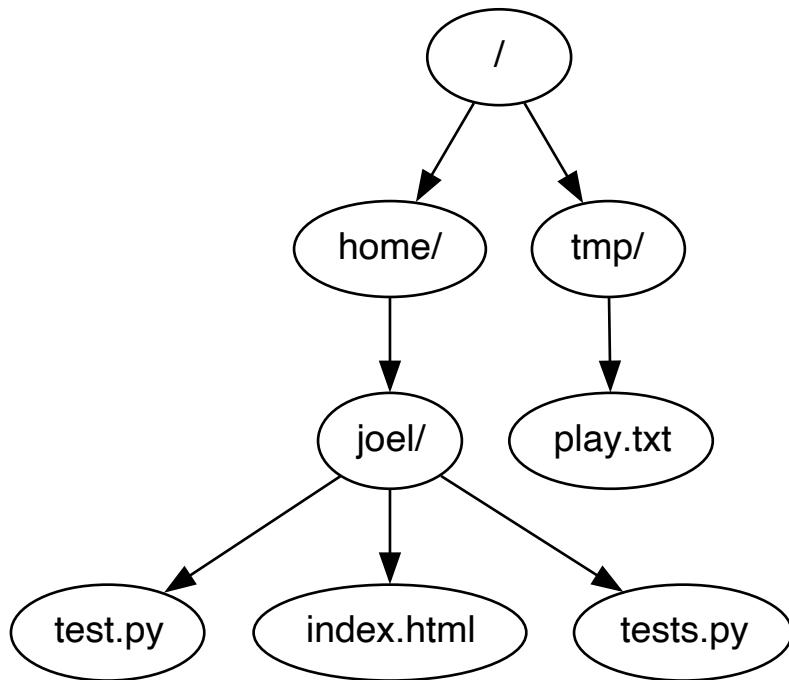


```
data = [ 1, [2, [3], 4], 5 ]  
  
function doubler(nums) {  
  for (let n of nums) {  
    if (Array.isArray(n)) {  
      doubler(n);  
    } else {  
      console.log(n * 2);  
    }  
  }  
}
```

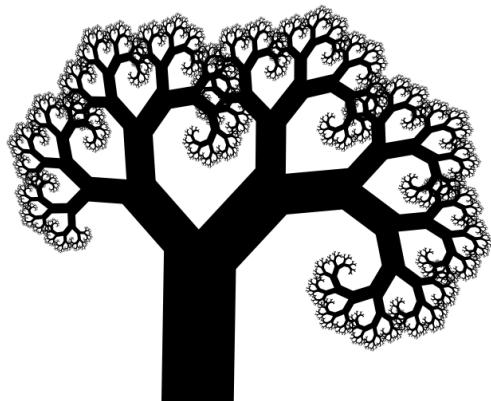
```
d([3]) "6"  
_____ ↓  
d([2,[3],4]) "4" [3]↑ "8"  
_____ ↓  
d([1,[2,[3],4],5]) "2" [2,[3],4]↑ "10" ↓
```

Recognizing Recursion

Filesystems



Fractals



Parsing

$$1 \times (2 + 3 \times (4 + 5 \times 6) + 7)$$

This is a particularly good, hard exercise to give yourself.

Nested Data

```
<html>
  <head>
    <title>Title</title>
  </head>
  <body>
    <h1>Body</h1>
    <ul>
      <li>One</li>
      <li>Two
        <ul>
          <li>Two A</li>
          <li>Two B</li>
        </ul>
      </li>
    </ul>
  </body>
</html>
```

Runtime

What's the runtime?

```
function sum(nums) {
  if (nums.length === 0) return 0;

  return nums[0] + sum(nums.slice(1));
}
```

$O(n^2)$ — we keep making new lists!

It also has $O(n^2)$ runspace — keeping all lists in memory!

Improving Runtime

Often, you can keep track of position in array, rather than slice:

```
function sum(nums, i=0) {
  if (i === nums.length) return 0;

  return nums[i] + sum(nums, i + 1);
}
```

Now runtime and runspace are $O(n)$

Accumulating Output

Given array of numbers, return even numbers

```

function evens(nums, i=0) {
  if (nums.length === i) return [];

  if (nums[i] % 2 === 0) {
    return [nums[i], ...evens(nums, i + 1)];
  }

  return evens(nums, i + 1);
}

```

Back to $O(n^2)$ – making all those lists!

Can solve with “helper recursion”:

```

function evens(nums) {
  let out = [];

  function _evens(nums, i) {
    if (nums.length === i) return;
    if (nums[i] % 2 === 0) out.push(nums[i]);
    _evens(nums, i + 1);
  }

  _evens(nums, 0);
  return out;
}

```

Back to $O(n)$

Accumulators

Often, can also solve with “accumulator”:

```

function evens(nums, out=[], i=0) {
  if (nums.length === i) return out;

  if (nums[i] % 2 === 0) out.push(nums[i]);

  return evens(nums, out, i + 1);
}

```

Back to $O(n)$

NOTE “Tail Call Optimization”

In some languages, this can be “tail-call optimized”, where the language can rewrite a recursive approach into a faster-performing loop approach during compilation or execution. This is an advanced but interesting feature, and you can learn more about it if you’re interested at [Advanced: Tail Call Optimization <http://2ality.com/2015/06/tail-call-optimization.html>](http://2ality.com/2015/06/tail-call-optimization.html)

Resources

How Recursion Works <<https://medium.freecodecamp.org/how-recursion-works-explained-with-flowcharts-and-a-video-de61f40cb7f9>>

