ReactJS

Build amazing Apps

and go have a drink

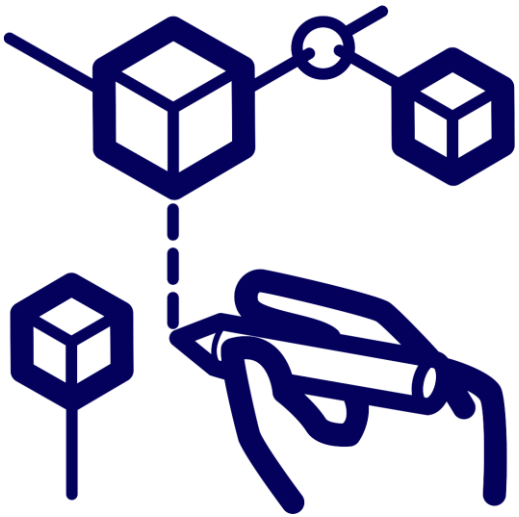
coding academy

The enter of the JS frameworks

- Web development has changed…
- Most of the apps today are built using web technologies
- For large scale projects, we just cannot get around with plain *javascript* or *jquery* any more
- We need a powerful framework to support all various aspects and life cycle of a web application



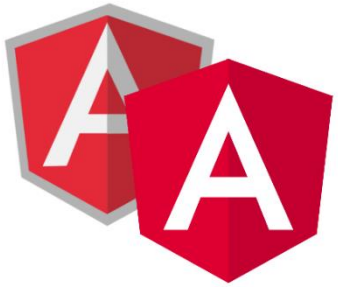
Good frameworks bring along

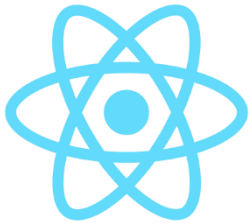


- A solid foundation so we can focus on our unique challenge
- Good separation of concerns
- Making the app easier to extend, maintain, and test

The enter of the JS frameworks

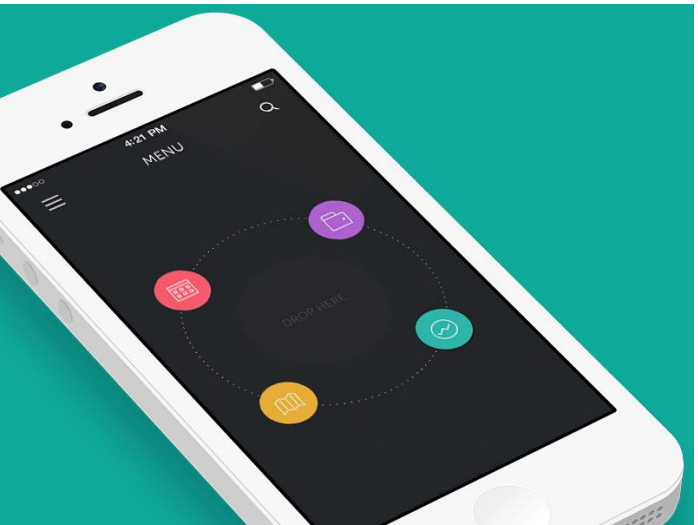
- We had Angular1 (now called angularJS)
  - It was great but had built-in problems
- Vue took Angular and made it a lot better
  - It has *underlying magic*
- New Angular has entered the scene
  - But its complicated
- React is powerful and the most popular choice today!

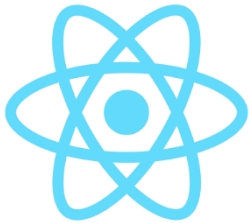




What’s ReactJS

- Framework (library) for building user interfaces
- It's capable of building modern applications
  - such as (whatsapp, facebook, Instagram, Netflix, Dropbox, etc..)





{coding academy

Why ReactJS

- It’s the most common
- Its battle tested
  - Used in large successful projects
- It can be used to build any app
  - simple to large applications
- It has a huge active community
  - thousands of supporting libraries

React - low-level-API

Here is the *React RAW (low-level) API* :

```
const MyApp = React.createElement(
    'h1',                                          // Type of element
    { className: 'header', title: 'Hola!' },       // Props and attributes
    'Hello World!'                                 // Children
)
```

```
const elContainer = document.getElementById('root')
const root = ReactDOM.createRoot(elContainer)
root.render(<MyApp />)
```

This is no fun :(

JSX

JSX is a syntax extension commonly used in React.

So instead of writing this:

```
const HelloWorld = React.createElement(
    'h1',
    { className: 'header', id: 'hello-world' },
    'Hello World!’
)
```

We will use **JSX**

(which is later compiled to the RAW React API)

```
const HelloWorld =
    <h1 className="header" id="hello-world">Hello World!</h1>
```

Babel & JSX

We code our View layer with JSX and Babel compiles it into JS that the browser can read.

```
const HelloWorld =
      <h1 className="header" id="hello-world">
        Hello World!
      </h1>
```

```
const HelloWorld = React.createElement(
  'h1',
  { className: 'header', id: 'hello-world' },
  'Hello World!'
)
```

Note the usage of: className instead of class
(class is a reserved word in Javascript)

```
▶ Warning: Invalid DOM property `class`. Did you mean `className`?
```

Index.html

Here is the <head> of our: *index.html* :

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <script src="lib/react.development.js"></script>
    <script src="lib/react-dom.development.js"></script>

    <link rel="stylesheet" href="assets/style/main.css">
    <title>React Project</title>
</head>
```

- React is composed from two libraries: React & ReactDOM
- Separating React from ReactDOM allows using react outside the browser (react-native)

- ReactDOM is the glue between React and the **DOM**

Index.html

Here is the <body> of our: *index.html* :

```
<body>
    <div id="root"></div>

    <script type="module/babel" data-presets="ca-preset" src="app.js"></script>

    <script src="lib/loadBabelModules.js"></script>
    <script src="lib/babel.js"></script>
    <script src="lib/babel-preset.js"></script>
</body>
```

```
// app.js
import { RootCmp } from './RootCmp.jsx'

const elContainer = document.getElementById('root')
const root = ReactDOM.createRoot(elContainer)
root.render(<RootCmp />)
```

coding academy

JSX Examples

```
const heading = <h1>Hello JSX</h1>

const games =
    <ul>
        <li>Car</li>
        <li>Doll</li>
        <li>Puzzle</li>
    </ul>
```

JSX Rule: Single top-most element

JSX expressions have a single top-most element

```
// This is valid.
const fruits = <ul>
                  <li>Apple</li>
                  <li>Orange</li>
               </ul>
```

```
// This is not valid.
const fruits = <h3>Fruits</h3>
               <ul>
                  <li>Apple</li>
                  <li>Orange</li>
               </ul>
```

coding academy

JSX Syntax

Embedding JavaScript expressions:

```
const fullName = 'Puki Ma'
const greeting = <h1>Hello {fullName}!</h1>
```

```
const imgIdx = (Math.random() > 0.5)? 1 : 2
const img = <img src={`./assets/img/${imgIdx}.gif`} />
```

Dynamic Attributes

We can assign dynamic attributes such as:

```
const dynSrc = 'someImgSrc.png'
const dynClass = 'special'

const HelloWorld =
    <div>
        <h2 className={dynClass}>Some Header</h2>
        <img src={dynSrc} alt=""/>
    </div>
```

Conditional Rendering

Option 1, Just use ifs in JS:

```
if (Math.rand() > 0.5) {

    btn = <button> Go Dark  </button>

} else {

    btn = <button> Go Light </button>
}
```

Conditional Rendering

Option 2, the Ternary Operator (short if)

```
const msg = <h1>
                {(age >= 18) ? 'Old enough' : 'Too young'}
            </h1>
```

Conditional Rendering

Option 3 – short circuit, using &&:

```
const age = 46
const baby = true
const cute = false

const tasty = <div>
                <h1>Conditional rendering - short circuit</h1>
                <ul>
                  <li>Chips</li>
                  {(baby && cute) && <li>Chocolate</li>}
                  {age > 15 && <li>Shakshuka</li>}
                  {age > 40 && <li>Soda</li>}
                </ul>
              </div>
```

Loops

Looping through an array:

```
const NamesList =  <ul>
                     {
                        names.map(name => <li>{name}</li>)
                     }
                    </ul>
```

For every name in the names array,
an <li> will be created

List Rendering - the key

- When rendering in a loop, we use the "key" attribute
- It helps React with rendering performance
- Key should be a unique and constant value for each item
- Usually, we will use the `item.id`

```
<ul>
    {users.map(currUser =>
        <li key={currUser.id}>
            {currUser.fullname}
        </li>
    )}
</ul>
```

Handling Events

Event handling in React is very similar to HTML , but:

- React events are named using **camelCase**, rather than lowercase
- With JSX you pass a function as the event handler, rather than a string

```
const btn = <button
                    onClick={() => {
                      alert('I Was Clicked!')
                    }}>
                       Click Me!
            </button>
```

coding academy

Handling Events

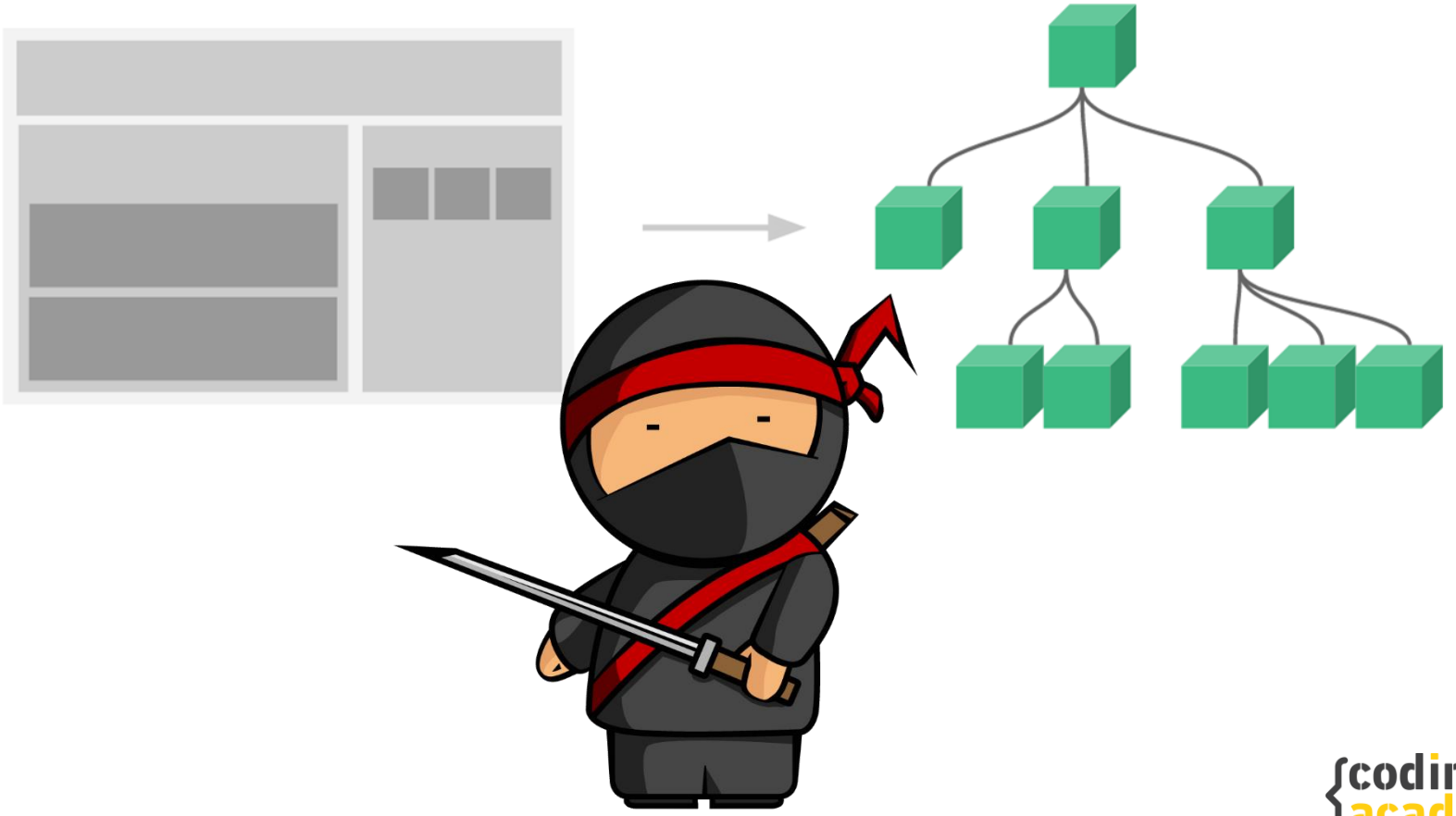
Here is another option

```
function handleClick(ev) {
    alert('I Was Clicked!')
}

const btn =
    (<button onClick={handleClick}>
        Click Me!
    </button>)
```

Level Done

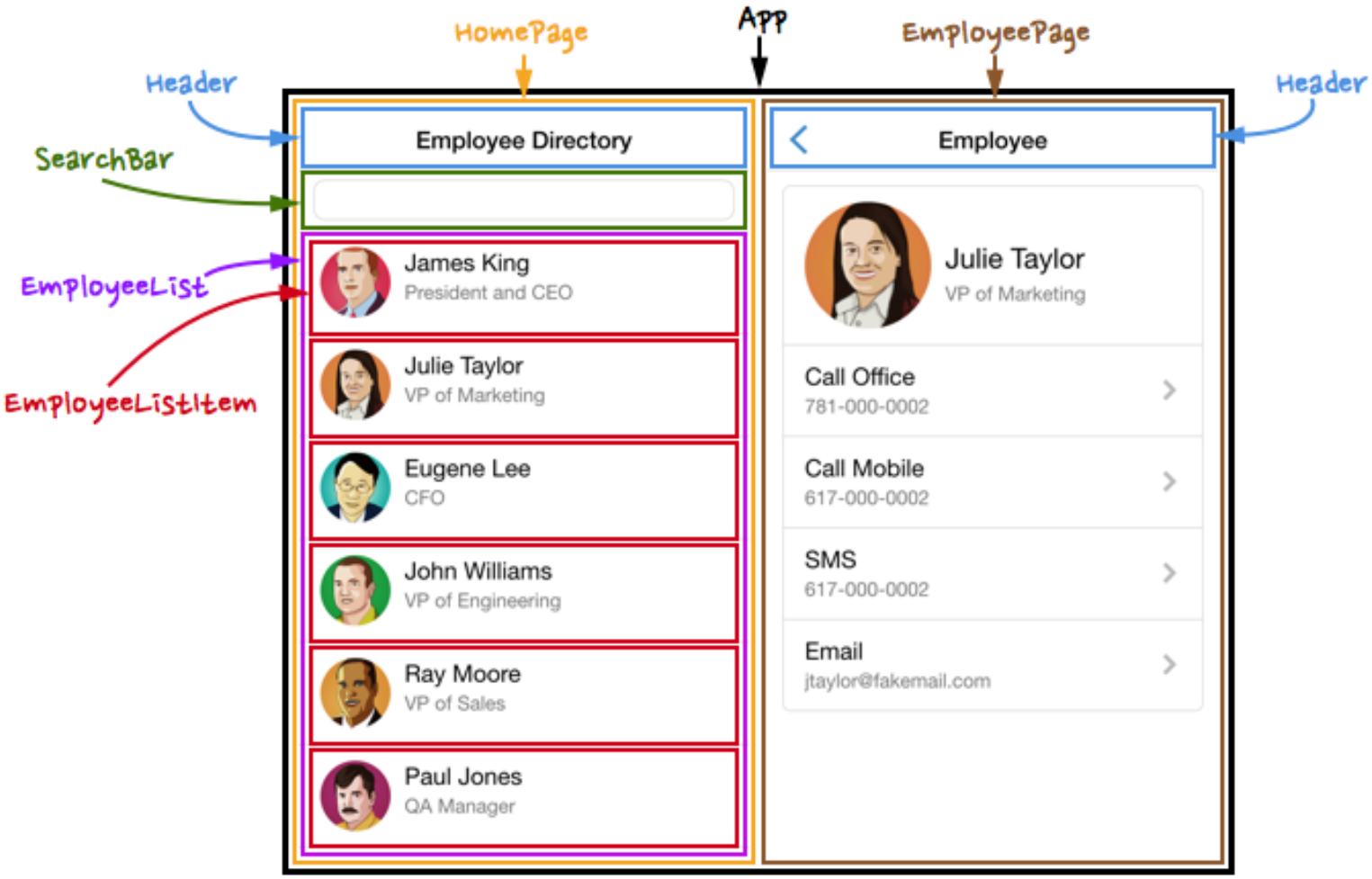
React and JSX

Components



Building UI with Components

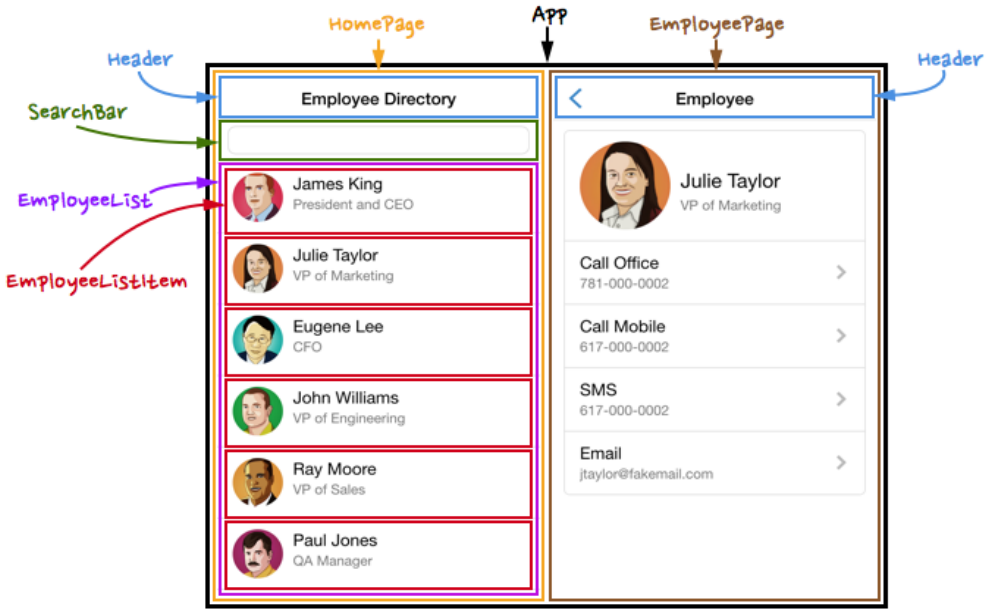
Web apps today are made of components:



coding academy

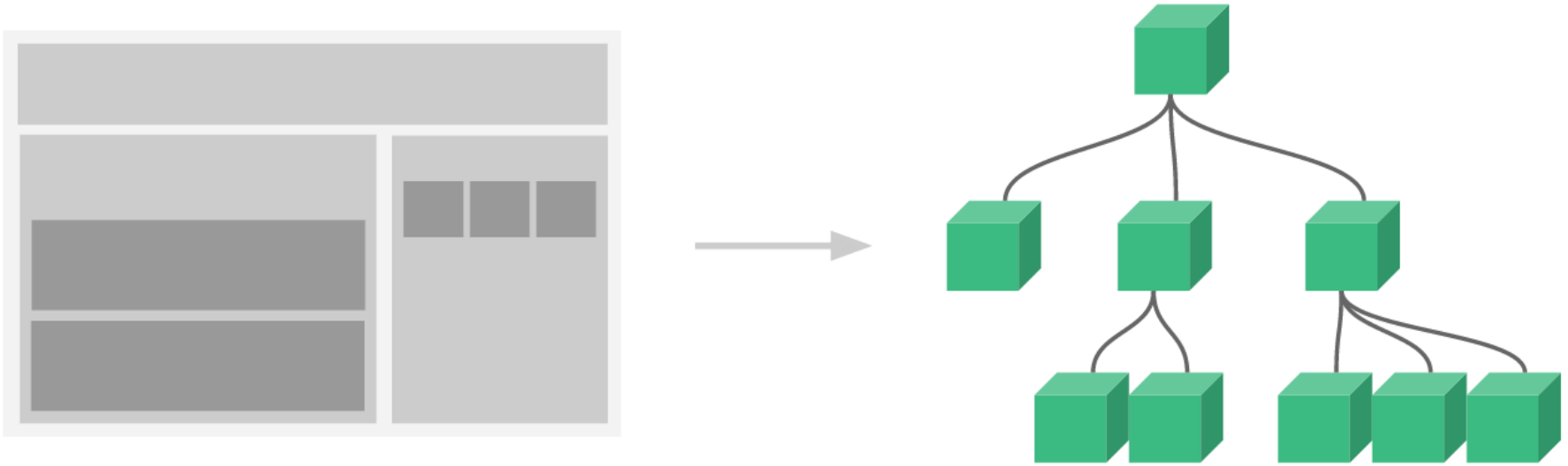
Define: Component

- A component is a small, reusable chunk of code, markup and style that is responsible for one job.
- That job is usually to render a piece of the User-Interface and handle it's events



Components architecture

Building big applications from small, self-contained, reusable components.



React Components

React components are functions
that returns a JSX

```
function SomeFuncCmp() {
    return <h1>Hello React Component</h1>
}
```

coding academy

Components

We "call those functions" in a special way:

```
function SomeFuncCmp() {
    return <h1>Hello Function Component</h1>
}


<SomeFuncCmp/>
<SomeFuncCmp/>
```

Component's Props

- "props" (properties) are passed to a component from its <Parent>
- Its like calling a function and sending some params

```
function Welcome(props) {
    return <h1>Welcome {props.name}!</h1>
}
```

```
<Welcome name={'Puki'} />
```

```
<Welcome name={'Muki'} />
```

Welcome Puki!

Welcome Muki!

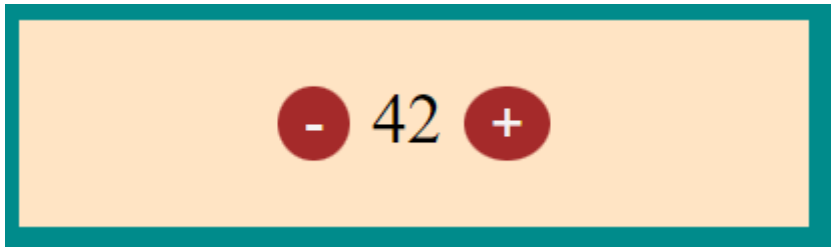
{coding academy

props are read-only

A component should not change its props
(only the <Parent> component is allowed)

```
function Welcome(props) {
    // This is not allowed!
    props.name = 'Baba'
    return <h1>Hello there {props.name}</h1>
}
```

Lets code a <SimpleCounter /> component



SimpleCounter.jsx

Components - State

- When coding components we need a place to hold the state
- Its like "local variables" of the component instance
- Changing the state causes the component to re-render

