CS50 Mobile App Development

with React Native

Lecture 1: JavaScript Fundamentals

Javascript is interpreted, like Python, and is dynamically typed, unlike C. There, you'd have to declare type *int* or *char** but in Javascript you simply declare a variable via *const* (or *let* or *var*) and the typing is taken care of behind the scenes.

We always want to use === in JS as opposed to == because the latter coerces the types (eg [] is == False) whereas you intend to check for exactly equality, which is what the former does.

In JS, if something isn't one of the primitive values (null, undefined, number, string, bool, etc), then it's an object. An object is utterly identical to a dict in JS, and so you can declare it via *new Object() or {}* and then simply by *dot notation* or *['like_this']* access its fields.

When you set one object equal to another, you're pointing to it not storing a whole new memory chunk for that object. So if you change the second object (which is really just a pointer to the first) then you inadvertently end up editing the first object, too. The reason it's important to understand memory and assembly is, concretely, because there will be many times code doesn't run as expected or programs don't work as intended, because of some nuance in the structure of a programming language and what is going on under the hood. Knowing about it will lead to a much easier time troubleshooting.

If you truly want to create a new object, don't set *const o2 = o1*, but do *const o2 = Object.assign({}, o1)* whereby you set the former equal to the latter. This is a shallow copy, though, so nested objects will still be assigned by reference, potentially leading to faults. Arrays are also stored by reference so if you set arr2 = arr1 and then changer arr1, then you change arr2.

All non-primitive types (objects like array) have some built in methods, like .toUpperCase and .length that come with the prototype (array) or even the parental prototype (object methods like .keys) and if there is a naming conflict you always go with the more specific method. Similarly, because these inbuilt prototypal methods are so often used, they've simulated even primitive types having them, so you can do num.toString(). The prototype of an object or instance or variable is stored by reference (not each of the 100 values in an array of arrays stores the prototypal methods of an array), they instead simply know that the elements of the array are prototyped by array, and store it by reference for when its needed.

Understanding the nuance behind a programming language is important for improved efficiency (writing more time & memory efficient as well as dev efficient) code as well as debugging, for any misuse of variable scoping (let, var, const) will lead to errors that you won't otherwise know the cause of. Today, const and let are standard. Var has scope until the end of the function, let has scope until the end of the code block.

JS engines create entire global objects under which things fall—window in the browser and global in CLI, and all variables and methods become a part of this global object, hence the fact that JS is an OOP language.

A short aside on programming paradigms:
These are really just different ways to think about implementing theoretical solutions. People agree on abstracted, theoretical solutions for how to solve a problem, but can make different developmental/design trade-offs for implementation. Different problems are best tackled by, or thought about, in different ways. Each method of structuring code has pros/cons.

Different languages are also structured to suit different paradigms based on what the creators of the language wanted the value prop of their language to be when they implemented it at scale—Ruby, Python, JS are OOP languages, OCaml/LISP is functional,

OOP is good at modeling and intuitive to how we think about the world. Everything is thought of as a "thing" with certain attributes and behaviors (properties and methods). Here, data and behavior are together part of objects.

Functional programming is based on i/o of black boxes and modularity—it's deterministic and data is not mutated. Data and behavior (state and methods) are fundamentally separate—one is a value, and another is a function, as opposed, to, say, both being part of the same object. This is good because since you don't assign values to variables (simply put an input through functions), you can't have problems with concurrent access to state (like variables).

Logic based programming is based on facts and rules. You define some axioms, then ask the computer whether certain conditions fulfill those. This is easy for people who think in terms of logic, and Prolog is an example of a language that works this way.

Procedural programming is a type of imperative programming where you simply say stuff you want to do. An example language is assembly, where you directly dictate what values you want to put in which registers, and then manipulating things accordingly. This style of programming is intuitive, because you're saying exactly what you want to happen and tweaking things accordingly.

Project 0: TO-DO App

Just as I have come to know the ins and outs of python syntax (things like [-5:] have become intuitive), doing multiple projects with Javascript will mean the niggly but important syntax like .createEventListener() and .appendChild() will become intuitive, as well as my ability to learn how to structure and architect small systems.

You should add event listeners for intended behavior when you create a certain object if you want it to have functionality later on. EG here I tried to add event listeners after all the TODOs had been created, but the listener/onclick is like a property you are supposed to imbue the todo-item with so that when they are clicked they are manufactured with that listening property constantly on (and therefore counters can be updated in the future without having to fiddle with individual id's and properties since you sorted them out as you made them).

Lecture 2: Advanced JavaScript & ES6

Since JS is evolving constantly (ECMA updates standards and JS developers update implementation to comply with those), you can use "transpilers" to make new, modern code legacy compatible. Babel, for example, takes your modern code and converts arrow functions into function() notation recognizable by legacy systems.

Closure refers to the creation of unexpected bugs as a result of nesting functions and having unintended differences in scope between the variables used within different levels of those functions.

An IFFE is an immediately invoked functional expression, whereby you invoke a function right after you declare it, and the function disappears afterwards. Don't fully understand scope nuances, closures and IFFE to be honest, and neither do I understand the vocational need to really get these except in passing.

Functions are "first-class" citizens in JS—they are objects like any other non-primitive thing, and are treated as such, being capable of being themselves input or output from other functions. Higher-order functions are these other functions, with map(), filter() and reduce() being the three most common of these.