Rithm Lecture on Recursion <[./index.html](#)>

Trees & Binary Trees

Trees

Goals

- Introduce terminology
- Create a tree class and methods
- Learn uses for trees

Terminology

node

basic unit

children

nodes directly below a node

descendants

nodes below a node

parent

node that is directly above a node

ancestor

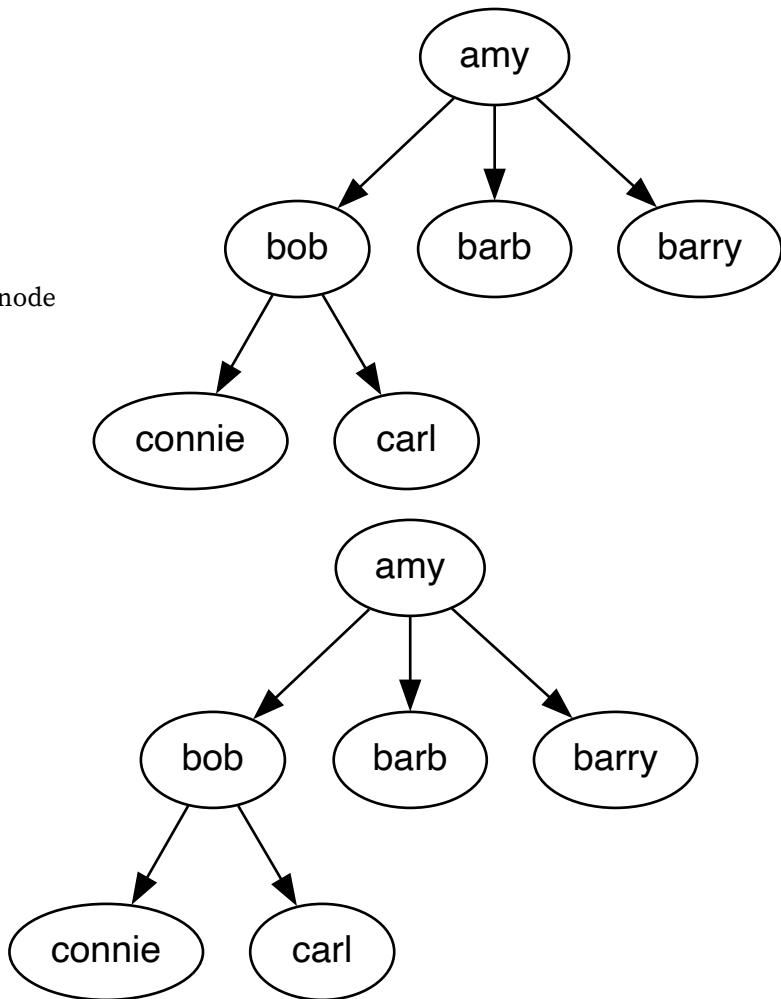
node that is above a node

root node

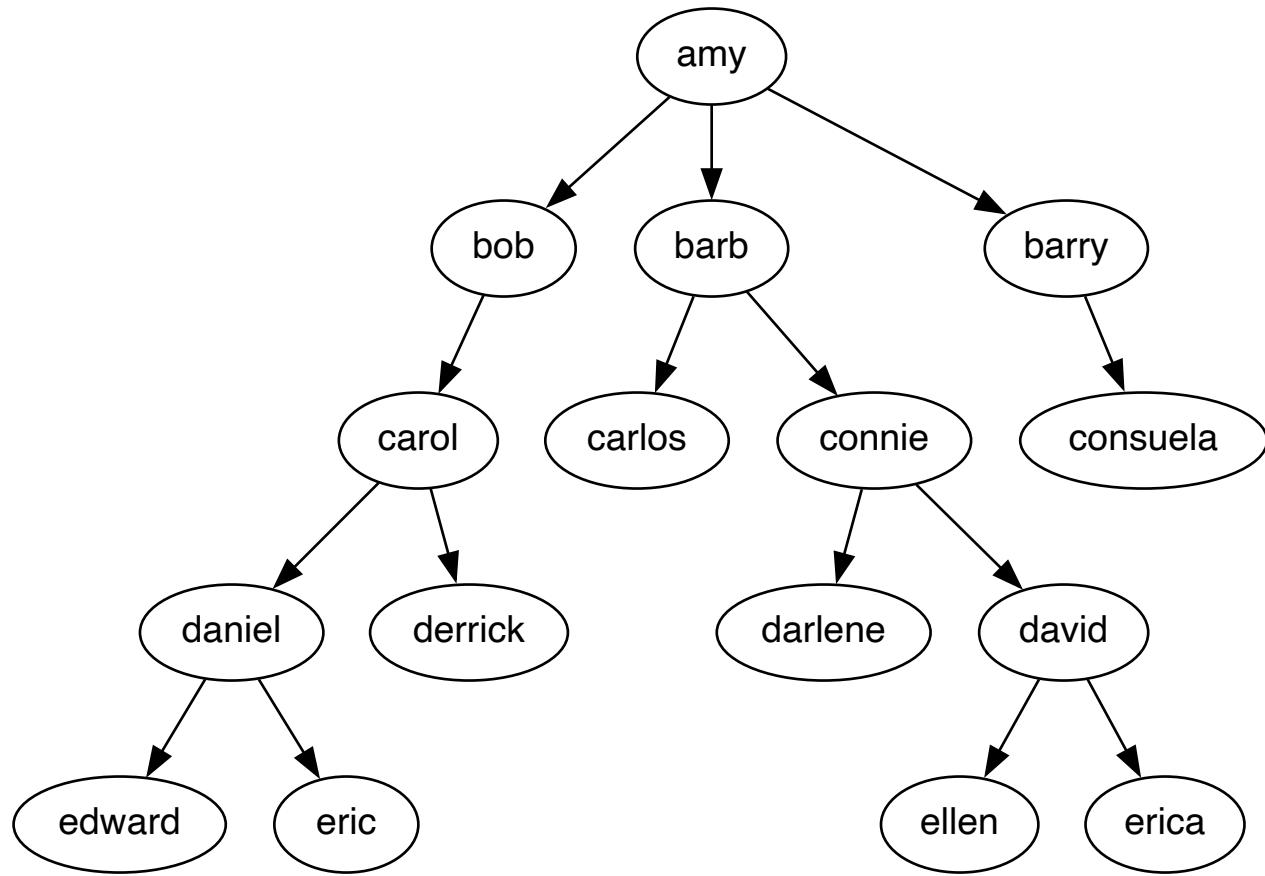
node at the top of tree

leaf node

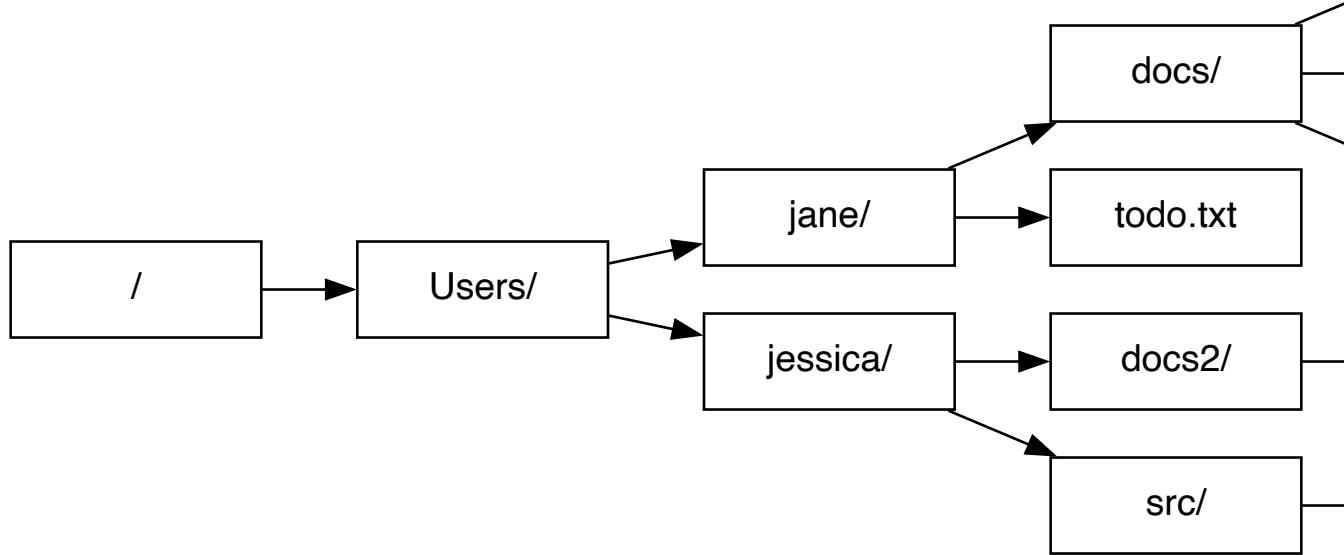
node without any children



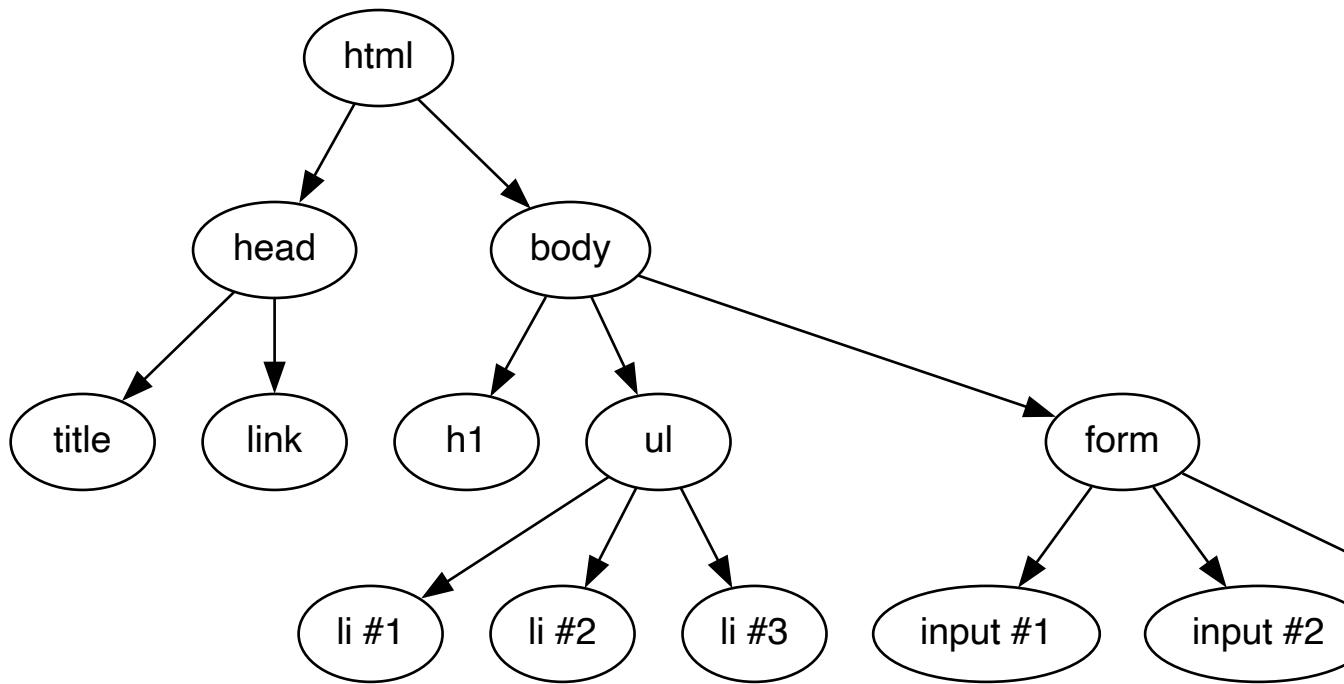
An Org Chart is a Tree



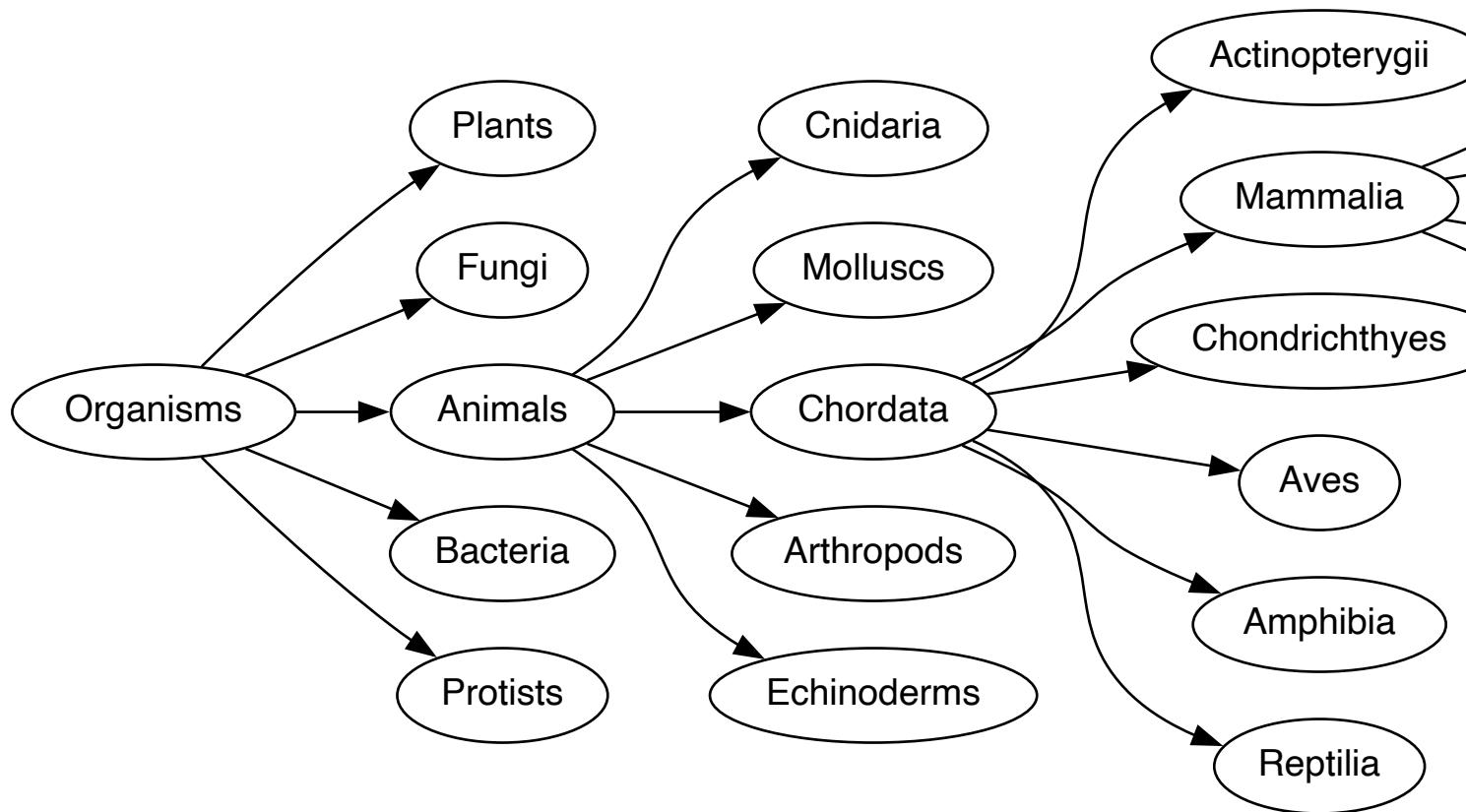
A Filesystem is a Tree



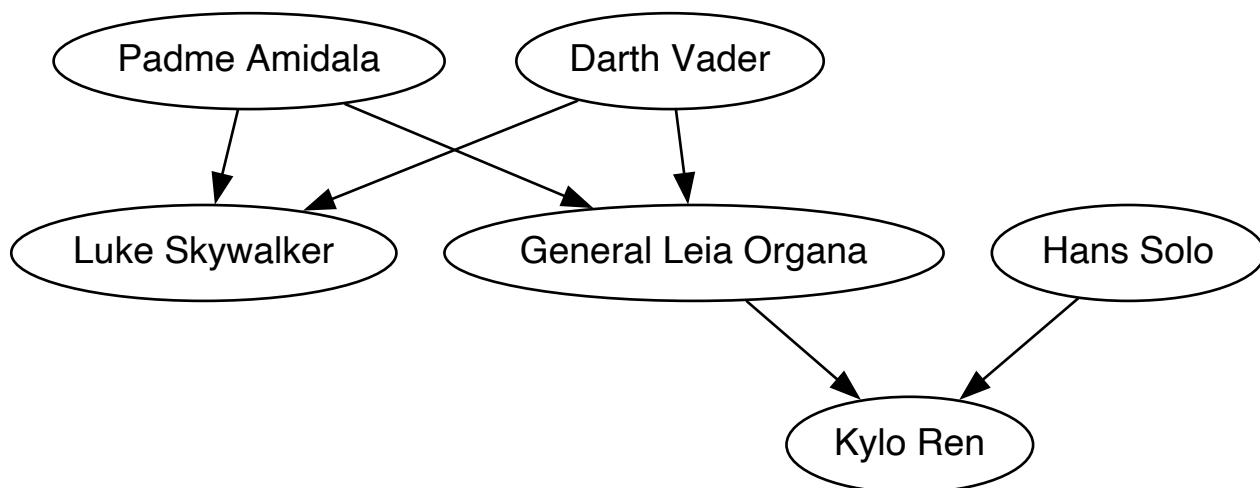
HTML DOM is a Tree



A Taxonomy is a Tree



This Is Not a Tree



