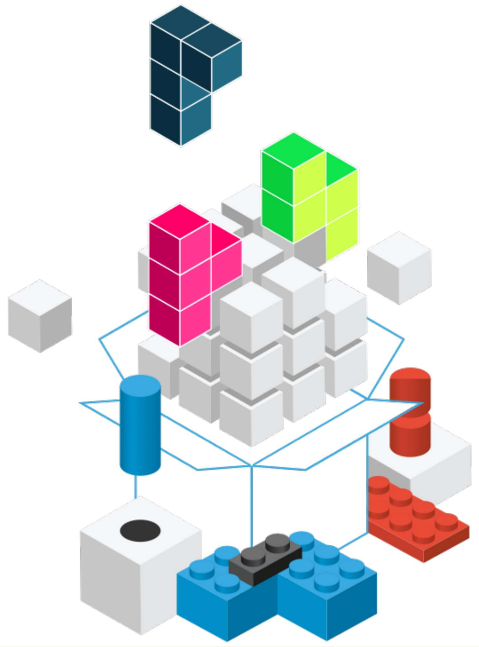
<WA1/>
<AW1/>
2023

Elements, JSX, Components

The Foundations of React

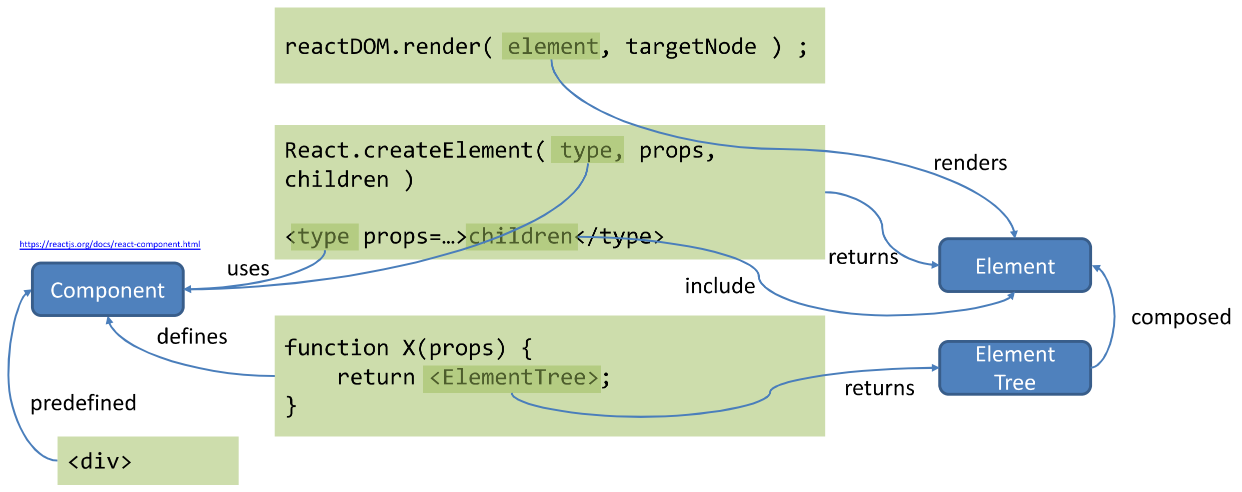
Fulvio Corno
Luigi De Russis
Enrico Masala



Outline

- React Elements
  - Creating
  - JSX language
- React Components
  - Defining

Conceptual Overview


reactDOM.render(element, targetNode) ;
React.createElement(type, props, children)
<type props=…>children</type>
function X(props) {
    return <ElementTree>;
}
<div>
https://reactjs.org/docs/react-component.html
Component
Element
Element Tree
renders
returns
include
uses
defines
predefined
composed
returns

https://react.dev/learn/your-first-component

Full Stack React, Chapter “JSX and the Virtual DOM”

Building block for describing web page content

REACT ELEMENTS

React Element

- An element is a plain object describing a component instance or DOM node and its desired properties
- A ReactElement is a representation of a DOM element in the Virtual DOM.
- It contains only information about
  - the component type (for example, a Button)
  - its properties (for example, its color)
  - any child elements inside it.
- Not an *instance* of a part of a page, but a *description* about how to construct it.

