React is a framework (library) designed to help us solve this exact problem. That is build highly interactive and “reactive” UIs. The key idea is to decouple the render of the current state from the updates to that state. You just need to answer to different questions. What do I want to the UI to look like at any given moment – more formally for any given state of the application – and how do I update that state based on user actions. We don’t have to answer the much trickier question of how do we want to update the UI in response to user actions. React takes care of efficiently propagating those state changes to the UI.

### Philosophy of React

1. Render the UI as it should appear for any given state of the application
2. Update the state as a result of user actions
3. Repeat (i.e. re-render UI with new state)

*The key conceptual idea is that those two steps are now decoupled and so simpler*

*The key technical enabler was efficient re-rendering when the data changes*
“Thinking in React”

1. Break the UI into a component hierarchy
2. Build a static version in React
3. Identify the minimal (but complete) representation of state
4. Identify where your state should live
5. Add “inverse” data flow (data flows down, callbacks flow up)

https://reactjs.org/docs/thinking-in-react.html
Recall that our state is just the three color components. Where should this state live (step 4 in “Thinking in React”)? We need this information in the sliders, i.e. in LabeledSlider, but also in ColorPicker to set the swatch color. Per the React documentation: “Often, several components need to reflect the same changing data. We recommend lifting the shared state up to their closest common ancestor.” Thus we will implement the state in the ColorPicker component (the closest common ancestor).

That state then “flows down” to the labeled sliders as props to those components. React components must act like pure functions with respect to their props. That is a component can't modify its props (this enables efficient updates). To communicate updates "back up" we supply a callback to the child that modifies the state in the parent (the “inverse” data flow or step 5 in Thinking in React).

Some important notes about modifying state:
• Do not modify state directly, instead use the setter.
• State updates may be asynchronous. React may batch updates, and so you shouldn't assume the state has actually changed after the call to the setter.
create-react-app (CRA) sets up a fully functional React application with all the necessary supporting infrastructure (e.g. Webpack for packaging production assets and Babel for transpiling). CRA is not the only way to setup a React application, and as with some of our other tools, e.g. the AirBnB ESLint configuration, we may not agree with some of the choices made by the CRA developers. However, it provides a robust starting point with many best practices built in.

For example, that compilation infrastructure is what makes it possible for use to JSX. Recall that since JSX is an extension to JavaScript, we will need a compiler to convert it to standard JavaScript. CRA integrates the Babel compiler to transpile JSX (and support features of ES6).
As we noted, CRA has integrated all of the tools we need to use recent JavaScript features (and have a pleasant experience doing so) and optimize our application for production. Generally speaking we don’t need to know a lot about it works under the covers (although the documentation is pretty good!).

**Typical CRA workflow**

1. Clone existing application repository
   
   `or npx create-react-app my-app` to create a new application

2. Install dependencies with npm install

3. Run development server with npm start
   
   Your application will automatically reload when you make changes

4. Run tests with npm test
   
   The tests will automatically rerun when you make changes
Typical CRA folder structure

my-app/
  README.md
  node_modules/
  package.json
  public/
    index.html
    favicon.ico
  src/
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg

NPM package infrastructure (present in every npm package)

All JS and CSS files must go in this directory. All of your work will happen in here.

Tests for the App component
Recall: Our test hierarchy

- Typescript or Flow annotations
- Linter
- PropTypes (dynamic)

Kent C Dodds “Write tests. Not too many. Mostly integration.”
The more specific we can make these requirements the more likely we are to catch type errors (generally true for all kinds of validation). Note that validation isn't the only purpose for providing `PropTypes`. Doing so is also a way of documenting the "type signature" of the component (analogous to a function signature in a statically typed language).

The code snippet provided demonstrates the use of `PropTypes` in a React component, specifically the `LabeledSlider` component. The `PropTypes` object is used to define the expected types for the properties `label`, `value`, and `setVal`. The `isRequired` property is also used to specify that these properties are required.

The code snippet includes a comment indicating a "code smell," suggesting that the `value` property should only accept strings or numbers, but the current implementation allows an object to be passed in, which is not ideal.

To catch errors and document component "signature," the code snippet provides an example of how to use `PropTypes` in a React component, specifically the `LabeledSlider` component. The `PropTypes` object is used to define the expected types for the properties `label`, `value`, and `setVal`. The `isRequired` property is also used to specify that these properties are required.

The code snippet includes a comment indicating a "code smell," suggesting that the `value` property should only accept strings or numbers, but the current implementation allows an object to be passed in, which is not ideal.

The documentation link provided is: [https://reactjs.org/docs/typechecking-with-proptypes.html](https://reactjs.org/docs/typechecking-with-proptypes.html)
A couple bits of syntactic sugar at work:

- Destructuring to “split” props object into its component properties, e.g. `const { record, submit } = props;`
- Spread operator to create a new record object (the ...record) part and then overwrite that with a new value for the name property.

By reviewing this code we can make inferences about the props and thus what types to specify. Note that if a prop is optional, as record is here, we want to specify a default value (even if that default value) is just null. ESLint should warn you about missing default props.
How can we style our application

- Static CSS files
- “Import” CSS files like code
  ```javascript
  import './ColorPicker.css'
  ```
- CSS-in-JS
  ```javascript
  const ColorLabel = styled.div{
    display: inline-block;
    width: 50px;
    text-align: left;
  ;
  }

  <ColorLabel>{props.label}</ColorLabel>
  ```

  “Style as code”

- We can include a static CSS file as an asset, i.e. the traditional approach and what you saw in CodePen examples. But this approach is not very modular and doesn’t necessarily work well with a component-based design as we would to have merge the styles for all components.
- We can “import” CSS files (using features of Webpack to bundle that CSS into the JavaScript file) for each component. The challenge is that by default the imported CSS exports all class names into the global selector scope creating potential for naming collisions.
- Implement CSS-in-JS. CSS-in-JS integrates styling into the components as JavaScript code (similar to our previous example in which we created the styles as JavaScript objects but with many more features, like handling differences in browsers).