- Trees need a root node — we don't have one!
- A node can only have one parent

Binary Trees/Binary Search Trees

These are different—and we'll cover later!

General trees are sometimes called “n-ary” trees, since they can have n (any) number of children.

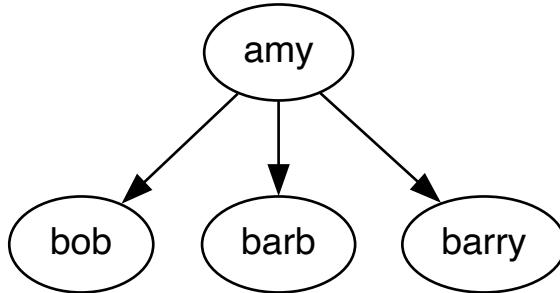
Trees in JavaScript

Node Class

```
class Node {  
  constructor(val, children = []) {  
    this.val = val;  
    this.children = children;  
  }  
}
```

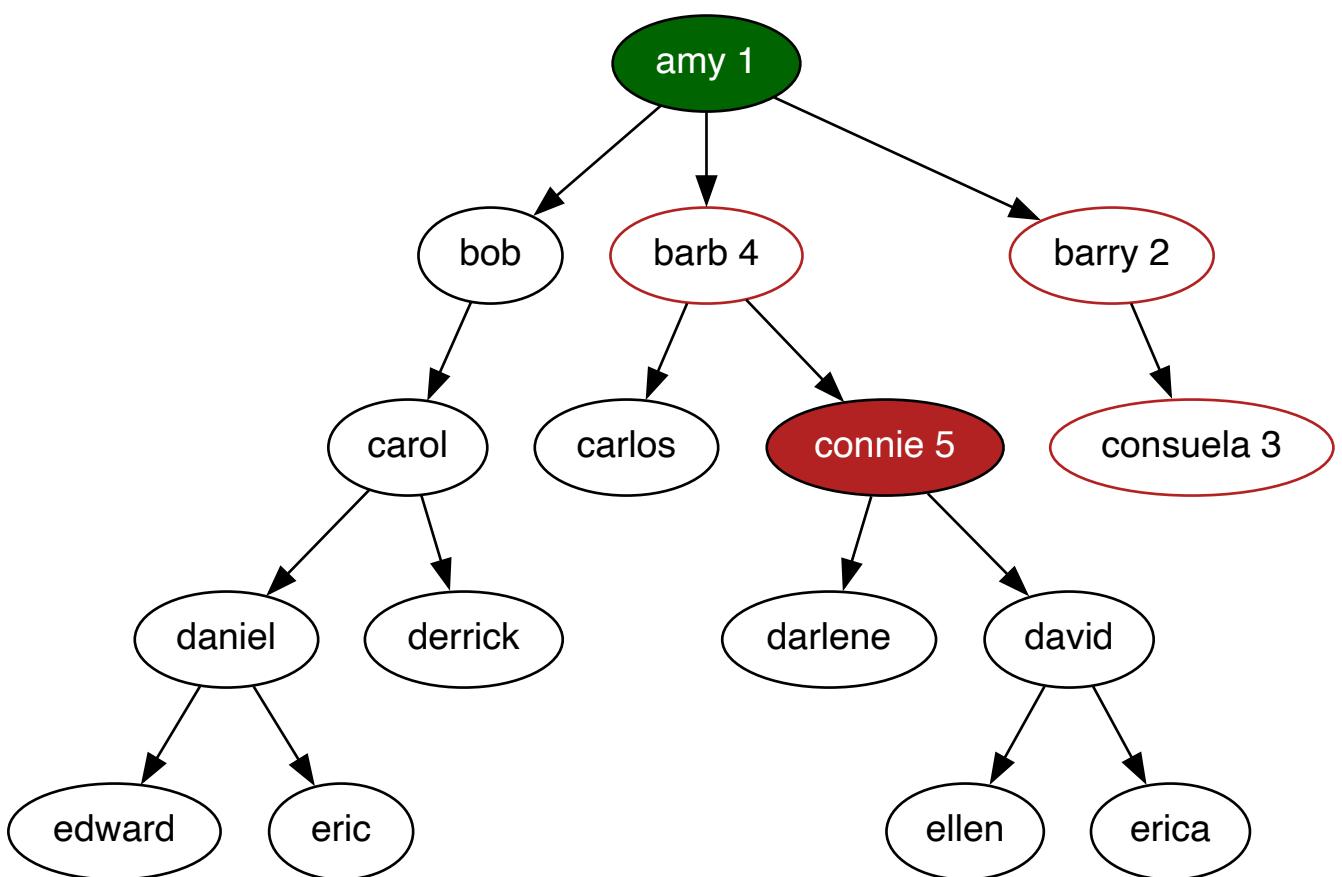
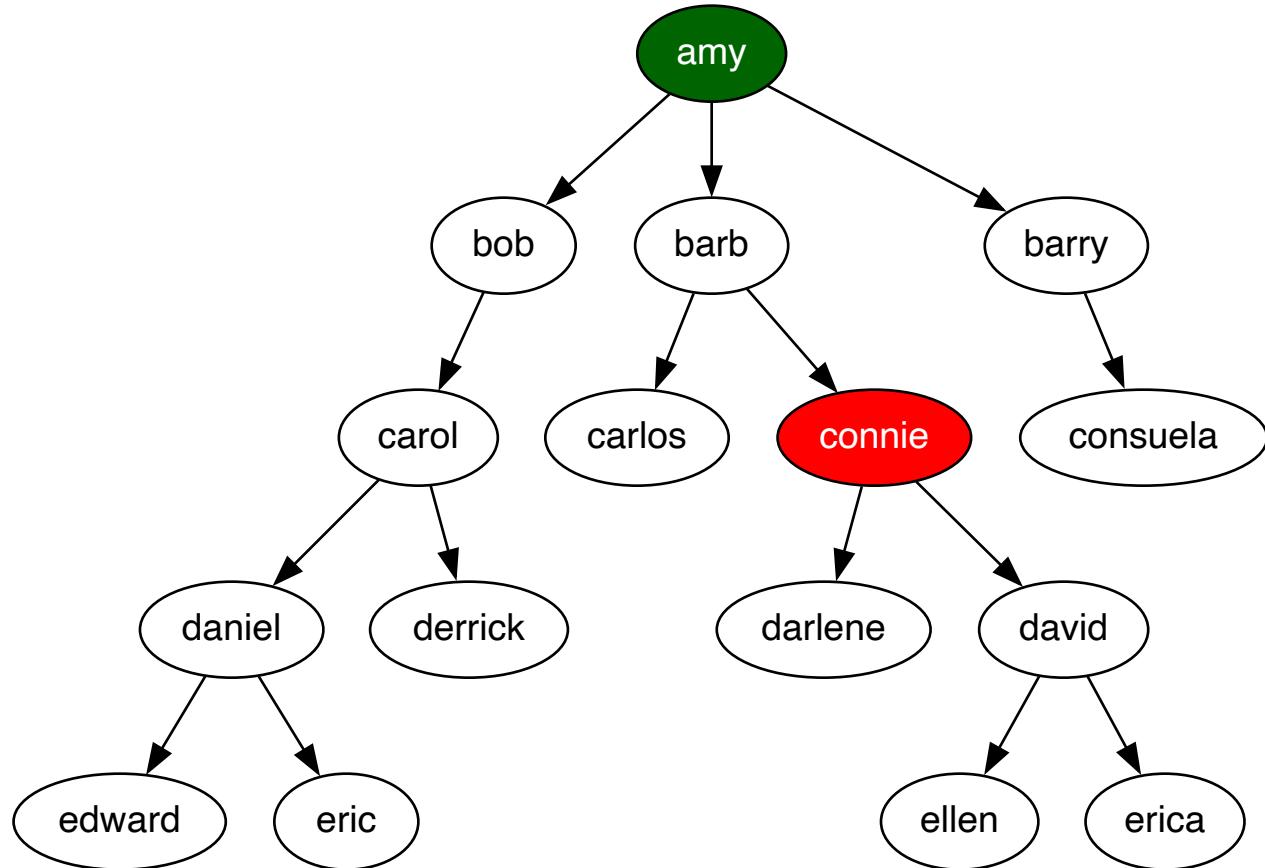
```
let amy = new Node("amy");  
  
amy.children.push(new Node("bob"));  
amy.children.push(new Node("barb"));  
amy.children.push(new Node("barry"));
```

```
let amy = new Node("amy",  
  [new Node("bob"),  
   new Node("barb"),  
   new Node("barry")])
```



