The static ⇒ dynamic transition made it easier for us to “stamp” multiple color pickers, transition to 4 color components (CMYK) and more. But it didn’t solve our state management problem. We still had lots of state and complicated interconnection (connecting slider to color swatch and span).

What if we wanted to change the span to a numeric input that could also change the color? How would you do so? The state management (and update) would get even more complicated.

How to implement an application in React (from Thinking in React)
1. Break the UI into a component hierarchy
2. Build a static version of react
3. Identify the minimal (but complete) representation of state
4. Identify where you state should live
5. Add inverse dataflow (data flows down, callbacks flow up)
“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
React has 3 main ideas (in my view...). I want to focus on #2, when implementing render think about rendering the HTML you want for the current state. Instead of worrying about binding, interaction, etc. just focus on rendering the UI you want. Then add the callbacks (inverse dataflow) that update that state (and trigger the rendering of an updated view).

The advantage of design patterns is that you can abstract out the parts of your solution that are the same every time. In the case of React, it handles efficiently re-rendering the HTML on the screen (the part that is the same in very application) so that you can focus on your application code.
Concisely expressing the view: JSX

// Example HTML
var heading = React.createElement("h1", null, "Hello, world!");

// Example component
var person = React.createElement(Person, {
    name: p.name,
    address: p.addr
});

// Example HTML
const heading = <h1>Hello, world!</h1>;

// Example component
const person = <Person name={p.name} address={p.addr} />

Props
Revisiting the color picker

```
render() {
  const { red, green, blue } = this.state;
  const colorBoxStyle = {
    width: '100px',
    ...  
    background: `rgb(${red},${green},${blue})`  
  };
  return (  
    <div>  
      <div id="color" style={colorBoxStyle} />  
      <LabeledSlider  
        label="Red"  
        value={red}  
        valueChange={value => {  
          this.setState({ red: value });  
        }}  
      />  
      ...  
    </div>  
  );
}
```
React from the top-down

create-react-app integrates

- Babel
- webpack
- Enzyme
- Airbnb
- ESLint
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).
How can we style our application

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

- We can include a static CSS file as an asset, i.e. the traditional approach. 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 int
- 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). o global selector scope creating a potential for naming collisions.
SoC is a design principle that each "unit" in a program should address a different and non-overlapping concern.

- HTML is content (only),
- CSS is style (only),
- Each component should be separate.

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!*

But we must answer:

- Is our application in a working state?
- Do we have the necessary HW/SW resources?
- How do we actually deploy?
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

- git push origin feature
- git branch -d feature
- 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 Berkeley CS169

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

A. True
B. False

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

- 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.