**Final Exam Review**

This is an outline for the final exam. Please review slides as well because all lecture material is fair game for the final exam.

\*\*If you have an ADS form and intend on using your accommodation, you must schedule your exam with the [testing center](https://www.towson.edu/accessibility-disability-services/testing.html). Due to scheduling conflicts, you will not be able to get accommodations if the exam is taken in the classroom at the regular time. Please schedule this a week in advance to avoid any issues.

1. **NodeMailer:**

* **Nodemailer is a popular Node.js module for sending email from your server. It lets you configure transport and then send messages in a few lines.**

**Code Example:**

**// install: npm install nodemailer**

**const nodemailer = require('nodemailer');**

**async function sendTestEmail() {**

**// 1. Create a transporter. For real apps, secure your credentials!**

**let transporter = nodemailer.createTransport({**

**host: "smtp.example.com", // your SMTP server**

**port: 587, // usually 587 or 465**

**secure: false, // true for 465, false for other ports**

**auth: {**

**user: "your\_user", // SMTP user**

**pass: "your\_pass", // SMTP password**

**},**

**});**

**// 2. Define the email options**

**let mailOptions = {**

**from: '"Sender Name" <sender@example.com>',**

**to: "recipient@example.com",**

**subject: "Hello from NodeMailer",**

**text: "This is a plain-text body",**

**html: "<b>This is HTML body</b>",**

**};**

**// 3. Send it**

**let info = await transporter.sendMail(mailOptions);**

**console.log("Message sent: %s", info.messageId);**

**}**

**sendTestEmail().catch(console.error);**

* 1. **Be able to send a simple email using NodeMailer**
* **Check above**
  1. **Know the three main protocols and their functionality**

**1. POP3 (Post Office Protocol v3)**

**How it works**

**Connect to the mail server.**

**Download all new messages to your device (e.g. your desktop mail client).**

**By default, delete those messages from the server after download.**

**Disconnect.**

**Key characteristics**

**One-way transfer: server → client only.**

**Stateless on the server: once downloaded (and deleted), the server “forgets” about those messages.**

**Simple protocol, minimal server-side storage or syncing.**

**Example scenario**

**You check mail on your laptop at home. POP3 downloads 20 new messages and removes them from the server.**

**Later, on your phone, you try to check mail again. Since the server copy is gone, your phone sees zero new messages—you only have those 20 on your laptop.**