Finding a Node

Starting at Amy, find Connie:



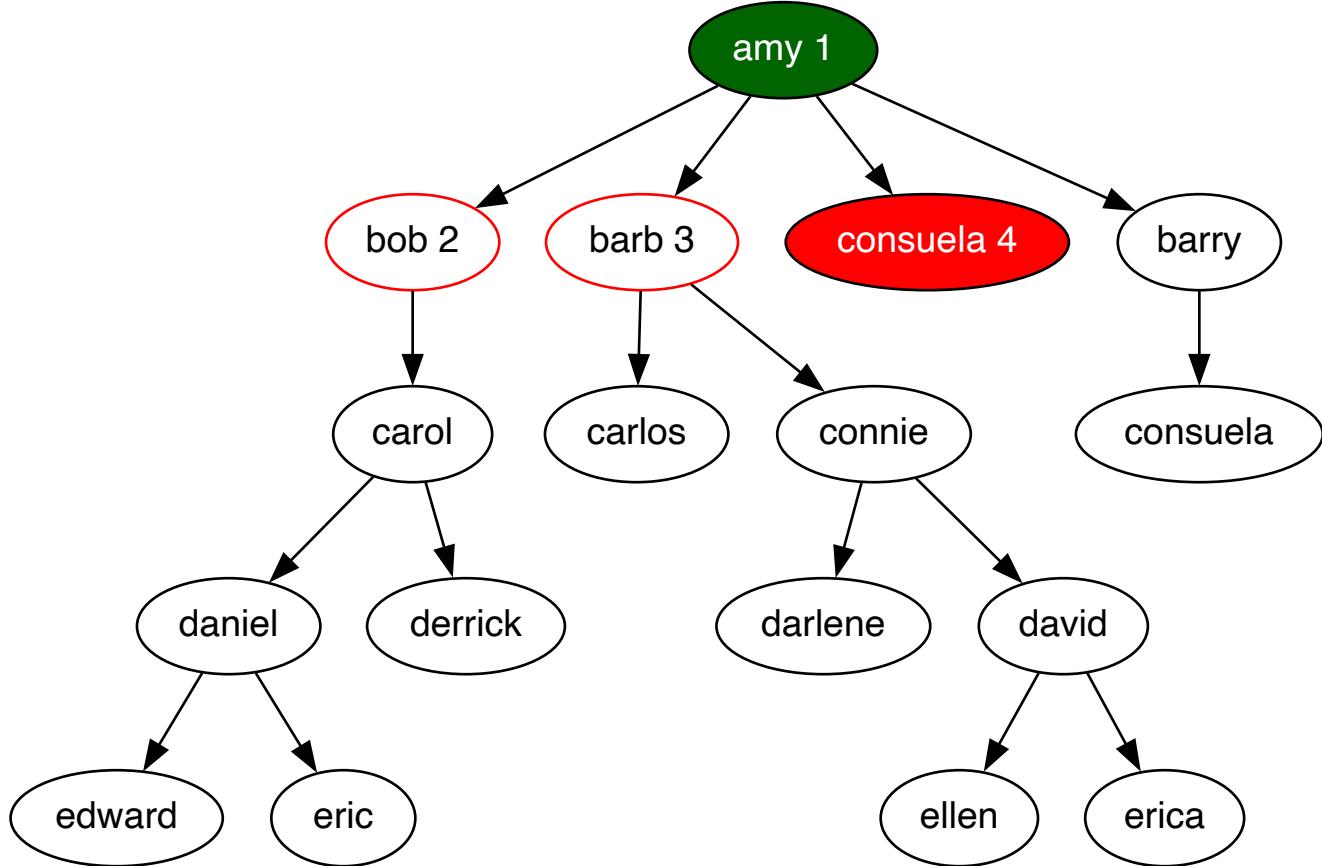
```

amy.find()
  find(v)
    let
      while
        let
          if
            for
          }
        }
      }
    }
  }
}
  
```

Depth First Search (uses stack)

Highest-Ranking Consuela

We hire another Consuela & we want to find highest-ranking one



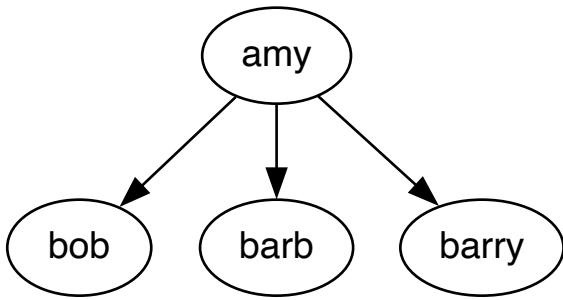
Breadth First Search (uses queue)

Tree Class

```
class Tree {  
  constructor(root) {  
    this.root = root;  
  }  
}
```

```
class Node {  
  constructor(val, children = []) {  
    this.val = val;  
    this.children = children;  
  }  
}
```

```
let org = new Tree(  
  new Node("amy",  
    [new Node("bob"),  
     new Node("barb"),  
     new Node("barry")]))
```



Do You Really Need a Tree Class?

Each node is, itself, a tree

It's useful to have a *Tree* class, though: then, you can find or change the head node

Can delegate to the head node for many operations:

```

class Tree {
  constructor(root) {
    this.root = root;
  }

  /** findInTree: return node in tree w/this val */
  findInTree(val) {
    return this.root.find(val)
  }

  /** findInTreeBFS: return node in tree w/this val */
  findInTreeBFS(val) {
    return this.root.findBFS(val)
  }
}

```

```

class Node {
  // constructor here

  /** find: return node obj w/this val */
  find(val) { }

  /** findBFS: return node obj w/this val */
  findBFS(val) { }
}

```

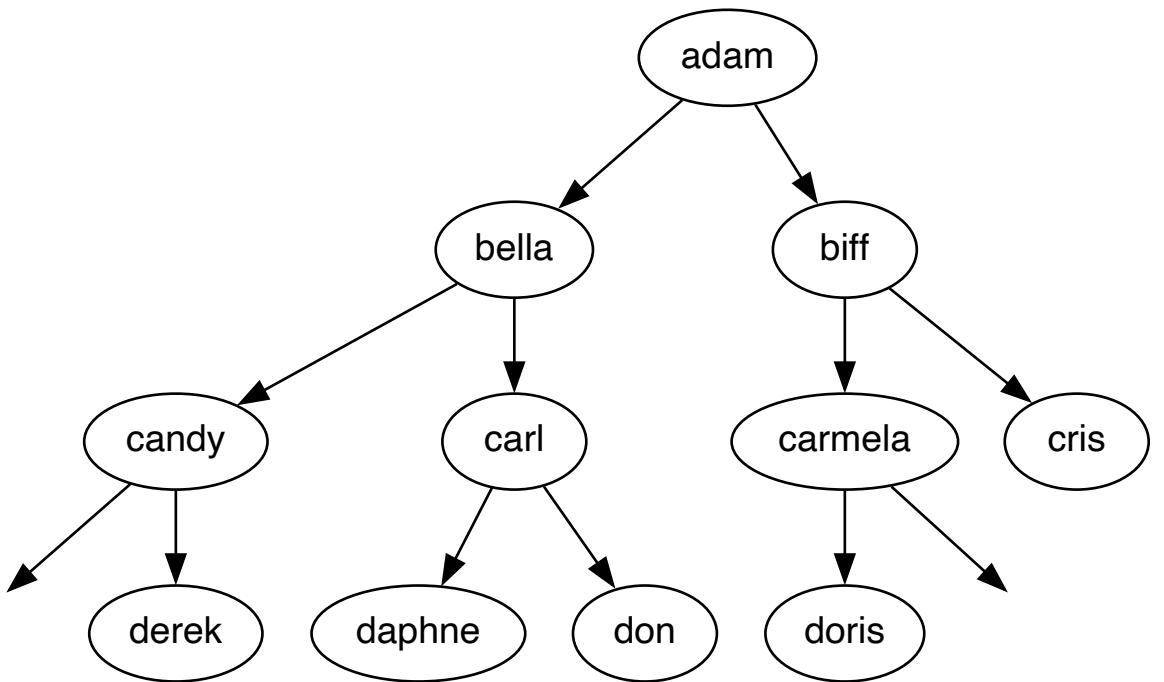
Also

Every linked list is a tree – but not every tree is a linked list

Binary Trees

General n-ary trees have nodes with 0+ children.