JS is a single-threaded, synchronous language. This means it reads your code line by line and in a single thread executes it in order. There is, however, async functionality (not in the order it's written) you can take advantage of or built around, if needed, though. To understand how the language can be both synchronous and at times asynchronous, we must understand:

Async is continuing while you wait on something to finish, and sync is waiting for it to finish before moving on. The truth is that JS is truly synchronous on an implementational level—the JS engine in the browser operates using an execution stack, browser API (to fitch and execute external library functions), event loop and function queue. So say you have a program with three functions each printing 1, 2, 3 but you tell the print1 to wait 1000ms, the print2 to wait 0ms, and leave the print3 as is. It will print 3, 2, 1 because after print1 was told to wait, it was sent to the browser API to execute the external wait function, and then the execution stack moved onto print2, which was then also sent, and then executed print3 by printing it. Then, as print2 got done it was moved to the function queue then stack then executed, printing 2. Then, finally, after a second, same happens to print3, resulting in what we saw as 3,2,1, as seemingly async behavior (out of order as programmed) but really operating synchronously on a lower level of abstraction (the stack, queue, API, event loop cycle).

Async functions in JS include setTimeOut() as above, as well as AJAX in jQuery, .fetch(), database calls via SQL if you're writing a node backend etc—things that rely on external libraries or functionality, we will continue executing our javascript while we wait for those things to get done (which makes sense in the context of AJAX and .fetch calls for example).

When you're doing stuff asynchronously in JS, like with the functions given above, you can easily get into "callback hell" where you're getting a branching tree for each time you make a callback spawns two more conditions—you can avoid this using promises which just has one error handler .catch() and so each callback only generates one other thing to do, which is handled by a .then(). There is yet a newer method of handling external callbacks called the async/await method in ES2017, which shows how languages and best practices evolve over time.

This is another neat feature, where you use it if you want to refer to the parent object in which it's being used in a ruction. So an object method's this would refer to the object, and then you could access other methods it has. It's context dependent, and you pick up an understanding of how it's used through practise. You can also use bind(), call() and apply() to manually bind a value to *this.something* in a particular context if you want it to be so.

Lecture 3: React, Props, State

When thinking about classes and instances, methods are those that work with instances, like .toString, and static methods are those that are strictly related to the parent class, like Date.now() which is the same for all instances of Date objects.

The *constructor*() function is one you include whenever you make a class to define its properties and methods (use *this*). You can also use *extends* to extend an existing class (in which case you must include super()) for the native implementation of certain methods (to expand on them in your own custom methods).

React is a declarative library that you use to write pages that update upon arrival of new information very easily. All of the diffing and resource-efficient updating is taken care of for you. Whereas vanillaJS is imperative, where you write the steps for how you want to do something to achieve a certain goal, HTML is declarative, where you simply declare what you want, and the browser takes care of displaying everything like you asked. Similarly, whereas you might imperatively code a guitar object by creating the head, neck, strings and body as objects and then manipulating their properties, you would declaratively, in React, simple create a <Guitar> and insert the <String> objects.

Typically, the browser APIs (getElementBy etc) are fiddly to work with because they force you to think on a low level of abstraction. React abstracts this away, so we just write the elements we want and the library takes care of what element has to be made under the hood to make that happen. It's like writing your HTML but in a JS window so you can write all the variability and code into the html. React is good because it componentizes things, making change easy to enact without repeating things like you may have to do imperatively (change this, and that, and that).

In React's JSX, lowercase tags like <li> are treated as HTML, and uppercase tags like <Slide> as custom components.

The reason that you use constructor() and super() is because you're making a component that isetending a class *component* that is already imbued with some properties that exist. Moreover, the reason that you add this.newFunction.bind() is to make sure that any method you create within that class is bound to that class so you can use *this* afterwards without fear of the transpiler not knowing what *this* refers to (binding it, as covered in the last lecture).

If, in a method you create, like incrementing a counter, you simple setState to a certain thing, and if you do it again you notice that it's only done once, showcasing how React batches up all the changes you make to state and diffs the virtual DOM against the actual DOM, which is why you have to make sure to reference prevState so it knows what to update, and is also important to understand the implementation/ low-level abstraction mechanics when architecting functions and debugging—if you didn't know how asynchronous batching works, you wouldn't understand that React bundles up the changes and makes them behind the scenes outside of the thread currently operating, and so wouldn't be able to understand why your two executions are being processed like one.

ToDo in react vs vanilla, pros/cons and structural differences

You should add event listeners for intended behavior when you create a certain object if you want it to have functionality later on. EG here I tried to add event listeners after all the TODOs had been created, but the listener/onclick is like a property you are supposed to imbue the todo-item with so that when they are clicked they are manufactured with that listening property constantly on (and therefore counters can be updated in the future without having to fiddle with individual id's and properties since you sorted them out as you made them). It's interesting that when you write the to-do functionality in vanilla (imperative) you have to tell the program what to do when you make the elements of the todo (createEl li, add checkbox to it, imbue it with this onClick) as opposed to *writing out the elements yourself (declaring them).*

And so you don't need to write any onToggle functionality for the checkbox since you can use the DOM API target.checked, but you *do* have to essentially create the checked (true/false) property when you make a React component because you don't have easy access to the DOM API (it's abstracted away—that's the point), and so maybe this particular app is actually easier to implement in vanillaJS.

It's also important to note that it's not binary—imperative vs declarative paradigm; there's a continuum. You can write stylistically in a way that leans either or, or somewhere in the middle. On the imperative side, to implement this app, you might do it like I did in the vanilla file, which is to make the elements, add things like the span and checkbox to them, then addTodo() by making a new element and adding that to the <ul>.

A step closer to declarative, we might abstract out the creation of the Todo that encapsulates the creation of the elements.

