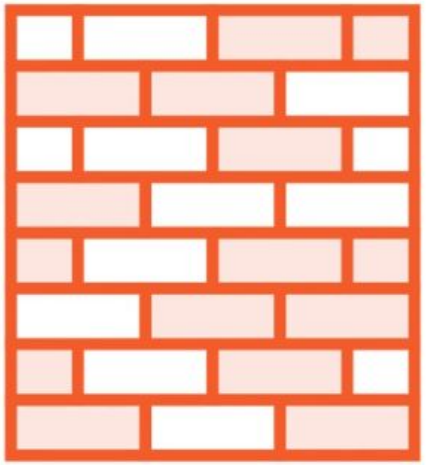
React

Fundamentals

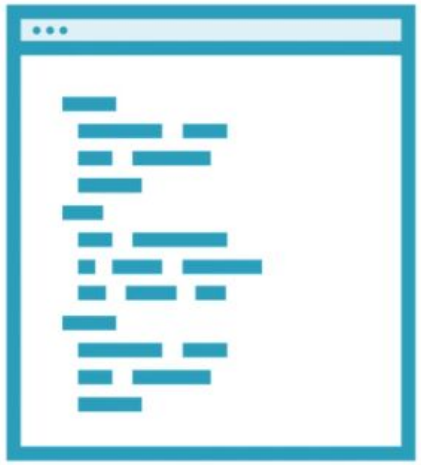
New Horizons®

The Power of React

Core React Features



Structure with components
Reusability
Have state



UIs declared in JavaScript
Rendered output changes
when state is updated

Efficiency with reconciliation
Only updates the parts
of the UI that changed

Components and JSX

- Components are JavaScript functions that return JSX
- JSX looks like HTML but it isn’t - it’s alternative way to write JavaScript
- It stands for JavaScript eXtension
- It is translated to JavaScript by a tool - usually Babel

Components and JSX

- This is a component:

  const Banner = () ⇒ <h1>This is a banner</h1>

- You can go to https://babeljs.io/

  You can input the code there and see how it is transformed

  NOTE: set React Runtime to Classic to see the JavaScript code in the “Try it out” section

Components and JSX

- This functionality comes from the basic core react
- To render the JavaScript on the screen we use another library in React called react-dom
- The separation is adequate if you want to create mobile app in which case react stays the same but you replace the react-dom with react-native

Components and JSX

- You can then nest components in other components:

```
const Greeting = () => (
    <div>
        <Banner />
        <h2 className=”highlight”>Greeting</h2>
    </div>
)
```

Components and JSX

- Note that the h2 is a built-in component using camelCase and it corresponds to DOM elements
- The custom components that we write are PascalCased
- Note the className attribute which corresponds to the HTML class attribute but since we are in JavaScript where *class* is reserved word, we need alternative

Components and JSX

- You can use a Class instead of a function:

```
class Greeting extends React.Component {
    render() {
        return (
            <div>
                <Banner />
                <h2 className=”highlight”>Greeting</h2>
            </div>
        )
    }
}
```

Components and JSX

- React team recommends functions although classes also work at this point in time
- New future React features will only be available in functions
- Functions are less verbose and easier to manage

Tools to Get Started

- Starting Point - template or boilerplate
- Transform JSX to JS
- Process JavaScript files
- Run a development server
- Automatically update browser when source file changes
- Create a production build

Tools to Get Started

- You can do it all by yourself but it takes time and effort to bundle everything and configure it
- You can use ready-to-go development environment and there are many options including:
  - create-react-app (CRA)
  - next.js

Tools to Get Started

- To use Next.js you need node.js
- You probably already have it but if you do not, download the latest stable version from nodejs.org
- Open a terminal in a folder of your choice and type:

  npx create-next-app globomantics

  Note: Use eslint in the process, “no” to everything else

Application Structure

- Open the newly created project in VS Code
- package.json - the npm package manager file with dependencies, devDependencies and scripts to run
- node_modules folder containing all downloaded packages
- styles folder with some CSS
- public folder which contains exposed that which will not be affected or changed during build process