Binary tree nodes can have 0, 1, or 2 children.



Binary tree nodes are usually structured with *left* and *right* properties, rather than an array of *children*:

```

class BinNode {
  constructor(val, left=null, right=null) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}
  
```

What Are They Good For?

Sometimes they're used to store data in a normal hierarchy, like a tree.

Often times, they have a “rule” about the arrangement:

- binary search trees
- min/max heap

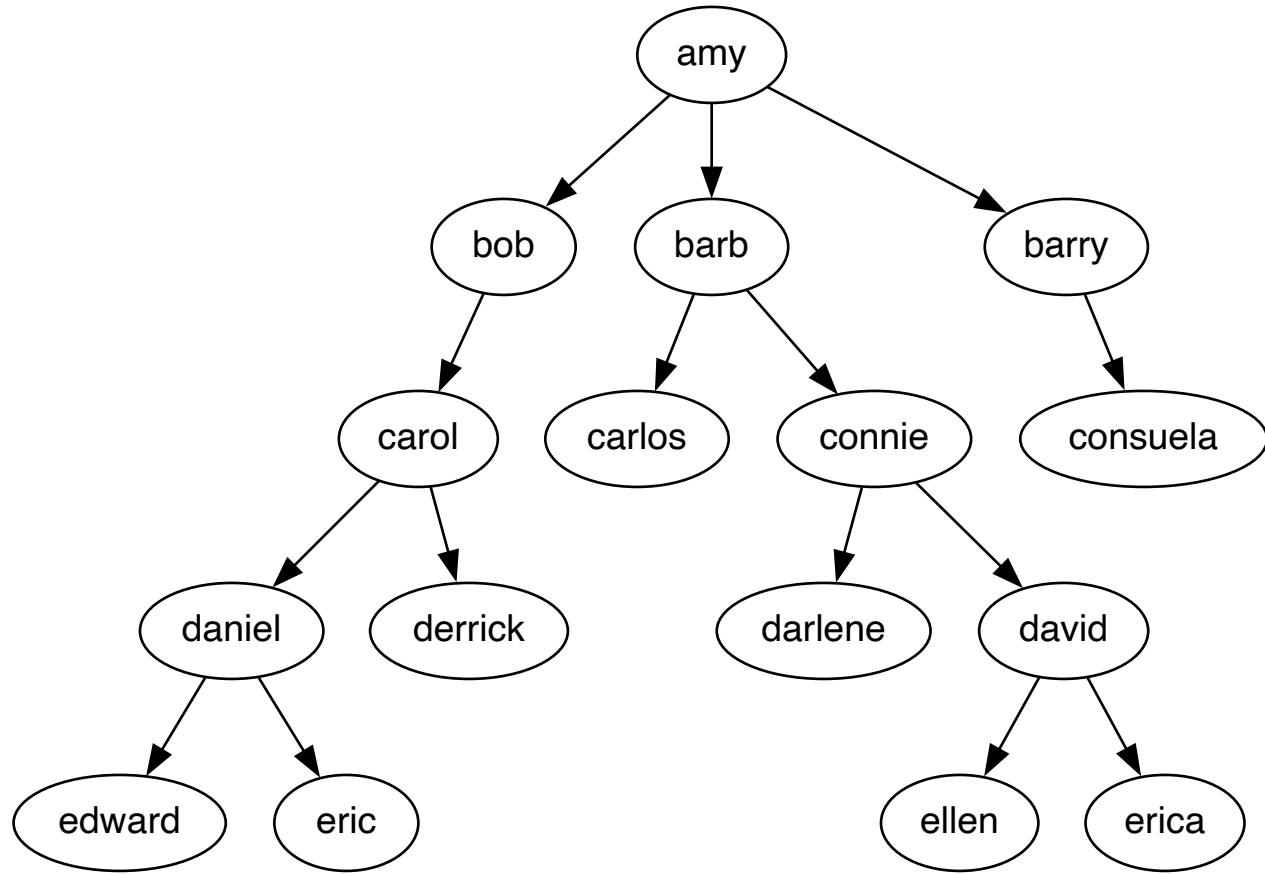
Other Trees

Less commonly, there are other n trees

One example is “quad-trees”, often used for geographic programs, to keep track of N/S/E/W information from a node.