A step even closer to declarative than that could be to use li.innerHTML = '<hr><input type' and so forth manually declaring what you want in your todos, but still *telling* (imperative) the program how you want to handle those.

A step even closer would be to implement the render() functionality yourself of updating the DOM <ul> as a function, and calling that on the custom made *renderTodo, addTodo* and *removeTodo* functions, storing the todo as a list in client side memory as an array of objects with keys of string, id, checked. This is much more declarative because you're effectively implementing a mini-React and declaring how you want a list to be shown, then updating every time you add or remove a todo by using the render() you created. You'd map the items in your array through renderTodo giving you a list of HTML items, then forEach append them to the list in render().

The final step is entirely declarative, which is to use React and simply write out the components as you want them. This may be easier or harder for some problems than others, and you should, over time, see that a todo list is easily componentized (though it is easy to implement in vanilla in this case too) and learn towards React.

Lecture 3: React Native

Same principles as React, state, props, etc., just using native mobile components instead of web components. It has multiple threads, including one for the UI and one for the JavaScript logic, which communicate using a 'bridge'—because of this, you can have the JS lock up by some (while(true)) or something and the UI will still be able to scroll and be responsive, just nothing will happen because business logic thread is locked up. This is different from React on the web, which conforms to the natural JS single-threading standard.

It's also important to note that while RN is a powerful prototyping tool, it is very limited in production. It was designed to reduce overlap in iOS and android code, not to be the single source of truth in itself. There are tons of misoptimisations for each platform to allow it to be platform agnostic, and if you want to deploy anything at remotely production-scale, you should write in Swift or Java, respectively, because each platform has its lions share of quirks and idiosyncrasies, if you're building a tech company you expect to scale and add complexity to. There are, some apps, even big ones, built on React Native—including the Olympics 2020 app, for example. It can be annoying to have RN in production if you're already writing native because unless 95% of your app is truly in RN, you end up having to manage and synchronize production in 3 environments, not 2.

A number of semantic differences include: button is Button, lists are ScrollView, div is View, span is Text, and more. Browser APIs like prompt() and getElementByID don't exist either, of course. CSS also doesn't exist, and functions like fetch and console and timeout all have to be imported (technically 'polyfilled') because they otherwise don't exist.

Stylistically, you use javascript objects passed in as props on a View or ScrollView or Text or something of the like. It's structurally analogous to CSS flexbox, and lengths are in unitless numbers. You can also import styleSheets from 'react-native' and assign these javascript object style definitions to IDs which you then refer to in the UI thread. You can read the docs to learn things like the fact that onClick is now onValueChange etc.You also need to consult the docs to know things like when you want to pass multiple styles in, you do that by passing an array, and that only certain components, like Buttons can be engaged with using certain event handlers, like Button and TouchableOpacity are amongst the only components that can be interacted with (so Spotify has all its clickable stuff as different versions of a Button if they used React Native, which they don't because they operate at scale).

You have two types of components—stateless (pure functional) which are just const comp = (props) => {func} which simply take in props and return a component, without touching state at all, and then you have React.Components which extend that and therefore have all the state functionality, and are used for lifecycle manipulation and more heavyweight logic.

Lifecycle methods include componentDidMount, componentDidUpdate, render(), shouldComponentUpdate, componentWillUnmount() and more. These can be thought as similar to hooks and event handlers.

Expo is simply a dev environment that handles much of the back-end and deployment pain of apps, kind of like Django for Python except it's also a XDE in addition to providing library/framework code.

PropTypes is a debugging tool that allows you to see exactly what props are being passed in where in expo dev mode to make your life easier when it comes to seeing under the hood.

Lecture 4: Lists, User Input

Whereas in web dev scrolling is taken care of by the browser, you have to add that functionality if things go off the screen manually when doing mobile dev by putting everything inside a <ScrollView>

Because we so often declare a constructor simply to define state (and bind component methods), react has made a shorthand for us where we can simply write state = {} and that will compile down into the constructor with the bound methods below it that you can declare like methodName = () => and it'll recognize that it needs to compile them into the constructor function.

When you're listing data sets in a scrollview, it's important to pass in a key prop to the component in which they're being rendered, because if you change the dataset, react can update its vDOM much more efficiently to see what changes have been made by comparing keys rather than data values, which it has no idea if they are unique, leading to much better performance, especially with even moderately large datasets.

If you want to pass multiple props in which all stem from one object, you can just do {…object} just as you dow with arrays (indeed, in Python, as well) in JS/React.

An important caveat to ScrollView is that it renders all of its children are rendered in the vDOM before it actually shows on page, and with large datasets that can take a while (eg mapping these datasets to components via a list) to render, whereas really you want stuff to appear on the screen as it's been processed instead of waiting for the whole dataset to render before showing anything in the ScrollView. FlatList is a component that allows you to do that—but you have to be careful manipulating state of these components because their lifecycles are nuanced—they are mounted and unmounted as things become visible or are scrolled past on screen, and so you don't want things' state to change and therefore their functionality to change just because they are not currently on the screen as that may affect other things in your app. There are libraries to handle this nuance to make it easier to manage state in this light, to be fair.
By reading the docs on FlatList, you'd see that it takes two arguments, including an object that defines how you want to list the data items, and then one with what the raw data is at all (in this case the list of contacts).

If you wanted to implement a function sorting contacts, you'd have to add the contacts to state and then make a button that changes state to .sort() the contacts. But since a contact is an object and there's no clear way to sort it, you have to pass in a method you create called compareNames or something that says take the first name and last name fields and compare them using > for this purpose, and return the sorted names, and then pass in *a new array* (by spreading the prevState.contacts) for sorting so that the FlatList knows to update (because *new props are passed in*) every time you press the sort button.