**Use POP3 when:**

**You have one primary device for mail.**

**You want to archive everything locally and don’t need to keep mail on the server forever.**

**Server storage is limited and you want to clear it out automatically.**

**2. IMAP (Internet Message Access Protocol)**

**How it works**

**Connect to the mail server.**

**List folders (Inbox, Sent, Drafts, etc.).**

**Fetch only the headers or bodies of new messages on demand.**

**Mark messages as read/unread, flagged, moved, or deleted—and these state changes live on the server.**

**Disconnect (or stay connected to receive updates).**

**Key characteristics**

**Two-way sync: any action you take (read, delete, move) is reflected on the server and thus on every device.**

**Stateful: server keeps track of which messages are read, flagged, in which folder, etc.**

**Selective download: you can fetch just headers first, then download full bodies or attachments as needed.**

**Example scenario**

**You read an email on your phone and mark it “read.”**

**Later on your laptop, you open your mail app and see that same message already marked “read.”**

**You move it into a “Project X” folder on your laptop—and on your tablet it’s also in “Project X.”**

**Delete one on your desktop, and it’s gone from all devices.**

**Use IMAP when:**

**You check mail from multiple devices (phone, laptop, tablet).**

**You need to keep your inbox/folders organized consistently everywhere.**

**Server storage is sufficient, and you want to offload local archiving.**

**2. IMAP (Internet Message Access Protocol)**

**How it works**

**Connect to the mail server.**

**List folders (Inbox, Sent, Drafts, etc.).**

**Fetch only the headers or bodies of new messages on demand.**

**Mark messages as read/unread, flagged, moved, or deleted—and these state changes live on the server.**

**Disconnect (or stay connected to receive updates).**

**Key characteristics**

**Two-way sync: any action you take (read, delete, move) is reflected on the server and thus on every device.**

**Stateful: server keeps track of which messages are read, flagged, in which folder, etc.**

**Selective download: you can fetch just headers first, then download full bodies or attachments as needed.**

**Example scenario**

**You read an email on your phone and mark it “read.”**

**Later on your laptop, you open your mail app and see that same message already marked “read.”**

**You move it into a “Project X” folder on your laptop—and on your tablet it’s also in “Project X.”**

**Delete one on your desktop, and it’s gone from all devices.**

**Use IMAP when:**

**You check mail from multiple devices (phone, laptop, tablet).**

**You need to keep your inbox/folders organized consistently everywhere.**

**Server storage is sufficient, and you want to offload local archiving.**

1. **Cheerio**
   1. **If given an HTML page, be able to parse it and return some result**

* **Cheerio is a jQuery like library for server-side HTML parsing and manupilating in Node.js. It loads HTML into a lightweight DOM you can querry with familiar CSS selectors just like jQuery but it runs in Node, without a browser.**
* **Example cheerio**
* const cheerio = require("cheerio");
* const axios = require("axios");
* const url = "http://quotes.toscrape.com/";
* async function scrapPage(url) {
* //accessing the website
* try {
* const { data: html } = await axios.get(url);
* const $ = cheerio.load(html);
* $(".author").each((i, element) => {
* if (i < 10) {
* console.log("The Author is: ", i + 1, ": ", $(element).text());
* }
* });
* } catch (error) {
* console.error("The error: ", error);
* }
* }
* scrapPage(url);

1. **Package.json**
   1. **The advantages of using it**

* **Package.json is a JSON-formatted manifest file at the root of your project that describes your project (name, version, description, author, license, etc.). List dependencies your code needs to run. Defines development only dependencies. Specifies scripts you can run with like npm run <name>. It has reproducible installs. Package-lock.json file lists the exact versions needed and will install on npm install command. npm ci (short for clean install) will remove your existing node\_module and only install exactly what is in package-locak.json. It is faster and ensures a completely fresh reproducible install.**

1. **React**

* **React is a JavaScript library focused on building user interfaces. You build the UI by using small, reusable components that manage their own state and render themselves. You describe what the UI should look like for a given state and React takes care of updating the DOM when that state changes.** 
  1. **Advantages of using React**
     1. **Virtual DOM**
* **React keeps an in-memory “virtual” representation of your UI.**
* **When your data changes, it diffs the new virtual tree against the old one and patches only what’s different in the real DOM—minimizing costly updates.**
  + 1. **Component-Based Architecture**
* **You break your UI into small, reusable components (e.g. <Header />, <TodoItem />).**
* **Each component owns its own markup, styling, and behavior, making your code easier to reason about and maintain.**
  + 1. **Declarative Syntax**
* **You describe what the UI should look like given some state, and React figures out how to update the DOM when that state changes.**
* **This leads to more predictable, bug-resistant code compared to manually manipulating the DOM.**
  + 1. **Rich Ecosystem & Tooling**
* **First-class support for routing (React Router), state management (Context API or Redux), and testing (Jest + React Testing Library).**
* **Create-React-App or Vite setups give you instant hot-reload, linting, and build pipelines out of the box.**
  1. **Differences between JSX and JavaScript**
     1. **Be able to write an interpret JSX code**
* **JSX – stands for JavaScript eXtensible Markup language. It allows you to right HTML like tags in your JavaScript files. It is a syntactic sugar. Under the hold uses tools like Babel to transform JSX into JavaScript understandable call for rendering.**
* **Example of JSX code:**

**function Greeting({ name }) {**

**return (**

**<div>**

**<h2>Welcome, {name}!</h2>**

**<button onClick={() => alert('Hi')}>Say Hi</button>**

**</div>**

**);**

**}**

* **This what that same code will look like when it is compiled to JavaScript version**