Advanced Ideas

Moving Up

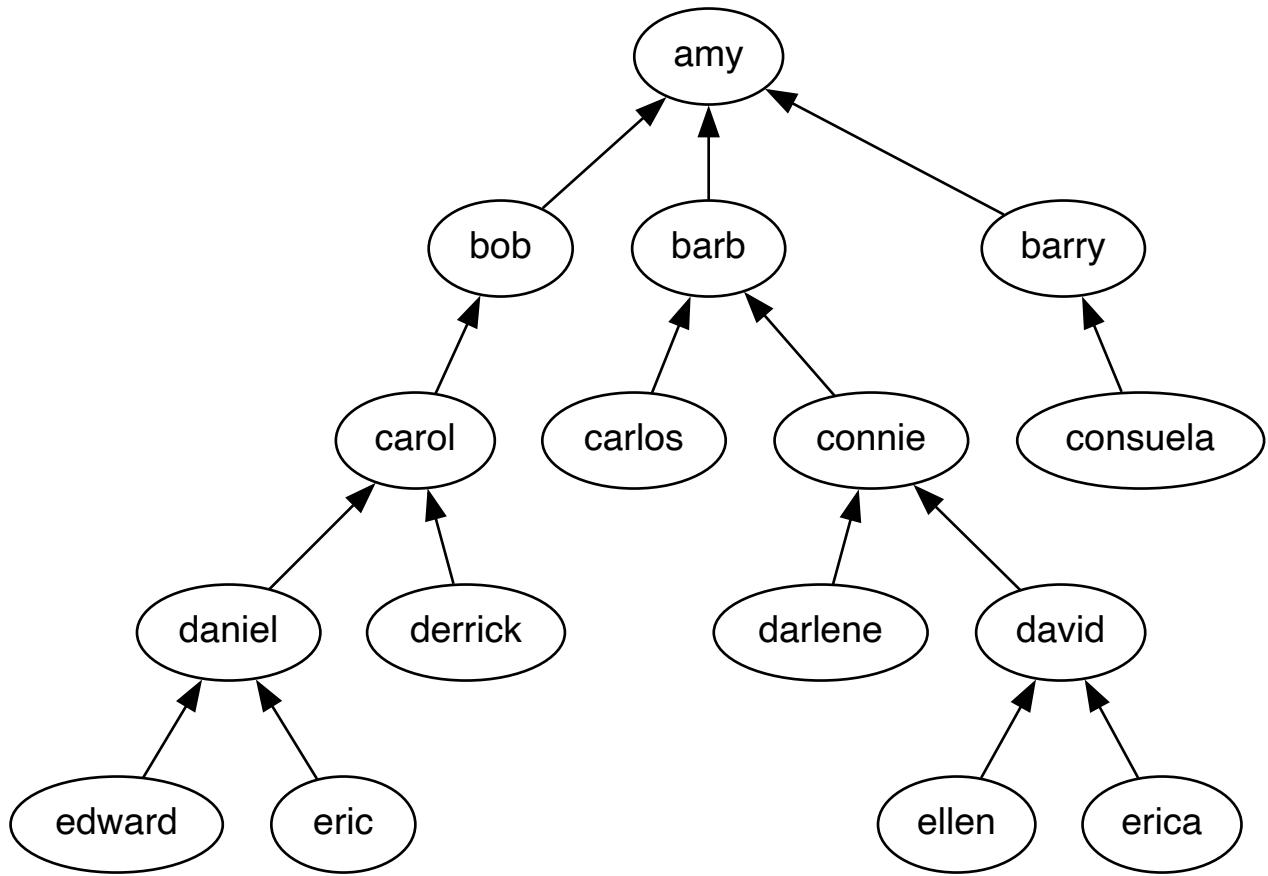


barb.children

It's not possible

barb.findAnce

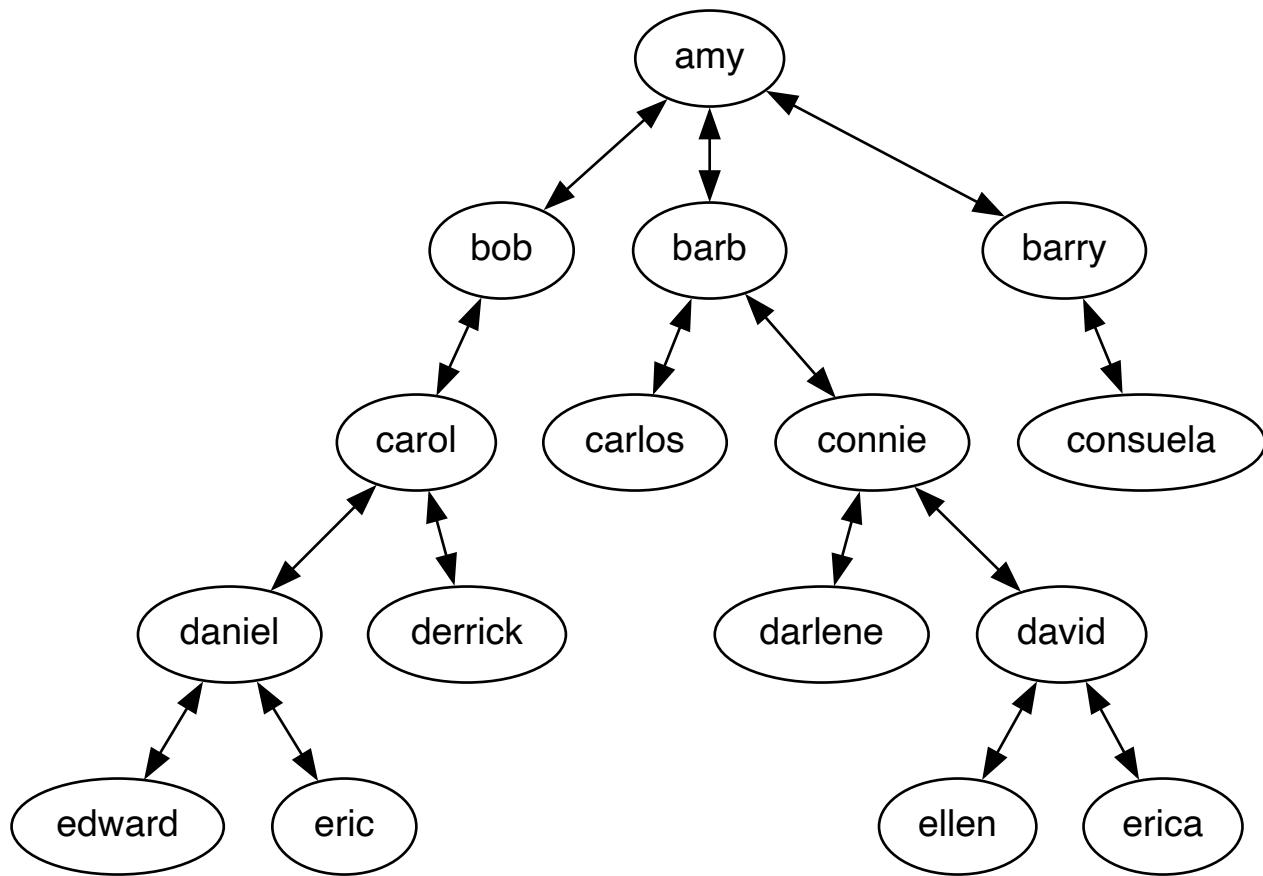
Some Trees Point Up



Of course, then you can't move down!

So you'd need to keep a list of all leaf nodes

Some Point Both Ways



```
class BidirectionalGraph {
    constructor() {
        this.value = '';
        this.parent = null;
        this.children = [];
    }
}
```

Resources

How to Not Be Stumped By Trees <<https://medium.com/basecs/how-to-not-be-stumped-by-trees-5f36208f68a7>>

Binary Search Trees and Hash Tables

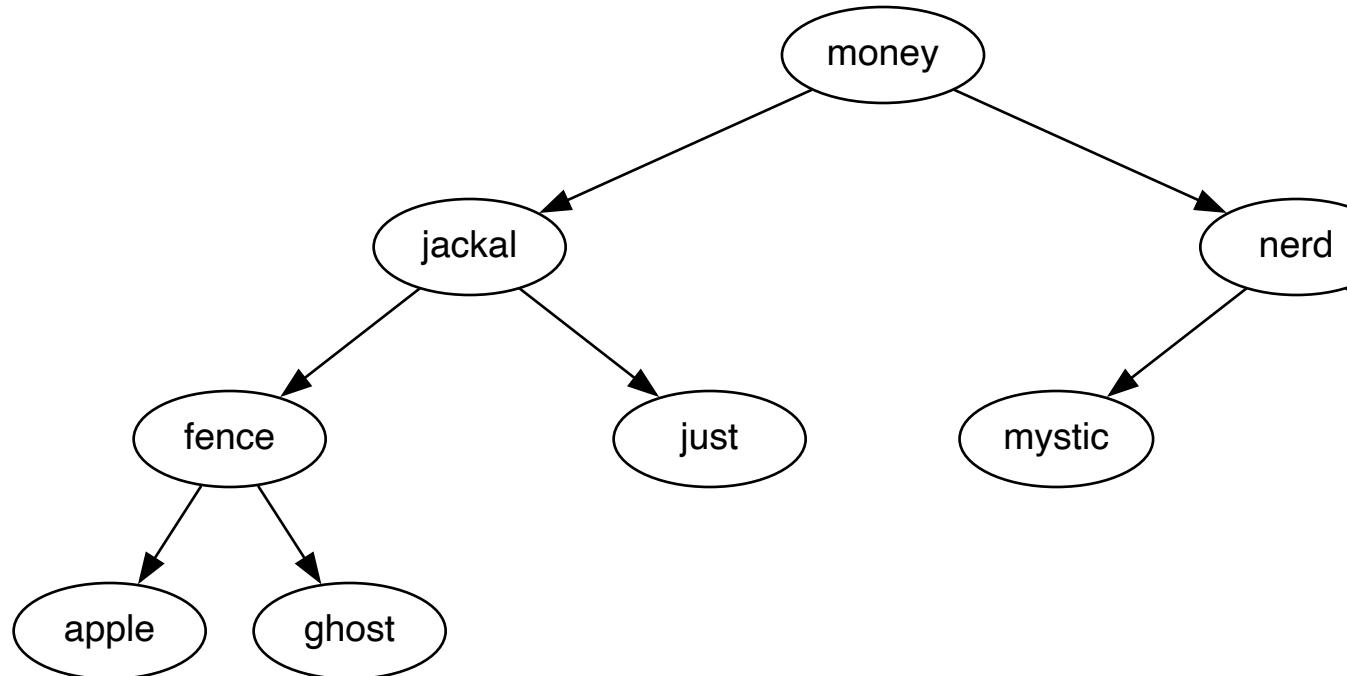
Binary Search Trees

A List of Words

Imagine this list of words:

apple, fence, ghost, jackal, just, money,
mystic, nerd, pencil, zebra

Binary Search Tree



- Also a tree, made of nodes
- But each node has a left and right child
- Has a “rule” for arrangement
 - Often used for fast searching

Implementing BSTs

Node Class

Node class is same as any other binary Node class:

```
class BinarySearchNode {
  constructor(val, left=null, right=null) {
    this.val = val;
    this.left = left;
    this.right = right;
  }

  // other methods here
}
```

Tree Class

Just like with n-ary trees, may not *always* need class for tree.

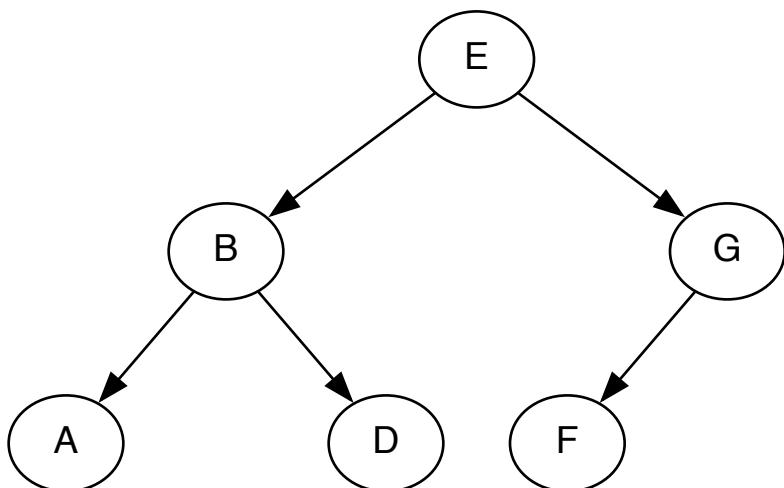
But it very useful for keeping track of root of tree:

```
class BinarySearchTree {
  constructor(root) {
    this.root = root;
  }

  // other methods here
}
```

Searching

Binary Search Tree Find



demo/bst.js

```
class Node { // ...
  find(sought) {
    let current = this;

    while (current) {
      if (current.val === sought)
        return current;

      current = (sought < current.val)
        ? current.left
        : current.right;
    }
  }
}
```

Starting at *E*, looking for *C*:

1. *C* comes before *E*, so go left to *B*
2. *C* comes after *B*, so go right to *D*
3. *C* comes before *D*, so go left to `None`
4. Drop out of `while` loop and return `None`

Every choice we make reduces # options by half!

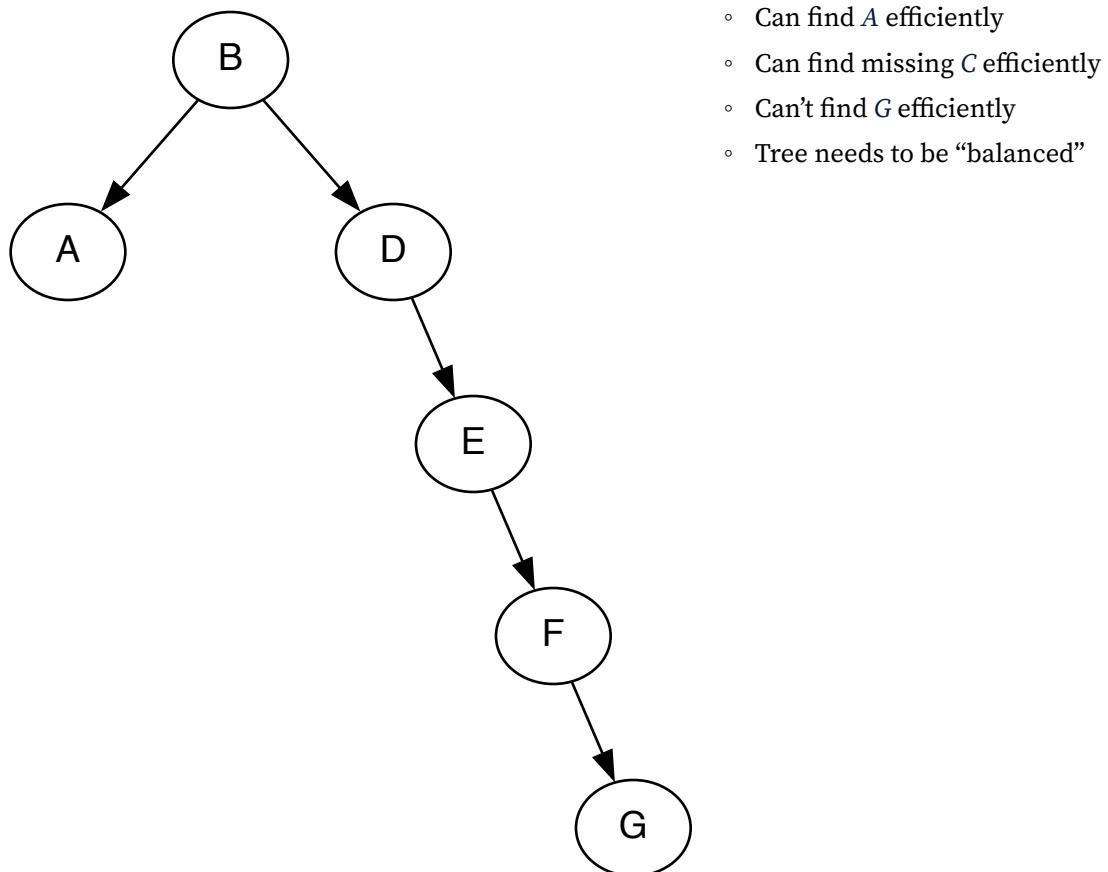
For *n* nodes, we need to search, at most O(log n) nodes

We can search >1,000 nodes in only 10 steps!