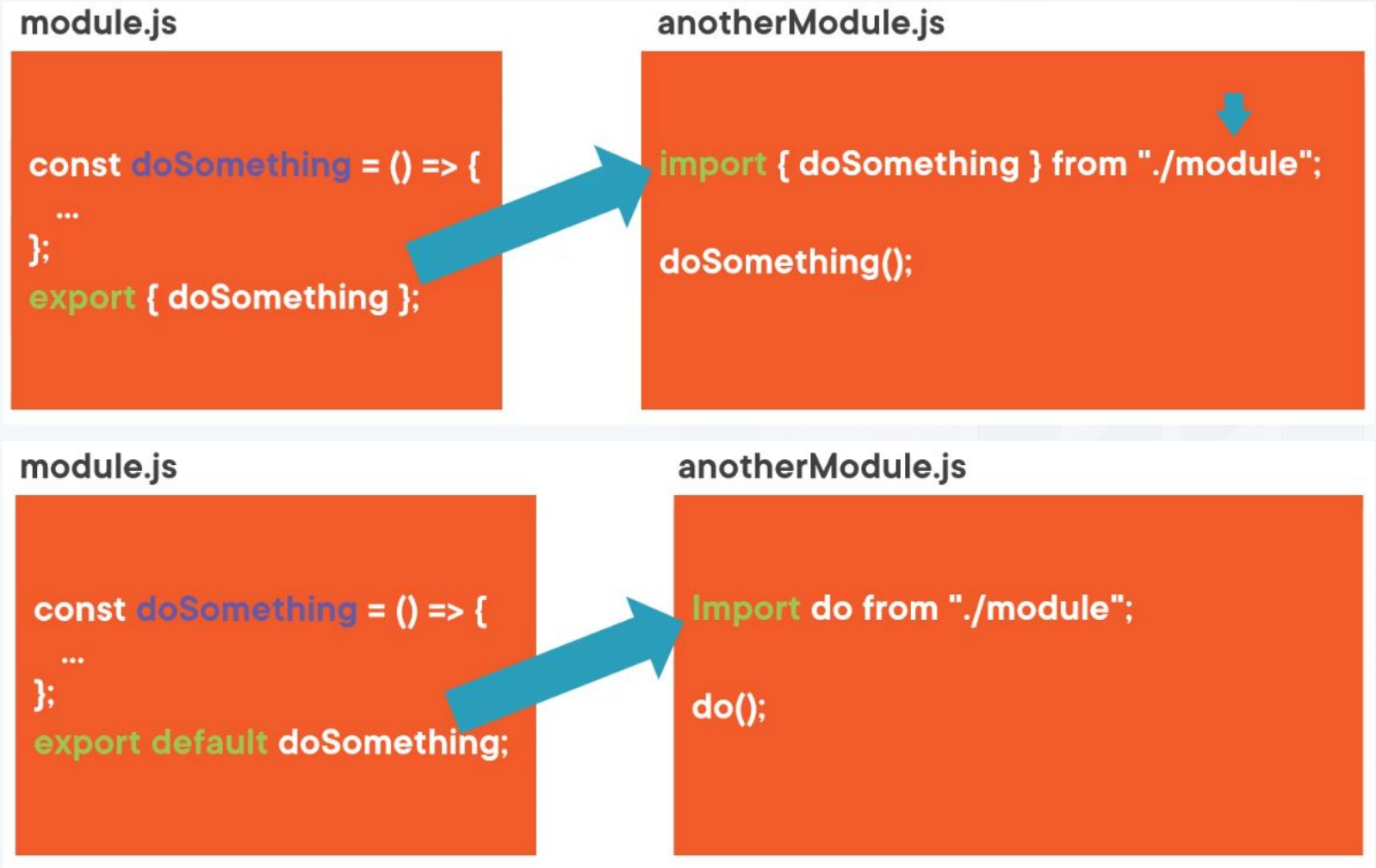
Application Structure

- In the terminal type: npm run dev
- Open a browser and go to: http://localhost:3000
- You can see the content on the screen in the file pages/index.js which contains a home component
- Note how changes to the file are instantly updated

JavaScript Modules Review

- Modules are JavaScript concepts not exclusive to React
- If you declare something (a function) in a JS file and you want to make it available to use in other such files (called modules), you have to export it first and then import it in the files that you want to use it in
- Non-exported code from the module is not available outside of it

JavaScript Modules Review



- Note the syntax:

JavaScript Modules Review

- You can export multiple items and then import them by separating them by commas in the curly braces
- Default export can only be single but you can use a different name for it once imported
- In React we export components that we want to reuse and we import them in the other parts of the app

JavaScript Modules Review

- Modules provide better code structure
- They bring reusability
- Encapsulation - non-exported code is private to module
- Needed for bundling - it helps figuring out the order when building everything into one big file at the end

Adding New Components

- It’s a good practice to have the name of the component be the same as the name of the containing .js file
- Delete everything in index.js and replace with:

```
const Index = () => {}


export default Index;
```

Adding New Components

- Create new components folder
- Add app.js and banner.js files with respective code
- Render the Banner inside of the app
- Render the App inside of the Index that we already have
- Note: make sure to handle all export / import statements
- Delete the _app.js file specific to next.js

ESLint

- Before we continue, let’s configure a Linter to help with problem detection and code styling
- ESLint is commonly used in many different applications, not only in React
- Next.js has a built-in React ruleset

ESLint

- Type in cmd / terminal: npm run lint
- You can see already that we have a problem and a suggestion to use the Image component instead of the HTML tag
- You can install the ESLint VS Code extension so you don’t have to ever run the command but instead see every error and warning in real time while coding

ESLint

- You can go to the VS Code Debug tab on the left
- Click on “Run and Debug” and select Web App (Chrome)
- Change the port to 3000 in the launch.json file
- Set a breakpoint somewhere and press the launch button
- You get controls to manage the debugging process
- You can also debug in the browser and install React Dev Tools extension

Styling Components

- You can load CSS files in Next.js using the _app.js file (yes, the one that we already deleted)
- Load a CSS file in the HTML root document
- Apply a CSS file on the component level
- Use the style attribute on a component

Styling Components

- Next.js does not have a default index.html file
- We can create one - in the pages folder add a file called _document.js
- Add <Main /> and <NextScript /> components
- Add links to Bootstrap and local CSS file
- You can copy example from the Next.js documentation