We also have a similar component called sectionList which takes in sectionHeaderList as well to partition the list into different sections (like in the actual contacts app on the iPhone).

There are two ways to design input components (<TextInput/>), and indeed components more broadly—either in a controlled or non controlled manner. In a controlled user input (which react recommends) overall you set the value of an input component, and handle the change the user makes by recording their change and resetting state to that (you are therefore controlling the output of what they are typing). Alternatively, you could leave it to the dom, and let it show up just because they're typing it, which is what happens normally. The reason they advocate controlled is because you can them manipulate the input easily without any web APIs (indeed this would be implemented using ref's which determine what's in the textbox in react) , for which React does not offer much support. This is why it might be weird at first, but you should get used to *determining* what the user writes as opposed to to merely *grabbing* it, which will inevitably require the use of functions to handleChanges as props on those controlled components.

Lecture 5: User Input, Debugging

The advantage of writing controlled components is that you can then easily validate inputs too, as you can check whether inputs fit certain criteria before re-setting state (which, definitionally, in a control component, determines what they see on screen).

You can cast a string into an int by prepending + to it and therefore verify a string is composed only of ints by doing +string because it returns false if there's any letters or punctuation in there—a useful verification tool.

Note that the second (optional) argument in this.setState takes in a callback function to execute after it has set state, useful for validating controlled input components. You could also use componentDidUpdate(prevProps, prevState) lifecycle method to the same end.

Remember that you can use Constants.statusBarHeight and things in the styleSheets. KeyboardAvoidingView(behaviour="padding/height etc") is what you can use if you're making a large form that's being obscured by the keyboard when typing.

A clever optimisation for the form input handlers is to make on general getHandler(key) function which returns *a function* that, in turn, edits the state of the application *at that key* with the value passed in. This way you can make hundreds of eventHandlers by calling the same function with a different argument (see this lecture for deets). But note that this way you are invoking a function to create a function, which then sets state, instead of doing it yourself with one function for each form field. In this sense, you're pitting cleanliness of abstraction and speed of dev time with efficiency of code, and your choice is a *design decision* you have to make based on what trade-offs feel right for you.

Debugging

You can access errors via console.error, and you can even *throw error("No password entered.")* in the case of certain events within your program. But more than throwing up phat red errors, warnings are more often used, as they were with propTypes (which uses the console.warn() to do so). This might be useful when writing or packaging code for others to use more than writing it for your own app to use.

You can also run RN in a chrome browser using their Node engine by going to the Expo shake page and opening in chrome then going to console, debug tools. This is nice because not just of the robust debugging tools they give, but also because their console is very powerful, and can .log(), amongst other things, things that the terminal is computationally unable to keep up with (like loops).
The inspector is used like the chrome DOM inspector so you can see which components have what styling properties and how they're laid out, normally used to debug UI/aesthetic problems. More powerfully, you can use react-devtools to debug your app's layout and functionality in real-time, as it's running, changing values to see the changes clearly and explicity. This, armed with perseverance and knowledge of how Advith debugs by literally addressing the error message and checking files himself, as well as incrementally building files up, should mean that building even complex apps should be straightforward, even if time consuming. I need to make sure to get into the good habits of writing clean, maintainable, scalable code, though, ready to pivot into a CTO position should that ever be necessary.

As a corollary, interesting to note how improvements in debug environments like this are exponentially leveraging, in that they allow millions of developers to save hours each, and produce apps that otherwise might not be as efficient, clean, and usable, and therefore save each of the users of *those* apps minutes, leading to potentially billions of users given a nicer experience and saved billions of hours—and such would be the leverage of being a foundational/senior member of the React/Angular team, which is an interesting take. Moreover, it's also interesting to note how companies always offer lots more products than we know (most people don't know FB owns Oculus, much less has ReactJS as a huge developer tool) that each need to be maintained, reviewed, analysed, improved, marketed, and scaled, part of the reason that these huge companies have so many employees.

Note that package.json contains the dependencies in JS, just like requirements.txt in python3. When using external libraries, you need to npm or pip3 install, either into the virtualenv (every project) or globally, in which case things like react, just once.

Lecture 6: Navigation

before, we'd been handling navigation by creating bools that determined whether certain components would be rendered, when they would be rendered or not, often adding this as showForm state. This gets cumbersome when you have many screens to show, as well as screen branches within *those*. Instead, we can abstract all that way by using the react-navigation library.

The fundamental building blocks in this library are navigators, which allow you to see the routes you can take like divs with tags at the bottom of the screen, routes, which correspond to the literal paths through the apps, and screen components, representing the different screens you get as you traverse those varying routes.

The basic way to create navigation is by using a library object called a SwitchNavigator:

```
Const Nav = createSwitchNavigator({
        routeName1: screen1,
        routeName2: screen2,
},
{       'initialRouteName': screenToStartWith,
}
)
```

And then when you want to design a button that goes to routeName2, you can pass in nav as props and do this.props.navigation.navigate('route2') as the, say, onPress for a function. If you have just Nav rendered in App.js because you've abstracted everything else away into its own files (as you should be doing) then you might encounter a situation where you need to pass in a certain prop that all the screens need access to, which you can do by passing in *screenProps* as the prop.