React.createElement (1/3)

- `React.createElement( type, props, children )`

- Type
  - String: a DOM node identified by the tag name (e.g., `'div'`)
  - React component class/function: a user-defined component

React.createElement (2/3)

- `React.createElement( type, props, children )`

- Props: a simple object {}, containing:
  - DOM attributes for DOM nodes (type, src, href, alt, …)
  - Arbitrary values for React components (even array- or object-valued)
    - Available as `props` in the Component body
  - Represented as object properties (not strings like HTML attributes)
    - Exceptions (reserved words): `class` → `className`, `for` → `htmlFor`

React.createElement (3/3)

- `React.createElement( type, props, children )`

- Children:
  - a ReactNode object, that may be:
    - A string or number: text content of the nodes
    - A ReactElement (that may contain a tree of Elements)
    - An array of ReactNodes
  - nested Elements to be rendered as children of the element

Conventions

- DOM Elements are always lowercase
  - `div p li img …`
- React Components are always uppercase
  - `WarningButton LoginForm TaskList …`
- The two types of elements can be mixed, nested, combined in any way
  - React uses *composition* and not *inheritance*
- Element trees describe **portions of the Virtual DOM**

https://react.dev/learn/writing-markup-with-jsx

Full Stack React, Chapter "JSX and the Virtual DOM"

React Handbook, Chapter "JSX"

A humane way of describing trees of ReactElements

JSX

JSX – JavaScript Syntax Extension

- Alternative syntax for React.createElement
- XML fragments inside the JS code
  - Syntax details: all tags must be </closed> or <selfclosing/>
- Transpiled by Babel into plain JS

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

Element/Component name
Props
Children / Text content

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
) ;
```

Components are expanded during *rendering*

```
<Button color='blue'>
OK!
</Button>
```

render

```
<button class='button button-blue'>
  <b>
    OK!
  </b>
</button>
```

🌲 Components encapsulate element trees (generated given their properties).

📷 React asks the Button component to render itself. It will generate a tree of elements, to replace this one.

↻ Repeat until only DOM nodes are present.

JSX Syntax

- May use `<tag>…</tag>` or `<tag/>` anywhere a JS expression is syntactically valid
  - Not only in Components
  - May also store in Arrays/Objects
  - After all, they are just ReactElements generated by React.createElement!
- May enclose in (…) for clarity

```
const element = <div className="main">Hello world</div>;
```

Note: use <tag/> if the component doesn't have any children

```
const element2 =  (<Message text="Hello world" />);
```

JSX Tag Name

- <Foo> is just React.createElement(Foo,…)
  - Foo must be in scope (imported or declared)

```
import CustomButton from './CustomButton';

function WarningButton() {
  return <CustomButton color="red" />;
}
```

JSX Attribute Expressions

- Tag attributes are converted to props of the ReactElement
- String attributes become string-valued props
  - `color="blue" -> {color: 'blue'}`
- Other objects may be specified as a JS expression, enclosed in { }
  - `shadowSize={2} -> {shadowSize: 2}`
  - `log={true}`
  - `color={warningLevel === 'debug' ? 'gray' : 'red'}`
- Any JS expression is accepted

JSX Children

- The *content* between the tags <tag>*content*</tag> is passed as a special property `props.children`
- Such content may be:
  - A string literal
  - More JSX elements (nested components)
  - Any {JS expression}
  - A {JS expression} returning an array of JSX elements (they are inserted as siblings)
  - A JS function (may be used as a callback by the Component)
  - Anything that the Component may understand (and render properly)

```
<MyComponent>Hello
world!</MyComponent>
```

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

JSX Child Expressions

- JS expressions in { } may be used to specify element children
- One child (or an array of children) are generated by an expression

```
const Menu = (<ul>{loggedInUser ? <UserMenu /> : <LoginLink />}</ul>)
```

  - <JSX> inside {JS} inside <JSX> inside JS. Totally Legit. 🤯
- `undefined`, `null` or Booleans (`true`, `false`) are not rendered
  - Useful for conditionally including children

```
return (<ul>
  <li>Menu</li>
  {userLevel === 'admin' && renderAdminMenu()}
</ul>)
```

Render Children Components

- In the component, you may render `{props.children}` to include the nested elements

```
return (
  <Container>
    <Article headline="An interesting
Article">
      Content Here
    </Article>
  </Container>
)
```

```
function Container (props) {
    return (<div className="container">
      {props.children}
    </div>);
}
```

Boolean HTML Attributes in JSX

- In HTML some attributes do not have a value. Their simple presence “activates” a behavior
  - HTML: `<option value='WA' selected>Washington</option>`
  - HTML: `<input name='Name' disabled />`
- In JSX, a Boolean value may be given
  - True, for the presence of the attribute (optional in recent React versions)
  - False (or nothing) for the absence of the attribute
  - JSX: `<option value='WA' selected={true}>Washington</option>`
  - JSX : `<input name='Name' disabled={true} />`

Comments in JSX

- There are **no** comments in JSX
- The HTML/XML comments syntax `<!-- … -->` does **not** work
- If you want to insert comments, you must do that in an embedded JS expression (using JS syntax inside `{}`)

  `{/* … */}`

- Yes, it’s ugly

DOM Attribute Names

- When passing props to a DOM native node, some differences exist
- Attribute names are camelCase
  - HTML onchange → JSX onChange
- The style attribute accepts an object and not a string
  - `<div style={{color: 'white'}}>Hello World!</div>`
  - Object keys are CSS Properties, and are camelCase (e.g., `margin-top` → `marginTop`)
  - Object values are CSS values, represented as strings

JSX Spread Syntax

- Shortcut syntax for passing all properties of an object as props to a React Component