Styling Components

- With Bootstrap present, we can add some classes and styling to the Banner component
- In order to style the logo we can provide CSS in a separate file specific only to the component
- This avoids scoping issues and the file will be bundled in the application via default Webpack functionality

Styling Components

- Create a banner.module.css file in components

- Add inside:

```
.logo {
    height: 150px;
    cursor: pointer;
}
```

- Import the CSS in banner.js and use a JS expression for the className attribute to apply it: {styles.logo} or destructure the import and use {logo} instead

Styling Components

- We can also use the style attribute for specific control
- It can take an object, second braces annotate the object
  style={{ fontStyle: “italic”}} //NOTE: camelCase here
- We can also definite the object as const in the component and pass it to the style attribute
- Since we are not exporting that object, other components won’t be able to access it

Styling Components

- Keep in mind that using the style attribute is discouraged (as it is in HTML)
- Use CSS files instead
- Brings separation of the CSS from the components
- CSS files also enhance the performance

Props

- Props are input values which can be set and passed by other components during the render process
- They use attribute style syntax and any type of value can be passed, strings, objects, array, etc:
  `<Component myprop=”somevalue” />`
- They are then received as an input parameter:
  `const Component = (props) ⇒ ...`

Props

- Let’s add the banner text as a prop in the app.js component:

  <Banner headerText=”Providing houses all over the world” />

- We then accept an object as a parameter to the components function:

  const Banner = (props) ⇒ { //props is the conventional name

- In the JSX replace the header text with an expression:

  {props.headerText}

Props

- Important Strict Rule

Props are read-only!!!

They are often passed down to other components by reference and/or are changed by the parent component

Props

- We can now render and reuse the Banner a second time with a different text (try it although we just need one)
- We can also destructure the props and only extract the ones that we need. Replace the component declaration: `const Banner = ({headerText}) => {`
- Change the expression in the JSX to using it directly: `{headerText}`

Props

- In the case of the banner, the text could be a bit longer and it may look weird to pass it as an attribute argument
- Instead we would like to pass it as the content such as:

  <Banner>Providing houses all over the world</Banner>
- We can then access a special prop called children which contains all markup between the component's tags:

  const Banner = ({children}) ⇒ { …

  {children} //replace this in the JSX

Props

- The children can contain more complex content than rather just text
- It will respect HTML tags so you can add , for example, or very long text with styling

Fragments

- Let’s create a basic table with no tbody for now as a new component called houseList.js
- The return statement needs to always return a single JSX element thus we need encompassing
- If we don’t want the extra div, we can use <React.Fragment> component instead
- There’s also a shorthand which is often used:

  <> … </>

Mapping Data

- Let’s declare some hardcoded data in the component in the form of array up top:

  `const houses = [...]`

- Let’s render the data in the tbody using the JS map function:

```
{ houses.map((h) => (
    <tr key={h.id}>
        <td>{h.address}</td>
        <td>{h.country}</td>
        <td>{h.price}</td>
    </tr>
))}
```

- Add the <HouseList /> component in app.js to display it on the screen

The Key Prop

- We had a special prop called key in the tr renders
- React needs it to be unique so it can index the items on the screen
- When we add new item, React can track the unique keys and insert it in the list
- If we didn’t have the key, React would have to re-render the whole list which would impact performance
- If you don’t have unique index, you can use the map function with second parameter which provides one for you: houses.map(h, index) ⇒ (...)

Extracting Components

- Let’s augment our logic and move the table rows into a separate component called houseRow.js
- It will receive the data to render in a prop called house
- We can then render the new component with the map function, still needing unique key for React:

```
{houses.map(h => <HouseRow key={h.id} house={h} />)}
```

Extracting Components

- A cool trick to do in order not to use the house object as prop is to destructure it to more basic props:

  ```
  const HouseRow = ({address, country, price}) ⇒ {
  ```

- You can then pass them one by one or just by spreading the original object:

  ```
  <HouseRow key={h.id} {...h} />
  ```

- However, use this at your own discretion as adding or changing the house object later will impact your props as well

Global Helper Functions

- We often have shared functionality between many (or even all) components and we need an easy way to access it without passing it around (as prop)
- Those functions can be defined and exported as separate helpers and then imported and used wherever we need them

Global Helper Functions

- Create new helpers folder
- Create a currencyFormatter.js file
- Implement JS object with international formatting for USD (note: object, not a component)
- Import it in HouseRow and use the format function to render the price of the house on the screen

Hooks

- Functions allowing access to React’s internal features
- Encapsulates complexity
- Names start with “use”
- You can also define custom hooks

Hooks

- Important rules when working with hooks:

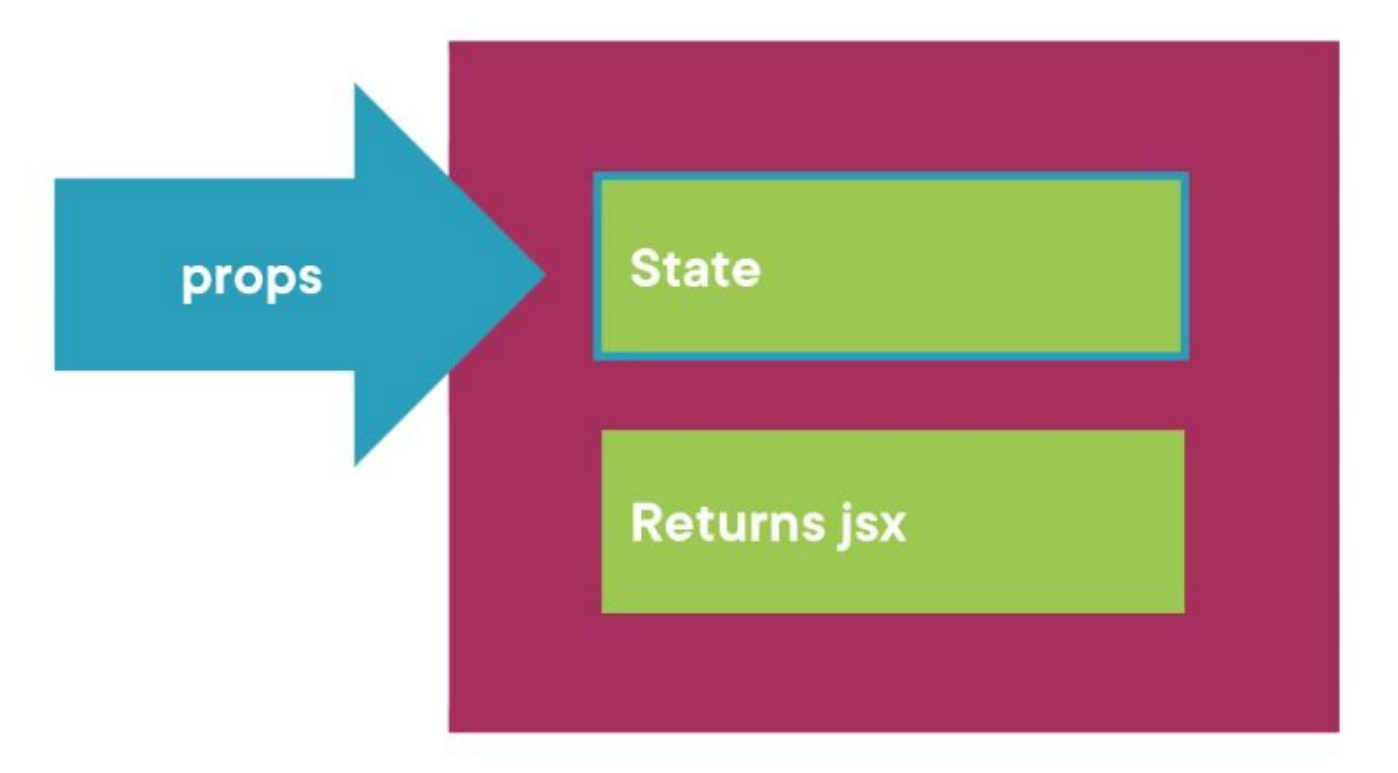
  - **Only call hooks at the top level** - do not include them in if statements or called them inside of a function which also ensures they are always called in the same order

  - **Only call hooks in function components** - calling them outside of the components will result in an error

- Examples follow to make better sense

State

- State is data kept internally by a component
- For comparison, props was data passed to the component



State

- In our example we render the houses on the screen but what if we want to add a new house to the array?
- React needs to know that the array actually changed so that it could be re-rendered and updated on the screen
- The state is automatically tracked but the hardcoded array that we defined up top is NOT

State

- Rename the houses array to houseArray - why in a sec
- In the component add the useState hook as follows:

  const [houses, setHouses] = useState(houseArray);
- houseArray is the initial value that we provide
- houses is the object that represents the current state
- setHouses is a function that we can use to change the state

State

- useState returns an array and we use array destructuring to assign the first value to a variable that holds the current state and the second value to a function used to change that state
- Calling the function lets React know that the state has changed so it will go and re-render the data on the screen

State

- Note that the state variable is called houses
- That means that we do not need to change the code in the JSX which is why we renamed the initial array to houseArray
- We are not using that array directly any more, it is just the initial value of the state

Setting State

- Create “Add” button below the table in houseList.js
- Create addHouse function inside the component
- Call the setHouses and add new value to the original
- Invoke that new function upon button clicks by reference

Props and State Interaction

- Note that clicking the button changes the state in HouseList
- Also note that this same data is actually prop value for the HouseRow component
- That means that props can change although we cannot do it from within the component itself but they do act like a state if they are changed from outside of the component

Reconciliation

- Open the inspector tool (F12) and monitor the table while clicking the “Add” button
- Note that only the new row is added but the whole table is not re-rendered on the screen
- React tracks changes in the data and only re-renders parts of the screen where changes occur instead of re-rendering everything all the time

Reconciliation

- Open the inspector tool (F12) and monitor the table while clicking the “Add” button
- Note that only the new row is added but the whole table is not re-rendered on the screen
- React tracks changes in the data and only re-renders parts of the screen where changes occur instead of re-rendering everything all the time

The Effect Hook

- React is pretty good at optimizing predictable component behavior
- If we give the same input (props), it will render the same JSX and output it on the screen the same way
- This is called using a “pure function”
- However, not everything is “pure” and we often have to implement unpredictable behaviors such as:
  API interactions, browser APIs (document, window), timing functions (setTimeout) and so on