But all of this is just to handle changes in screen like a web page redirecting. You also want people to be able to go down rabbit holes, like going down a stack of screens, able to go forward or backward with the slide of a finger left or right—which is done using the *stackNavigator*. A key difference here is that the states of the previous screens are maintained for easy access, which does not happen with the *switchNavigator*. You can implement it the same way as above, but with the additional fact that since it tracks history you can use the this.props.navigation.goBack() functionality as well. The reason data structures/algorithms are important IRL is because that's how you implement all of these cool libraries (vDOM, stackNavigator) and, by extension, are more able to debug/see through them when using them.

You can also use *static NavigationOptions* to "style" different screens (eg change the title, header colour etc). And all of this is just to implement and streamline the standard things that any user expects from a navigation interface (sizing, sliding, clean labels, etc).

There may also be instances where you take in some data and want to pass it to a different different screen as you route to that second screen from the first screen, you can use this.props.navigation.getParam('paramName'). You can also use setParam() on the same page to alter the data that was passed in. This feature of navigation.get/setParam is particularly useful in passing data between the title/header of pages and the screen components themselves, where you can make the navigationOptions into a function taking in the *navigation* object from this.props.navigation and then accessing its properties to style, say, the header, in the same way. An example of the power of this is to make any routing button you want in the header by using its access to the navigation object by passing in headerRight in the navigation function to be a <Button onPress={() => navigation.navigate('routeName')} />

Using *debugger* when debugging with chrome devtools can be useful to sort of have it console.log for you as you go through the code to see what functions in which files are being hit when you press certain buttons.

If you want to open a new screen, use navigate.push() as it'll refresh the front-end even if you're going to a new version/instance of the same template screen. These are stack functions, so naturally pop(), popToTop() etc exist to allow you to navigate the stack screens easily.

You can also compose navigators, eg by having a switch navigator between the login screen and main app, then a stack navigator within the main app. And make sure never to render a <Navigator> inside of a screen component, instead always reference them to each other when declaring navigators in the first place, routing them to screenViews as appropriate.

Just as we have stack and switchNavigators, we can create tabNavigators that do what it says on the tin, render different stacks or tabs when you click on a certain tab (tabs could be contacts, settings, etc) but all of which are still under the 'main' umbrella of the login tab.

You can grab icons from a library called Ionicons which has a bunch of SVG and use a ternary operator when styling a stack.NavigatorOptions so that a different color icon shows up when it's focused or not (as is normal in mobile apps).

Lecture 7: Data

You have React components, and indeed devices, talk to each other by talking to a shared parent, which in the case of devices and companies is a server, and interfacing with a server is done by using their API (their gate that allows processed information out and unprocessed information you're giving to them to be passed into their architecture). But api is a more general term for anything abstracted away from you—React components have APIs that you interact with by passing props (you're not writing the HTML yourself, the library is and you're passing data, that is, props, into this "API" that is doing this for you, even though it's on your computer and not some other server). In this sense, API gets to be generalized to mean anything where you're using some external functionality that can require information passed into it. Another example are DOM APIs where you're using getElementByID, something implemented by the Javascript creators to manipulate the DOM. Even locally, you interact with the state of classes by invoking methods, and so methods are the "Gateway" by which you can manipulate what is going on within a class, and therefore the methods are the "API" of that class.

A short aside on callbacks, promises, async, and JS

Most programs wait for code on line 1 to finish before going onto line 2, even if line 1 takes some time. JS is different in that it moves straight on—normally things like I/O take time (reading, writing large datasets to a DB, getting a large DB from some API), and you don't want web interfaces to freeze up/block (which is what JS is responsible for) while this is happening. Async just means something takes some time, and other things happen while it is processing, and it happens at some point in the future. And so in anticipation of the point where the async action has completed, you define a function that outlines what to do, and this is called the callback function. The order in which things in JS happen is not top-to-bottom, cleanly, like in other langs like C or python, but it jumps around based on what's completed.

People used to write modular functions that each handled part of callback hell, and generalize error messages, but that still had some complexity. .fetch() was difficult to implement, but now that Mozilla has done it, it's a very intuitive way to handle async JS.

Nowadays, instead of AJAX, we use .fetch().then().catch(), where the parent method returns a promise, which is just the name of the object that contains code that will return with either a status of fulfilled, ie it contains the data it was sent to get, or rejected, in that it didn't. In the common fulfilled case, a field of which is the data you'll request, in the .json() field, and then you can manipulate the data easily going forwards as shown below. The promise object is an innovation because it helps devs avoid callback hell because the way its structured.

```
fetch('http://example.com/movies.json')
  .then((response) => {
    return response.json();
  })
  .then((data) => {
    console.log(data);
  });
```

A modern implementation is using async/await, implementing things as if they were synchronous by doing under-the-hood trickery. This is why try()catch() was created, for error handling in this pseudo-synchronous manner to allow the program to manipulate the error if it ever came up. A useful way to write JS for string casting is `${JS}` where the JS will be cast into a string and stored in whatever variable you assign it to.

When you're sending a POST req to an endpoint, make sure to include in the object you send:

headers: {'content-type': 'application/json'}
body: JSON.stringify({username, password:password})

You can also use *throw new Error(msg)* in the case that !response.ok. The way that backends (eg authentication by querying a DB) is implemented is the same as web, but most of the functionality in the app doesn't come from back-end processing—some of the data wrangling and algorithmic work you'd normally see only in a back-end is embedded into the RN functions with only administrative things like login on the real backend of the mobile app.

Project 2: Movie Browsing App

When rendering stuff, you can only put JSX components, nothing else. When you do use JS (eg mapping state to make a list of components), you're still only putting components after compilation of the JS.

With async functions, it's important to catch() errors, as well as validate outputs that rely on those async functions in case they have to activate before the response is fetched and you need to render them with an empty element, for example.