The latter will be our preferred approach. Here we use the styled-components library to create React components that incorporate CSS. As we will see in a moment, as React components, our styled components can take props that change the styles. Here we are specifying that ColorLabel is a component, built around an HTML div element that has the styling shown. We can then use it (ColorLabel) like any other React component.
Separation of Concerns (SoC) will be a recurring topic this semester, but in short, SoC is a design principle that each “unit” in a program should address a different and non-overlapping concern.

In this context, a common SoC argument around HTML/CSS is that HTML should specify content (only) and CSS should specify the style (only), i.e. separate style from content. Proponents of CSS-in-JS also make a SoC argument, but that one component should be entirely separate from the others.
Deployment: Closing the loop

Programs that are never deployed have not fulfilled their purpose. We must deploy!

To do so we must answer:

• Is our application in a working state?
• Do we have the necessary HW/SW resources?
• How do we actually deploy?
Continuous Integration (CI)

- Maintain a single repository
  * With always deployable branch
- Automate the Build (Build is a proper noun)
  * And fix broken builds ASAP
- The Build should be self testing
- Everyone integrates with master frequently
  * Small “deltas” facilitate integration and minimize bug surface area
- Automate deployment
  * Practice “DevOps” culture

Martin Fowler “Key practices of Continuous Integration”

CI emphasizes frequent small integrations (hence the name)

There are two related concepts:
* Continuous Deployment: Every change automatically gets put into production, and thus there are many production deployments each day.
* Continuous Delivery: An extension of CI in which SW is deployable throughout its lifecycle, the team prioritizes keeping SW deployable, and it is possible to automatically deploy SW on demand.

https://martinfowler.com/bliki/ContinuousDelivery.html

We will be aiming for a Continuous Delivery-like workflow in which our applications start and stay deployable throughout the development process. As with CI, this reduces the complexity (and risk) of deployment by enabling us to do so in small increments. And Continuous Delivery facilitates getting user feedback by frequently getting working SW in front of real users. Although to mitigate risk companies will often first deploy for a small subset of users.
Git workflow for CI

- Branching is cheap in Git
- We will use features branches to segregate changes until integration
- The “master” branch remains deployable

Master is always “deployable”
- Tests pass
- No incomplete features

Short-lived branch for single feature

https://www.atlassian.com/git/tutorials/using-branches
But we are rarely working alone. On a team we need to make sure we stay in sync and create opportunities to get a second pair of eyes on our code (i.e. create opportunities for code review).
Git/GitHub workflow with CI

- Alice
  - `git branch -d feature`
  - `git push origin feature`

- Github
  - `git checkout master`
  - `git pull --prune`
  - CI server tests branch and merged code
  - Merge PR
Student advice: Branch-per-feature

- “Aggressive branch-per-feature minimized merge conflicts”
- “With this many people you NEED branch-per-feature to avoid stepping on each other”

Our goal is to work efficiently as a project team. *Practice now the processes you will need in your project!*

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
True or False? The development team's goal of launching new features is in conflict with the operations team's goal of ensuring services stay live and usable.

Answer: A

From the Google SRE handbook: "At their core, the development teams want to launch new features and see them adopted by users. At their core, the ops teams want to make sure the service doesn’t break while they are holding the pager. Because most outages are caused by some kind of change—a new configuration, a new feature launch, or a new type of user traffic—the two teams’ goals are fundamentally in tension."
In furtherance of the first principle, in some settings there is a single team that is responsible for the entire application lifecycle from development to testing to deployment. The role of automation is to improve efficiency, reduce the chance for human error and provide always up-to-date documentation of the workflow.

DevOps: A preview

- Involvement of the operations function in each phase of a system’s design and development
- Heavy reliance on automation versus human effort
- The application of engineering practices and tools to operations tasks
The operational work involved in supporting a service should realistically scale how as the service grows by 10X?

A. \( O(1) \): Just one time efforts to add resources  
B. Sublinear: There will be additional work required as a function of service size  
C. \( O(n) \): The effort will have to grow linearly with demand  
D. Greater than \( O(n) \): Increasing scale means increasing complexity

Answer: A

Again from the Google SRE handbook: "An ideally managed and designed service can grow by at least one order of magnitude with zero additional work, other than some one-time efforts to add resources." To do so, one needs highly automatic systems.

Automation (and engineering practices) are what enables that constant effort scaling. Automation goes beyond just provisioning resources, it is techniques like automatically rolling out changes to a small fraction of users, detecting errors (through monitoring) and then automatically rolling back the changes!

Although DevOps wasn't a thing (and thus not a job), the role of site reliability engineer (SRE) is the closest to DevOps as a job. Popularized by Google, SREs are engineers who focus on running products and "create systems to accomplish the work that would otherwise be performed, often manually, by sysadmins."
As described previously the 3-tier architecture is a design pattern. PaaS factor out the common elements of that architecture. For example the Heroku PaaS provides the “presentation tier” and the “persistence tier” and the portions of the “logic tier” that wrap around your specific application.

*aas (e.g. PaaS) “factor out” the common needs*

*aaS and the cloud has eliminated all physicality from the process but also change the dynamic from provisioning (and decommissioning) HW infrequently to doing so frequently forcing automation (even by otherwise small-scale users).

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.