**function Greeting(props) {**

**return React.createElement(**

**'div',**

**null,**

**React.createElement('h2', null, 'Welcome, ', props.name, '!'),**

**React.createElement(**

**'button',**

**{ onClick: function() { alert('Hi'); } },**

**'Say Hi'**

**)**

**);**

**}**

* **Key Rules in JSX**
  + **Component Names Must Be Uppercase: <MyComponent /> vs. <mycomponent />**
  + **Use className not class, because class is a reserved JS word.**
  + **Self-closing tags need the slash: <img src="..." />.**
  1. **Be able to write and interpret actual React code**

**// Greeting.jsx**

**import React from 'react';**

**function Greeting(props) {**

**// 1. props is an object passed from the parent**

**// 2. We can destructure for convenience:**

**const { name, onWave } = props;**

**// 3. Return JSX describing the UI for this component**

**return (**

**<div>**

**<h1>Hello, {name}!</h1>**

**<button onClick={onWave}>👋 Wave</button>**

**</div>**

**);**

**}**

**export default Greeting;**

**What’s happening here?**

1. **Component definition:**
   * **function Greeting(props) declares a function component named Greeting.**
   * **Components must start with a capital letter.**
2. **Props:**
   * **props is the object containing data and callbacks passed from the parent.**
   * **In this case we expect { name: string, onWave: () => void }.**
3. **Destructuring:**
   * **const { name, onWave } = props; pulls out those specific values for easier use.**
4. **JSX return:**
   * **The <div> with a single root wraps an <h1> and a <button>.**
   * **{name} embeds the name value.**
   * **onClick={onWave} wires the button’s click event to the onWave callback.**

**Using this component**

**In your App.jsx you might write:**

**import React from 'react';**

**import Greeting from './Greeting';**

**function App() {**

**const sayHi = () => alert('Hi there!');**

**return (**

**<div>**

**<Greeting name="Abel" onWave={sayHi} />**

**<Greeting name="Siham" onWave={sayHi} />**

**</div>**

**);**

**}**

**export default App;**

** You render <Greeting /> twice with different name props.**

** Both buttons use the same sayHi function when clicked.**

* + 1. **Components**
       1. **Why components are important**
* **A component is important because it is a reusable self-contained piece of UI, Its ability to encapsulate everything in the component like the markup, styling and logic. You can assemble complex UI by nesting components. It is also the best when it comes to maintainability. Because the smaller files are focused on piece.**
* **A component example:**

**// Greeting.jsx**

**import React from 'react';**

**/\*\***

**\* A simple component that says hello to a given name.**

**\* - props.name: the name to greet.**

**\*/**

**function Greeting(props) {**

**return (**

**<div className="greeting">**

**<h2>Hello, {props.name}!</h2>**

**</div>**

**);**

**}**

**export default Greeting;**

* + 1. **Props**

**Props (short for “properties”) are the mechanism React uses to pass data into a component from its parent. Props (short for “properties”) are the mechanism React uses to pass data into a component from its parent.**

**How Props Work**

1. **Parent passes props via JSX attributes:**

**jsx**

**<Greeting name="Abel" age={30} />**

1. **Child component receives them in its function signature:**

**jsx**

**function Greeting(props) {**

**// props === { name: "Abel", age: 30 }**

**return <h1>Hello, {props.name}! You are {props.age}.</h1>;**

**}**

1. **Destructure for clarity:**

**jsx**

**function Greeting({ name, age }) {**

**return <h1>Hello, {name}! You are {age}.</h1>;**

**}**

**Why Props?**

* **Data Flow: Keeps data flowing downward from parent → child, making your UI predictable.**
* **Reusability: You can reuse the same component for different inputs:**
  + **<Greeting name="Abel" age={30} />**
  + **<Greeting name="Siham" age={28} />**
* **Immutability: Props are read-only inside the child. To change data you lift state to a common parent.**

**// UserCard.jsx**

**function UserCard({ username, email }) {**

**return (**

**<div className="card">**

**<h2>{username}</h2>**

**<p>{email}</p>**

**</div>**

**);**

**}**

**// App.jsx**

**function App() {**

**const user = { username: "Abel", email: "abel@example.com" };**

**return <UserCard username={user.username} email={user.email} />;**

**}**

* + 1. **State**

**1. What Is State?**

* **State holds data that a component owns and can change over time.**
* **Unlike props (which a component receives from its parent and must not modify), state is private and mutable within the component.**