When commenting within a component, you must remember to put it inside {} as it only complies as a comment if it's interpreted as javascript.

Lecture 8: Expo Components, with Charlie Cheever

The premise of expo is to abstract away a lot of the functionality of mobile development and make full use of all the data and functionality collected and offered by the mobile hardware, and the docs cover everything from using maps to calendars to camera and more, whereby you simply do things like pull in 'gyroscope' and treat it like a programming object.

You can use Expo.MapView, taking args like style=flex:1, initialRegion by checking the docs (eg grabbing users location by using a Permission and setting it to a variable and then setting that to the InitialRegion, all asynchronously). You can read the docs to see things like the fact that you can use Expo.MapView.Marker which takes coords= and title= props to set a pin on the map on the mobile app. Another thing you can do is grab coords of a street using Expo.Location.geoCodeAsync("StreetName") amongst other things. You can also change Google VS Apple Maps (default) by changing the provider. This is the kind of thing you'd either indirectly use to gauge location for targeted ads, matching algorithms, etc, or directly for features like Snap Maps.

You can also do a similar thing with contacts on the users' iphone, by getting permission from Expo.Contacts, to do things like pull a random contact to text for fun in a mobile app, or something of the sort (all of these are done async because you're pulling from an external API).

You can also make a compass by pulling in images of a needle in a NSEW background and have it rotate based on the output of Expo.Magnetometer and a little mathematical trickery (check the lecture and docs if you ever need to do this). The field looking at how to combine ancillary data from many sensors is called sensor fusion, and it looks like how you can physically model, say, the 3-dimensional layout of the phone base on the gyroscope and magnetometer using some mathematical trickery. To upload images to <Image source={require("./ImageName")}. You can also use these kind of things to make the screen always point up regardless of the orientation of the phone etc, which can be useful in vision applications etc.

Make sure to remove listeners using the Unmount() lifecycle method to avoid wastage.

You can make multimedia apps using Expo.Video, with props like autoplay, size, etc. You can even use Expo functions like Expo.Audio.setAudioModeAsync to do things like have audio play even when silent mode is on, etc. You can use lots of different props given in the documentation to do things like reset the frame to the starting frame once its done running, how it looks when resizing, and more. You also need to implement functions (by using the Expo APIs) for pausing and starting when clicked, all of which are synchronous.

You can also do conditional rendering by having if() outside of the return() part of the render() method. You can await promises by using await Promise.all([promiseFunctionReturnList]) on a lifecycle method so that none of the business logic will execute before you have all the dependencies ready (thus avoiding errors like I had when making the movie browsing app).

Inside the return of render() you can also do things like {this.state.img && <Image source={require(URI)}> || null} as a form of conditional render because the && is just a neat way of rendering the image if this.state.img exists, and otherwise rendering null. You can use Expo.ImagePicker.launchCameraAsync to get the camera and then asynchronously store the images taken as a variable that you can then render() in the <Image> component, and same with camera roll. You can also manipulate the orientation etc of these images using ImageManipulator. You can also have a customizable camera interface built into your app, and create new functionality to do things like switch between the front and back views onPress of the component.

Other APIs you can use include push notifications, FB/Google login, fingerprint scanner, and more. TLDR; there are lots of abstracted methods you can use to take full advantage of the hardware and glean useful insights from these to be able to add more personalized, complex functionality to your app.

Lecture 9: Redux

The whole point of React is that the application will *react* automatically to changes in state, rendering them efficiently as needed, making it conceptually lot simpler to implement.

The problem comes with scale: Facebook started noticing bugs when they developed many chat applications in different forms—web on bottom right, pulled out from left, mobile app, web interface, and there was not completely synchronous nature between them because it was hard to keep up with what views were being changed by what controllers acting on information from what stores. And so to simplify this architecture, they invented an internal state management library called Flux, characterized by unidirectional data flow, a single source of truth, and a functional paradigm that was cleanly, and in deed clearly, modular. This inspired a modern, simpler version to emerge called Redux, which essentially operates under the same principles to make state and props management in applications of even moderate complexity and size painless and intuitive.

In redux, you have some state which is merged with a function passed in to change the state. This is dispatched/used/enacted whenever there is an update to state needing to be made, and is done aptly by the dispatcher, which is the single source of truth that makes changes to the models/database/state. It exposes/gets state for viewing using the getState() method, but can only be updated using dispatch().

Programming paradigms heavily influence development of libraries and frameworks since those are all about thinking about some given implementation in a novel, more intuitive way as well as choosing certain abstractions over others. In this case, Redux is inspired by functional programming in that the dispatcher enacts changes to state in a functional manner. Therefore, understanding these paradigms quickly cuts to understanding the heart of both why and how libraries and implemented, making picking them up much easier as everything becomes syntax and implementation details, and you can make tech stack decisions more easily as you see the pros/cons of different stacks deeply, not just to do with things like speed or ease of development.

The whole point is to make the Store an API, a black box, where you just getState and dispatch(update), and the update is applied through the function you define in the Store upon its creation. At that point, we don't care how state is managed under the hood, whether it's an object or some complicated tree for performance reasons—just that it's easy to get, manipulate, and debug.

But often times your state is nuanced enough that one variable passed in is not enough information about what in state should be changed, or exactly how—more arguments and business logic is needed. Here come in *actions,* which take *type* and *payload* which allow you to, in a functional manner, make suitable changes to the state (by calling the reducer in different scenarios, so you don't change currentUser when they are trying to add a contact to their contactsList) all under one roof which just checks which type of action was input.

