

UI Patterns in React

- We've seen UI patterns in plain HTML/CSS/JS
- How does React change things?
 - Components
 - CSS per component?
 - incl. media queries!
 - Reusable
 - state variables in component
 - passed props to component
 - HTML based on state/props
 - classes based on state/props
 - Output HTML based on state/props

Reusable Components

- So far Components containers for custom content
 - Used to organize
- "Reusable" components also exist
 - Can be used multiple times
 - For similar output
 - Customized based on values passed
 - Organize a different way
 - Consistent
 - Do repetitive work once

Making a Card Component

Different options

- Wrapper Component
- Pass parts as props (prop-driven)
- Wrapper Subcomponents

Option 1: Card Component as Wrapper

Goal of how to use:

```
<Card className="card" onClick={onClick}>
  <h3 className="card__title">Jorts</h3>
  <img
    className="card__pic"
    alt="a smug orange cat sitting with tail curled around front paws"
    src={jortsPic}
  />
  <p className="card__text">
    It has been 0 days since a Trash Can mishap
  </p>
  <button
    className="card__link"
    aria-label="Read more about Jorts"
  >
    Read More
  </button>
</Card>
```

Details about using a Wrapper style Component

- The `<Card>` component generates the wrapper
- Contents of `<Card>` element put inside wrapper
- `<Card>` not in charge of styling contents
- `<Card>` passed onClick (for whole card)
 - `<Card>` does not add behavior to parts
- `<Card>` does not do MUCH in this style

How to create a Wrapper-style component

- Contents of element are passed as `children` prop
 - Automatic React behavior
- Component decides:
 - What to put around `children`
 - Whether to show `children`
- Component can't easily ALTER `children`

```
function Card({ className, onClick, children }) {  
  return (  
    <div  
      className={`card-container ${className}`}  
      onClick={onClick}  
    >  
      {children}  
    </div>  
  );  
}
```

Option 2: Card Parts as Props

Goal of how to use:

```
<Card
  className="card"
  onReadMore={onReadMore}
  title="Jorts"
  pic={jortsPic}
  alt="a smug orange cat sitting with tail curled around front paws"
  text="It has been 0 days since a Trash Can mishap"
  linkText="Read More"
/>
```

Details about using a prop-driven Component

- The `<Card>` component generates inner-elements
 - Uses values from props for their contents
- `<Card>` has a LOT of control
 - But can only do things the way it was coded
- Can decide what elements to add interaction to
 - Ex: `onReadMore` might be `onClick` on link only
- Can be VERY detailed in props
- and/or limited flexibility

How to create a props-driven component

```
function Card({
  className, onReadMore,
  title, pic,
  alt, text, linkText,
}) {
  return (
    <div className={className}>
      <h3 className={` ${className}__title`} >{title}</h3>
      <img className={` ${className}__pic`}
        alt={alt}
        src={pic}
      />
      <p className={` ${className}__text`} >{text}</p>
      <button className={` ${className}__link`}
        onClick={onReadMore}
        aria-label={`Read More about ${title}`}
      >
        Read More
      </button>
    </div>
  )
}
```

Pros/Cons of Prop-driven component

- More consistent!
 - Easier to use if data-driven!
 - Classnames auto-generated from base
- MUST be consistent
 - Contents will ALWAYS be same structure
 - Must have same data for each instance
 - Can change base className
 - But can't add extra
 - Aria-label auto-generated
 - Might not always be "good"

Option 3: Wrapper with Subcomponents

Goal for use:

```
<Card>
  <CardTitle>Jorts</CardTitle>
  <CardPic src={jortsPic} alt={alt}/>
  <CardText>
    It has been 0 days since a Trash Can mishap
  </CardText>
  <CardLink onClick={onClick}>Read More</CardLink>
</Card>
```

Details about Subcomponents

- Here `<Card>` doesn't do much
 - But other components (subcomponents) do
 - Provide classnames and behaviors
 - Operate on their `children` prop
 - Like wrapper style component
- Subcomponents assume `<Card>`
- Pro: More flexible than prop-driven
- Pro: Provides benefits over wrapper
- Con: Subcomponents are mentally coupled
 - Look at multiple files to understand output

Which approach is best?

- You know the answer

It Depends

- Consider pros/cons
- Do you need flexibility?
- Do you need consistency?
- Do you need simplicity?

Creating any Component based on these

- Not just writing once
- All about making changes + reusing

Reusable Button Component

- A straightforward example
 - Still many parts to consider
- `onClick` handler passed as prop
- Text as prop or content (`children`)?

```
<Button onClick={onClick}>Demonstrate!</Button>
```

Simple Example

```
function Button({ children, onClick }) {  
  return (  
    <button onClick={onClick}>{children}</button>  
  );  
}
```

No benefit to this component!

- Could add some extra options to give it a benefit
 - Like an `visual` prop
 - Will handle `"link"` or `"switch"`
 - Change appearance to match

More to think about

"Reusable" means it handles other needs too!

- `disabled` prop?
 - `readonly` prop?
- `type` prop?
- `visual` prop?
- `className` prop?

Medium Example

```
function Button({
  children,
  className,
  disabled="false",
  onClick,
  type="button",
  visual="button",
}) {
  let buttonClass = "button";
  if (visual === "link") {
    buttonClass = "button-link";
  }
  return (
    <button
      className={` ${buttonClass} ${className}`}
      disabled={disabled} type={type}
      onClick={onClick}
    >
      {children}
    </button>
  );
}
```

Creating a dropdown menu UI

- Open/close on click
- Similar to hamburger menu demo
 - State for open/close class (if media query)
 - State for show/hide (if no media query)
- We didn't talk about a reusable Component

Deciding on Approach

- We just saw many options
 - Decide on approach
 - Consider what is important for THIS case
 - Simplicity of using?
 - Consistency of generated HTML?
 - Flexibility of content?
 - Flexibility of styling?