**2. The useState Hook**

* **In function components, React gives us the useState hook to add state:**

**import React, { useState } from 'react';**

**function Counter() {**

**// 1. Declare a state variable `count`, initialized to 0.**

**// `setCount` is the function you call to update it.**

**const [count, setCount] = useState(0);**

**// 2. Render the UI based on `count`**

**// and provide a way to update it via the button.**

**return (**

**<div>**

**<p>Current count: {count}</p>**

**<button onClick={() => setCount(count + 1)}>**

**Increment**

**</button>**

**</div>**

**);**

**}**

**useState(0)**

* **The argument 0 is the initial state.**
* **Returns a two-element array: [stateValue, updaterFunction].**

** count**

* **Holds the current state value.**
* **You can read it anywhere in your component.**

** setCount**

* **Call this to enqueue a state update.**
* **React will re-render the component with the new count.**

**Why State?**

* **Interactive UIs: Track form inputs, toggle visibility, manage counters, etc.**
* **Reactivity: When you call the updater function (in the example that is setCount), React schedules a re-render so the UI stays in sync with the data.**
  + 1. **Conditional Rendering**
* **Conditional rendering in React means you show or hide parts of your UI based on state or props. Instead of manually manipulating the DOM, you express what should be rendered when certain conditions are met.**
* **Using the Ternary Operator**

jsx

function Greeting({ isLoggedIn }) {

return (

<div>

{isLoggedIn

? <h1>Welcome back!</h1>

: <h1>Please sign in.</h1>

}

</div>

);

}

* Using the Logical AND Operator

Great for “show this only when true”:

jsx

function Alert({ message }) {

return (

<div>

{message && <p className="alert">{message}</p>}

</div>

);

}

* If message is a non‑empty string, the <p> appears; if message is falsy ("", null, undefined), nothing renders.

**If/Else Inside the Component**

When logic gets more complex, you can decide before returning:

jsx

function StatusMessage({ status }) {

let msg;

if (status === 'loading') {

msg = 'Loading…';

} else if (status === 'error') {

msg = 'Something went wrong.';

} else {

msg = 'Data loaded successfully.';

}

return <p>{msg}</p>;

}

* 1. **React Native**
     1. **How good mobile hybrid solutions work vs bad ones**
* **How good mobile hybrid solutions work vs bad ones. Hybrid apps let you use web technologies like JSm HTML and CSS to build mobile apps across platforms. React Native strikes a balance by compiling to real native components rather then running in a WebView**

| **Aspect** | **Good Hybrid (e.g. React Native)** | **Bad Hybrid (WebView-based)** |
| --- | --- | --- |
| **UI Performance** | Renders real native widgets (e.g. <Text>, <View>) via bridge; smooth 60 fps animations. | Renders HTML/CSS in a WebView—janky scrolling, input lag. |
| **Access to Native APIs** | Direct modules or community plugins give near-full access to camera, sensors, files. | Limited or slow via JavaScript bridges; often need custom WebView plugins. |
| **Developer Experience** | Hot Reloading / Fast Refresh; you write one codebase but get two native apps. | You write web code, but debugging device quirks in a WebView can be painful. |
| **Look & Feel** | Uses each platform’s native controls and styling conventions by default. | UI feels “foreign”—web widgets don’t automatically match iOS/Android patterns. |
| **Bundle Size** | App bundle includes needed JS and native modules—kept fairly small with code splitting. | Often bloated by shipping a full browser engine inside the app. |
| **Community & Ecosystem** | Large ecosystem of native-bridge libraries (e.g. React Navigation, Reanimated). | Mostly generic JS libraries—few tackle device-specific UX nuances. |

* + 1. **The Message Queue**
       1. **Its importance and functionality**
* **In ReactNative, JavaScript runs in its own thread separate from the native UI thread. The message Queue also called the bridge is how these two worlds communicate.**

**Why the Bridge?**

* Native views can’t directly execute JS, and JS can’t directly manipulate native objects.
* The **bridge** serializes messages (method calls, property updates, events) into JSON and passes them back-and-forth.