```
function Counter() {

  const [count, setCount] = useState(0)

  return <div>
           <p>You clicked {count} times</p>
           <button onClick={() => setCount(count + 1)}>
              Click me
           </button>
         </div>
}
```

coding academy

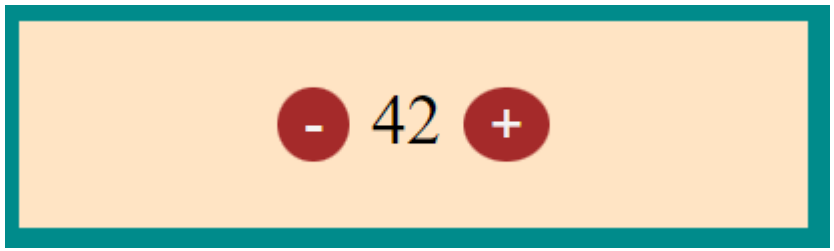
Components - State

Changing the state in such way causes the component to re-render and display the updated value

```
const { useState } = React

export function Counter() {
    const [count, setCount] = useState(0)

    return <section className="counter">
        <p>You clicked {count} times</p>
        <button onClick={() => setCount(count + 1)}>
            Click me
        </button>

    </section>
}
```

Lets code a <SimpleCounter />

```jsx
const { useState } = React

export function SimpleCounter() {
    const [count, setCount] = useState(0)

    return <section className="simple-counter">
        <button onClick={() => setCount(count - 1)}>-</button>
        {count}
        <button onClick={() => setCount(count + 1)}>+</button>

    </section>
}
```

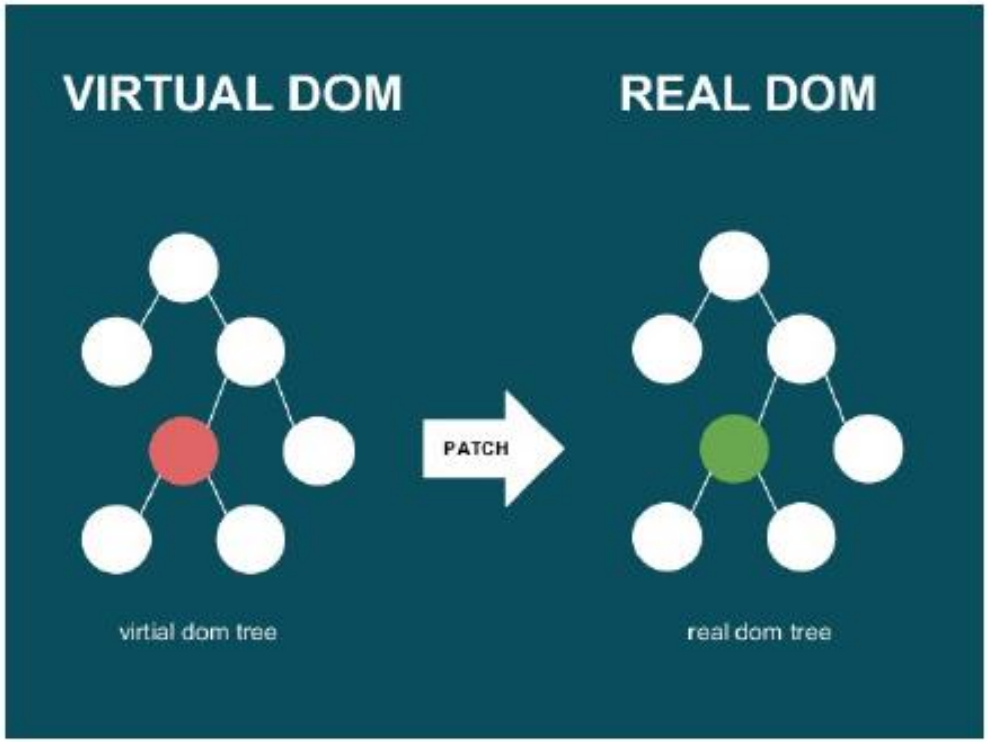


coding academy

Virtual DOM

In React, for every DOM Object, there is a corresponding "virtual DOM object."

- Its like a lightweight copy
- Manipulating the virtual DOM is fast, because nothing gets drawn onscreen
- React than can optimize what to change in the real DOM

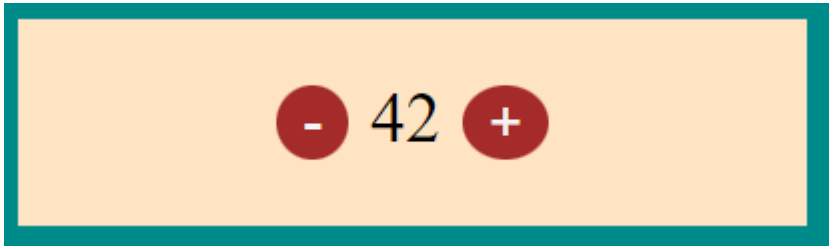


coding academy

So when we change something in the DOM:

- The entire virtual DOM gets updated.
- The virtual DOM gets compared to what it looked like before the update.
- React figures out which objects have changed.
- DOM Updates get optimized
  (e.g. – replace the all <tbody> if all <td>s were changed)
- The changed objects, get updated on the *real* DOM.
- Change appear on screen

Let's add an initial value prop

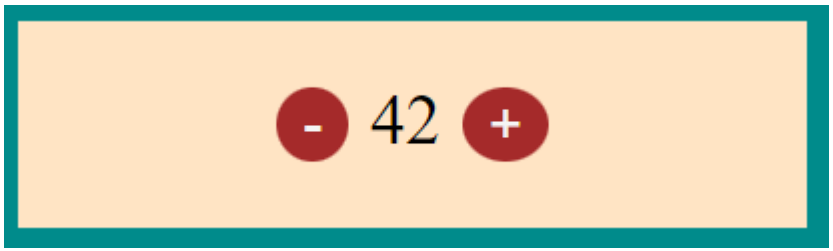



```
const { useState } = React

export function SimpleCounter(props) {
    const [count, setCount] = useState(props.val || 0)

    return <section className="simple-counter">
        <button onClick={() => setCount(count - 1)}>-</button>
        {count}
        <button onClick={() => setCount(count + 1)}>+</button>

    </section>
}
```

```
<SimpleCounter val={42} />
```

Let's add a callback prop

```
export function SimpleCounter(props) {
    const [count, setCount] = useState(props.val || 0)

    function handleClick(diff) {
        setCount(count + diff)
        props.onModified && props.onModified(count + diff)
    }

    return <section className="simple-counter">
        <button onClick={() => handleClick(-1)}>-</button>
        {count}
        <button onClick={() => handleClick(1)}>+</button>

    </section>
}
```



Destructering props

It is better to use `Destructering` with the `props` so the API of the component is more obvious to parent components:

```
export function SimpleCounter(props) {}

// Destructering the props
export function SimpleCounter({val, onModified}) {}
```

Let's use our <SimpleCounter /> component in some parent component


```
const { useState } = React
import {SimpleCounter} from './SimpleCounter.jsx'

export function UserPreview() {

    const [user, setUser] = useState({fullname: 'Puki Reactof', score: 7})

    function whenModified(newScore){
        setUser({...user, score: newScore})
    }
    return <section className="user-preview">
        <h2>{user.fullname}</h2>
        <h3>Score: {user.score}</h3>
        <SimpleCounter onModified={whenModified} val={user.score} />
    </section>
}
```

coding academy

Emitting to <Parent>

Child component may emit (report)
an event to it's <Parent>

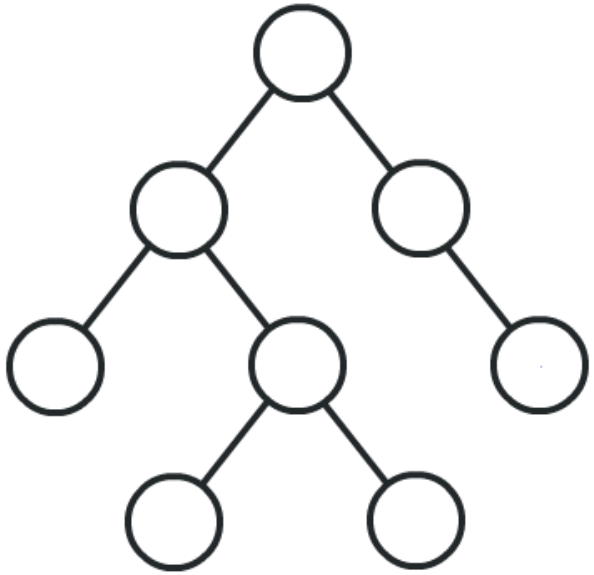
This is done by passing a **callback function** from the <Parent>

```
<SimpleCounter onModified={whenModified} val={someVal} />
```

The <Child> can call that function:

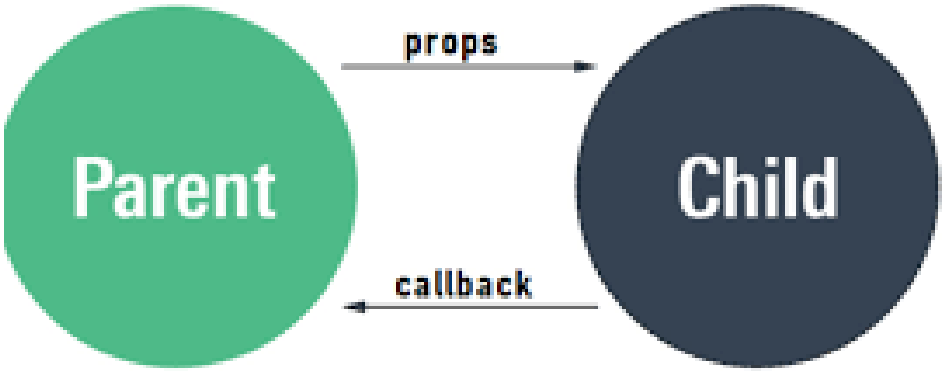
```
props.onModified('someData')
```

This call might lead to updated props…



Composing components

- Components are meant to be used together, most commonly in a parent - child relationship.
- This is achieved via: passing props and executing callbacks



Passing Callbacks to Components

It is common to pass a callback function as a prop to a child component, so it can notify the parent component when a certain event occurs

```
function ParentCmp() {
    function handleSomething() {
        alert('Something happened in ChildCmp')
    }
    return <section>
        <ChildCmp onSomething={handleSomething} />
    </section>

}

function ChildCmp({onSomething}) {
    return <button onClick={onSomething}>
        Do Something
    </button>
}
```

React devtools

Let's review React Dev-Tool extension:

component's life cycle

- There are few stages in the life of a component:
  - When it is initially **mounted** to the DOM
  - When it is **updated**
  - When it is being **unmounted** (removed from DOM)

- For those, we will use the hook: useEffect

Those life-cycle hooks allow hooking into the component's lifecycle

coding academy

component's life cycle - mounted

The initialization logic of a component is usually placed here:

```
function Cmp() {

    useEffect(()=>{
        console.log('Cmp mounted')
    }, [])

    return <div>Cmp!</div>
}
```

component's life cycle - updated

We can register a function to run whenever some `state` or some `prop` is changed:

```
function Counter() {
    const [count, setCount] = useState(0)

    useEffect(() => {
        document.title = count
    }, [count])

    return <div>
              <p>You clicked { count } times</p>
              <button onClick={ () => setCount(count + 1) }>
                   Click me
              </button>
           </div>

}
```

coding academy

component's life cycle - unmounted

We can register a function to run when the component is being removed from the DOM:

```
const { useState, useEffect } = React

export function MouseMonitor() {
    const [pos, setPos] = useState({ x: 0, y: 0 })
    useEffect(() => {
        document.onmousemove = (ev) => {
            setPos({ x: ev.clientX, y: ev.clientY })
        }
        return () => {
            document.onmousemove = null
        }
    }, [])
    return <div>
        x:{pos.x} y:{pos.y}
    </div>
}
```

The effect function will run when component mounts
and the function we return will run when component unmounts

coding academy

Using Refs

- Sometimes we need a direct access to a DOM element, we will use a Ref
- Refs are available after the component is mounted to DOM

```
function DemoRef() {
    const inputEl = useRef(null)
    const onButtonClick = () => {
      // `current` points to the mounted input element
      inputEl.current.focus()
    }
    return <div>
             <input ref={inputEl} type="text" />
             <button onClick={onButtonClick}>Focus the input</button>
           </div>

  }
```

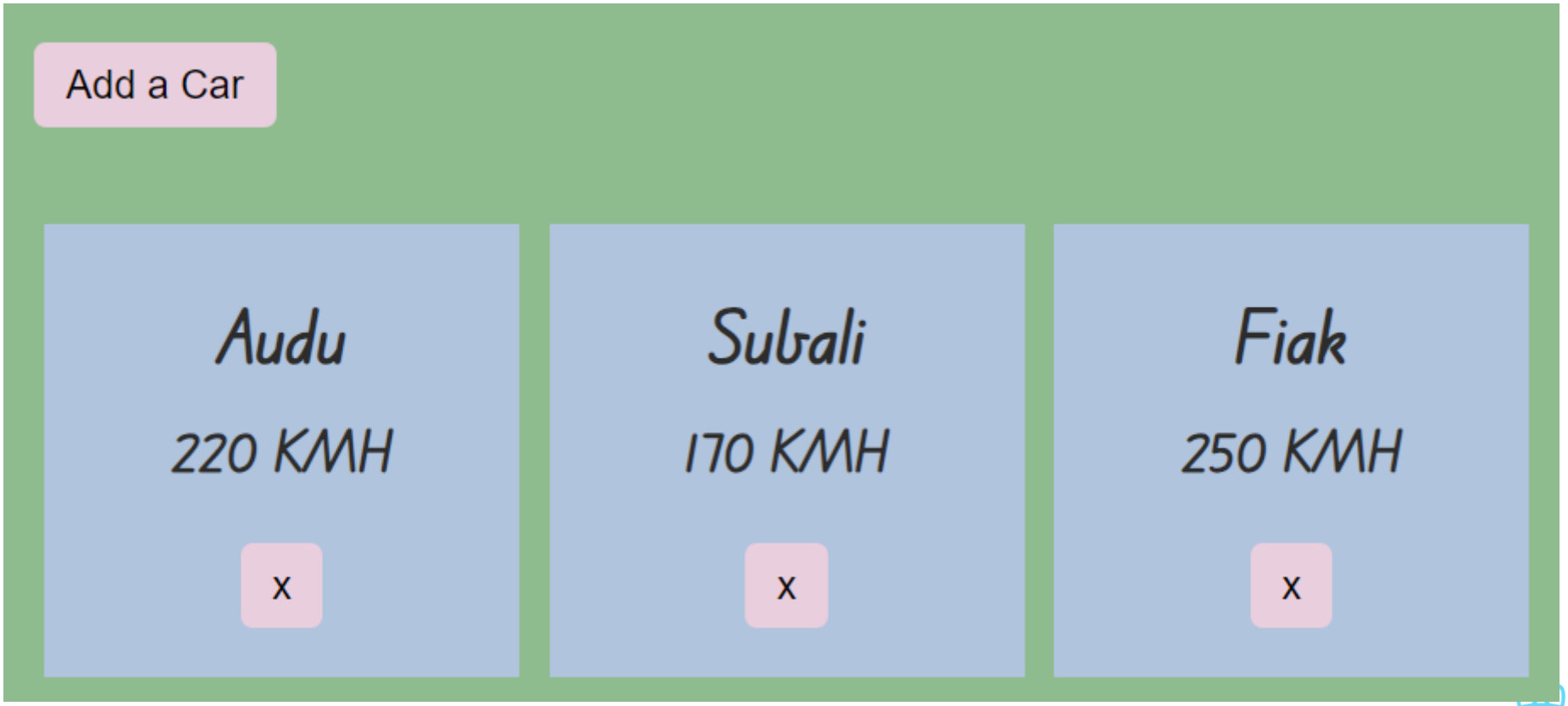
a <SimpleTimer> component

useRef can also be used for keeping some mutable values around, as the same ref pointer is received on every render

```
const {useEffect, useRef, useState} =  React
export function SimpleTimer() {
    const [count, setCount] = useState(0)
    const intervalIdRef = useRef()
    useEffect(() => {
        console.log('Counter Mounted')
        intervalIdRef.current = setInterval(() => {
            setCount((prevCount) => prevCount + 1)
        }, 1000)

        return () => {
            console.log('Counter going down')
            clearInterval(intervalIdRef.current)
        }
    }, [])
    function stopCounting() {
        clearInterval(intervalIdRef.current)
    }
    return <section>
        <span>{count}</span>
        <button onClick={stopCounting}>Stop</button>
    </section>
}
```

Example

Let's setup a basic car app



Car App - Model

Let's get some data from a service

```
const { useState, useEffect } = React
import {carService} from '../services/car.service.js'

export function CarIndex() {
    const [cars, setCars] = useState([])

    useEffect(()=>{
        carService.query()
            .then(setCars)
    }, [])

    return (
        <section className="car-index">
            <h2>List of Cars</h2>
            <ul>
                {cars.map(car => <li key={car.id}>{car.vendor}</li>)}
            </ul>
        </section>
    )
}
```

coding academy

Example

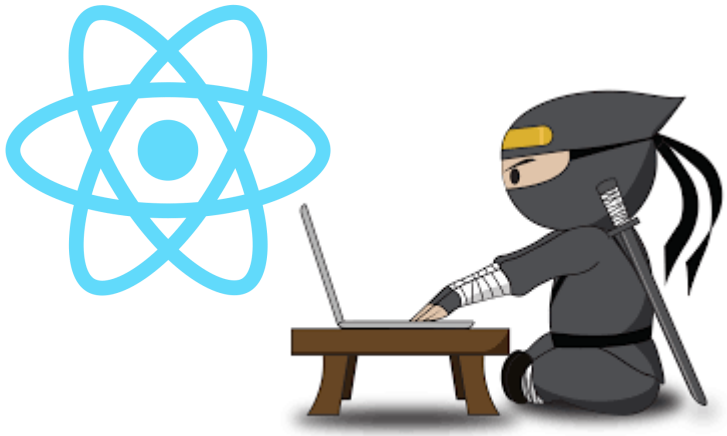
```
const { useState, useEffect } = React
import { carService } from '../services/car.service.js'

export function CarIndex() {
    const [cars, setCars] = useState([])

    useEffect(() => {
        carService.query()
            .then(setCars)
    }, [])

    function onRemoveCar(carId) {
        carService.remove(carId)
            .then(() => {
                setCars(cars.filter(c => c.id !== carId))
            })
    }

    return (
        <section className="car-index">
            <h2>List of Cars</h2>
            <ul>
                {cars.map(car => <li key={car.id}>
                    <h3>{car.vendor}</h3>
                    <h4>{car.maxSpeed} KMH</h4>
                    <button onClick={() => onRemoveCar(car.id)}>x</button>
                </li>)}
            </ul>
        </section>
    )
}
```

coding academy

Level Done

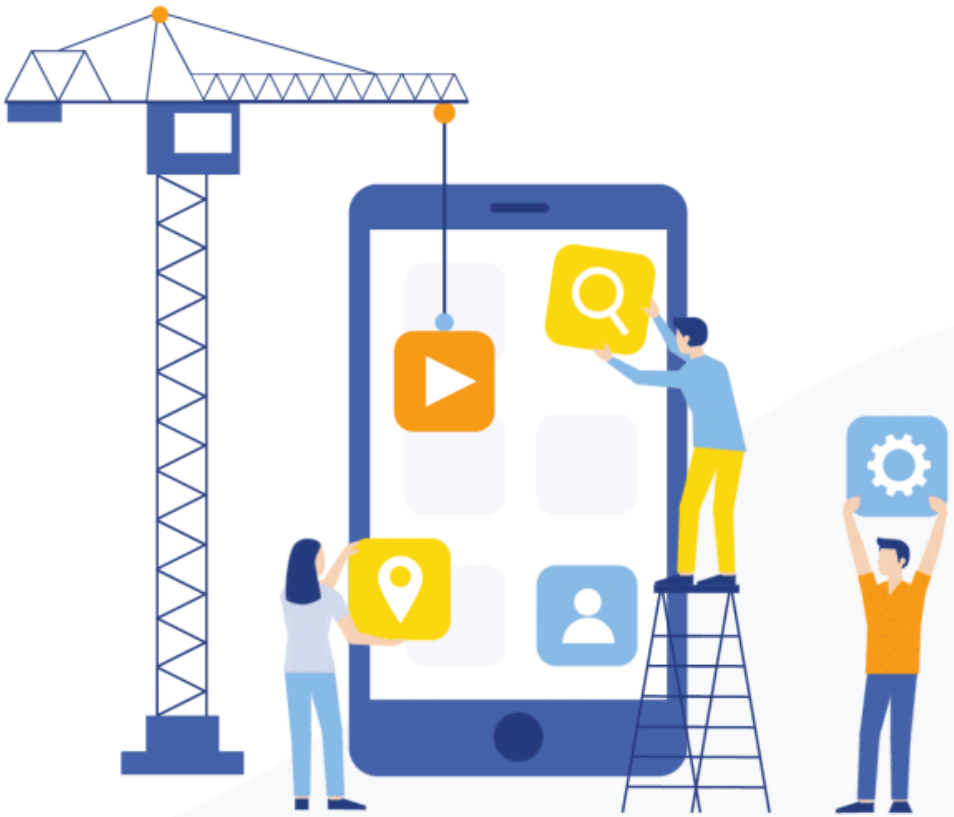
React Components

Handson Time

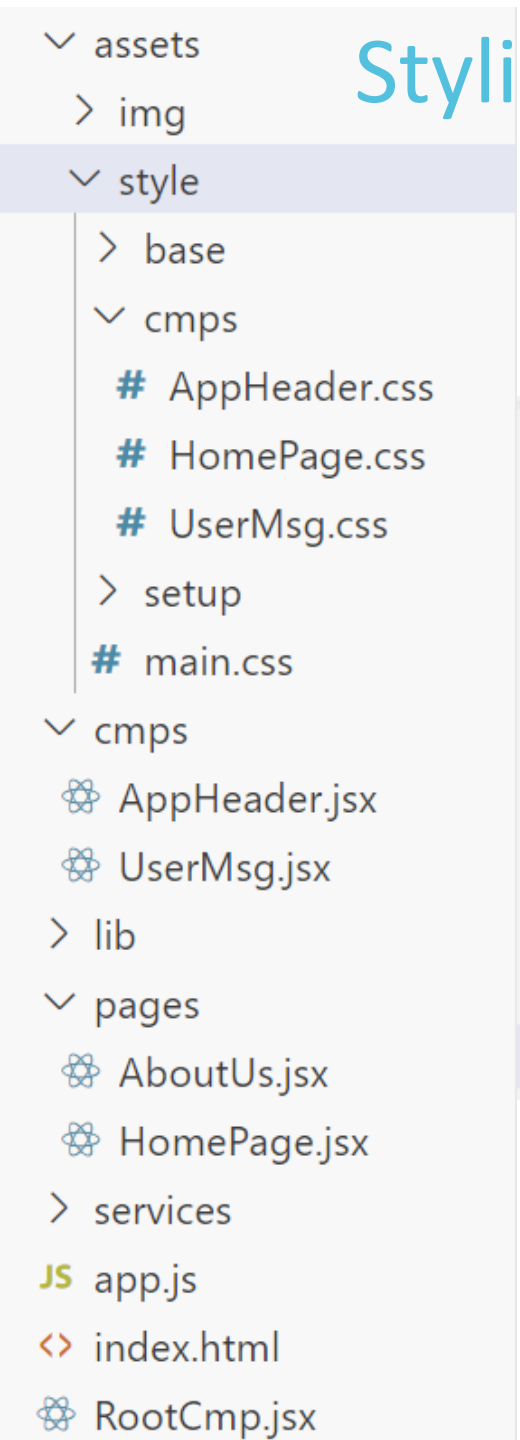


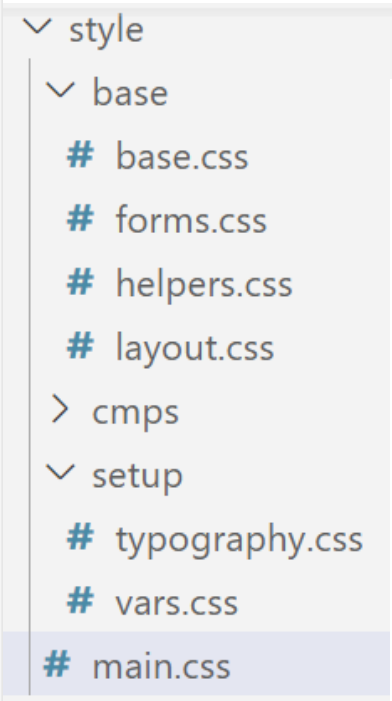
coding academy

App Structuring



Styling architecture





main.css

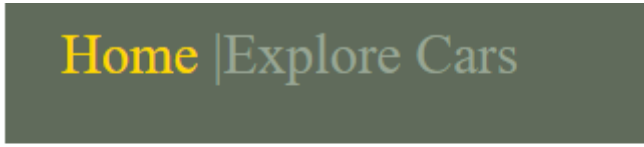
assets > style > # main.css