A level of simplicity beyond this is taking the level which checks which *type* the action is, and embedding that into the necessary reducer itself, and then calling all the reducers relevant to some data. EG if you have several reducers that make different changes to contacts, like lenContactsReducer and addContactReducer, you could call them both every time, and they would return *state* passed in unless the *type* of dispatch action sent was GET_LEN_CONTACTS or ADD_CONTACT, respectively. These would be plopped into combineReducer() as the next paragraph details.

All of this was to understand the architecture and anatomy of redux. In reality, you don't make a store class or implement much of the functionality from scratch, but call redux library methods like createStore(). You also use combineReducer() to put all the individual reducers on the keys they alter in the state, instead of handwriting a parentReducer() function to do that for you. You also abstract each feature (actions vs reducers vs store etc) into each one of the files that redux creates for you when you spin it up.

You access state using store.getState() throughout your application, as needed, same for dispatch methods, import them and in the views on things like onPress, you just use an action you pull in from redux/actions in the same repo—hence the idea of SSoT. Still, there exists some functionality like updating all necessary props/state onStoreChange that hasn't been implemented. Of course, 'react-redux' is a library that implements higher order components that both do this, and have connect() map the state part we care about (in the case of contactsListScreen that's the contacts, not other data) to the props passed into that state, so you also don't have to rewrite any props stuff that you've written, but get the same efficiency of react which only passes things down as necessary, in a more intuitive, functional manner. Just make sure to wrap your entire app in <Provider store={store}> to make sure that it knows what store your app is linked to.

Over time, you really do come to see how the fragility and complexity of these enormous systems is something to behold. We somehow manage to get *tens of thousands of engineers, all across the world*, to contribute to a single, enormous, repository, that, to the user, takes in an input (eg Google search) and give a single, highly optimized and useful, output. There's so many things that can go wrong, so many moving parts, and human error on top of that. Having many, many abstractions, checks and balances, and a rigorous culture is what ensures that these systems achieve exactly what we want, and reliably, at that. In this sense, Google and Faceook's repos are the modern analogs of the Leuna ammonia plant, solving many technical problems in the process of being built (creating abstractions) and contributing enormous value to the modern world.

Lecture 10: Async Redux, JavaScript Tooling

Each action has an action type, and in the previous lecture we implemented action creators, which essentially took an update and returned an action, which you could then store.dispatch whenever you wanted (add a user, for example). But so far we have to go and dispatch that action ourselves, and when we're waiting on data from something like fetch we can't hardcode the dispatch, and have to create an asynchronous function.
To implement async functionality with redux, you can't change the simpleRedux implementation of store (just stores data/state) or reducers (that would make them impure and harder to use) and so you need to make an action creator, for example for login functionality (which hits some login API/some other server to authenticate credentials). To do so, you make a function that *returns a function* (HOF) and essentially dispatches an action saying LOGIN_SENT and then upon .fetching() the result, you .then dispatch LOGIN_SUCCESS, or .catch a LOGIN_REJECTED. A level of abstraction further, you pass into this function dispatch(login) and it creates then dispatches actions in the right order when it gets the right responses from external APIs.

Redux middleware is a concept important in design async functionality with Redux, and I don't fully understand it. If and when it comes time to implementing Redux in an app, rewatch that part of the lecture, do the redux tutorial and then implement it.

You should use Redux when you will be doing a significant amount of passing props and state *between* unrelated components/pages because then you can avoid bugs by having a SSOT that you reference often. If your app's architecture is not like this, then normal state management will suffice. It can be good to think through this architecture to make the decision before you start, or you can, equally, build it out and if/when you start to run into troubles (forget what state is in a particular component/struggle to pass props through multiple components) and the movement of data between components becomes even remotely confusing, spend some time switching to Redux so as you scale it only gets easier to monitor state/SSOT. In other words, if you find that:

- you're passing in lots of props
- you're forgetting to update some props or parts of state
- you're passing in state into a deeply nested component
- you're passing state/props between unrelated pages/components many times

Amongst other things, then you should switch to redux which allows you to grab state from an importable object (the store). For most projects, even medium sized ones, you should be fine without it if you just master React.

Redux is good for debugging not just because of the SSOT architecture, but also because actions, reducers, store all do different things, and so if you know there' a problem with how your data is being changed when you send an action, and it's not as intended, you can check the reducer functions to see what's going on under the hood.

Important JavaScript tools:
- NPM
- Babel transpiler
- @std/esm allows *import* in Node
- Chrome devtools (incl. for assessing performance)
- ESLint (a tool that scans your files to adhere to certain company-wide code organization decisions—consensus on use of arrow functions, styling, making sure you declare variables that are used, etc, so that everyone can swap from one codebase to another and understand everything easily without spending a long time getting used to the method and philosophy behind how others write code. ESLinst identifies the parts of the code that don't comply with a standard—which you can write yourself or extend from GitHub—and then Prettier is an integration for it that will actually change the code so it *does*). Its fixes are somewhat superficial, and it doesn't understand logic or change design decisions or abstraction decisions for you, but its a generally good tool for aesthetic/syntactic decisions.
- Prettier
- Flow/TypeScript (used for checking types, eg when you change a function prototype to take a number in as an integer and not a string but you don't change an implementation of the function dee pin your app somewhere it can be an incredible pain to discover that and make the change. By using TS (which is compiled down into pure JavaScript), you can discover these errors quickly as it is more rigid and picky with typing to make sure everything works as intended, otherwise you're notified.

You can change package.json to add scripts like lint file1 file2, and then npm run every time before starting the server so that you can see any errors that have come up in the changes that you've made/ pushed.