We can search >1,000,000 nodes in only 20 steps!

Balancing

Valid But Badly Balanced



Balancing Trees

Easy ways to get reasonably balanced trees:

- shuffle values for tree randomly, and then insert
- or sort values, then insert from the middle working out

Self-Balancing Trees

There are algorithms for BSTs that can balance themselves.

You don't need to understand how they work, but you might see them referenced:

one common type is called an *AVL Tree*.

Traversal

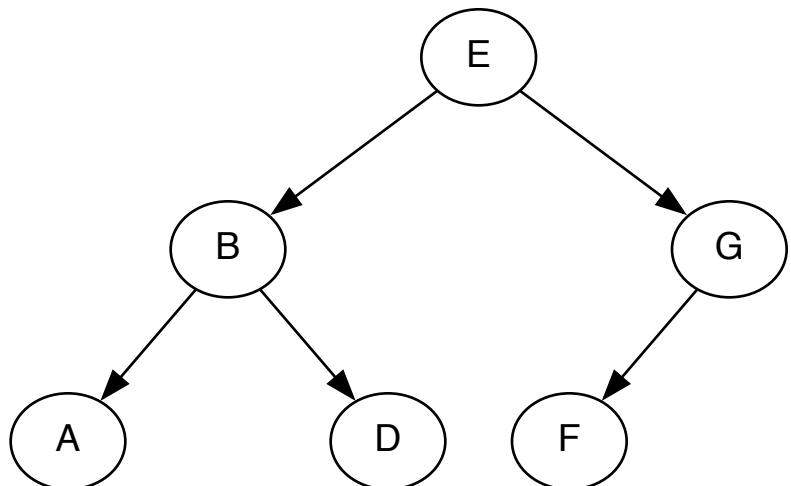
Often, you don't want to look at every node in a BST

That's the point — you can search without looking at each!

But sometimes you will want to traverse entire tree

In Order Traversal

```
traverse(node) {  
    if (node === null) return;  
    traverse(node.left);  
    console.log(node.val);  
    traverse(node.right);  
}
```

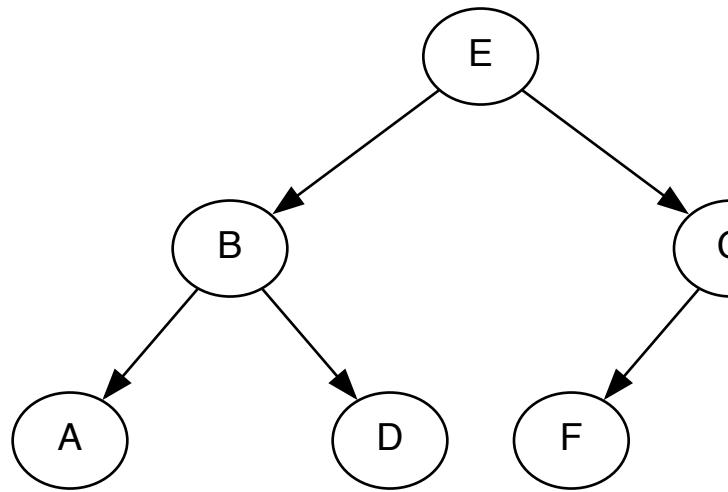


“traverse left, myself, traverse right” is “in-order”:

A → B → D → E → F → G

Pre Order Traversal

```
traverse(node) {  
    if (node === null) return;  
    console.log(node.val);  
    traverse(node.left);  
    traverse(node.right);  
}
```

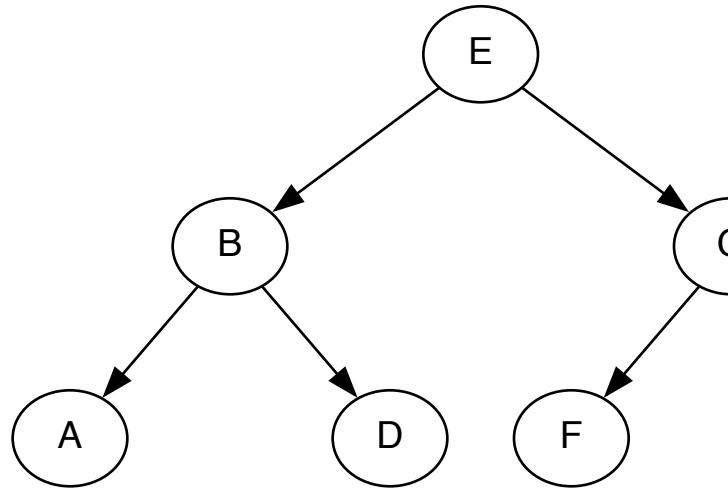


“myself, traverse left, traverse right” is “pre-order”:

E → B → A → D → G → F

Post Order Traversal

```
traverse(node) {  
    if (node === null) return;  
    traverse(node.left);  
    traverse(node.right);  
    console.log(node.val);  
}
```



“traverse left, traverse right, myself” is “post-order”:

A → D → B → F → G → E

Maps

Map

Map
Abstract Data Type for mapping $key \rightarrow value$

Hash Table (also called Hash Map)

Efficient implementation for the Map ADT

```
let petsToAges = {  
  "Whiskey": 6,  
  "Fluffy": 2,  
  "Dr. Slitherscale": 2,  
};
```

- Javascript: *Map* or `{}`
- Python: *dict*
- Ruby: *Hash*
- Java: *HashMap*
- Go: *map*

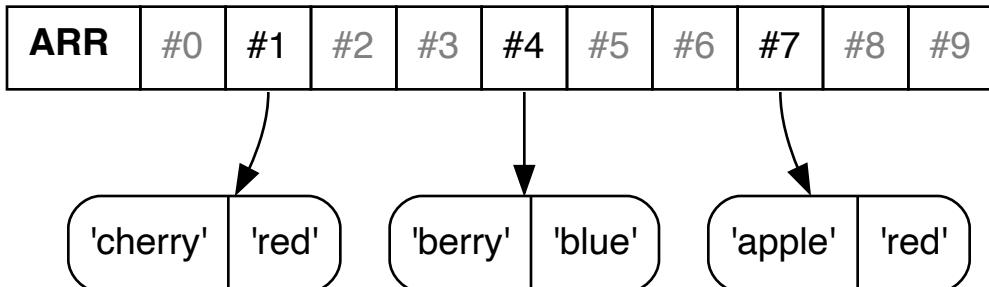
Typical API

<i>set(key, val)</i>	<i>size()</i>
Sets <i>key</i> to <i>val</i>	Number of items in the map
<i>get(key)</i>	<i>keys()</i>
Retrieve values for <i>key</i>	Iterable of keys
<i>delete(key)</i>	<i>values()</i>
Delete entry for <i>key</i>	Iterable of values
<i>has(key)</i>	<i>entries()</i>
Is there an entry for <i>key</i> ?	Iterable of key/value pairs

Hash Tables

```
let fruits = {apple: "red", berry: "blue", cherry: "red"};
```

- It'd be awesome to keep this in some sort of magic array
 - Get $O(1)$ time for many operations

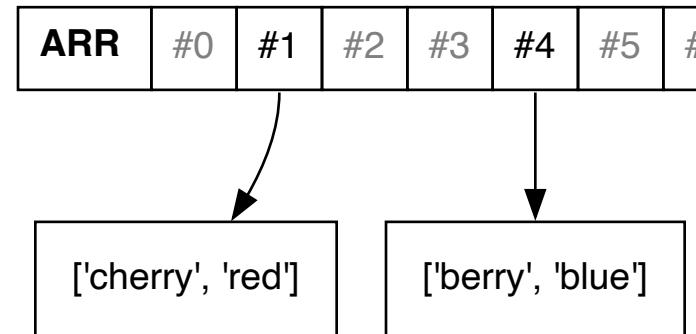


But how could we know that "apple" is index #7?

Hashing!