```
1   /* Setup */
2   @import 'setup/vars.css';
3   @import 'setup/typography.css';
4
5   /* Base */
6   @import 'base/base.css';
7   @import 'base/helpers.css';
8   @import 'base/layout.css';
9   @import 'base/forms.css';
10
11  /* Components */
12  @import 'cmps/UserMsg.css';
13  @import 'cmps/HomePage.css';
14  @import 'cmps/AppHeader.css';
```

```jsx
const { useState} = React

import { Home } from './pages/Home.jsx'
import { CarIndex } from './pages/CarIndex.jsx'

export function RootCmp() {
    const [page, setPage] = useState('home')

    function onSetPage(ev, page) {
        ev.preventDefault()
        setPage(page)
    }

    return <section className="main-layout app">
        <header className="app-header">
            <h1>React Car App</h1>
            <nav>
                <a href="" className={(page === 'home') ? 'active' : ''}
                    onClick={(ev) => onSetPage(ev, 'home')}>
                    Home
                </a> |
                <a href="" className={(page === 'car') ? 'active' : ''}
                    onClick={(ev) => onSetPage(ev, 'car')}>
                    Explore Cars
                </a>
            </nav>
        </header>
        <main>
            {page === 'home' && <Home />}
            {page === 'car' && <CarIndex />}
        </main>
    </section>
}
```

Basic Routing

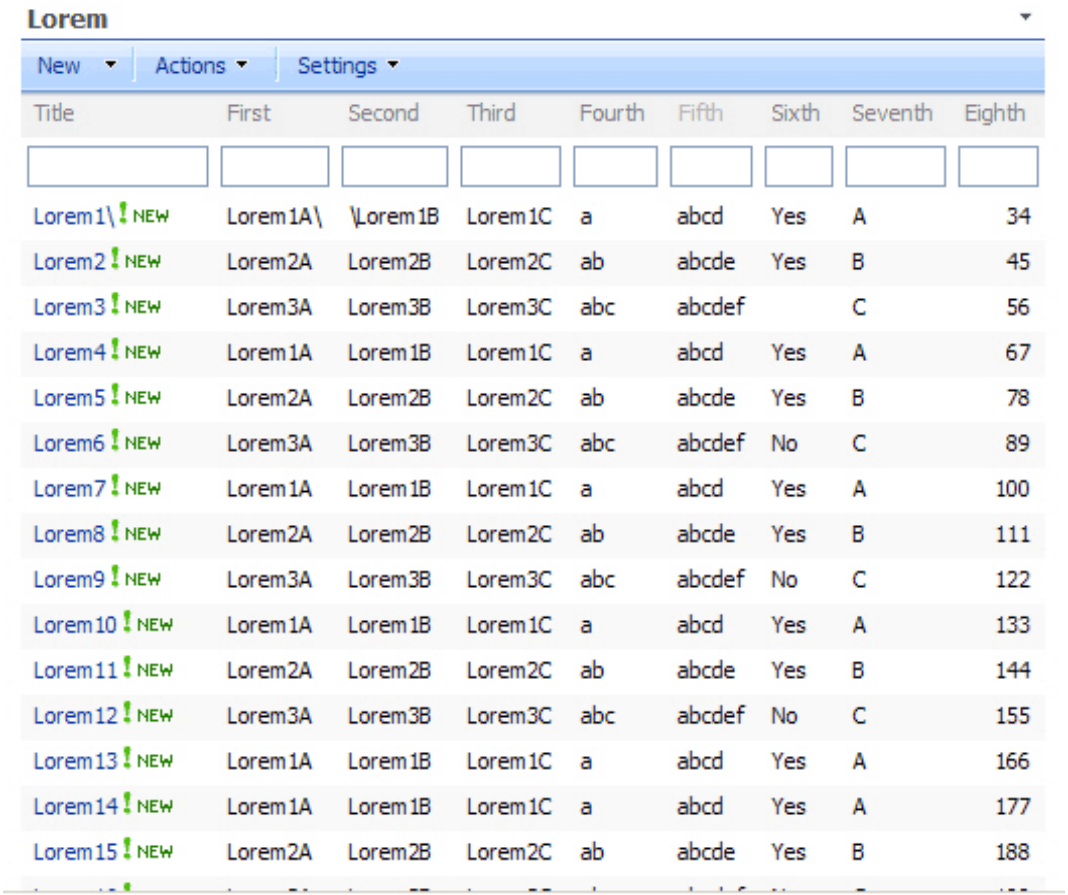
Let's setup *kind-of* routing in our root component



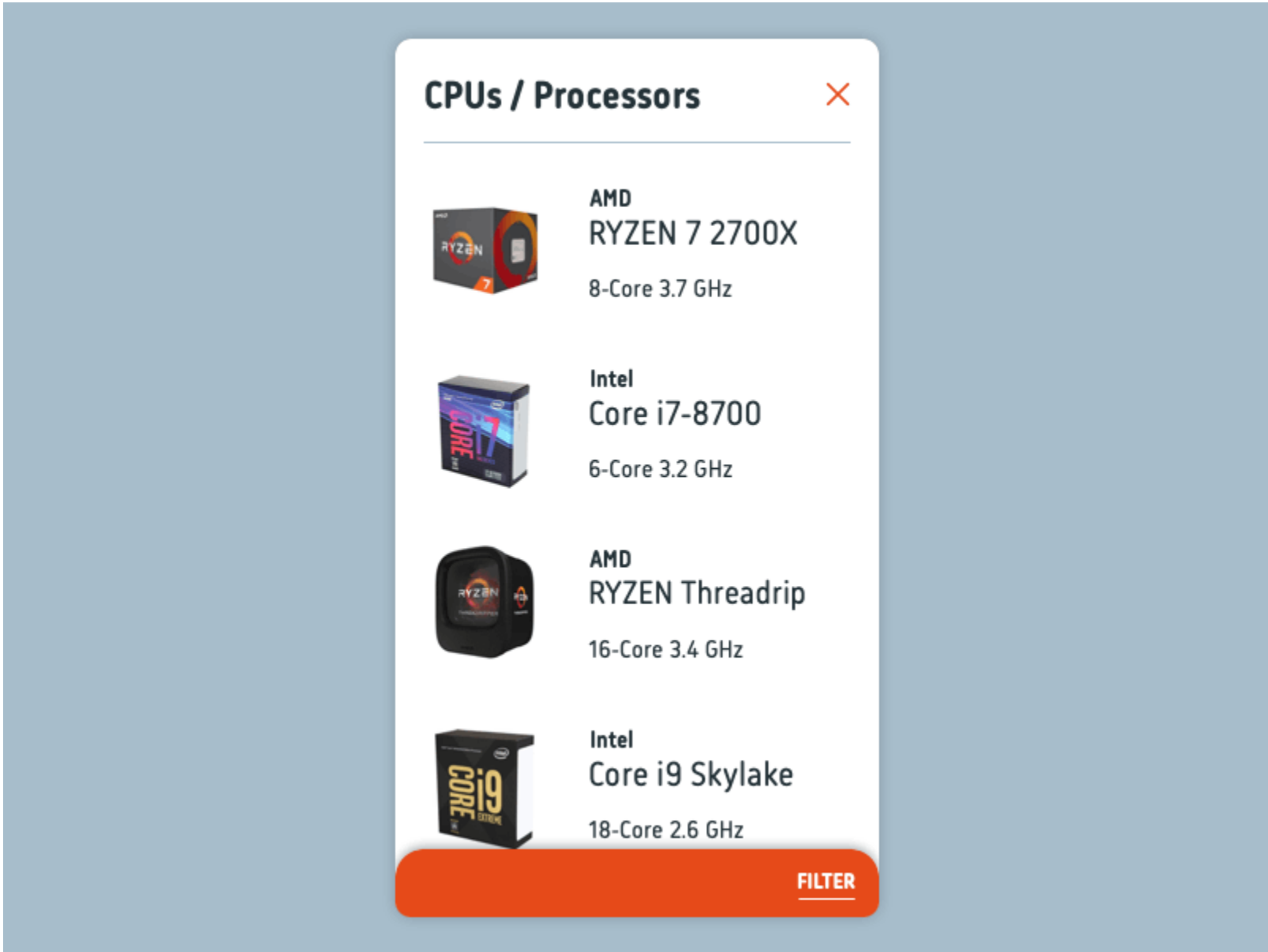
coding academy

List of items

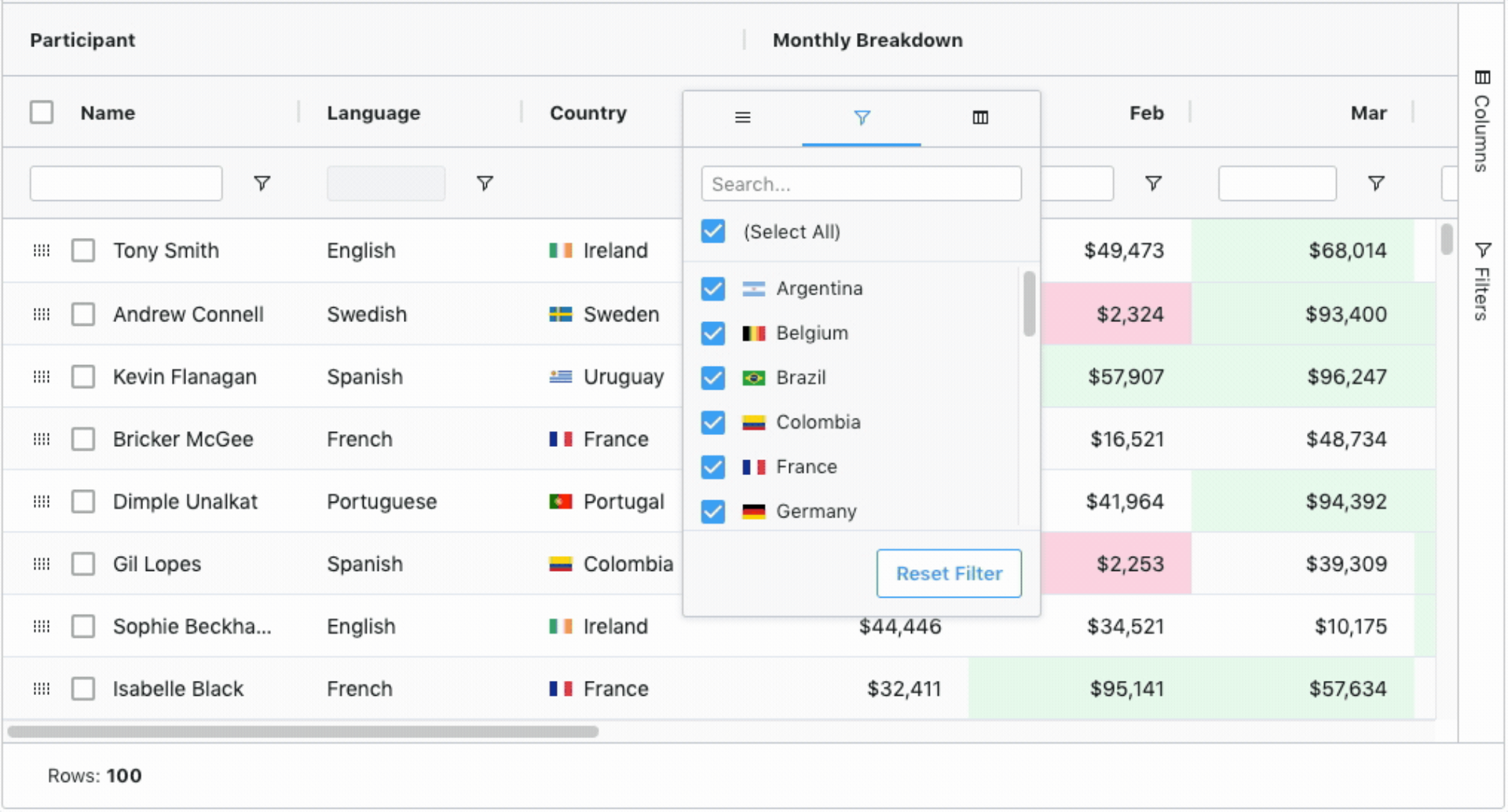
- In many application, we will have the need to present a list of items
- We usually also need to filter and sort that list



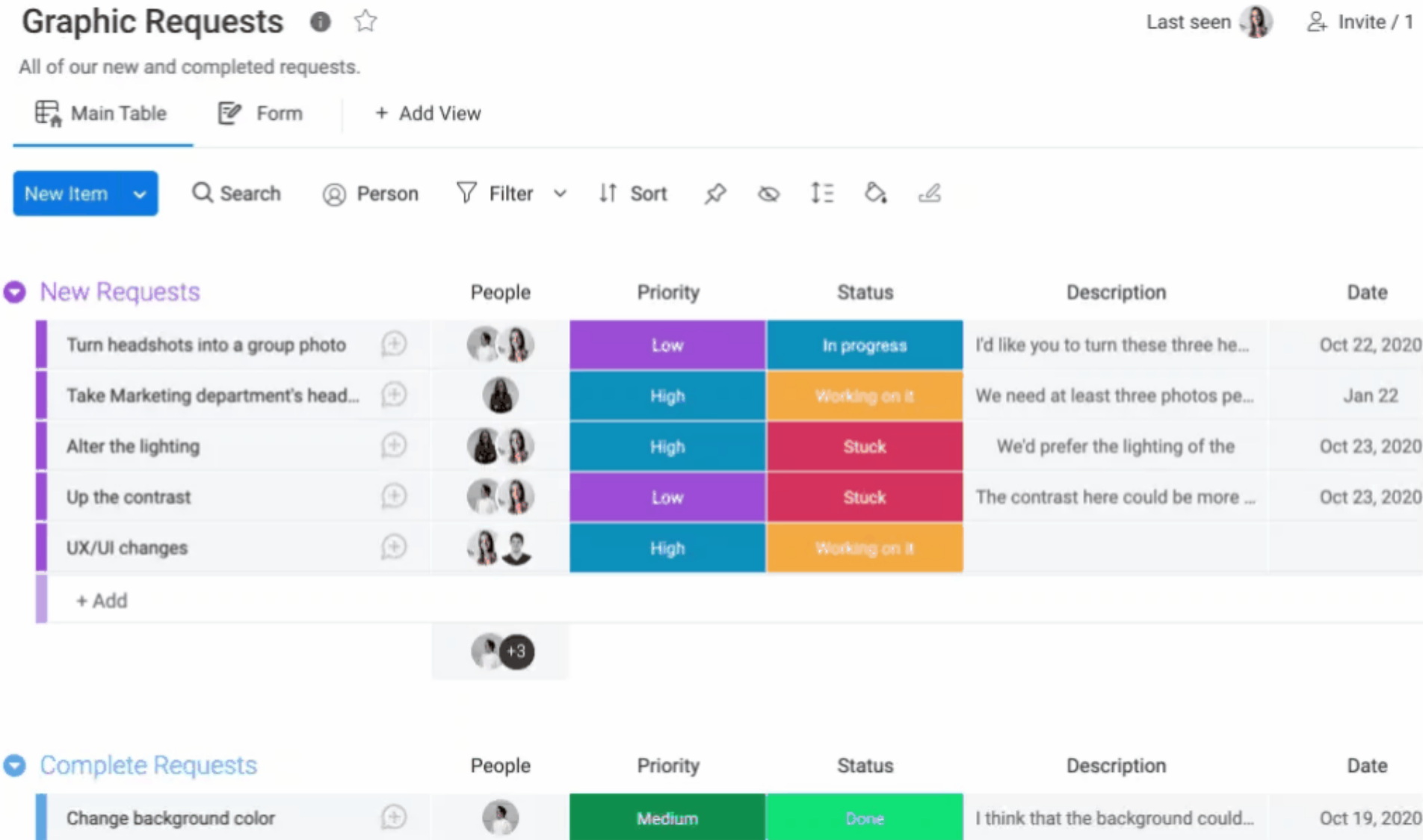
Another example



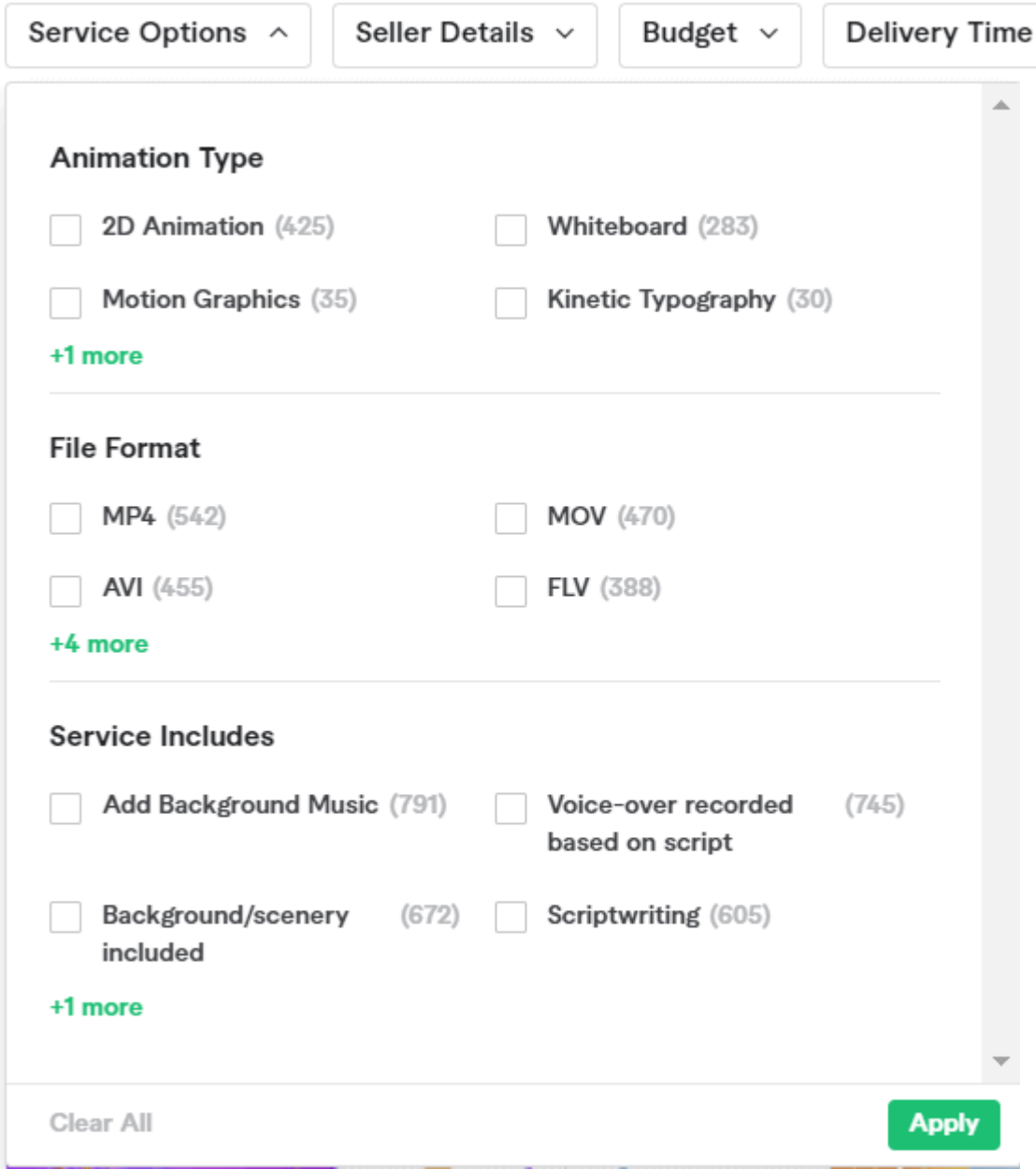
Another example



Example - Monday

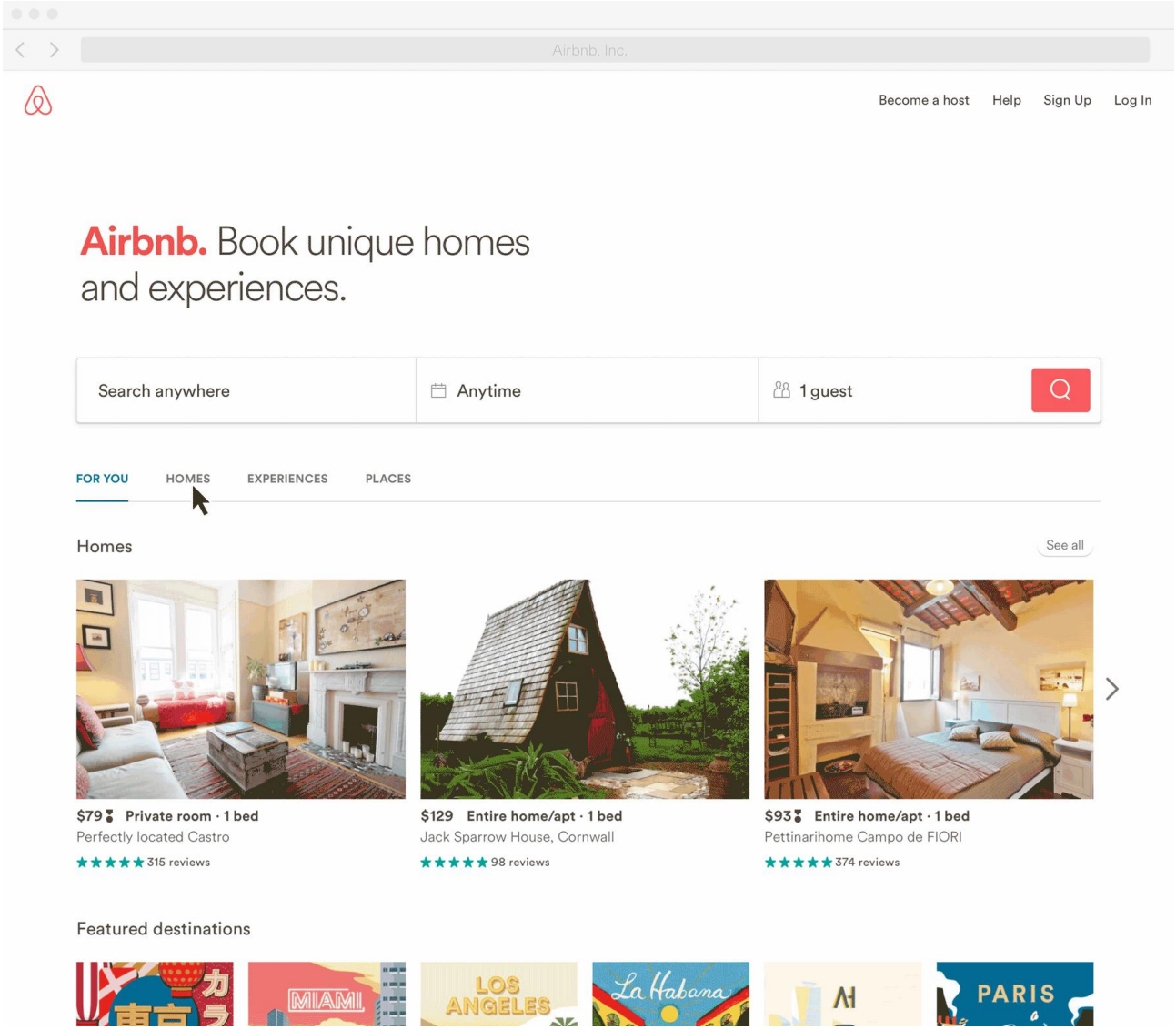


Example - Fiverr



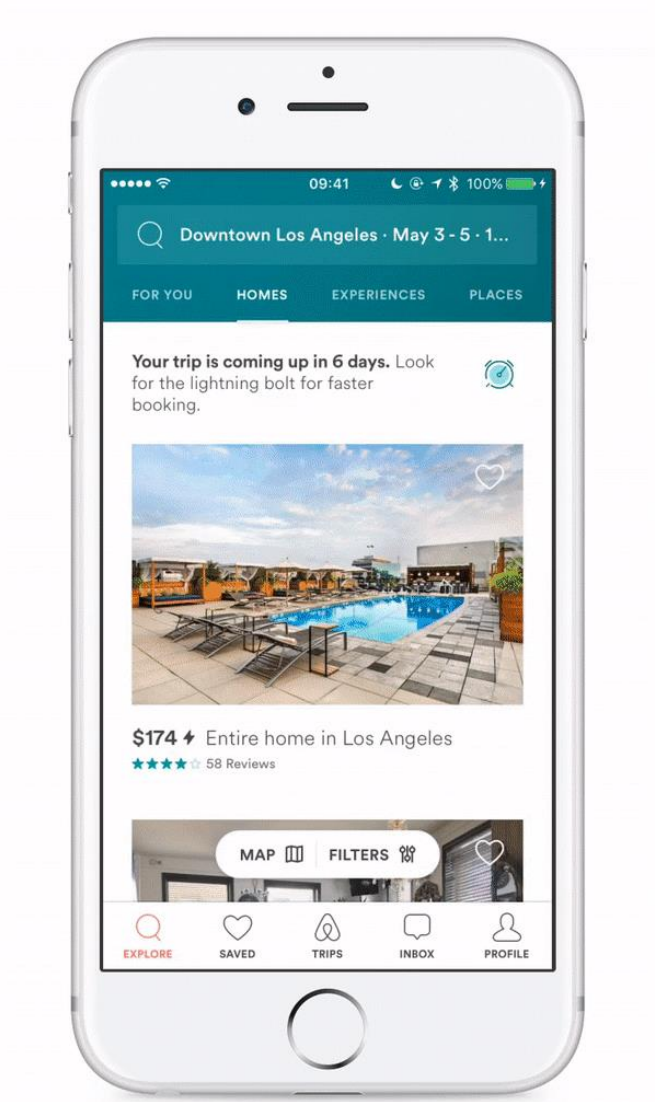
coding academy

Example - airbnb



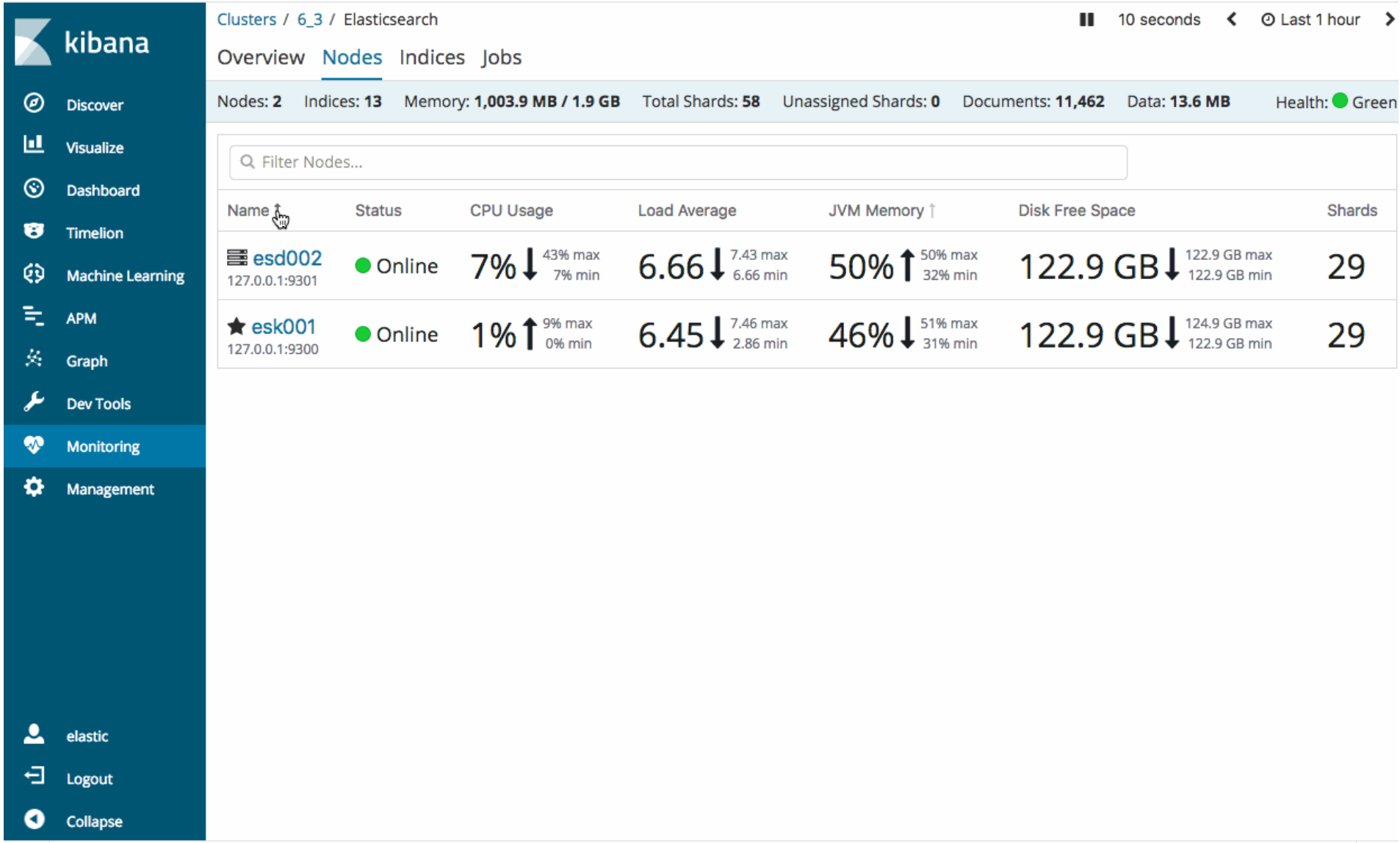
{coding academy

Example - airbnb



coding academy

Example - sorting



Many applications share the following structure

- It is common to have an <ItemIndex> component, getting a list of items from an itemService
- It then renders an <ItemList>, rendering <ItemPreview> in a loop.
- It also uses an <ItemFilter> for filtering/sorting the list
- In addition, we will usually have a <ItemDetails> and <ItemEdit> as separate pages

the <ItemIndex>

- This is a smart component and usually the most complex one
- It gets the item list from the service

- It renders <ItemList> to show the items and sends down callbacks for the needed actions which will use the itemService to perform

- It also renders <ItemFilter> and when updated, it sends down an updated list to <ItemList>

the <ItemIndex>

- This is a smart component and usually the most complex one.
- It starts by getting the item list from the service:

```
const [cars, setCars] = useState([])

useEffect(()=>{
    carService.query()
        .then(setCars)
}, [])
```

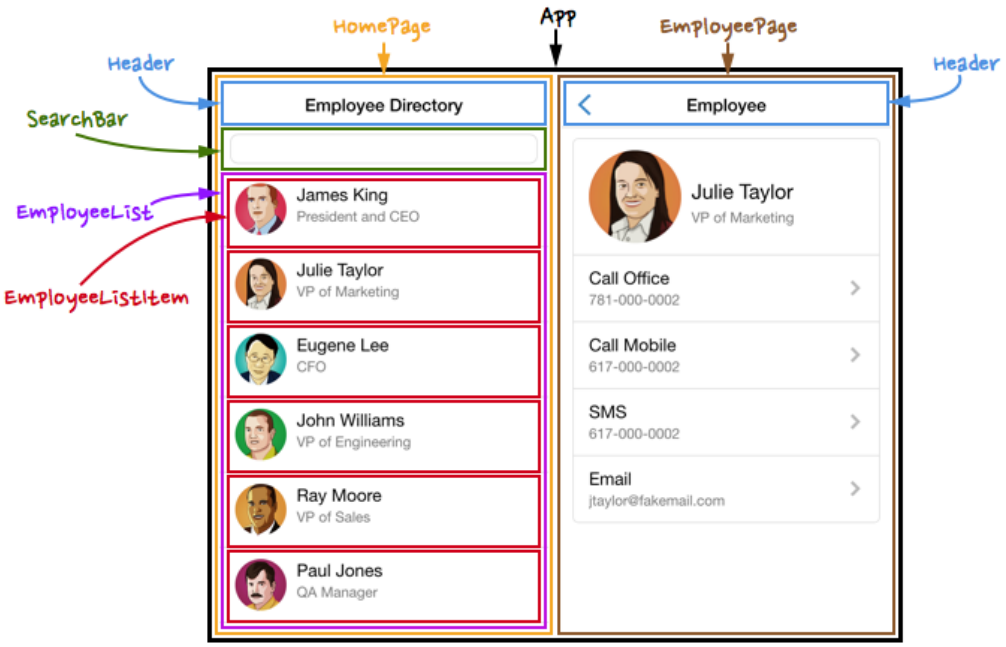
the <ItemIndex>

Here is a basic structure

```
export function CarIndex() {
    const [cars, setCars] = useState([])
    const [filterBy, setFilterBy] = useState({})
    const [selectedCar, setSelectedCar] = useState(null)

    useEffect(() => {
        carService.query(filterBy)
            .then(setCars)
    }, [filterBy])

    return (
        <section className="car-index">
            <h2>Cars</h2>
            {!selectedCar && <section>
                <CarFilter onFilter={setFilterBy} />
                <CarList cars={cars} onSelect={setSelectedCar} />
            </section>}
            {selectedCar && <section>
                <h2>Selected Car</h2>
                <pre>{JSON.stringify(selectedCar, null, 2)}</pre>
                <button onClick={()=>setSelectedCar(null)}>Close</button>
            </section>}
        </section>
    )
}
```

the <ItemList>

It’s where the user sees a list of preview items



the <ItemList>

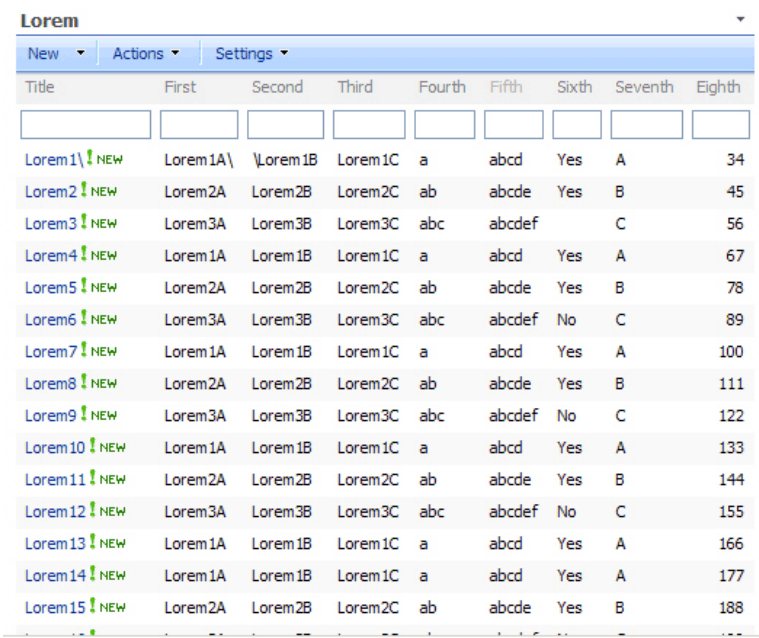
It is usually a simple component getting items as prop and mapping them to <Preview> component

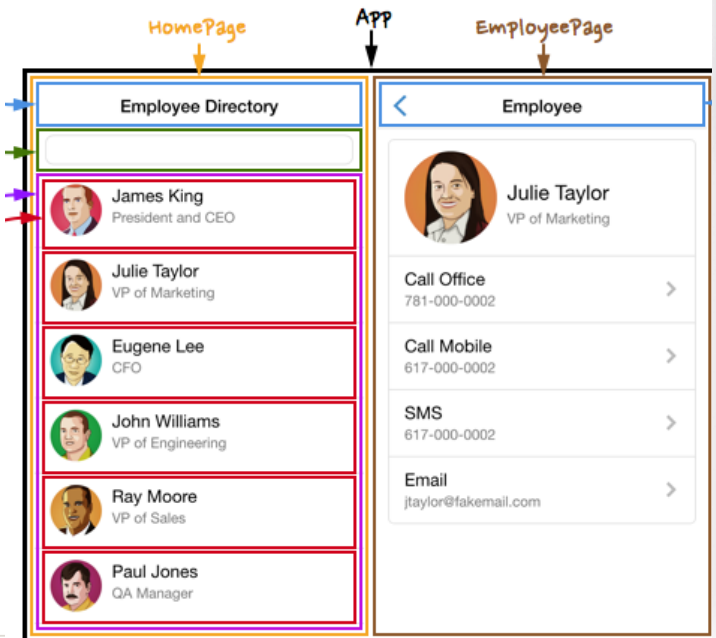
For example – a car list:

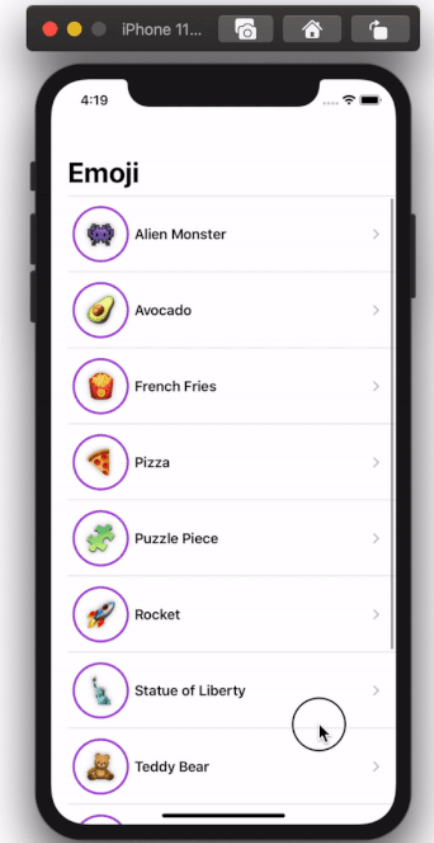
```
import { CarPreview } from './car-preview.jsx'

export function CarList({ cars, onSelectCar }) {
    return <section className="car-list">
        {cars.map(car =>
            <CarPreview key={car.id} car={car}
                onSelectCar={onSelectCar} />
        )}
    </section>
}
```

the <ItemPreview>

- It's a preview of the item, usually links to another page with the full details
- It gets an item as a prop and renders
- Commonly reports back to his parent (emits) for any important task

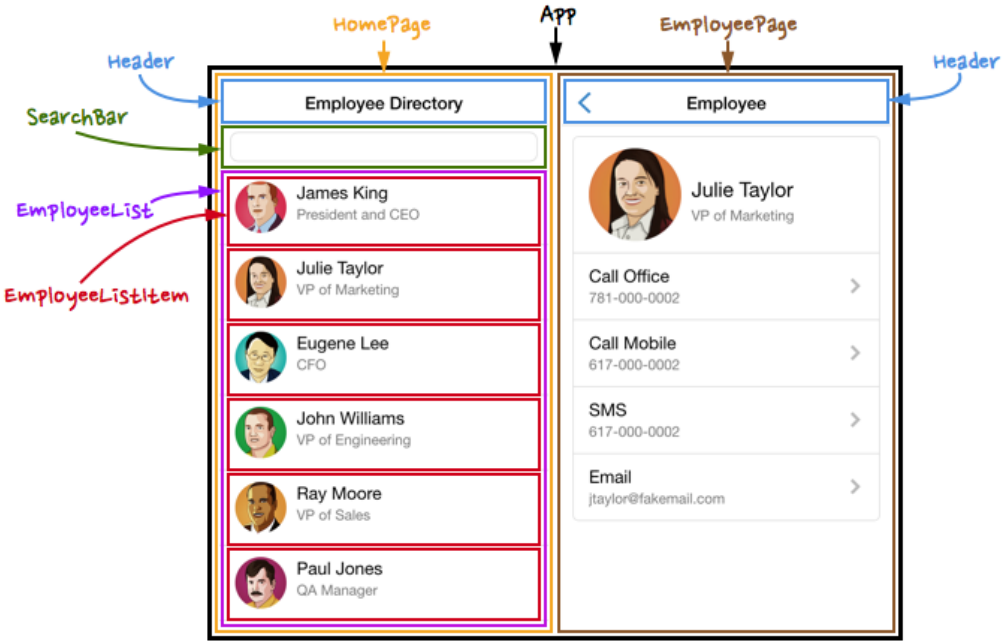


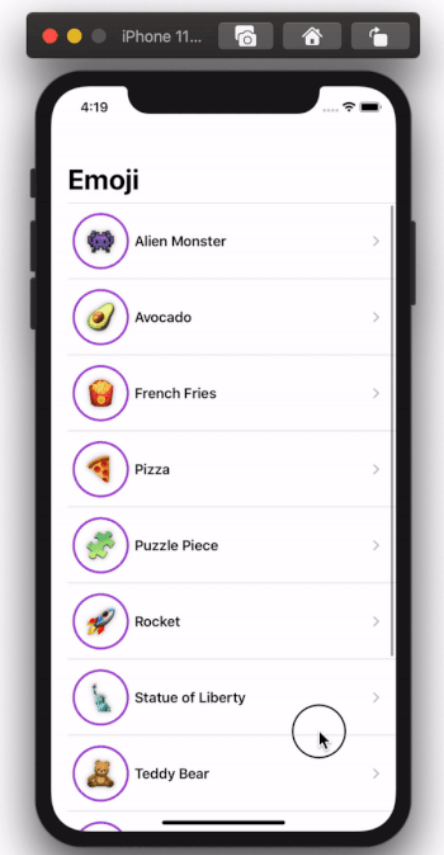




the <ItemDetails>

- Display the entire details of the items
  - Including reviews, and more
- It works directly with the service for getting fresh full details for the item





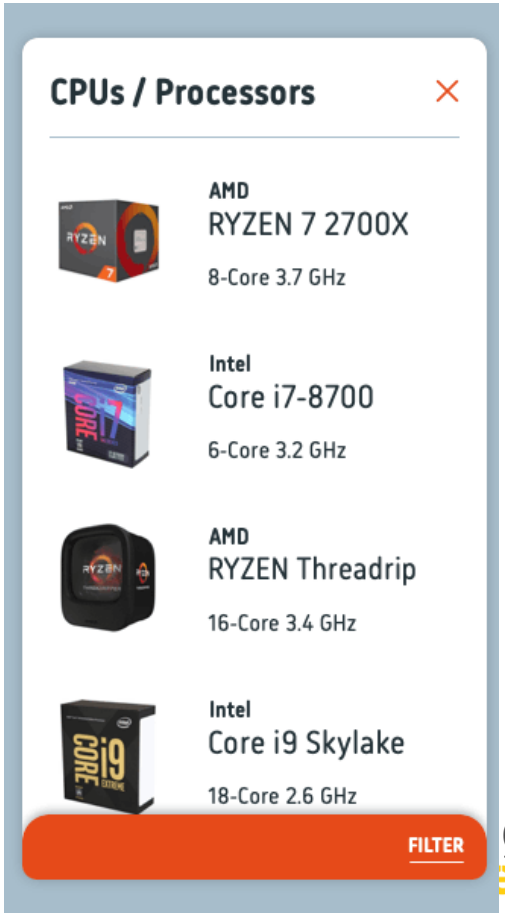
the <ItemFilter>

- It's where the user filters the list
- It usually reports an updated filter to his parent component (calls a provided callback)

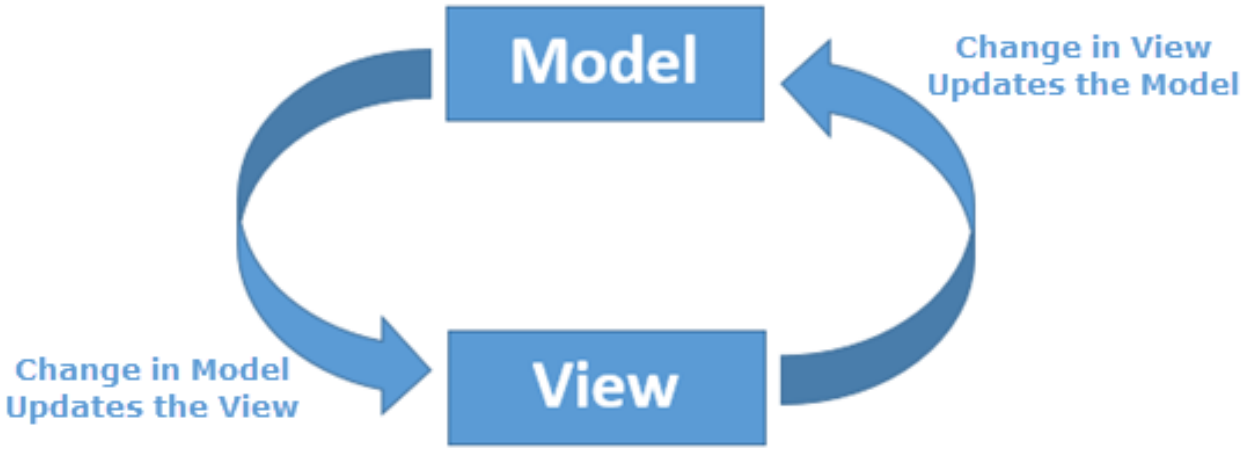
Lorem

New ▾ | Actions ▾ | Settings ▾

Title	First	Second	Third	Fourth	Fifth	Sixth	Seventh	Eighth
Lorem1\ NEW	Lorem1A\	\Lorem1B	Lorem1C	a	abcd	Yes	A	34
Lorem2 NEW	Lorem2A	Lorem2B	Lorem2C	ab	abcde	Yes	B	45
Lorem3 NEW	Lorem3A	Lorem3B	Lorem3C	abc	abcdef		C	56
Lorem4 NEW	Lorem1A	Lorem1B	Lorem1C	a	abcd	Yes	A	67
Lorem5 NEW	Lorem2A	Lorem2B	Lorem2C	ab	abcde	Yes	B	78
Lorem6 NEW	Lorem3A	Lorem3B	Lorem3C	abc	abcdef	No	C	89
Lorem7 NEW	Lorem1A	Lorem1B	Lorem1C	a	abcd	Yes	A	100
Lorem8 NEW	Lorem2A	Lorem2B	Lorem2C	ab	abcde	Yes	B	111
Lorem9 NEW	Lorem3A	Lorem3B	Lorem3C	abc	abcdef	No	C	122
Lorem10 NEW	Lorem1A	Lorem1B	Lorem1C	a	abcd	Yes	A	133
Lorem11 NEW	Lorem2A	Lorem2B	Lorem2C	ab	abcde	Yes	B	144
Lorem12 NEW	Lorem3A	Lorem3B	Lorem3C	abc	abcdef	No	C	155
Lorem13 NEW	Lorem1A	Lorem1B	Lorem1C	a	abcd	Yes	A	166
Lorem14 NEW	Lorem1A	Lorem1B	Lorem1C	a	abcd	Yes	A	177
Lorem15 NEW	Lorem2A	Lorem2B	Lorem2C	ab	abcde	Yes	B	188



two-way-data-binding



When working with inputs we need to create a two-way data bindings:

- The data is placed in the input value
- When user modifies its reflected in the model

two-way-data-binding

When using `<input>`, `<textarea>` and `<select>` we bind the value and the onChange event

Changes in the component's state are automatically reflected into the view and vise-versa