The Effect Hook

- In those cases we utilize useEffect which takes a function as a parameter:
  useEffect(() => { //perform the effect });
- This function will be executed once React is done running the component's pure functions and the browser has been updated (such as fetching data from the API)
- Let's add the houses data from an API next

The Effect Hook

- Add the api folder and the houses.json files to your project (provided by the instructor)
- Replace the initial state of the houses to be empty arr []
- Add useEffect to the HouseList component
- Create async function that gets a response from the API, extracts the data from it and calls setHouses with result

The Effect Hook

- Well done!
- Game over! ...but wait, if we press the “Add” button...
- The reason is simple - useEffect is being called EVERY time the component finishes with re-renders and browser updates
- We then re-initialize the houses with the API value...
- So what we would like to happen is to run the useEffect only once

Effect Hook: Dependency Array

- We can pass a second parameter to the useEffect function which is a dependency array
- The function will be only called upon initial component render and changes to the values listed in that array
- To run an effect only upon initial component render, you can simply pass an empty array as dependency

Effect Hook: Dependency Array

- Change your code to include this:

  `useEffect(() => { ...}, []);`

- Also note that pressing the button in this case updates the state of the component but it does **not** persist the data in the API

- As an exercise you can try to add functionality to save the new data in the API and also extract the button into a new component

Effect Hook: Separation of Concerns

- You can have multiple useEffect calls within a component
- Don’t try to squeeze everything in one place and also use different dependency arrays for different functions

Effect Hook: Cleaning Up

- You can return a function from the useEffect which will be called once the component is removed from the UI (unmounted)
- Example: unsubscribe from an event on leaving the page

```
useEffect(() => {
    //subscribe code
    return () => {
        //unsubscribe code
    };
}, []);
```

The Memo Hook

- Assume we have complex calculation that we have to perform inside of the component:
  `const result = timeConsumingCalculation(houses);`
- This will be performed on each re-render and could occur quite frequently slowing down the application
- Instead, we can memoize the result and pull it from the cache every time we need it

The Memo Hook

- That happens using the useMemo hook:

```
const result = useMemo(() => {
    return timeConsumingCalculation(houses);
}, [houses]);
```

- As you can see, there's a dependency array here as well
- Value will be automatically recalculated upon changes to houses but not otherwise

The Ref Hook

- Persist values that survive re-renders without causing a re–render on change

- It is often used to gain access to JavaScript and DOM objects (more about this later):

```
const TextInputWithFocusButton = () ⇒ {
    const inputEle = useRef(null);
    const onButtonClick = () ⇒ inputEle.current.focus();
    return (
        <>
            <input ref={inputEl) type=”text” />
            <button onClick={onButtonClick}>Focus the input</button>
        </>
    ); }
```

Conditional Rendering

- We often have logic that changes the way elements are rendered on the screen
- We can implement such changes straight into JSX. Let's say we want to display prices differently in the HouseRow:

```
<td className={`${house.price >= 500000 ? "text-primary" : ""}`}>
```

Conditional Rendering

- We can also determine whether or not to render something by using the logic AND operator to determine if a value is truthy such as:

```
{house.price && (
    <td className={`${house.price >= 500000 ? "text-primary" : ""}`}>
        {currencyFormatter.format(house.price)}
    </td>
)}
```

- If house.price is falsy, the rest won't be evaluated and displayed

Conditional Rendering

- Build a house.js component which will display information for a single selected house
- Use conditional rendering to determine if the house has an image to display and show a default if not
- Your instructor will provide house images and a default image helper (with base64 encoded image)

Conditional Rendering Components

- We would like to display the newly created component once the user clicks on a row in the table
- That means replacing the HouseList component in app.js with the House component instead
- We will need some conditional logic here as well to determine which component to render

Conditional Rendering Components

- Let’s add a state in app.js to keep track of the selected house

```
const [selectedHouse, setSelectedHouse] = useState();
```

- Let’s also remove the HouseList from JSX and replace with the following logic instead:

```
{selectedHouse ?
    <House house={selectedHouse} /> : <HouseList />}
```

Conditional Rendering Components

- We face a problem here: how does the App know to call the setSelectedHouse when the click to trigger that will happen in the HouseRow component which is several levels down the hierarchy?
- We kind of know how to do that already - we can pass the setSelectedHouse function down the chain as a prop `<HouseList selectHouse={setSelectedHouse} />`
- You might have to use parentheses in JSX for new line items

Conditional Rendering Components

- In the HouseList component we need to annotate the function being received as a prop:

```
const HouseList = ({ selectHouse }) => ...
```

- We then pass it down to the HouseRow component as well:

```
<HouseRow key={h.id} house={h}
selectHouse={selectHouse} />
```

Conditional Rendering Components

- Finally in the HouseRow we can annotate the prop too:

  const HouseRow = ({ house, selectHouse }) ⇒ {

- And then invoke the function once we click on the row:

  <tr onClick={() ⇒ selectHouse(house)}>

- Clicking on a row should now change the displayed component in App

Custom Hooks

- We can also extract the logic from the HouseList component into a separate custom hook
- This will extract the logic into a separate function, helping with the separation of concerns and providing reusability
- When the state in the hook changes, the component that uses it will re-render
- Reusing the hook brings isolated state - each instance is separate

Custom Hooks

- Create new hooks folder with useHouses.js inside
- Move the house state logic from HouseList into the new file including the useEffect section
- Return the values that you need to use at the end:

```
return { houses, setHouses };
```

- Add them back up in the HouseList component as:

```
const { houses, setHouses } = useHouses();
```

Custom Hooks

- Let’s add some additional logic to our new custom hook by displaying a loading indicator while the list is generating
- Create loadingStatus.js in the helpers folder with just some plain text indicators for now
- Create additional component call loadingIndicator.js which will display a short <h3> title with text - feel free to make this component as fancy as you’d like

Custom Hooks

- In our useHouses.js hook we can now add state logic up top:

  const [loadingState, setLoadingState] = useState(loadingStatus.isLoading);

- On top of the fetchHouses function we can set it to load:

  setLoadingState(loadingStatus.isLoading);

- And at the end after the API call is done we can set it back:

  setLoadingState(loadingStatus.loaded);

Custom Hooks

- You can actually add try / catch blocks around the API and set the loading state to error if you catch an exception
- Make sure to return the loading state from the hooks function at the end so you can access it from outside:

```
return { houses, setHouses, loadingState };
```

Custom Hooks

- Back in HouseList we can check if the status is loading and

  if it is, we can return early from that function, displaying the status and ignoring the rest of the code (this is by the way also a conditional rendering):

```
const { houses, setHouses, loadingState } = useHouses();

if (loadingState !== loadingStatus.loaded)
    return <LoadingIndicator loadingState={loadingState} />;
```

useCallback

- Here’s an advanced task: let’s create a functionality that will allow us to pull data from the API but using arbitrary url as right now we are always pulling from the “api/houses”
- We can create a separate hook with the request logic for the API and also move the loading functionality in there
- All of this can then be reused in other parts of the application with different URLs for different pages

useCallback

- Create in hooks a file called useGetRequest.js
- Move all the async fetch logic inside but use the url as a prop
- Make sure to keep the loading logic as well and export it
- Without using useCallback yet, use the new hook to fetch the data in the useHouses hook
- Check what happens on the screen and implement useCallback

useCallback - What Happened?

- We need the [get] dependency array so we fire the useEffect only once on render and then only if the function changes (which means different url)
- When the function returns we call setHouses which changes the state and causes a re-render to useHouses
- At that point useHouses is reinitialized, it calls useGetRequest again which creates the get function anew, which then calls useEffect
- You can see the endless loop here

useCallback - What Happened?

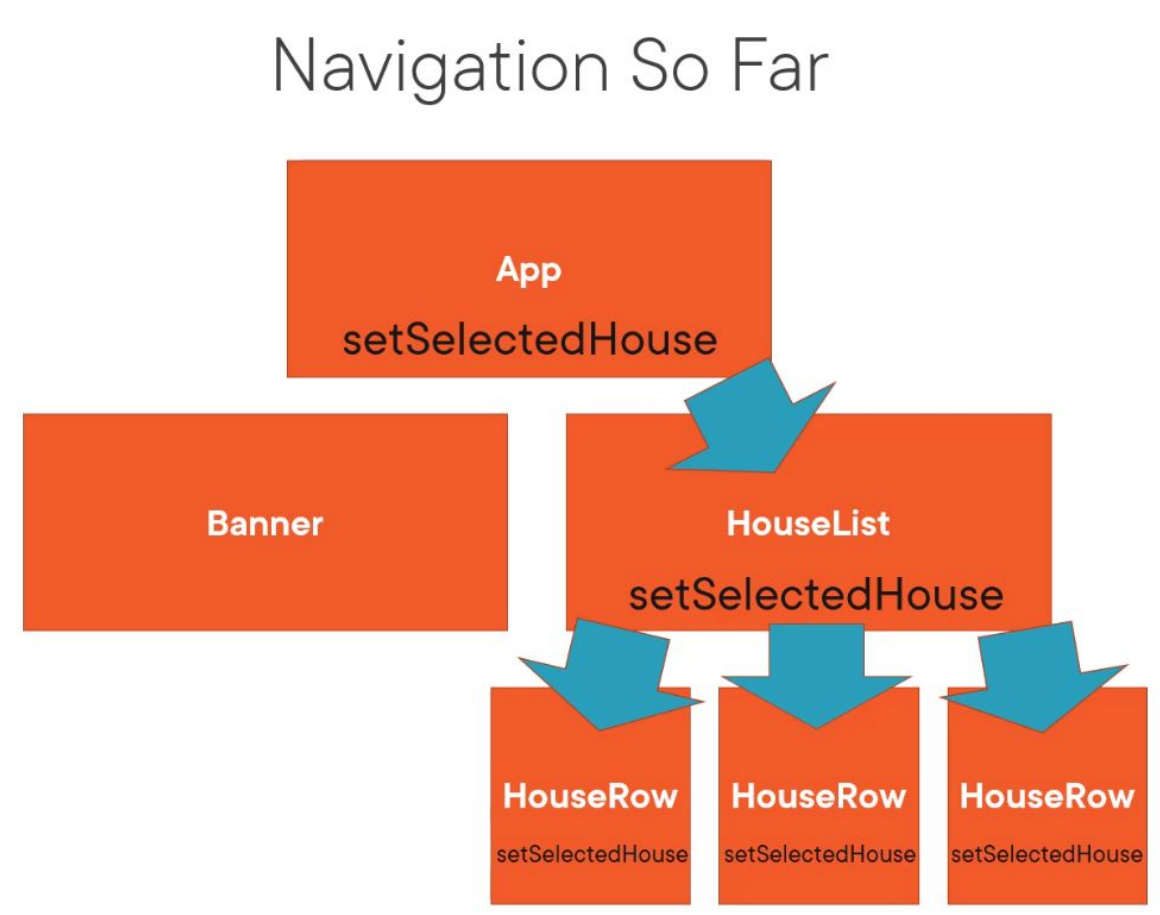
- In short, each re-render creates get again which changes the reference to that function causing useEffect to fire again
- This is why we want to useCallback to cache the function to be the same an reuse it on re-renders if needed **WITHOUT** creating a new one every time
- The [url] dependency is needed in the useCallback as it is external reference to the function (in our case it doesn't change)