```
const welcome = {msg: "Hello", recipient:
"World"} ;

<Component
  msg={welcome.msg}
  recipient={welcome.recipient} />
```

```
const welcome = {msg: "Hello", recipient:
"World"} ;

<Component {...welcome} />

// properties of the welcome object
// are "spread" as individual props
// with the same name
```

JSX Spread Example (Property Passthrough)

```
const Button = props => {
  const { kind, ...other } = props;
  const className = kind === "A" ? "ABtn" : "BBtn";
  return <button className={className} {...other} />;
};

const App = () => {
  return (
    <div>
      <Button kind="primary"
        onClick={() => console.log("clicked!")}>
        Hello World!
      </Button>
    </div>
  );
};
```

- The ‘kind’ property is “consumed” by <Button>
- All other properties (…other) are passed to the child <button>
- In this way, <App> can specify the kind to Button and all other properties to “pass through” down the hierarchy

JSX Syntax Reminders

- The HTML `class` attribute is called `className`
  - Useful to add CSS classes for layout (e.g. `className='d-block vh-100'`)
- The HTML `for` attribute is called `htmlFor`
- HTML entities (&lt; &amp; &copy; &star; etc…) may not be supported directly in older JSX
  - Use the corresponding Unicode character (< & © ☆) inside a string in JS `{'☆'}`
  - Alternatively, use a Unicode Escape sequence: `{'\u2606'}`
    - See: https://www.toptal.com/designers/htmlarrows/

https://react.dev/learn/passing-props-to-a-component

https://react.dev/learn/thinking-in-react

Full Stack React, Chapter “Advanced Component Configuration with props, state, and children”

Putting together the building blocks

REACT COMPONENTS: INTRO

Declaring Components

Components (as functions)

```
const Button = ( props ) => (
        <Element>...</Element>
);

function Button(props) {
        return <Element>...</Element> ;
}
```

- **Components**:
- Take **props** as their input
- Return the **elements** as their output

Components (as functions)

- Defined as function statement, function expression or arrow expression
- Receive (`props`) argument
- Must `return` a React Element tree
- The returned elements are function of the props
- Must be a **pure function** (no side-effects) and **idempotent**
- State and lifecycle may be managed with the *Hooks* mechanism

Tips for Creating Components

- It is normal to create many different “small” components
- Each component is constructed by *composing* other components
  - Components may be repeated (with different props)
  - It’s up to the parent to determine the children’s props
- If a component becomes too complex, try to *extract small re-usable parts* as independent components

Lists and **Keys** (1/2)

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map(
    (number) => <li>{number}</li> );
  return (<ul>{listItems}</ul>);
}

Function App(props) {
  const numbers = [1, 2, 3, 4, 5];
  return <NumberList
numbers={numbers}/>;
}
```

- `NumberList` generates a `<ul>` containing `<li>` for each of the numbers in `props.numbers`
- Whenever you construct a list of elements, you must pass a unique **key** attribute to identify each item
- Unique keys help React identify which items have changed, are added, or are removed.

https://react.dev/learn/rendering-lists

Lists and **Keys** (2/2)

- Always assign to each item in the list a special ‘`key`’ attribute, with unique values
  - `<li key={number}>{number}</li>`
- Most likely, we may reuse unique IDs from the data itself
  - `<li key={todo.id}>{todo.text}</li>`
- Keys **must** be specified when building the array of components
  - Usually in the `.map()` call, in the ‘container’ component
- Uniqueness is only required within *the same list*
  - *Not globally* on the page
- Keys are *not* available as `props` in the component

React Fragments

- A component should always return a tree of elements, with a single root.
- To return a list of elements, you must include them in some "container" (such a <div>)
  - This generates an "extra" DOM node, and in some contexts it might be invalid
- The special node `<React.Fragment>` may be used to wrap a list of element into a single root.
  - React.Fragment will not generate any node at the DOM level
- A shortcut syntax for fragments is `<>` … `</>`

License