```
export function SimpleInput() {

    const [user, setUser] = useState({ fullname: 'Puki', score: 87 })
    function handleNameChange(ev) {
        const { value } = ev.target
        setUser((prevUser) => ({ ...prevUser, fullname: value }))
    }

    return <input type="text"
                  value={user.fullname}
                  onChange={handleNameChange}
    />

}
```

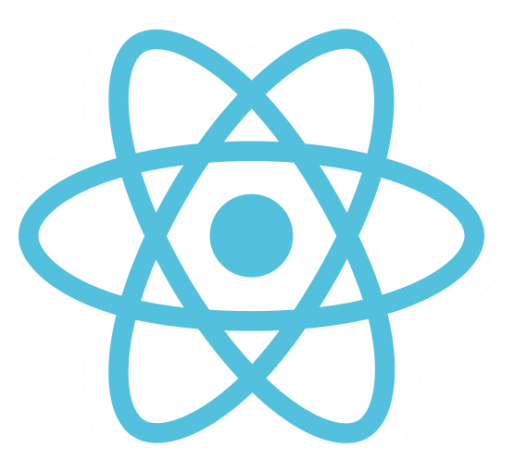
A simple form

Let's review

```
export function SimpleForm() {

    const [user, setUser] = useState({ fullname: 'Puki Reactof'})
    function handleNameChange(ev) {
        const { value } = ev.target
        setUser((prevUser) => ({ ...prevUser, fullname: value }))
    }

    function saveUser(ev) {
        ev.preventDefault()
        console.log('Saving user', user)
    }

    return <form onSubmit={saveUser}>
        <input type="text"
            name="fullname"
            value={user.fullname}
            onChange={handleNameChange}
            placeholder="Full name"
        />
        <button>Save</button>
    </form>
}
```

coding academy

ReactJS

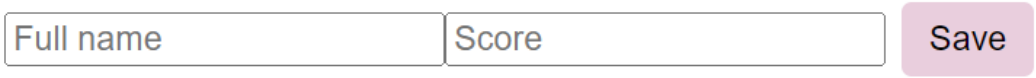


coding academy

```jsx
export function SimpleForm() {

    const [user, setUser] =
       useState({ fullname: 'Puki Reactof', score: 87 })

    function handleChange(ev) {
        let { value , type , name: field } = ev.target
        value = (type === 'number') ? (+value || '') : value
        setUser((prevUser) => ({ ...prevUser, [field] : value }))
    }
    function saveUser(ev) {
        ev.preventDefault()
        console.log('Saving user', user)
    }

    return <form onSubmit={saveUser}>
        <input type="text"
            name="fullname"
            value={user.fullname}
            onChange={handleChange}
            placeholder="Full name"
        />
        <input type="number"
            name="score"
            value={user.score}
            onChange={handleChange}
            placeholder="Score"
        />
        <button>Save</button>
    </form>
}
```

A simple form

Let's review



coding academy

Forms are tricky

The `handleChange` function should handle different types of inputs: and can look a bit different for different forms.

For example, in the previous example, we need to handle numbers correctly:

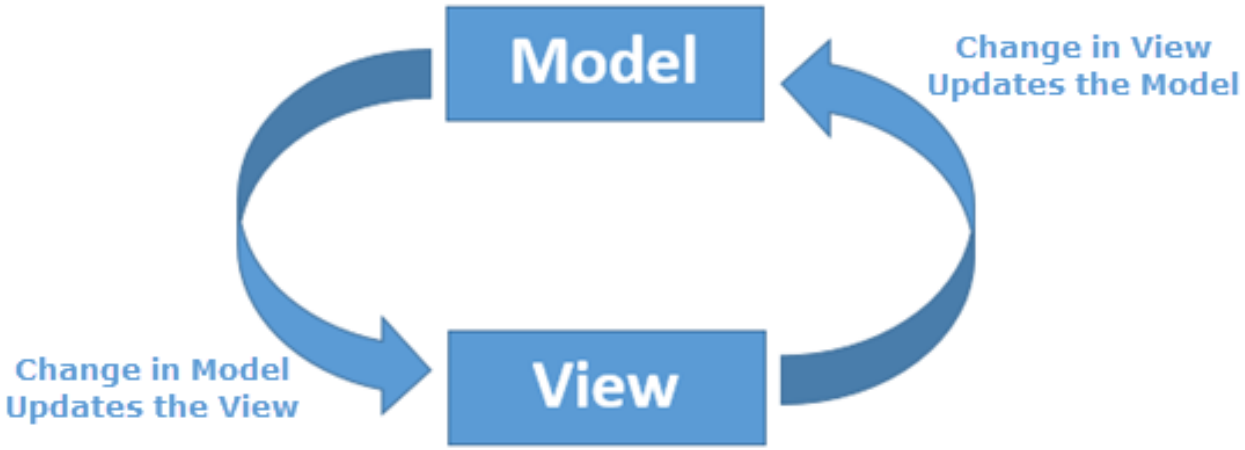
```
const [user, setUser] = useState({ fullname: 'Puki Reactof', score: 87 })
function handleChange(ev) {
    let { value , type , name: field } = ev.target
    value = (type === 'number') ? (+value || '') : value
    setUser((prevUser) => ({ ...prevUser, [field] : value }))
}
```

Forms are tricky

If we add a checkbox and a range inputs,
we will add some more conditions:

```
function handleChange({target}) {
    var {value, name: field} = target
    switch (target.type) {
        case 'range':
        case 'number':
            value = +target.value || 0
            break
        case 'checkbox':
            value = target.checked
            break

    }
    setUser((prevUser) => ({ ...prevUser, [field] : value }))
}
```

Revisit: two-way-data-binding



We created a two-way data bindings:

- The data is placed in the input value
- When user modifies its reflected in the model

coding academy

Form validity

The validity of the form, can be computed:

```
const isValid = user.fullname && user.score >= 0

<button disabled={!isValid}>Save</button>
```

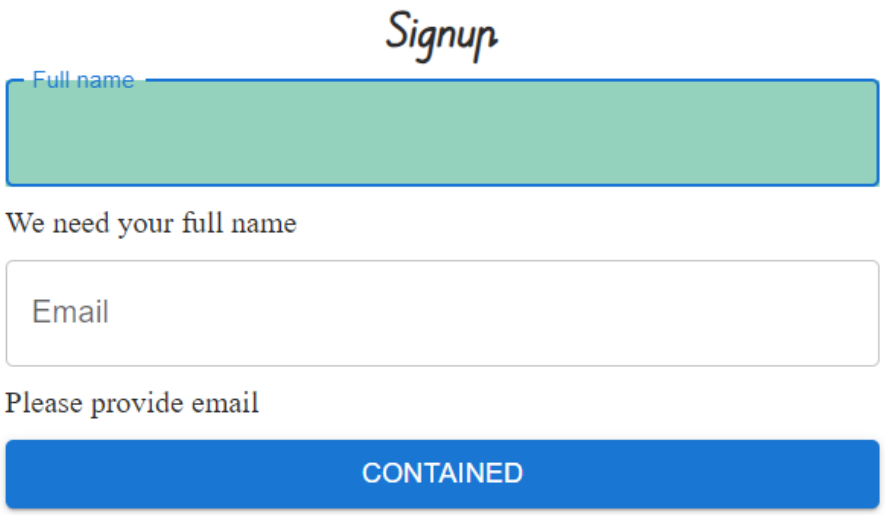
Forms

Forms are sometimes more complex

We will later meet libraries that focus on forms, such as:

```
const SignupSchema = Yup.object().shape({
    fullName: Yup.string()
        .min(2, 'Too Short!')
        .max(50, 'Too Long!')
        .required('We need your full name'),
    email: Yup.string().email('Invalid email').required('Please provide email'),
})
```

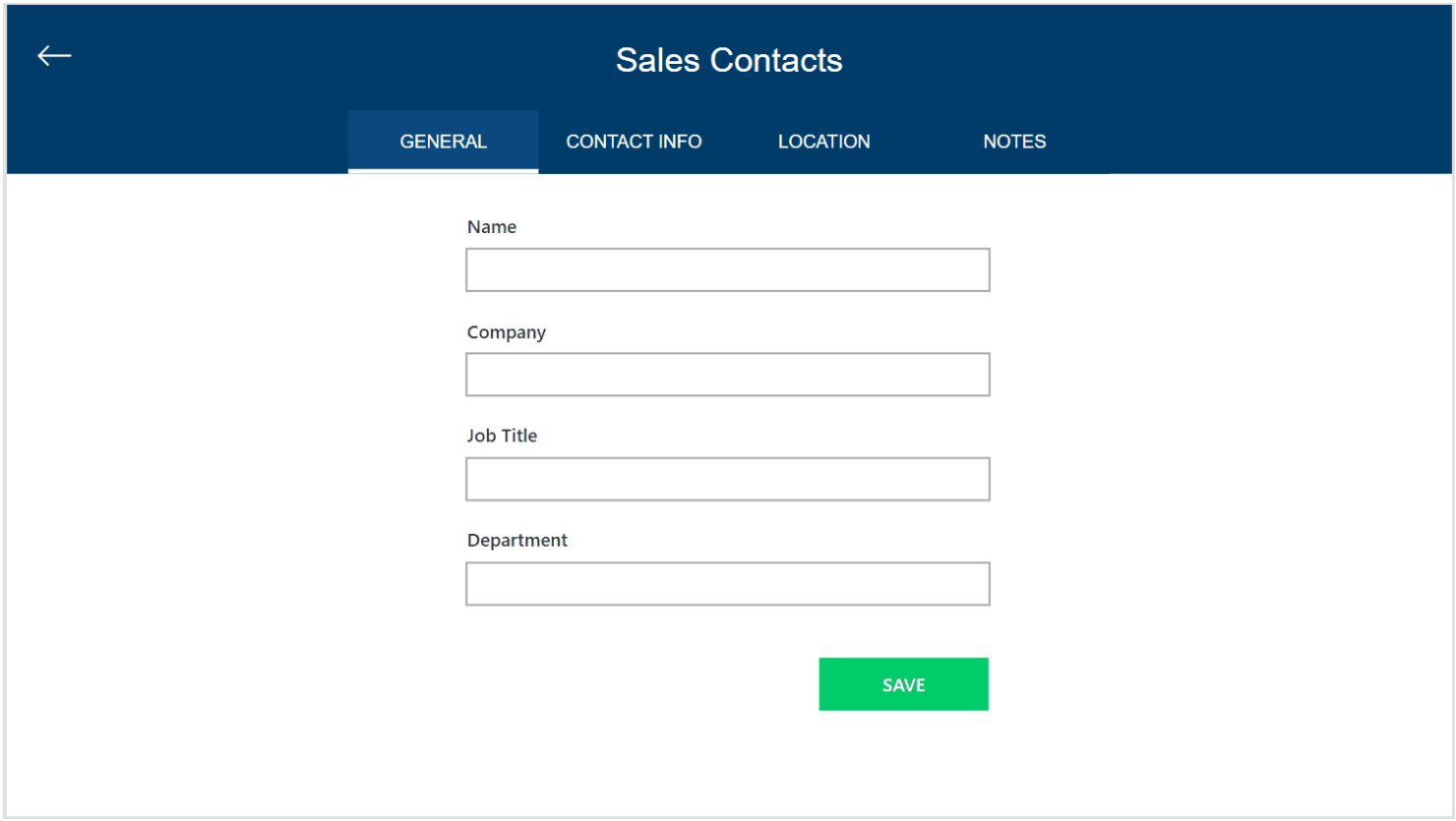
But for now we are good



the <ItemEdit>

It’s where the user edits the item

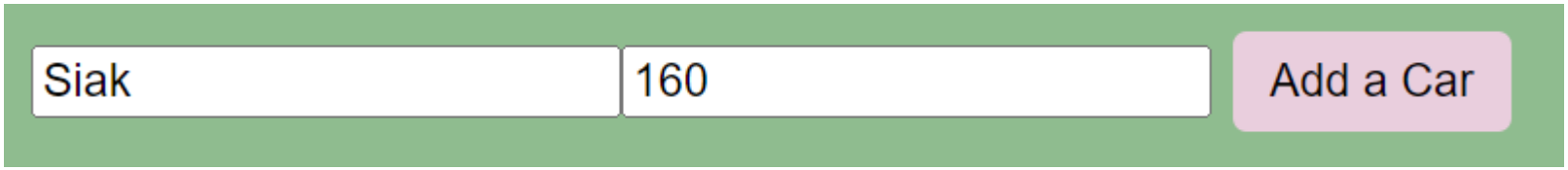
It usually works directly with the service for adding and updating an item



Form Submit

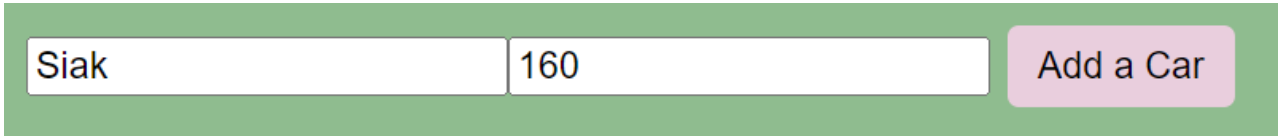
We will usually wrap inputs inside a form, handle the submit event and prevent the default reload

```
function addCar(ev) {
    ev.preventDefault()
    console.log('Saving car', car)
}
```



Form Submit

Let’s add a car

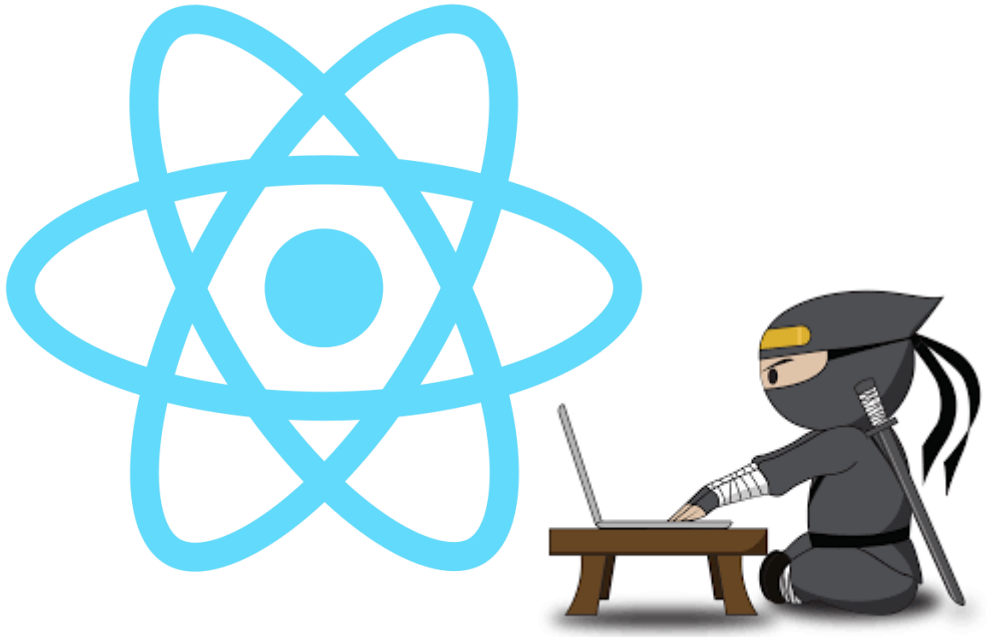



```
function addCar(ev) {
    ev.preventDefault()
    carService.save(this.carToEdit)
        .then(savedCar => {
           setCars([...cars, savedCar])

        })
        .catch(err => {
            console.error('Failed to save car', err)
            // TODO: show error msg to user
        })
}
```

Level Done

App Structuring

Handson Time



coding academy

Animations

Animations are an important tool, lets use animate.css

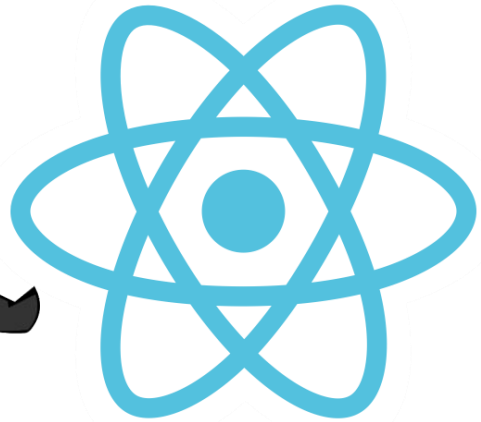
```
// In our utilService

function animateCSS(el, animation) {
    const prefix = 'animate__'
    return new Promise((resolve, reject) => {
        const animationName = `${prefix}${animation}`

        el.classList.add(`${prefix}animated`, animationName)

        function handleAnimationEnd(event) {
            event.stopPropagation()
            el.classList.remove(`${prefix}animated`, animationName)
            resolve('Animation ended')
        }

        el.addEventListener('animationend', handleAnimationEnd, { once: true })
    })
}
```

Animations

Now, we can easily animate elements

```
// use a ref to animate some element
const titleRef = useRef()
…
<h3 ref={titleRef}>Here is some Ipsum</h3>
…
function onSomething() {
   utilService.animateCSS(titleRef.current, 'pulse')
}
```

ReactJS



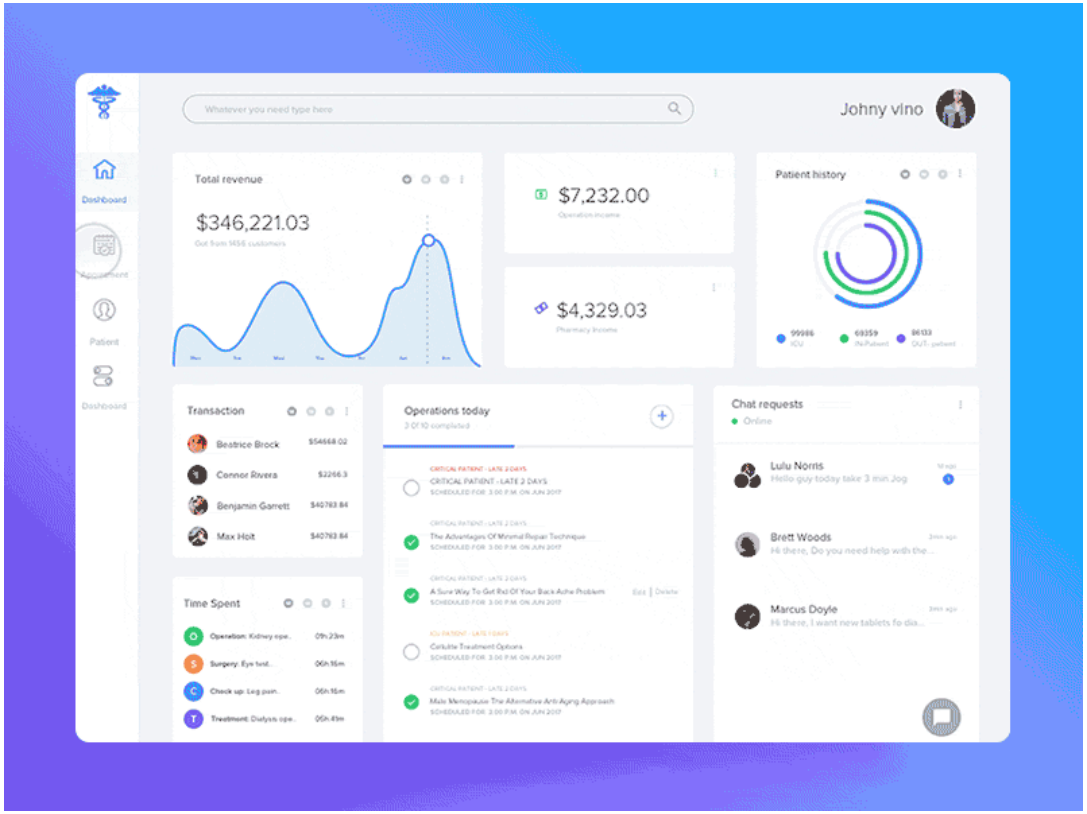
Routing

and Single Page Applications

coding academy

Single Page Apps (SPA)

Single page application (SPA) allows the user to navigate between views without reloading the entire page



coding academy

Routing in Single Page Application

- Add the needed libs to your project
  - react-router, react-router-dom, history

```
// in RootCmp.js
const Router = ReactRouterDOM.HashRouter
const { Route} = ReactRouterDOM

<Router>
    <section className="app"></section>
</Router>
```

<Routes> and <Route>

Next, we match paths with components (pages):

```
import { Home } from './pages/Home.jsx'
import { About } from './pages/About.jsx'

const { Routes, Route } = ReactRouterDOM
//...