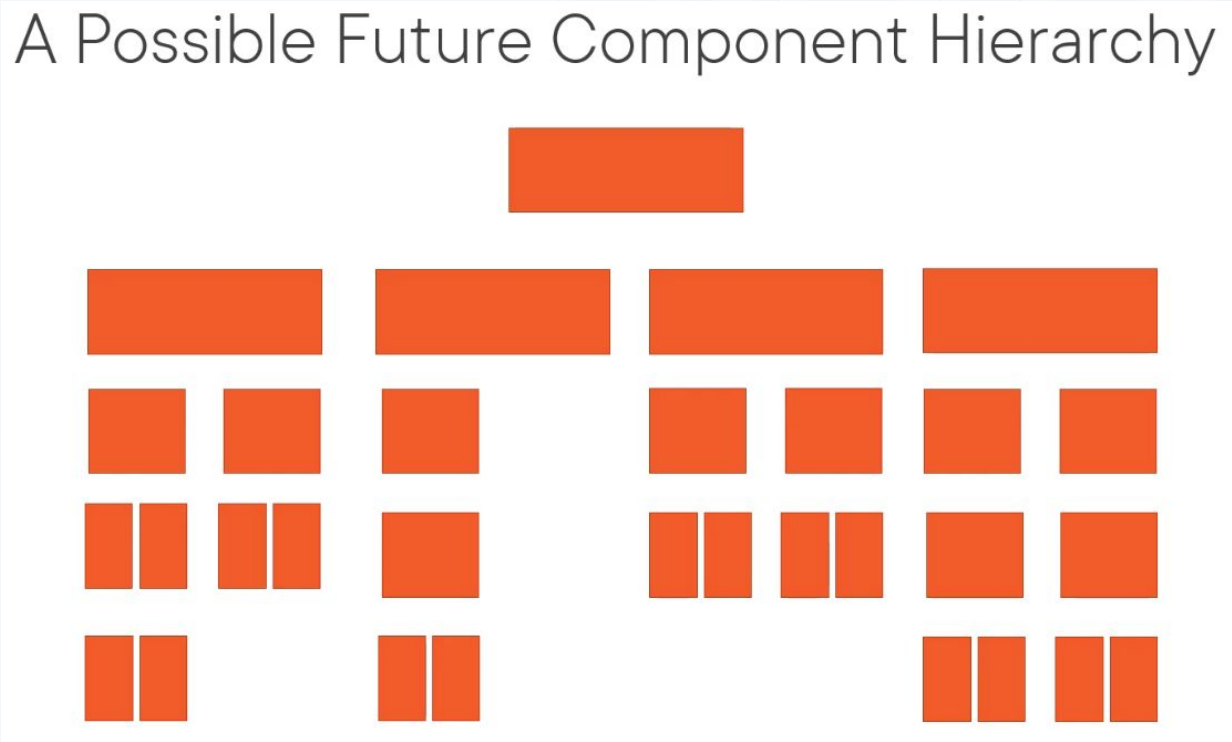
useCallback

- We shall see useCallback again but you can see the benefits
- You can argue that this final state of the code does not need the useHouses hook any more as we can put the useEffect logic back in the houseList component - feel free to shuffle things around as an exercise

Context

- We have simple navigation so far but that can easily change





Context

- Passing down something as prop is great but in a huge project that could be quite cumbersome and hard to maintain
- Instead we'd like to have a place where components could read data from without having the need to pass it down as a prop
- This is where the context comes into play

Context

In component providing context or separate module

```
const context = React.createContext("default value");
```

In the component providing context

```
<context.Provider value="some value">
   //children
</context.Provider>
```

In child components

```
const value = useContext(context);

<context.Consumer>
  { value => /* render something based on value */ }
</context.Consumer>
```

Context

- Note that child components that use the context do not need to be direct children to the context provider component

- Context is often declared in top level components and exported from them so children can then import and use it

- IMPORTANT NOTE: When the value of the context changes, all consumer components will be re-rendered

Context

- Create navValues.js in helpers with possible nav values
- Remove all the setSelectedHouses functionality
- Create context in App on top with default values

  (outside of the component and export it at the end)
- Wrap the Banner component as context provider:

  <navigationContext.Provider> ...