Hash Table

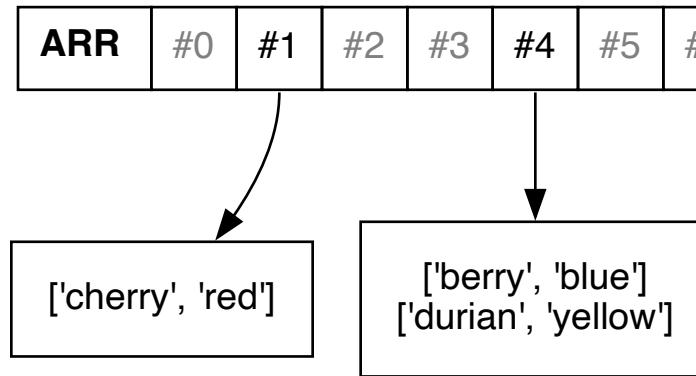
```
apple → 7  
berry → 4  
cherry → 1
```



Oh no! Two keys hash same?

```
apple → 7  
berry → 4  
cherry → 1  
durian → 4
```

Solution: Each bin is array of [key, val] s



HashTable Runtimes

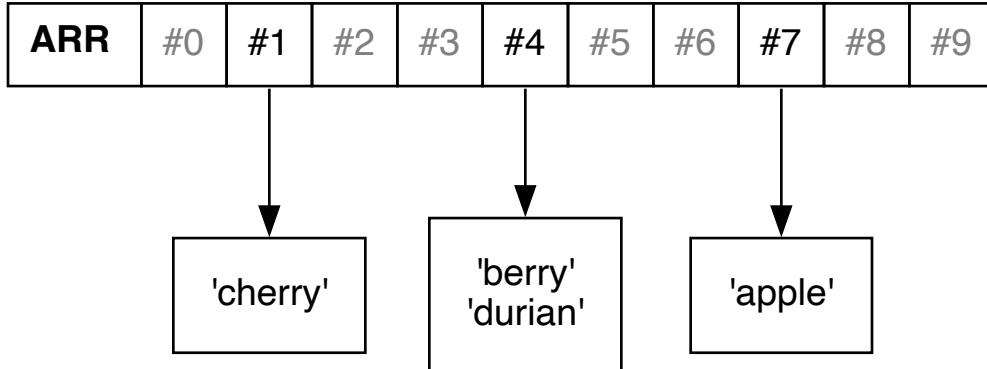
set	O(1)
get, has	O(1)
delete	O(1)
size	O(1)
keys, values, entries	O(n)

Resizing

- To ensure efficiency, good implementation shrink/grow array
 - Often aiming to keep it ~75% occupied
- This means some `.set()` and `.delete()` calls will take longer
 - If shrink/grown by proportion (eg, double/halve), will be “amortized O(1)”

Sets

```
fruits = new Set(['apple', 'berry', 'cherry', 'durian'])
```



- A *Set* is just a *Map* without values
- Same runtime characteristics

JavaScript Types

Map

- Built-in type for mapping
- Keys can be any type
 - Retrieval uses `==` to match
- Keeps keys in order of insertion
- Amortized $O(1)$ for set/get/delete

```
let fruits = new Map([
  ["apple", "red"], ["berry", "blue"]
]);

fruits.set("cherry", "red");

// some methods return map, so can chain
fruits.set("cherry", "red")
  .set("durian", "yellow")
  .delete("apple")

let berryColor = fruits.get("berry");

// how many objects in my Map?
fruits.size; // O(1)
```

Object

- POJO; can use for mapping
- Prior to *Map* (2015), was only way!
- Keys can only be strings
 - Numbers stringified: `1` → “`1`”
- Keys **not always** in order of insertion
- Amortized $O(1)$ for set/get/delete
- Better to use *Map* for mapping

```
let fruits = {  
    "apple": "red",  
    "berry": "blue",  
};  
  
fruits.cherry = "red";  
fruits["durian"] = "yellow";  
  
let berryColor = fruits.berry;  
let cherryColor = fruits["cherry"];  
  
// how many items in my object?  
Object.keys(fruits).length; // O(n)
```

Keys can be a few other less common things, such as Javascript “Symbol” types, though these are uncommon for use in mapping (this is more common when making special methods for OO). The ordering of keys can also at times be a bit complex when you have different types of keys.

IMPORTANT Insertion order not guaranteed!

Do note carefully that JS objects **do not guarantee** that keys are returned in the same order they were added. When keys are “number-like” (keys are strings, but a number-like key would be a string of a number, like “`42`”), they are **not** kept in order of addition, but instead are always returned in strict numerical order, before all other keys.

```
let o = {  
    b: 10,  
    a: 20,  
    7: 30,  
    2: 40,  
};  
  
Object.keys(o); // ['2', '7', 'b', 'a']
```

When you want to preserve the order of addition, use a *Map*:

```
let o = new Map([  
    ["b", 10],  
    ["a", 20],  
    [7, 30], // works the same with key = "7"  
    [2, 40],  
]);  
  
o.keys(); // MapIterator {'b', 'a', 7, '2'}
```

Set

- Built-in type for sets
- Keys can be any type
 - Retrieval uses `==` to match
- Keeps keys in order of insertion
- Amortized $O(1)$ for set/get/delete

```
let fruits = new Set(["apple", "berry"]);  
  
fruits.add("cherry");  
fruits.has("apple"); // true
```

Python Types

Dictionary

- Built-in type for mapping
- Keys can be any *immutable* type
- Keeps keys in order of insertion
- Amortized $O(1)$ for set/get/delete

```
fruits = {"apple": "red", "berry": "blue"}  
also_can = dict(apple="red", berry="blue")  
  
fruits["cherry"] = "red"  
  
fruits["berry"]      # error if not there  
fruits.get("cherry") # or None  
  
# dict comprehension  
{x: x * 2 for x in numbers if x > 5}
```

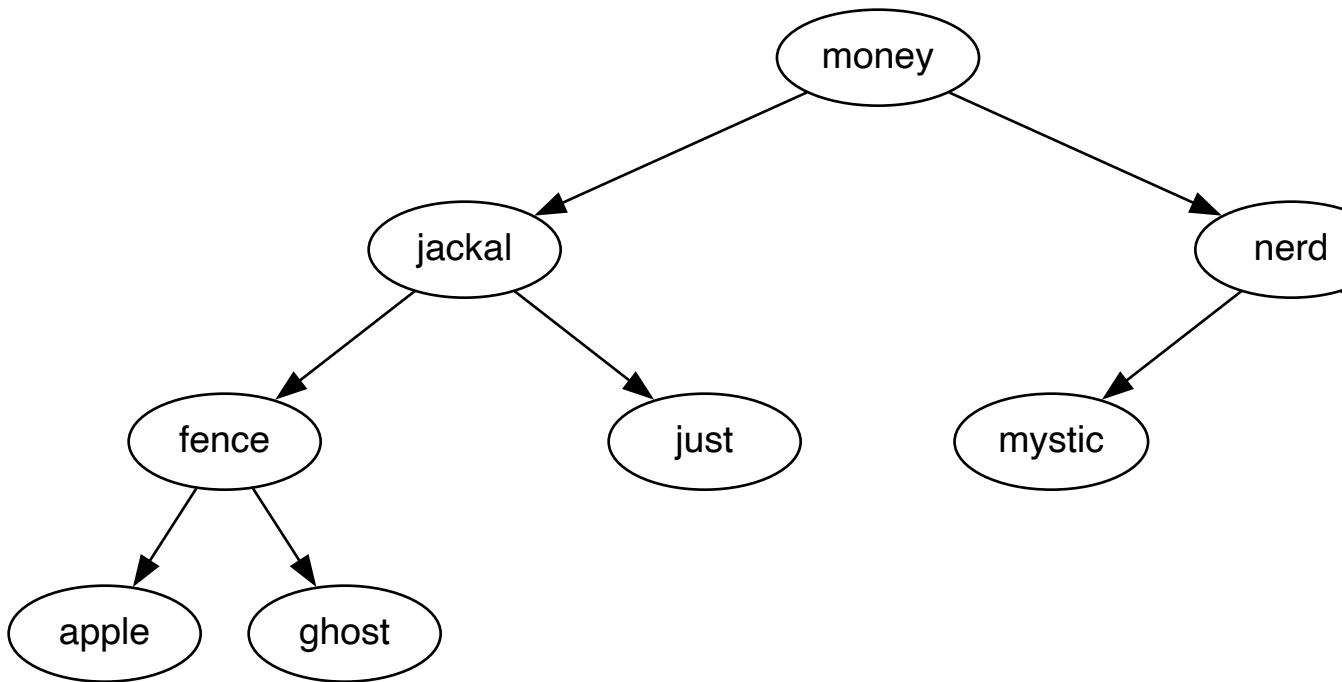
Set

- Built-in type for sets
- Keys can be any *immutable* type
- Key order not guaranteed
- Amortized $O(1)$ for set/get/delete
- Has awesome built-in set operations
 - Union, intersection, symmetric difference, subtraction
 - For JS, can get with *lodash* library

```
moods = {"happy", "sad", "grumpy"}  
  
dwarfs = set(["happy", "doc", "grumpy"])  
  
# union, intersection, and symmetric diff:  
moods | dwarfs  # {happy, sad, grumpy, doc}  
moods & dwarfs  # {happy, grumpy}  
moods ^ dwarfs  # {sad, doc}  
  
# subtraction  
moods - dwarfs  # {sad}  
dwarfs - moods  # {doc}  
  
# set comprehension  
{n for n in some_list if n > 10}
```

Binary Trees vs Hashmap

How do they compare?



Hashmaps

- O(1) lookup/addition/deletion
- Have know exactly what you're looking for
- Can't find "first word equal or after banana"
- Can't find range of "words between car and cat"

Binary Search Trees

- O(log n) lookup/addition/deletion
- Can search for exact value, or inequalities
- Can search for ranges
- Often used to implement indexes in databases

Resources

[Leaf It Up To Binary Trees <<https://medium.com/basecs/leaf-it-up-to-binary-trees-11001aaf746d>>](https://medium.com/basecs/leaf-it-up-to-binary-trees-11001aaf746d)

[Trees & Binary Search Trees video <<https://dev.to/vaidehijoshi/trees--binary-search-trees--basecs-video-series-5e38>>](https://dev.to/vaidehijoshi/trees--binary-search-trees--basecs-video-series-5e38)

[Taking Hash Tables Off the Shelf <<https://medium.com/basecs/taking-hash-tables-off-the-shelf-139cbf4752f0>>](https://medium.com/basecs/taking-hash-tables-off-the-shelf-139cbf4752f0)

[Hashing Out Hash Functions <<https://medium.com/basecs/hashing-out-hash-functions-ea5dd8beb4dd>>](https://medium.com/basecs/hashing-out-hash-functions-ea5dd8beb4dd)