By looking at ESLint, Prettier, TS, you see how important a clean, uniform dev standard is because of how expensive engineering time is. You're not making good use of funding/talent if they're spending time finding and fixing niggly, non-business-logic-related bugs as opposed to quickly developing and deploying new features that "delight the customer".

Lecture 11: Performance

Always juggling *trade-offs* between performance and complexity. Most optimizations are unecessary, as they make something fast faster, which the user normally can't notice—only want to optimize in a bottleneck, and there are lots of tools built out to help find and measure these bottlenecks, tracking the UI/JS threads' performance in *perf mode* of Expo, as well as Performance in Chrome dev tools (which gives a nice flame chart/temporal bar chart of DOM rendering if you start and stop recording in the console). It's crucial to actually *use these tools* that we've found in this lecture and the previous ones—there's literally nothing you've gained form the 4 hours spent watching the two lectures if you don't use and get used to and benefit from these more efficient tools at our disposal.

SectionList is a virtualized list that only renders what you need to see at the time—built in performance handled under the hood for you.

A solid understanding of CS is important for larger organisations, especially in infra roles, because things like performance optimization are very important there, and are mostly about understanding how things are implemented and transported and manipulated as data structures under the hood via various algorithms and networking protocols, and so then you can streamline those.

One common misoptimisation is to render things too often—React takes care of most of it but if you change props unrelated to the UI often, it renders the same UI anyways, if you use lifecycle methods unnecessarily for a given UI or involving updates that could be done outside, if you don't give your long lists keys for the vDOM to sort them by, and if you, in Redux, map the *whole store* when really a certain component page needs only a small part of state, you have unnecessary re-rendering—all of which slows down the app's usability if done often enough/will make it more painful for the user. This happens more often than you'd think, with actions that are making small changes re-rendering entire data structures.

You can use class based pureComponents to build in the act of making components only update props when they've changed, and go even further to use the lifecycle shouldComponentUpdate() to check to update based on whether a *single* (or few) props/parts of state have changed.

Redundancy can also arise from changing props passed into a parent component of a subtree in the vDOM unnecessarily which causes the whole subtree to then re-render. If you have a component where you declare style/function inline (eg onPress) that is recreated each render, as opposed to using a property of the class based comp, for example.

Finally, there is often redundancy when a component mounts. Creating a function like *funcName = () =>* defines a class method like class.funcName, whereas doing funcName() {} inside a class creates a method, which happens to be inside the class (and then needs to be bound to the class). The former is re-instantiated every time an object of that class is instantiated, whereas the latter is just a method, like any other, declared once.

Animations are an interesting special case, because they use both the UI and JS thread several times per second to function smoothly, and each time this is done data needs to be serialized etc, and this is an expensive operation. Blocking/occupancy on either thread causes problems since its a constantly running thing. You can use an inbuilt API called Animation to make these computations run on the native thread and render on the JS/UI threads, done by declaring a new Animation.value and following the docs there by executing Animated stuff every time ComponentDidMount().

It's easy to fall into the trap of thinking that performance is some esoteric worry that only infra engineers at large companies need to worry about, but a general thing to be aware of in designing systems is that the scale and scope of problems rises very much, very fast—in fact, much faster than most computation can be done. By tweaking a problem statement slightly, you go from taking 0.1 millisecond to 1 second, which doesn't seem like a problem on the user side, but is a 10^4 order change—one more of those tweaks/grievances and your users are waiting hours for a simple change. Optimise early, optimise early, optimise often.

Lecture 12: Deploying & Testing

You deploy by building in Expo then uploading to app store (see docs for more info) and you can simply upload new JS to give the users who have your app over the air updates after testing the changes rigorously on the local server.

There are unit, integration, and UI (e2e) tests, all helped by the Jest testing framework, which you can run with npx jest. You can import functions you want to test into a test.js file and use something like:

const sum = require('./sum.js')
test('sums 1 and 1', () => {

```
                    expect(sum(1, 1).toBe(2))
          })
```
And make scenarios testing it in many ways. Aside from more readable tests and output, Jest also automatically runs whenever a file in which a function you imported changes, and tells you whether the changes break anything or not. You can also add script one-liners like "test:watch": "jest - -watch" in package.json to be able to use those abbreviations in CLI. Also gives you useful functions to do things like compare objects using toEqual instead of === since that compares objects by reference.

In the case of functions we often don't want to copy/paste many tests rewriting the whole function in the expect() and so you can simply append .toMatchSnapshot() after the expect to have it compare the output to the historical output of the function, and will simply notify you when something *changes* as opposed to doesn't match some hardcoded output, and you can update this desired snapshot comparison point with npx jest - - - -u. You can then group many similar tests in a describe() group and abstract functionality away further.

You can test async functionality by using Jest's mock functions in case you use functions that rely on third party APIs/libraries/services, you can cut those out and have it test your core functionality simulating some response from the library.

Beyond unit tests like the above, you can do integration tests by running components outside of an app using a React-test-renderer library made specifically for this purpose. You check these by matching snapshots, same as before. You can make a describe() group that has several tests, incrementally increasing sophistication of functionality, eg when developing a button you have a test *it('renders),* and then *it('redirectsOnPress')* etc to see exactly which step something is failing in when you run it and it fails some test.

Test-driven development is using the tests more as a specification guiding what IO should be in the finished product as opposed to an afterthought to make sure they do what we want.

Jest even tells you your coverage % so you know how much has been covered, no matter how you spin it from lines of code to branches to functions, and gives you a table for how much each file you've hit has been tested. There's also a sick library called detox being made by Wix that does a complete automated e2e test of your app as if it were a user.