Context

- Next we need to figure out how to change the value provided to the context (if not specified, it’s just the default)
- We can create and keep a state of that value in App:

```
const [nav, setNav] = useState(navValues.home);
...
<navigationContext.Provider value={nav}>
```

Context

- We now need to expose the setNav function to other components so they can use it and change the navigation
- We can create a wrapper to this function with useCallback as it will be provided to all components in our application and we would like to avoid unnecessary re-renders:

```
const navigate = useCallback(
    (navTo) => setNav(navTo)
, []); //empty dependency array
```

Context

- But still how do we pass this function down?
- If we pass it directly in the context (i.e. {nav, navigate}) each component calling that function with different value will cause it to change the context thus re-render every other component that uses it
- We can instead include that function in the state as part of an object that we will pass down
- The reference to that object will always be the same in that case

Context

- Let’s change the code so far to use object as state:

```
const navigate = useCallback(
    (navTo, param) => setNav({ current: navTo, param, navigate })
, []);


const [nav, setNav] = useState({ current: navValues.home,
navigate });
```

Context

- Now the function to change the navigation is part of the context, it's cached in there and every component could access it without re-rendering everything else
- We need logic that will react on nav state changes now and render the corresponding components
- We could write a simple switch statement but with bigger projects that will make the JSX quite long and ugly
- Instead we can extract that logic into a separate component

Context

- Create new ComponentPicker.js component
- It will take the nav as prop and render respective component
- Add code to provide error page / message if value is wrong
- Insert this new component in App below the Banner:

```
<ComponentPicker currentNavLocation={nav.current} />
```

Context

- We have a problem though, the House component needs a house prop value
- Let’s go to the HouseRow component and useContext there to access the navigate function and call it once the user clicks on a row
- When the user clicks on a row we know the selected house so can we just pass that as second parameter to the navigate:
  <tr onClick={() ⇒ navigate(navValues.house, house)}>

Context

- We can go back to App now and add a second parameter to the navigate function like this:
  (navTo, param) ⇒ setNav({ current: navTo, param, navigate }),

- We don’t need to change the initial state value as there is no param at that point - it will only be added to that object at later calls of the setNav function

Context

- We can now go to House, useContext there, destructure the param out and rename it to house for convenience:

```
const House = () => {
    const { param: house } = useContext(navigationContext);
```

Context

- Adding navigation now become very easy because anyone can access the navigationContext
- Let’s add functionality to the Banner to return to the HouseList upon clicking on the logo:

```
const { navigate } = useContext(navigationContext);
...
onClick={() ⇒ navigate(navValues.home)}
```

Context

- All of this is great... for this small application
- However, even clicking on a row does not change the URL in the browser
- Clicking the back button does not work and linking to a house page directly is not possible
- Use Next.js router or React Router instead

When to Use Context?

- When the same state has to be passed to many components
- Keep in mind that this leads to re-rendering
- Components become dependent on state so their reusability become more difficult
- Hidden state - if you start writing new component you have no idea that the context is there

User Input and Forms

- In order to access the values of HTML input elements we need to create a references to them in React to extract values

- We'd need to convert their internal values to React state so we can track and update it

- We then implement the created state into components

User Input and Forms

- We can bind the value to a state variable but we also need to track changes by intercepting the onChange function

```
const [ firstname, setFirstname ] = useState("Alice");

return (
  <input type="text" value={firstname}
     onChange={(e) => setFirstname(e.target.value)} />
);
```

User Input and Forms

- If we handle forms, we have to take over the submit action

```
const [ person, setPerson ] = useState({ firstname: "Alice", lastname: "Doe"});
const submit = (e) => {
  e.preventDefault();
  //submit person to API
};
return (
  <form onSubmit={submit}>
    <input type="text" value={person.firstname}
        onChange={(e) => setPerson({ …person, firstname: e.target.value})} />
   <input type="text" value={person.lastname}
        onChange={(e) => setPerson({ …person, lastname: e.target.value})} />
  </form>
);
```

User Input and Forms

- We can also use a onChange handler function and object state

```
const [ person, setPerson ] = useState({ firstname: "Alice", lastname: "Doe"});
const change = ((e) => setPerson({ …person, [e.target.name]: e.target.value });

return (
  <form onSubmit={submit}>
    <input type="text" name="firstname" value={person.firstname}
       onChange={change}/>
   <input type="text" name="lastname" value={person.lastname}
        onChange={change} />
  </form>
);)
```

- The spread operator is used with the [...] syntax to just replace the value in question. In JS that syntax is called computed property syntax

User Input and Forms

- Add the useBids.js hook and bids.json files to your project
- Create new component bids.js that uses the bids hook to add new bid values via inputs: bidder and amount
- Add the new component in the House component and connect it to work with the currently displayed house

User Input and Forms

- A form usually needs much more: validation, error messages, etc.
- You can also write your own code but you can also use external libraries
- Formik (formik.org) is an example

Questions?

Thank you!