Demonstration of prop-driven

```
function Demo() {  
  const menu = [  
    { label: 'Famous Cats',  
      submenu: [  
        { label: 'Internet Cats', path: '/internet.html' },  
        { label: 'Military Cats', path: '/military.html' },  
      ],  
    },  
    { label: 'About Us',  
      submenu: [  
        { label: 'Founders', path: '/founders.html' },  
        { label: 'Purpose', path: '/purpose.html' },  
      ],  
    },  
  ];  
  return (  
    <>  
    <DropDownMenu menu={menu}/>  
    </>  
  );  
}
```

Reusable Accordion Component

State that marks each element open/close

- Can do with actual rendering output
- a11y with `aria-expanded` attribute

Example Subcomponent Style (Mostly)

```
<Accordion>  
<AccordionSection title="Are Cats Nocturnal">  
Cats are "crepuscular" - most active at dawn and dusk,  
which is not the same as being nocturnal  
</AccordionSection>  
<AccordionSection title="When did Cats become domesticated">  
Cats domesticated humans about 10,000 years ago, trading  
their services as pest controllers for worship and care  
</AccordionSection>  
</Accordion>
```

Explaining the Subcomponent example

- `<Accordion>` component doesn't do much
 - Could be skipped!
- `<AccordionSection>`
 - accepts `title` prop (slightly prop-driven)
 - Has internal state to track open/close
 - does/does not render `children` prop
 - based on state
 - Title (button) includes `aria-expanded="true/false"`

Why did I use this style? (For this example)

These true for example, not for all cases

- No interaction between section data
- Content wasn't data driven
 - "Hardcoded" text
 - Easiest to edit by seeing
- Could still be generated via a loop w/data
- Title was kept as data
 - Let Component do work of formatting element

Modal in React

- Old style (separate div) still complicated!
- Using `<dialog>` still simple
 - But with one special requirement
 - Must access `<dialog>` node
 - To call `.showModal()`
 - To call `.close()`
 - Shouldn't use `querySelector()` with React!
 - So what do we do?

useRef hook

- We've seen `useState` and `useId`
 - Now `useRef`
- Two purposes
 - Store a value without cause re-renders
 - Advanced and unusual need
 - We don't need it
 - Get a reference to a particular element node
 - Unusual need
 - Necessary to interact with element API
 - Such as assigning focus
 - Or `.showModal()` and `.close()`

useRef syntax

- import just like `useState` or `useId`

```
import { useRef } from 'react';
```

- Component function calls it to get a value

```
const someRef = useRef(); // Give it a good variable name
```

- Assign this value as the `ref` prop of an element

```
<dialog ref={someRef}>...</dialog>
```

- `YOUR_REF_VARIABLE.current` will be the DOM node

```
<button onClick={() => someRef.current.showModal()}>Open</button>
```

Simple useRef example

```
function Demo() {  
  const dialogRef = useRef();  
  return (  
    <>  
      <button onClick={() => dialogRef.current.showModal()}>  
        Open  
      </button>  
      <dialog ref={dialogRef}>  
        Here is my Modal Content  
        <button onClick={() => dialogRef.current.close()}>  
          Close  
        </button>  
      </dialog>  
    </>  
  );  
}
```

Modal as a Component

- showModal/close not render-based
 - Doesn't match general React pattern
- Hard to do as *reusable* component
- Not hard to do as per-content component

Summary - Components

Components can exist for different reasons

- Often a mix of reasons
- Important: Semantic names
- Break up/simplify large content
- Reusable/Repeatable Components
 - Consistency of generated HTML
 - Reduces visual clutter to edit
 - Flexible options

Summary - Component Techniques

Different ways to create a component

- Often mixed together
- "Wrapper" Component
- "Prop-driven" Component
- "Subcomponents"

Summary - "Wrapper" component

- Displays content around `children` prop
- Pro: Lots of control over content
 - Allows desired inconsistency
- Con: No automation of content
 - Risks undesired inconsistency
 - More manual effort

Summary - Prop-driven Component

- Pass content(s) as props
 - Can be Components themselves!
- Pro: Has control over contents
- Pro: Can automate content interactions
- Con: Limited flexibility of content
- Con: Can be hard to read in JSX

Summary - Subcomponents

Highly Coupled "related" Components

- Pro: Medium Content Flexibility
- Pro: Medium Manual Repetition
- Pro: Easy to read JSX
- Con: Multiple coupled parts
 - Hard to edit/detail

Summary - Card Component

- Tricky to find balance
 - Flexibility of Content
 - Automation of Parts
 - Ease of Reading/Editing

Summary - Button Component

- Mix of styles:
 - Many Prop-based options
 - `children` content
- Much better reusable Component than Card

Summary - Dropdown Menu

- Works well with data in props
- Hard to anticipate HTML

Summary - Accordion Component

- Data like Dropdown Menu?
 - Allows control
 - Hard to visualize content
- Separate Sections
 - Easy to read content
 - Hard to coordinate multiple sections

Summary - Modal Component

- Need `useRef` to interact with `<dialog>`
- Result: Harder to generalize as reusable
 - Not impossible, just harder
- Still fairly easy to use per content