1. **Docker**
   1. **Why is Docker useful when dealing with full-stack applications?**
      1. **Team collaboration**
   2. **Explain what a Docker Registry is**
      1. **Be able to list two**
   3. **Best version of Linux for simple Node.js app containerization**
   4. **Very basic commands of a Dockerfile**
2. **GraphQL**

* **GraphQL is an open-source query language for APIs and a corresponding runtime for executing those queries against your data. It was created by Facebook in 2012 and publicly released in 2015.**
  1. **How is GraphQL different than a RESTful API**
     1. **The advantages of using GraphQL**

 **Endpoint Structure**

* **REST:** Multiple endpoints (/users, /posts, /users/1/posts, etc.), each returning a fixed data shape.
* **GraphQL:** Single /graphql endpoint. Clients specify in each query exactly which fields and nested data they want.

 **Data Fetching**

* **REST:** Can cause **over-fetching** (getting fields you don’t need) or **under-fetching** (needing several round-trips for related data).
* **GraphQL:** Eliminates both: you request precisely the data you need—even deeply nested—in one round-trip.

 **Typing & Schema**

* **REST:** Often loosely typed or documented via OpenAPI/Swagger. The server defines response schemas per endpoint.
* **GraphQL:** Built on a **strongly-typed schema** that describes every object, field, and argument. Clients can **introspect** the schema for real-time docs and autocompletion.
  1. **How can GraphQL and a pre-existing RESTful API work together?**

**Create a GraphQL Layer**

Stand up a new GraphQL server (e.g. Apollo Server or express-graphql) alongside your existing REST service, exposing a single /graphql endpoint.

**Define Your Schema**  
Match your REST resources with GraphQL types and fields. For example, if you have /users/:id and /users/:id/posts, your schema might include:

graphql

CopyEdit

type User {

id: ID!

name: String!

posts: [Post!]!

}

type Post {

id: ID!

title: String!

body: String

}

type Query {

user(id: ID!): User

}

**Write Resolvers that Call REST**  
In each resolver, make HTTP calls to the underlying REST API:

js

CopyEdit

const resolvers = {

Query: {

user: async (\_, { id }) => {

const res = await fetch(`https://api.example.com/users/${id}`);

return res.json();

}

},

User: {

// When a client asks for “posts” on a User, fetch from REST:

posts: async (parent) => {

const res = await fetch(

`https://api.example.com/users/${parent.id}/posts`

);

return res.json();

}

}

};

**Why This Helps**

* **One endpoint** (/graphql) for all reads
* **Nested queries** let clients fetch a user and their posts in a single round-trip
* You can **incrementally adopt** GraphQL without rewriting the REST backend
  1. **Setting up a simple Schema given a scenario**
     1. **Schemas with relations**

Let’s take a small scenario—say a blogging platform with **Authors** and their **Posts**—and define a GraphQL schema that models their relationship.

graphql

CopyEdit

# 1. Define the Author type

type Author {

id: ID! # unique identifier

name: String! # author’s name

bio: String # optional biography

posts: [Post!]! # relation: an author’s list of posts

}

# 2. Define the Post type

type Post {

id: ID! # unique identifier

title: String! # post title

content: String # post body

author: Author! # relation: the post’s author

}

# 3. Root Query type

type Query {

# Fetch a single author by ID

author(id: ID!): Author

# Fetch all posts (optionally filter by author)

posts(authorId: ID): [Post!]!

}

**What’s happening here?**

1. **Author.posts: [Post!]!**
   * Implies each Author object has a field posts returning a non-null list of non-null Post objects.
   * Establishes a one-to-many relationship: one author → many posts.
2. **Post.author: Author!**
   * Each Post has exactly one Author.
   * This back-reference lets clients “nest upward” in queries.
3. **Query.posts(authorId: ID): [Post!]!**
   * By accepting an optional authorId argument, clients can ask for all posts or restrict to a specific author.
   1. **Queries**
      1. **Be able to make Queries given a Schema**
         1. **Nested Queries for specific data as well**
   2. **Mutations and how they work**
   3. **Subscriptions and how they work**
   4. **Setup a simple GraphQL endpoint**