<Routes>
    <Route path="/" element={<Home />} />
    <Route path="/about" element={<About />} />
</Routes>
```

Navigation with <Link>

We use the <Link> component

```
const { Link } = ReactRouterDOM

//...
<Link to='/car'>Back to list</Link>
```

We can examine the DOM, it renders <a> elements

Navigation with <NavLink>

In our main navigation, let's show the user which link (page) is currently active

we will use NavLinks:


```
const { NavLink } = ReactRouterDOM

export function AppHeader() {
    return <header className="app-header"
        <h1>React Car App</h1>
        <nav>
            <NavLink to='/'>Home</NavLink> |
            <NavLink to='/about'>About</NavLink> |
            <NavLink to='/car'>Explore Cars</NavLink>
        </nav>
    </header>
}
```

Home Sweet Home

```
nav a.active {
    color: yellow;
}
```

coding academy

Route Params

- Setting Up Route Parameters:

```
<Route path="/car/:carId" component={CarDetails}/>
```

- Getting the Route Parameters in the matched component:

```
// in CarDetails.jsx
const { carId } = useParams()
```

Imperative Navigation

```
export function MyCmp() {
    const navigate = useNavigate()

    function onSaveCar() {
        // save the car
        // then navigate
        navigate('/car')
    }
}
```

- Meaning - Navigating from Code instead of using <Link> or <NavLink>
- For example – in the EditItem page, after the ItemService reports a successful save, we want to navigate back to the list

<CarDetails>

Let's examine

```
import { carService } from '../services/car.service.js'
const { useParams, useNavigate, Link } = ReactRouterDOM

export function CarDetails() {

    const [car, setCar] = useState(null)
    const params = useParams()
    const navigate = useNavigate()

    useEffect(() => {
        loadCar()
    }, [])

    function loadCar() {
        carService.get(params.carId)
            .then(setCar)
            .catch(err => {
                console.error('err:', err)
                showErrorMsg('Cannot load car')
                navigate('/car')
            })
    }
```

coding academy

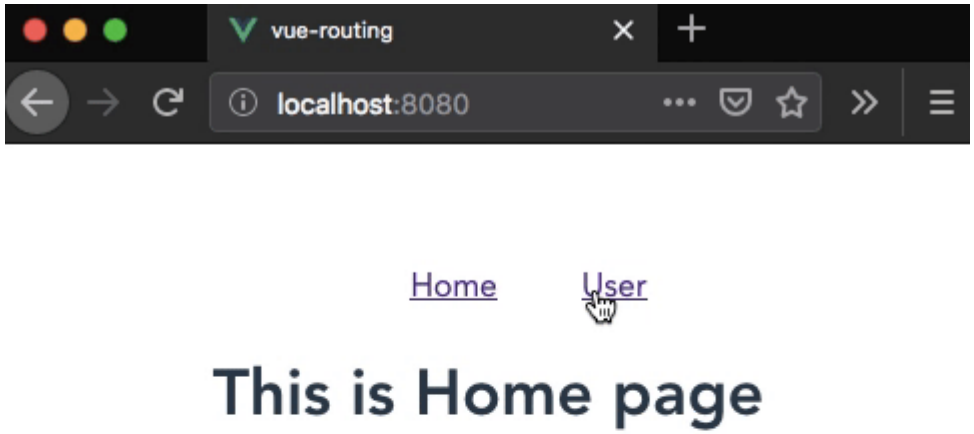
<CarDetails>

Let's examine

```
    function onBack() {
        // If nothing to do here, better use a Link
        navigate('/car')
        // navigate(-1)
    }

    if (!car) return <div>Loading...</div>
    return (
        <section className="car-details">
            <h1>Car Vendor: {car.vendor}</h1>
            <h1>Car Speed: {car.maxSpeed}</h1>
            <button onClick={onBack}>Back</button>
        </section>
    )
}
```

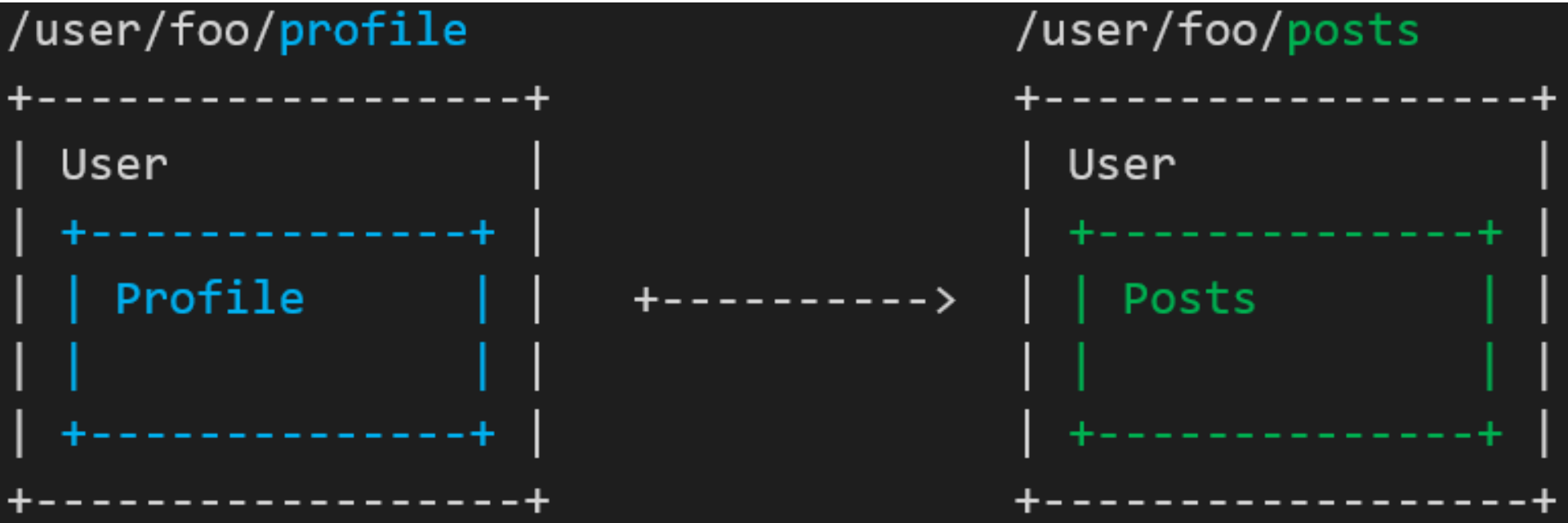
Nested Routes

Larger apps may be composed of several levels of routing.



Nested Routes

Segments of the URL corresponds to a certain structure of nested components



Nested Routes

Let's add some nested routes in our app:

```
<Routes>
    <Route element={<Home />} path="/" />
    <Route element={<AboutUs />} path="/about" >
        <Route element={<AboutIndex />} path="/about" />
        <Route element={<AboutTeam />} path="/about/team" />
        <Route element={<AboutGoal />} path="/about/goal" />
    </Route>
    <Route element={<CarIndex />} path="/car" />
</Routes>
```

The Next / Previous scenario

- We want to add: Previous car and Next car buttons to our app
- This will provide the user another navigation method between different cars

coding academy

Getting the next/prev cars

When getting a full car by id,
let's add the needed data:

```
// car.service.js
function get(carId) {
    return storageService.get(CAR_KEY, carId)
        .then(_setNextPrevCarId)
}


function _setNextPrevCarId(car) {
    return storageService.query(CAR_KEY).then((cars) => {
        const carIdx = cars.findIndex((currCar) => currCar.id === car.id)
        const nextCar = cars[carIdx + 1] ? cars[carIdx + 1] : cars[0]
        const prevCar = cars[carIdx - 1] ? cars[carIdx - 1] : cars[cars.length-1]

        car.nextCarId = nextCar.id
        car.prevCarId = prevCar.id
        return car
    })
}
```

The Next / Previous scenario

- In our <CarDetails> Let's add a link to the next car:

```
<Link to={`/car/${car.nextCarId}`}>Next car</Link>
```

- Note that when routing from '/car/xxx' to '/car/yyy' the Route's path remains '/car/:id'

- So, the component stays alive and doesn't re-mount

- Still, we need to get the data of another car and update the cmp's data accordingly

- Let's see how

Reacting to Route Params changes

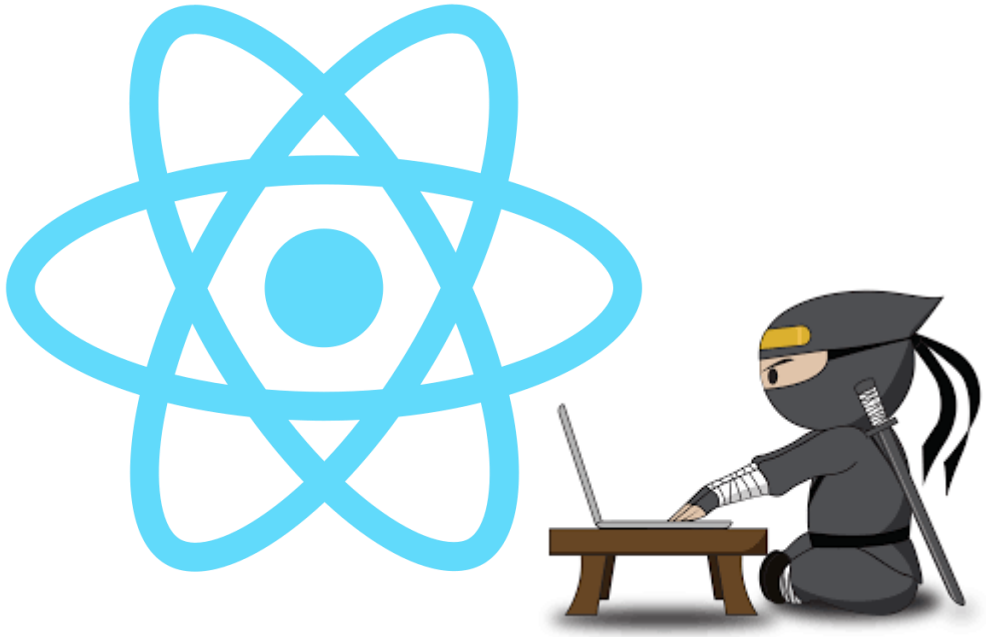
Whenever the `carId` changes –
we reload the data:

```
const [car, setCar] = useState(null)
const params = useParams()

useEffect(() => {
    loadCar()
}, [params.carId])

function loadCar() {
    carService.get(params.carId)
        .then(setCar)
        .catch(err => {
            console.error('err:', err)
            showErrorMsg('Cannot load car')
            navigate('/car')
        })
}
```

Level Done

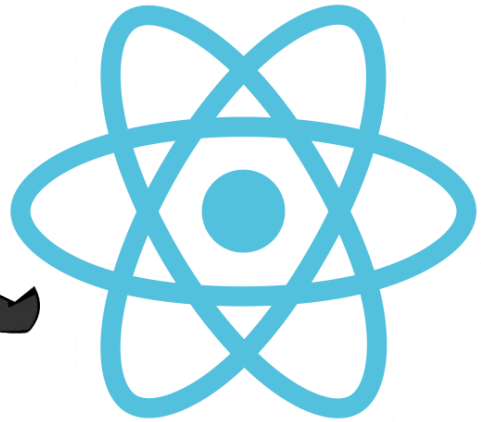
SPA Routing

Handson Time



coding academy

ReactJS



Some Extra Features

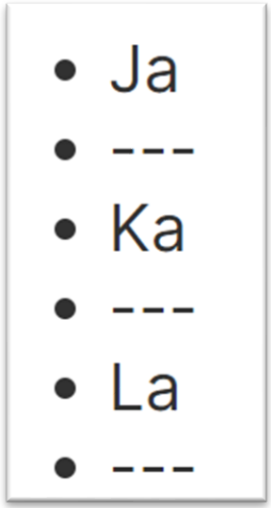
React.Fragment

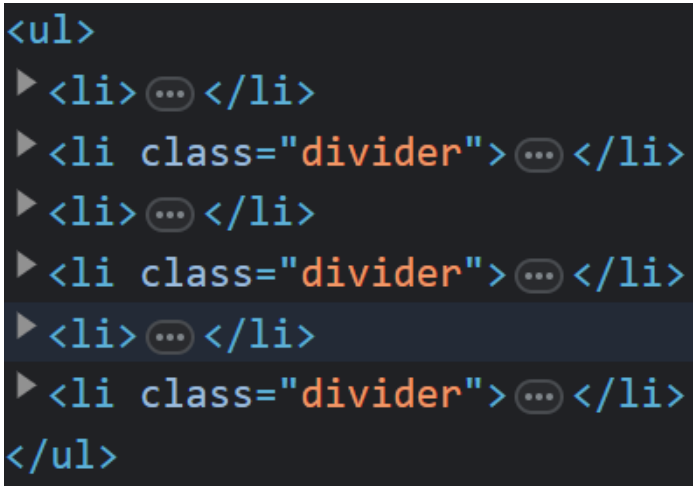
Sometimes it is needed for a component to return multiple elements.

Fragments are used to group some children without adding extra parent node to the DOM

Example:

```
<ul>
   {['Ja', 'Ka', 'La'].map(name =>
      <React.Fragment>
         <li>{name}</li>
         <li className="divider">---</li>
      </React.Fragment>)}
</ul>
```

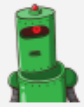




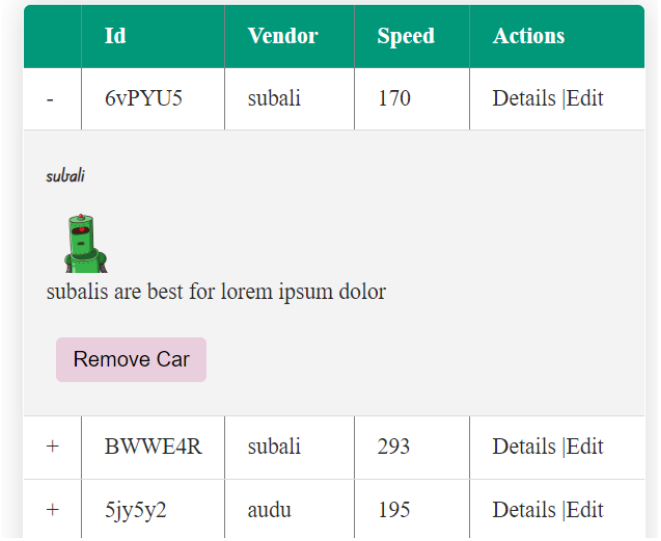
coding academy

React.Fragment - Example

Let's look at another example
we need to build a table with an option to expand the row for extra details:

	Id	Vendor	Speed	Actions
-	6vPYU5	subali	170	Details |Edit
	subali  subalis are best for lorem ipsum dolor Remove Car			
+	BWWE4R	subali	293	Details |Edit
+	5jy5y2	audu	195	Details |Edit

Example: DataTable




```
import { DataTableRow } from "./DataTableRow.jsx"

export function DataTable({ cars, onRemoveCar }) {
    return <table border="1">
        <thead>
            <tr>
                <th></th>
                <th>Id</th>
                <th>Vendor</th>
                <th>Speed</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>
            {cars.map(car =>
               <DataTableRow key={car.id} car={car} onRemoveCar={onRemoveCar} />)}
        </tbody>
    </table>
}
```

```jsx
const { useState, Fragment } = React
const { Link } = ReactRouterDOM
export function DataTableRow({car, onRemoveCar}) {

    const [isExpanded, setIsExpanded] = useState(false)

    return <Fragment>
        <tr>
            <td onClick={() => {
                    setIsExpanded(!isExpanded)
                }}>
                {(isExpanded) ? '-' : '+'}
            </td>

            <td>{car.id}</td>
            <td>{car.vendor}</td>
            <td>{car.maxSpeed}</td>
            <td>
                <Link to={`/car/${car.id}`}>Details</Link> |
                <Link to={`/car/edit/${car.id}`}>Edit</Link>
            </td>
        </tr>
        <tr hidden={!isExpanded}>
            <td colSpan="5">
                <img src={`https://robohash.org/${car.id}`} style={{maxWidth: '50px'}} />
                <p>{car.vendor}s are best for lorem ipsum dolor</p>
                <button onClick={() => onRemoveCar(car.id)}>Remove Car</button>
            </td>
        </tr>
    </Fragment>
}
```

DataTableRow

	Id	Vendor	Speed	Actions
-	6vPYU5	subali	170	Details |Edit

subali

subalis are best for lorem ipsum dolor

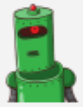
Remove Car

+	BWWE4R	subali	293	Details |Edit
+	5jy5y2	audu	195	Details |Edit

We have an expandable table!

	Id	Vendor	Speed	Actions
-	6vPYU5	subali	170	Details |Edit

subali



subalis are best for lorem ipsum dolor

Remove Car

+	BWWE4R	subali	293	Details |Edit
+	5jy5y2	audu	195	Details |Edit

Using Style

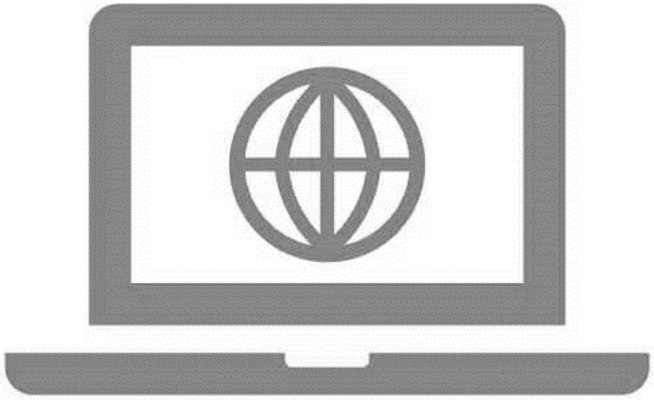
- Generally, we prefer using classNames over using the style attribute
- However, inline styling is sometimes needed for setting dynamically-computed values
- The style attribute accepts a JS object with camelCased properties

```
<span title={item.title}
   style={{ height: item.value + '%' }}>
       {item.value + '%'}
</span>
```

```
const msgStyle = {
    margin: '10px',
    backgroundImage: 'url(' + imgUrl + ')',
}

function Cmp() {
    return <div style={msgStyle}>Great day to be alive</div>
}
```

Car service

Let's add some grouping

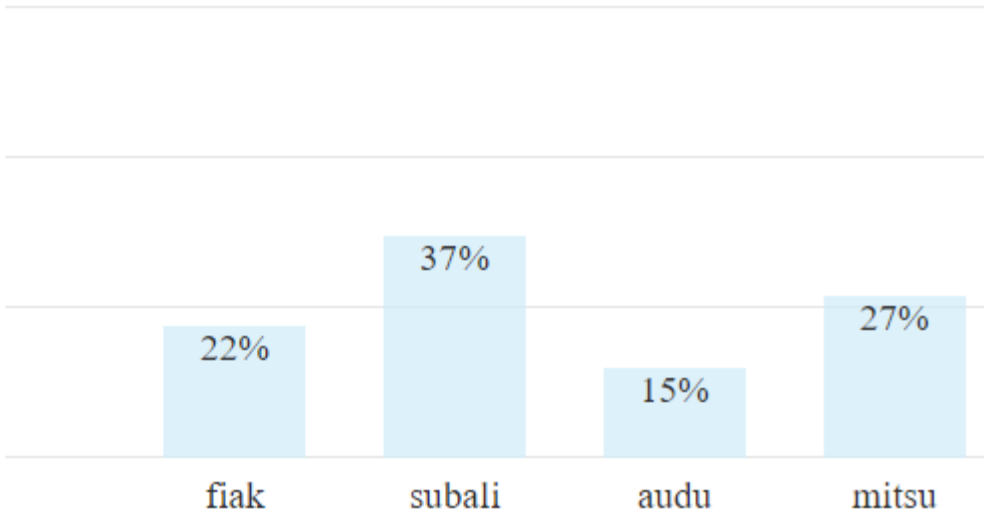


Example: Cars Dashboard

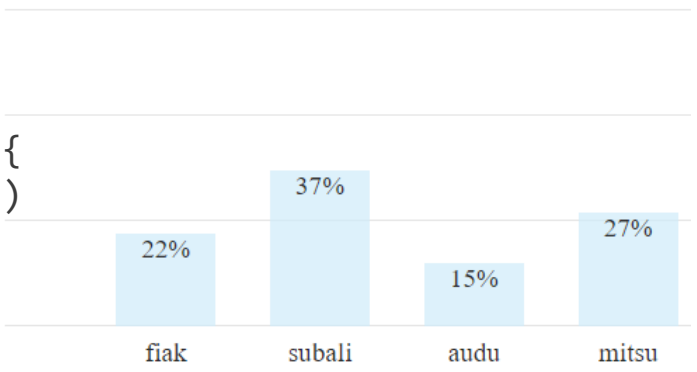
Dashboard

Statistics for 41 Cars

By Vendor



Cars Demo Data




```
// in car.service
function _createCar(vendor, maxSpeed = 250) {
    const car = getEmptyCar(vendor, maxSpeed)
    car.id = utilService.makeId()
    return car
}
```

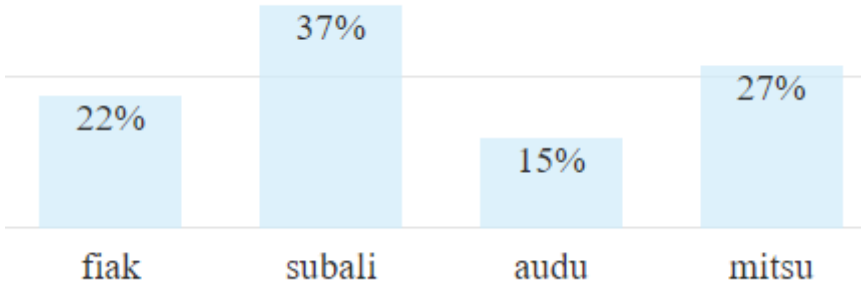
```
function _createCars() {
    let cars = utilService.loadFromStorage(CAR_KEY)
    if (!cars || !cars.length) {
        cars = []
        const vendors = ['audu', 'fiak', 'subali', 'mitsu']
        for (let i = 0; i < 41; i++) {
            const vendor =
              vendors[utilService.getRandomIntInclusive(0, vendors.length - 1)]
            cars.push(_createCar(vendor, utilService.getRandomIntInclusive(80, 300)))
        }
        utilService.saveToStorage(CAR_KEY, cars)
    }
}
```

coding academy

<Chart> component

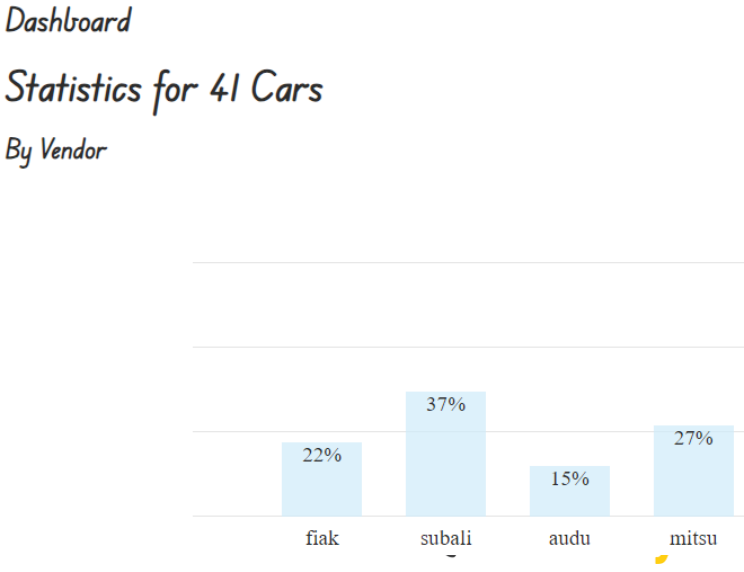
```
▼ (4) [{…}, {…}, {…}, {…}] i
  ▶ 0: {title: 'fiak', value: 22}
  ▶ 1: {title: 'subali', value: 37}
  ▶ 2: {title: 'audu', value: 15}
  ▶ 3: {title: 'mitsu', value: 27}
    length: 4
```

