Web Development and API Design

Lesson 04: SPA State Handling

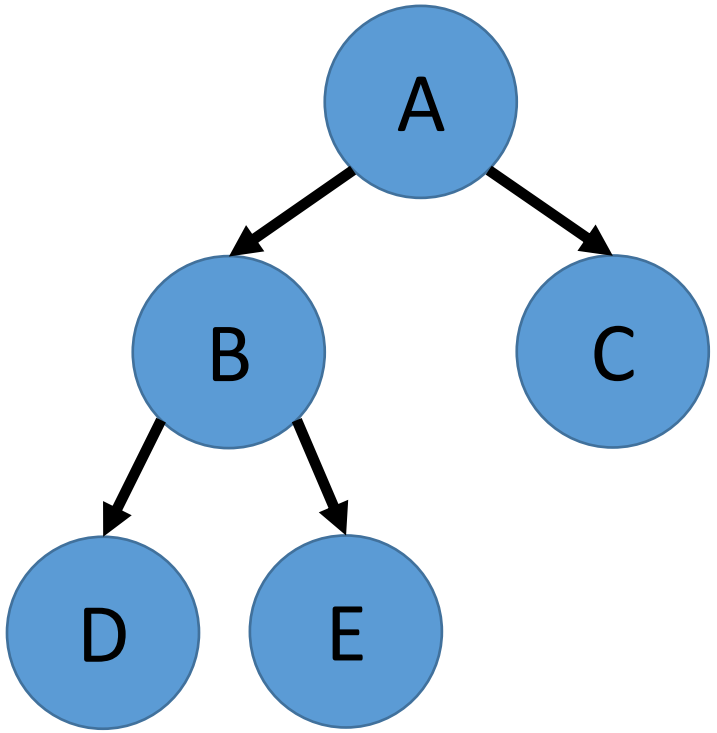
Prof. Andrea Arcuri

Goals

- Understand how state is handled in a multi-component React application
- Learn how to write unit tests for *React* components

Parent-Children Tree Hierarchy

- Each node can see its children
- Node cannot directly see the parent
  - a component could be re-used in many different places
- A component might need to modify the state of parent/ancestors, or of other siblings
  - eg, B might need to interact with C, but B only sees its state, and the state of its children D and E
- *How to do that?*



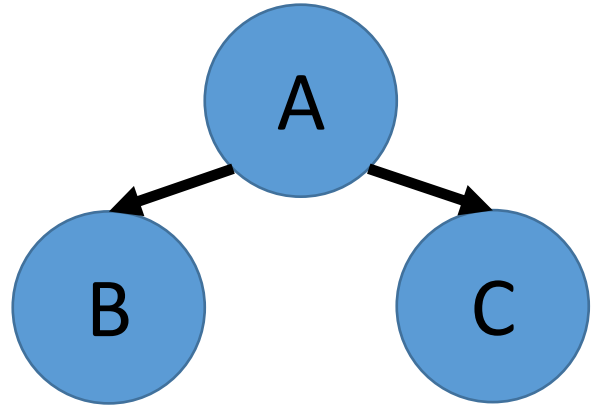
Properties

- When a parent creates a child X, it can pass to it some properties (*props*), similarly to HTML attributes
  - however, recall that JSX is transformed into JS, and such "*attributes*" are actually replaced by the appropriate calls to the *React* library
- Ex., <X *foo*=5 />
  - here the component X takes as input the prop *foo* with value 5
- *Props* should be considered as **immutable** state
  - you can read them, but not supposed to modify them in the child
  - can read/use *props* inside *render()*
  - wrongly modifying a *prop* in the child does NOT trigger a new *render()* call

Callbacks

- The content of a *prop* could be a function
- Such function has the scope of the parent node
  - it can then call *setState()* in the parent
  - can access the other siblings
- If a child cannot directly interact with parent, parent can pass a callback as a *prop* to enable it

State Lift Up



- If two components depend on the same state, such state needs to be lifted up in a parent component
- Parent will provide callbacks to manipulate such state
- Recall: when state of a component is modified, all its children are re-rendered
- Ex.: when **<B>** calls the function *x(),* this executes the function *A.callback()*, which, if it changes the state of **<A>**, it will re-render **<C>** as well besides **<A>** and **<B>**

```
<A>
 <B x=callback />
 <C x=callback />
</A>
```

Complex State

- This approach to handle shared state works well for many kinds of applications
  - However, if you have a complex application, with a deep component tree and high relations between components, it might not scale well
- **React Context**: "*provides a way to pass data through the component tree without having to pass props down manually at every level*"
- **Redux**: a popular library to handle such scalability issue
  - one single source-of-truth for the state, and not spread around for each component
  - components need to be hooked into the *Redux* data-store, to be able to re-render automatically at each relevant state change
  - *Redux* can be complex to use, and it is an overkill for what we need in this course. So will not use it, albeit you need to have an idea of what it is

Testing React Components

- A React component has a *render()* method that generates HTML
- The HTML tags can have event handlers that trigger state change
  - e.g., clicking on a button or hoovering with the mouse over a tag
- When state changes due to a triggered event, the HTML might change
- *How to unit test such behaviors?*

Simulate a Browser

- For **unit** testing (and not **system** testing), using actual browser is an overkill
- We need to run a fake, headless browser directly in JS, inside *NodeJS* runtime
- This can be done with libraries like *Enzyme* and *JSDOM*
- We will use a base HTML page in which we will dynamically *mount* the component we want to test
- We will simulate clicks and other events, and see what HTML strings are generated in reaction to those events

CSS Selectors

- When we need to interact with a web page with *Enzyme*, need to specify the HTML elements
  - eg, which button to click, and which text area to fill
- 2 main languages to select elements in HTML: *CSS Selectors* and *XPath*
- *CSS Selectors* are the same as when writing *.css* files
  - *".foo"*: select all HTML elements with *class* attribute "foo"
  - *"#foo"*: select element with *id* attribute "foo"
  - etc.