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
Props flow down, callbacks flow up

const [red, setRed] = useState(0);
...

Props

Callbacks

Red: 0
Green: 0
Blue: 0

Single source of truth!
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.”
PropTypes in action

LabeledSlider.propTypes = {
    label: PropTypes.string.isRequired,
    value: PropTypes.oneOfType([PropTypes.string, PropTypes.number]).isRequired,
    setValue: PropTypes.func.isRequired,
};

Bit of a “code smell”

Catch errors and document component “signature”
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>

“Style as code”
Really a debate about separation of concerns (SoC)

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
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”
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
Git “solo” branching workflows

- `git checkout -b feature`
- `git checkout master`
- `git merge feature`
- `git commit -m "..."`
- Make sure tests pass
Git/GitHub workflow with CI

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

2. Github
   - PR
   - CI server tests branch and merged code
   - `git checkout master`
   - `git pull --prune`
   - 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
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.
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
Client (e.g. browser) - Internet - Site

Client-Server

HTTP & URI

HTML, JSON, ...

3-tier Architecture

Web Server (e.g. Apache, NGinx)

Presentation Tier

App. Server (e.g. NodeJS)

Logic Tier

Database (e.g. Mongo, PostgreSQL)

Persistence Tier

Routing & Controllers (e.g. Express)

Models (e.g. knex, objection)

MVC

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