```
export function Chart({ data }) {
    return (<ul className="chart">
        {
            data.map((item) => <li key={item.title}>
                <span title={item.title}
                      style={{ height: item.value + '%' }}>
                      {item.value + '%'}
                </span>
            </li>)
        }
    </ul>)
}
```



Example: Cars Dashboard

```jsx
const { useEffect, useState } = React
import {Chart} from '../cmps/Chart.jsx'
import { carService } from '../services/car.service.js'

export function Dashboard() {

    const [cars, setCars] = useState([])
    const [vendorStats, setVendorStats] = useState([])

    useEffect(()=>{
        carService.query()
            .then(setCars)
        carService.getVendorStats()
            .then(setVendorStats)
    }, [])


    return (
        <section className="dashboard">
            <h1>Dashboard</h1>
            <h2>Statistics for {cars.length} Cars</h2>
            <h4>By Vendor</h4>
            <Chart data={vendorStats}/>
        </section>
    )
}
```



Cars Grouped by Vendor

```
// in car.service
function _getCarCountByVendorMap(cars) {
    const carCountByVendorMap = cars.reduce((map, car) => {
        if (!map[car.vendor]) map[car.vendor] = 0
        map[car.vendor]++
        return map
    }, {})
    return carCountByVendorMap
}
```

```
{fiak: 9, subali: 15, audu: 6, mitsu: 11}
```

```
function getVendorStats() {
    return storageService.query(CAR_KEY)
        .then(cars => {
            const carCountByVendorMap = _getCarCountByVendorMap(cars)
            const data = Object.keys(carCountByVendorMap)
                .map(vendor =>
                    ({ title: vendor,
                       value: Math.round((carCountByVendorMap[vendor]
                                          / cars.length) * 100) }))
            return data
        })
}
```

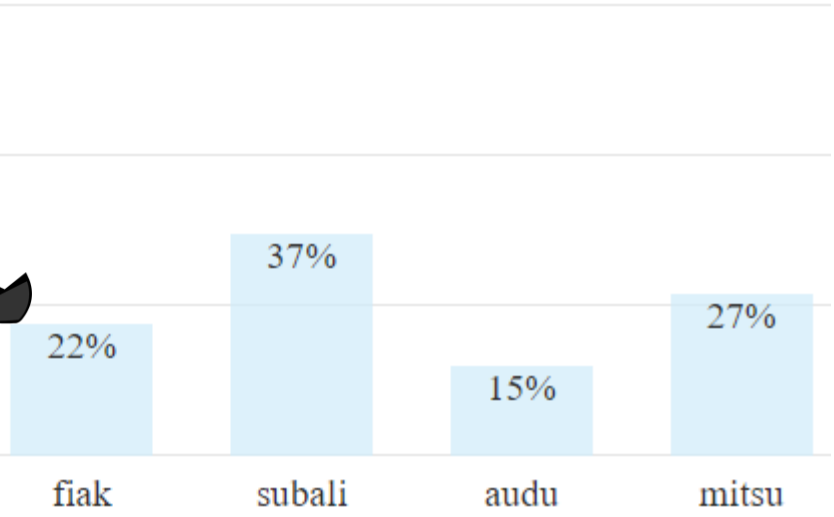
```
▼ (4) [{…}, {…}, {…}, {…}] i
  ▶ 0: {title: 'fiak', value: 22}
  ▶ 1: {title: 'subali', value: 37}
  ▶ 2: {title: 'audu', value: 15}
  ▶ 3: {title: 'mitsu', value: 27}
    length: 4
```

We have a Dashboard

Dashboard

Statistics for 41 Cars

By Vendor



{coding academy

React - Search Params

Let's integrate

127.0.0.1:5502/index.html#/car?txt=f&minSpeed=200

Filter Cars

Vendor: f

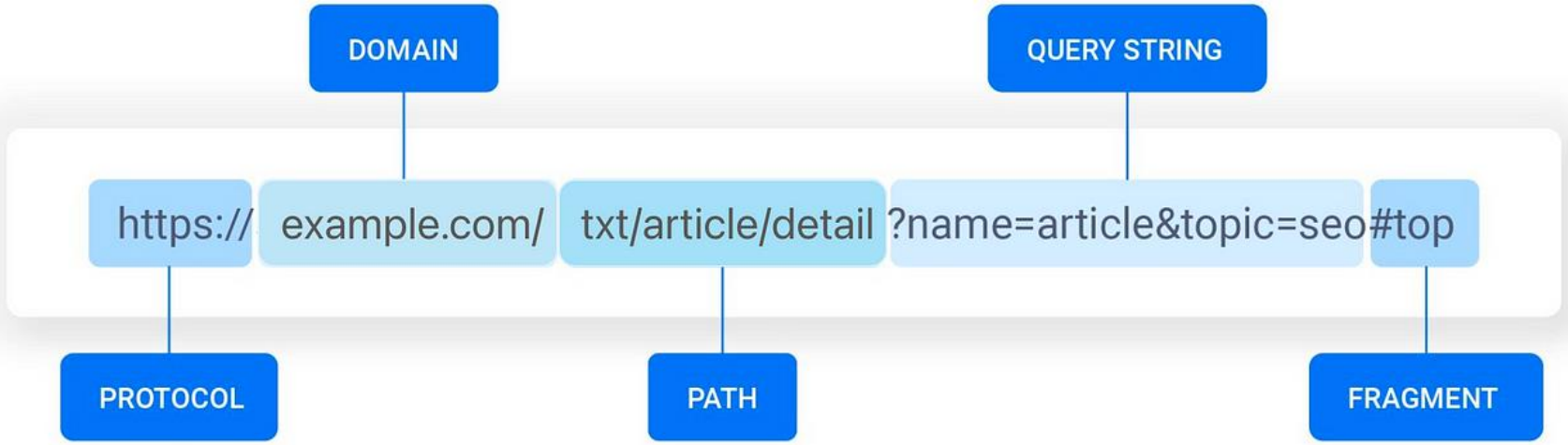
Min Speed: 200

coding academy

Working with Search Params

(AKA Query-Params, Query-String)

- Reminder: Search params are a defined set of parameters attached to the end of a url.
- They are used to help define specific content or actions based on the data being passed

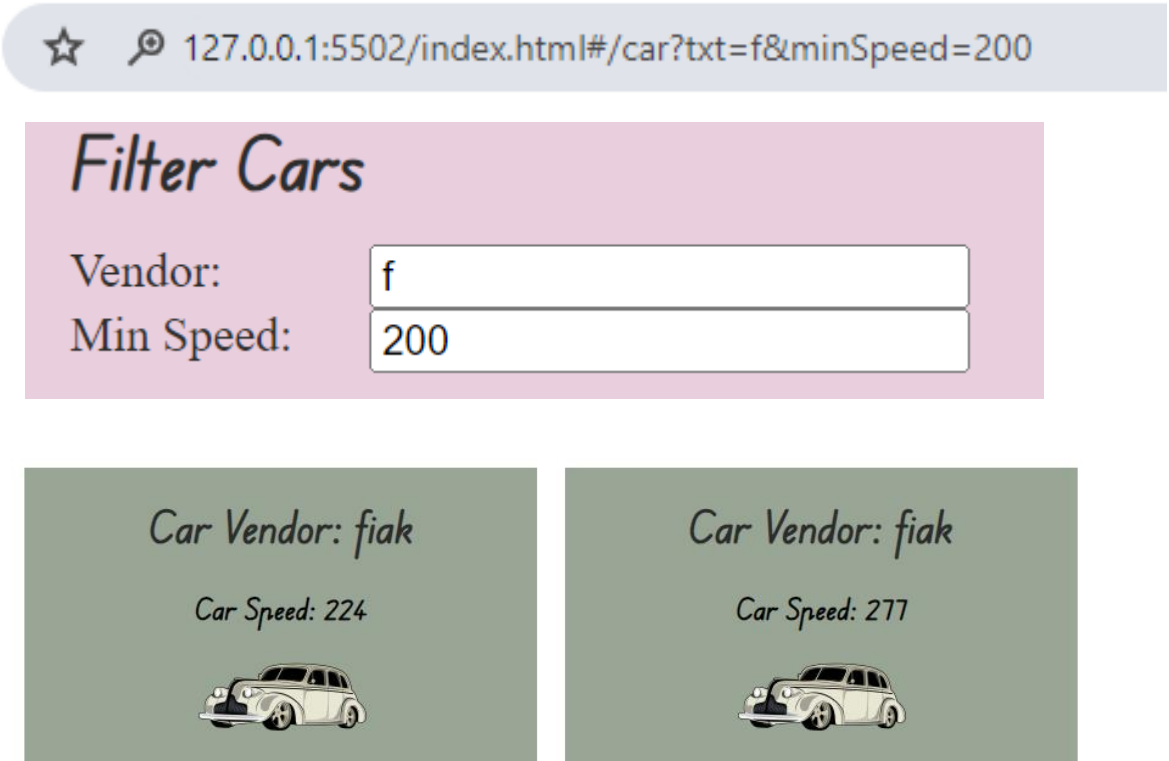


coding academy

Working with Search Params

Let's integrate

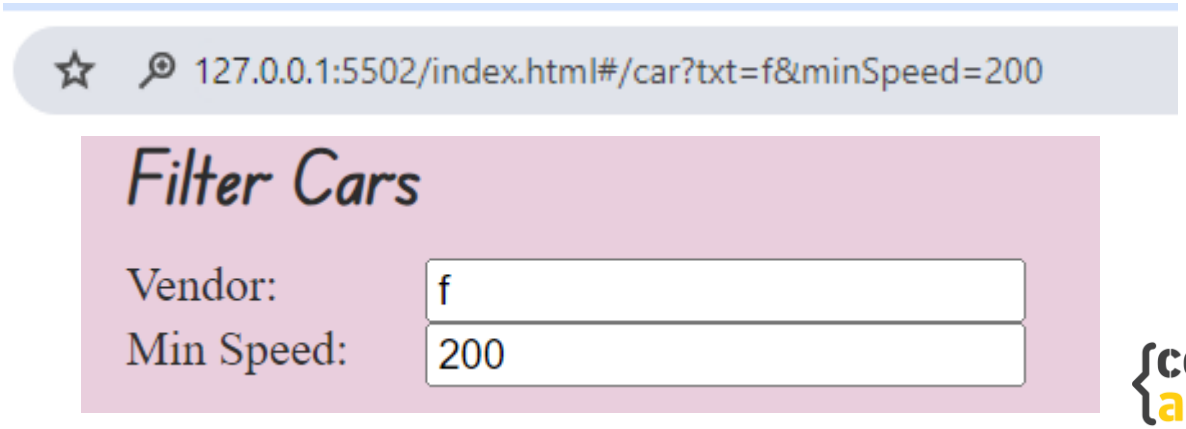
- Let's integrate search params in our car filtering
- This will make the results - bookmarkable
- Meaning that I can send someone a link to filtered results



coding academy

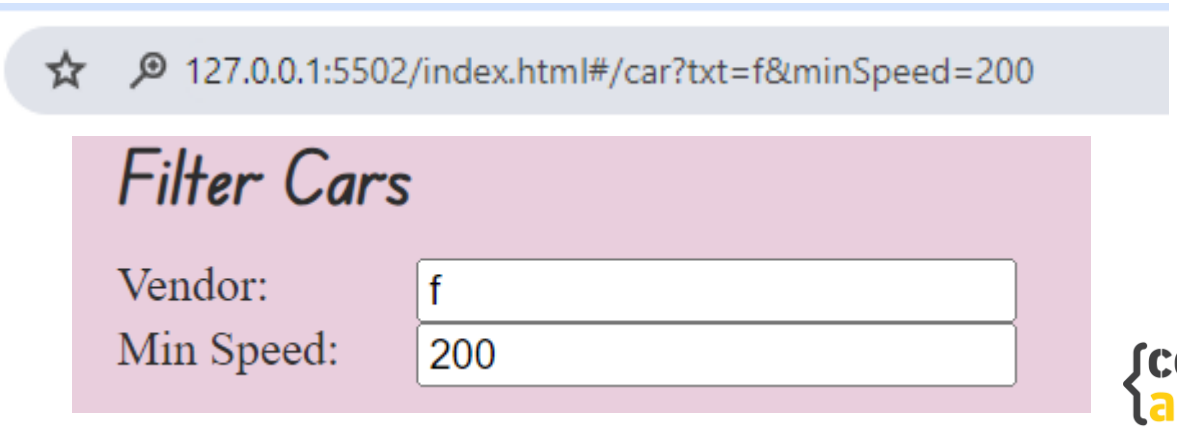
Working with Search-Params

```
// in CarIndex:

const { useSearchParams } = ReactRouterDOM

// Special hook for accessing search-params:
const [searchParams, setSearchParams] = useSearchParams()

const defaultFilter = carService.getFilterFromSearchParams(searchParams)

const [filterBy, setFilterBy] = useState(defaultFilter)

useEffect(() => {
        setSearchParams(filterBy)
        carService.query(filterBy)
            .then(cars => setCars(cars))
            .catch(err => {
                console.eror('err:', err)
                showErrorMsg('Cannot load cars')
            })
}, [filterBy])
```



Working with Search-Params

```
// in CarService:

function getDefaultFilter() {
    return { txt: '', minSpeed: 0 }
}

function getFilterFromSearchParams(searchParams) {
    const defaultFilter = getDefaultFilter()
    const filterBy = {}
    for (const field in defaultFilter) {
        filterBy[field] = searchParams.get(field) ||
                          defaultFilter[field]
    }
    return filterBy
}
```



We have search-params integration

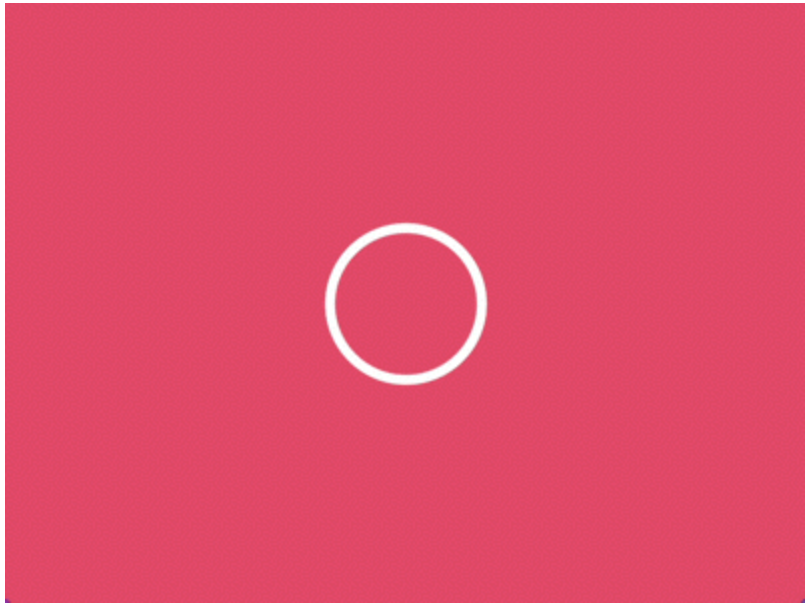
127.0.0.1:5502/index.html#/car?txt=f&minSpeed=200

Filter Cars

Vendor: f

Min Speed: 200

coding academy



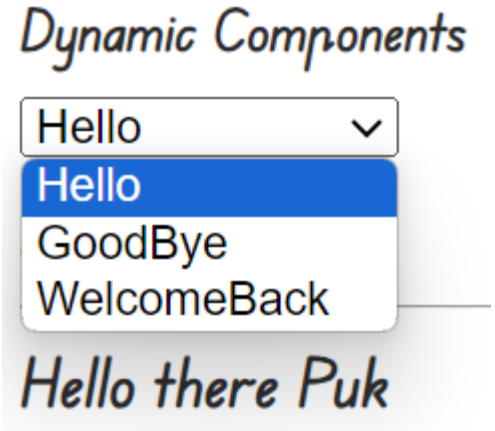
Dynamic Components

Dynamic Components

It is sometimes useful to work with dynamic components

Let's start with a simple example:

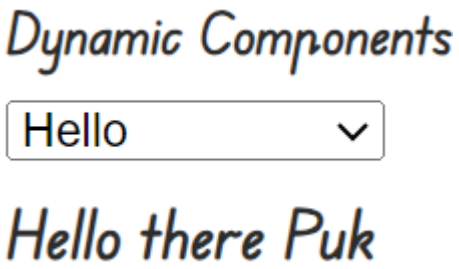
```
function Hello({ name }) {
    return <h1>Hello there {name}</h1>
}
function GoodBye({ name }) {
    return <h1>Bye {name}</h1>
}
function WelcomeBack({ name }) {
    return <h1>Welcome back {name}</h1>
}
```



Dynamic Components

The `DynamicCmp` is a component (function) that returns the correct component

```
function DynamicCmp(props) {
     switch (props.cmpType) {
          case 'Hello':
               return <Hello {...props} />
          case 'GoodBye':
               return <GoodBye {...props} />
          case 'WelcomeBack':
               return <WelcomeBack {...props} />
     }
}
```



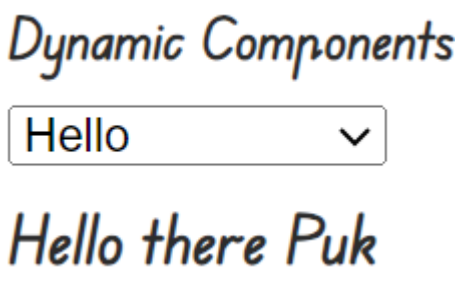
This idea is also called: Higher-Order-Component

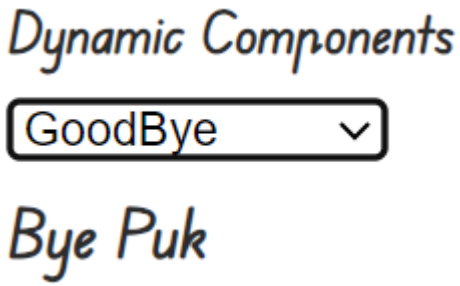
Dynamic Components

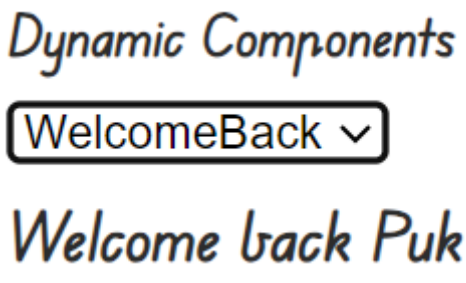
Let's allow the user to switch the component:

```
const [cmpType, setCmpType] = useState('Hello')
```

```
<h4>Dynamic Components</h4>
<select onChange={ev => setCmpType(ev.target.value)}>
    <option>Hello</option>
    <option>GoodBye</option>
    <option>WelcomeBack</option>
</select>
<DynamicCmp name="Puk" cmpType={cmpType} />
```







Dynamic Components

Here is another example, looping through some components:

```
const pageCmps = ['Hello', 'Hello', 'WelcomeBack']
```

```
{
    pageCmps.map((ct, idx) =>
        <DynamicCmp name="Puk" cmpType={ct} key={idx} />)
}
```

Hello there Puk

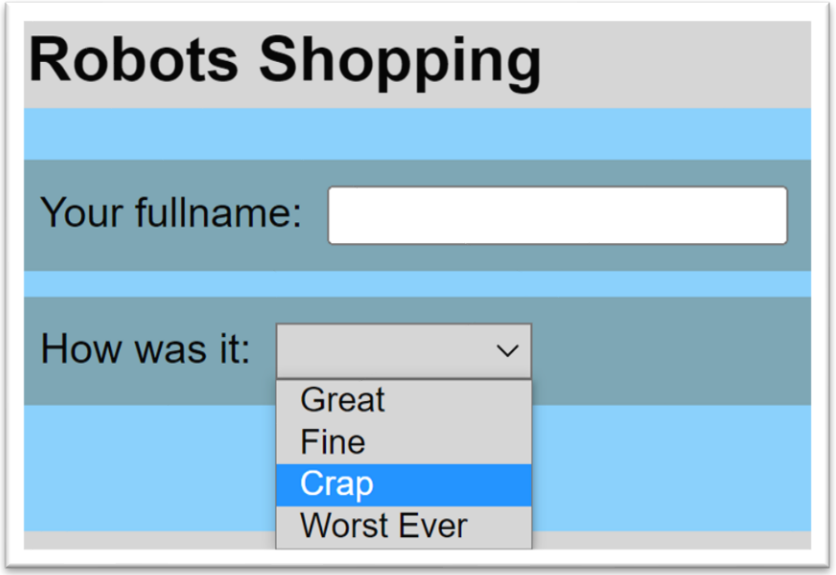
Hello there Puk

Welcome back Puk

Dynamic Components - Example

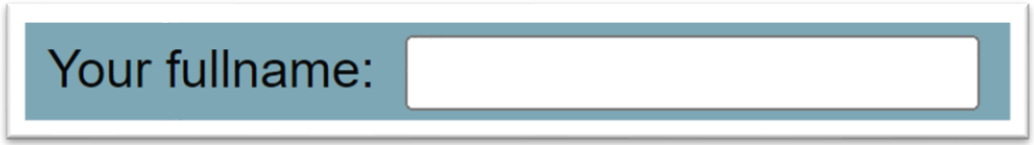
Let's build a dynamic survey based on the following JSON:

```
const survey = {
    title: 'Robots Shopping',
    cmps: [
        {
            type: 'TextBox',
            id: 'c101',
            info: {
                label: 'Your fullname:'
            }
        },
        {
            type: 'SelectBox',
            id: 'c102',
            info: {
                label: 'How was it:',
                opts: ['Great', 'Fine',
                'Crap', 'Worst Ever']
            }
        }
    ]
}
```



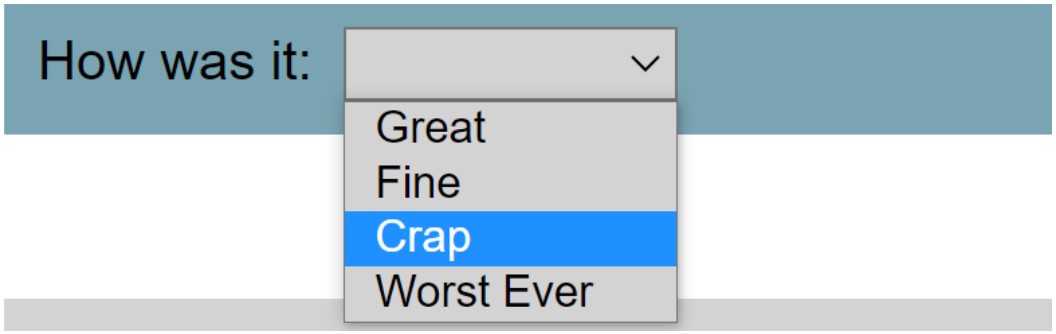
coding academy

<TextBox> component




```
function TextBox({ info, val = '', onChangeVal }) {
    const { label } = info
    return (
        <label>
            {label}
            <input type="text" value={val} onChange={(ev) => {
                onChangeVal(ev.target.value)
            }} />
        </label>
    )
}
```

<SelectBox> component



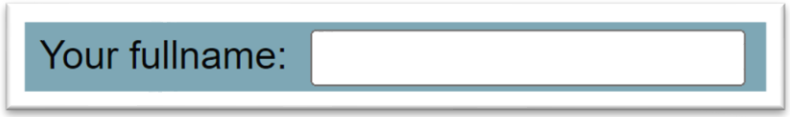

```
function SelectBox({ info, val = '', onChangeVal }) {
    const { label, opts } = info
    return (
        <label>
            {label}
            <select value={val} onChange={(ev) => {
                onChangeVal(ev.target.value)
            }}>
                <option value=""></option>
                {
                    opts.map(opt => <option key={opt}>{opt}</option>)
                }
            </select>
        </label>
    )
}
```

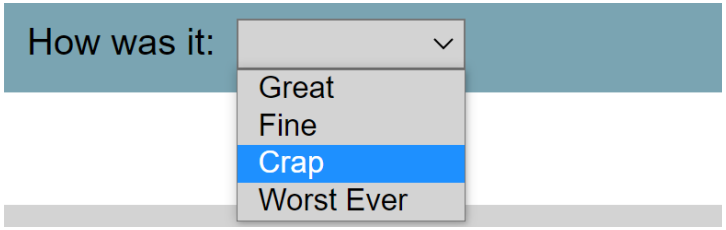
coding academy

The Dynamic Component

Here is the (higher-order) <DynamicCmp>:

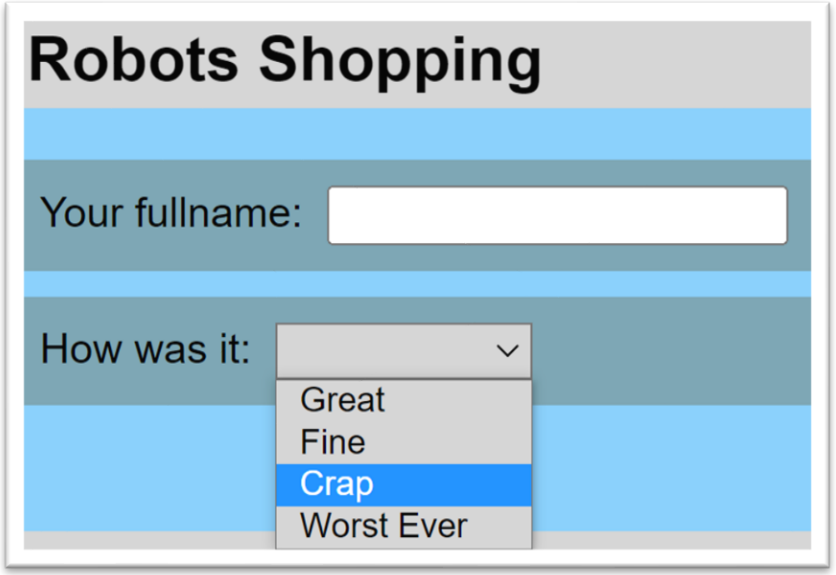
```
function DynamicCmp(props) {
    switch (props.type) {
        case 'TextBox':
            return <TextBox {...props} />
        case 'SelectBox':
            return <SelectBox {...props} />
    }
}
```





Back to our JSON

```
const survey = {
    title: 'Robots Shopping',
    cmps: [
        {
            type: 'TextBox',
            id: 'c101',
            info: {
                label: 'Your fullname:'
            }
        },
        {
            type: 'SelectBox',
            id: 'c102',
            info: {
                label: 'How was it:',
                opts: ['Great', 'Fine',
                'Crap', 'Worst Ever']
            }
        }
    ]
}
```



coding academy

Let's hold a map for the user answers

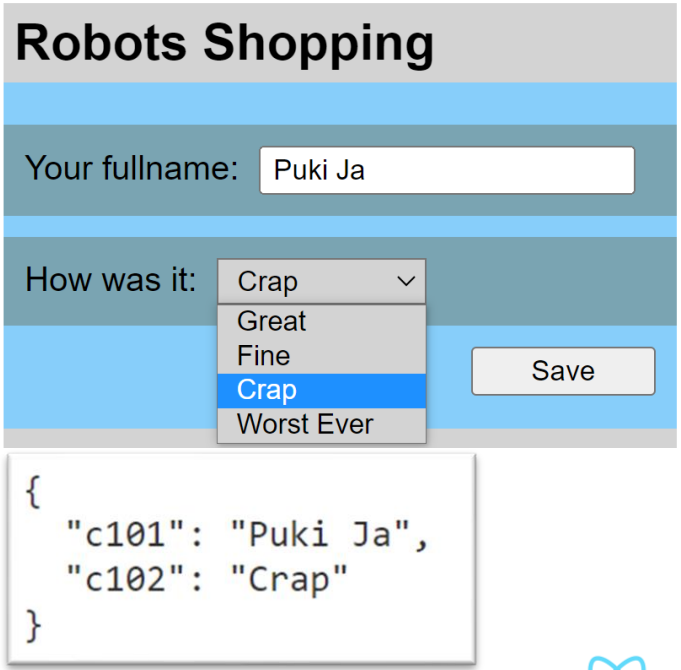
```
import { surveyService } from
     '../services/survey.service.js'

const { useState, useEffect } = React

export function SurveyIndex() {
    const [survey, setSurvey] = useState(null)
    const [answersMap, setAnswersMap] = useState({})

    useEffect(() => {
        surveyService.getById()
            .then(setSurvey)
    }, [])

    function onChangeVal(id, val) {
        const answersToSave = { ...answersMap }
        answersToSave[id] = val
        setAnswersMap(answersToSave)
    }

}
```

```
{
  "c101": "Puki Ja",
  "c102": "Crap"
}
```

The Survey

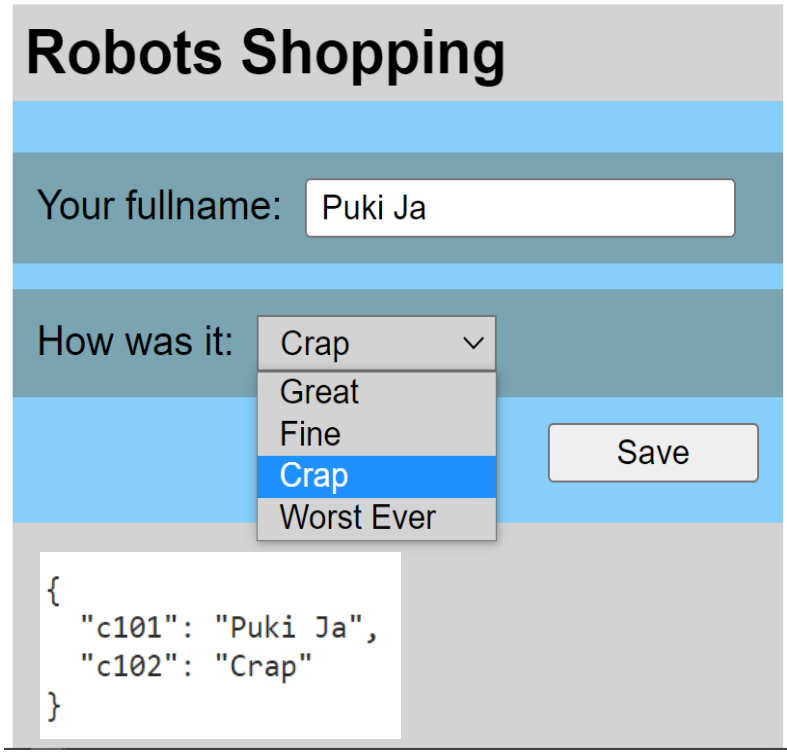
We render a component for each question:

```
<section className="survey-index">
    <h2>Survey - {survey.title}</h2>
    {
        survey.cmps.map(cmp => <div key={cmp.id} >
            <DynamicCmp
                type={cmp.type} info={cmp.info}
                val={answersMap[cmp.id] || ''}
                onChangeVal={(val) => {
                    onChangeVal(cmp.id, val)
                }}
            />
        </div>)
    }

    <pre>
        {JSON.stringify(answersMap, null, 2)}
    </pre>

</section >
```



We have a dynamic survey

```
const survey = {
    title: 'Robots Shopping',
    cmps: [
        {
            type: 'TextBox',
            id: 'c101',
            info: {
                label: 'Your fullname:'
            }
        },
        {
            type: 'SelectBox',
            id: 'c102',
            info: {
                label: 'How was it:',
                opts: ['Great', 'Fine',
                'Crap', 'Worst Ever']
            }
        }
    ]
}
```



Dynamic Components

Robots Shopping Today

Your fullname: Puki

Robot Type: FeedMeBob

Features: Responsive ☑ Accurate ☑ Resourceful ☐ Flexible ☐ Simple ☐

How was it: Fine

Quality: ○ 1 ○ 2 ◉ 3

Tune your photo:

Zoom:

Size:

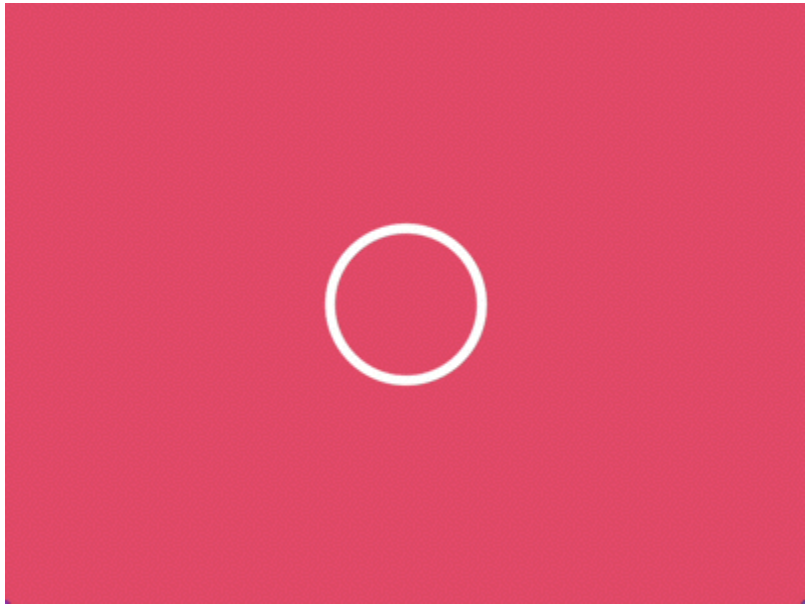
Angle:

vignette:


```
[
  "Puki",
  "FeedMeBob",
  [
    "Responsive",
    "Accurate"
  ],
  "Fine",
  3,
  {
    "size": 200,
    "zoom": 1,
    "angle": 0,
    "vignette": "20"
  }
]
```

We can add more cmp-types and build various dynamic forms

{coding academy



Dynamic Components

Done

coding academy

Having Children

```
<FancyBox>
   <h3>{someTitle}</h3>
   <button>Tell me More</button>
</FancyBox>
```

Having Children

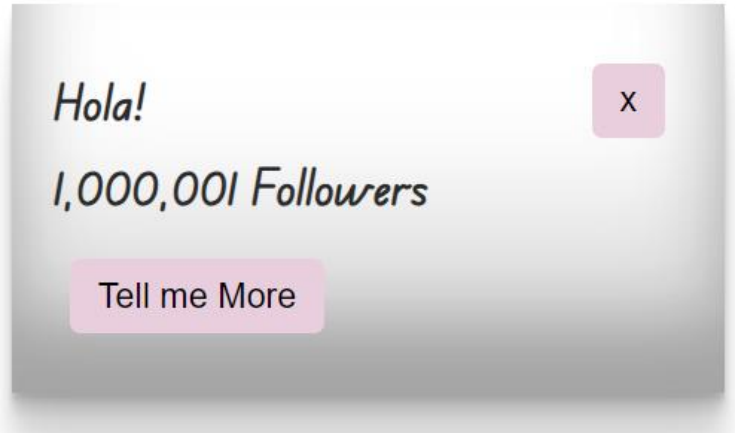
We can pass in some content into the <Child> component

```
<FancyBox>
   <h3>{someTitle}</h3>
   <button>Tell me More</button>
</FancyBox>

// This cmp has no children:
<SomeCmp />
```

coding academy

Having Children

This is especially common for container components such as <Sidebar> or <Dialog> that represent generic “boxes”




```
function FancyBox(props) {
    return <div className="fancy-box">
        <button style={{ float: 'right' }} onClick={props.onClose}>x</button>
        {props.children}
    </div>
}
```

```
<FancyBox onClose={() => console.log('ok, closing')}>
   <h3>{count.toLocaleString()} Followers</h3>
   <button onClick={onViewMore}>Tell me More</button>
</FancyBox>
```

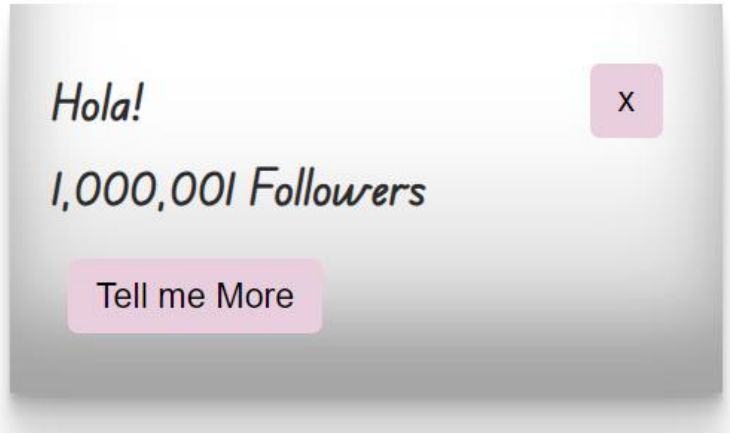
{coding academy

React Components API

```
<FancyBox title="Hola!" onClose={() => console.log('ok')}>
   <h3>{count.toLocaleString()} Followers</h3>
   <button onClick={onViewMore}>Tell me More</button>
</FancyBox>
```

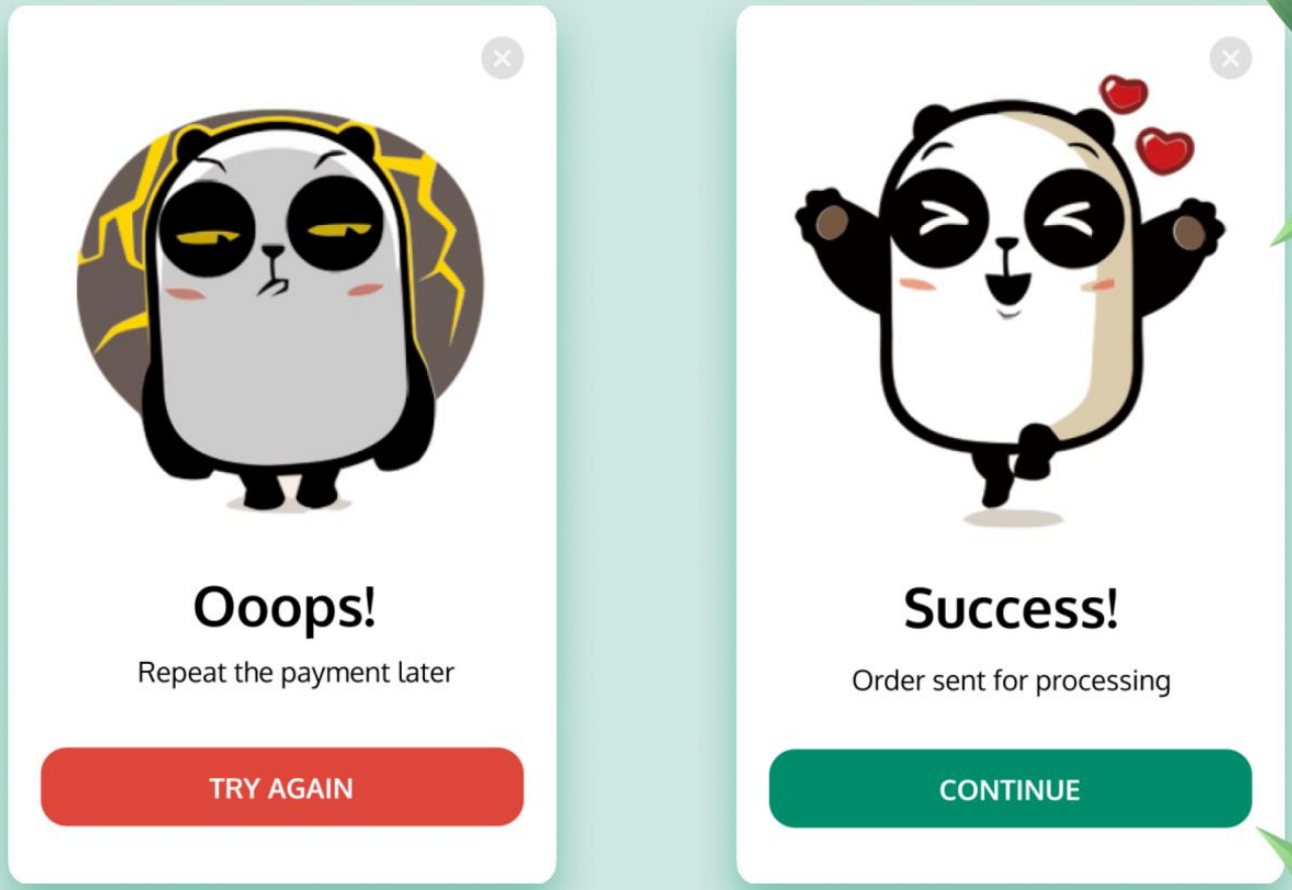
The API for a React component comes in three parts:

- **props** allow the parent component to pass data into the component
- **events** allow the child component to trigger a function on the parent
- **children** allow the parent to compose the child component with extra content




```
function FancyBox(props) {
    return <div className="fancy-box">
        <button style={{ float: 'right' }} onClick={props.onClose}>x</button>
        <h3>{props.title}</h3>
        {props.children}
    </div>
}
```

coding academy

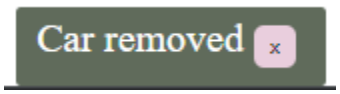


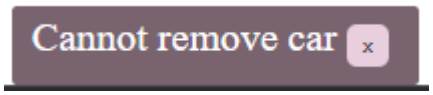
<UserMsg> component

coding academy

<UserMsg> component

- We need a way to display messages to the user
- Those can be success or error messages
- The message should have a close button
- The message will close itself after 3 seconds
- We need a way to activate the <UserMsg> from any component
- Using props-drilling everywhere is exhausting
- We need a **non-parent-child** communication
- We will use a technique called **event-bus**





coding academy

The event bus

- Let's meet the `eventBusService`
- It exposes two functions:

```
// Use this function to subscribe to an event
on(evName, listener){ ···
},
// Use this function to emit an event
emit(evName, data) { ···
}

// Example for using the service
eventBusService.on('some-event', (data)=>{
    console.log('Got some-event with data:', data)
})
eventBusService.emit('some-event', 100)
```

Unsubscribing

eventBusService.on() returns an unsubscribe() function we can call when the subscribing component is being unmounted

```
const unsubscribe = eventBusService.on('some-event', data=>{
    console.log('Mee too:', data)
})

// Just as example - unsubscribe after 2 secs
setTimeout(()=>{
    unsubscribe()
}, 2000)
```

Let's build a <UserMsg> component

```jsx
import { eventBusService } from "../services/event-bus.service.js"
const { useState, useEffect } = React

export function UserMsg() {

  const [msg, setMsg] = useState(null)

  useEffect(()=>{
    const unsubscribe = eventBusService.on('show-user-msg', (msg) => {
      setMsg(msg)
      setTimeout(closeMsg, 3000)
    })
    return unsubscribe
  }, [])

  function closeMsg(){
    setMsg(null)
  }

  if (!msg) return <span></span>
  return (
    <section className={`user-msg ${msg.type}`}>
      <button onClick={closeMsg}>x</button>
      {msg.txt}
    </section>
  )
}
```

coding academy

Showing user messages

We want an easy way to emit the `'show-user-msg'` msg

Lets add some useful functions to our `eventBusService`

```
export function showUserMsg(msg) {
    eventBusService.emit('show-user-msg', msg)
}
export function showSuccessMsg(txt) {
    showUserMsg({txt, type: 'success'})
}
export function showErrorMsg(txt) {
    showUserMsg({txt, type: 'error'})
}
```

Car removed x

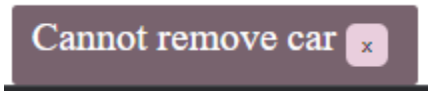
Cannot remove car x

{coding academy

Using the <UserMsg> component

In the <CarIndex>:

```
import { showSuccessMsg, showErrorMsg }
   from '../services/event-bus.service.js'


    function onRemove(carId) {
        carService.remove(carId)
            .then(() => {
                showSuccessMsg('Car removed')
                const updatedCads = cars.filter(car => car.id !== carId)
                setCars(updatedCads)
            })
            .catch(err => {
                console.log('OOPS', err)
                showErrorMsg('Cannot remove car')
            })
    }
```



We can also use it in the <CarEdit>

Another example

we build a shop, and we want to show the shopping-cart items count at the page header:

```
// in my <AppHeader> component, useEffect:
eventBus.on('added-to-cart',
    (productId) => {/* Update count label */ }
)
```


```
// in my <Shop> component when user buy something:
eventBus.emit('added-to-cart', {productId: 't101'})
```

The event bus - summary

- So sometimes, far-away components need to communicate with one-another
- We will use an **event bus service** to connect those components

Level Done

Event Bus

We are Ready

coding academy