React Basics

Pau Fernández Jose L. Muñoz-Tapia

Universitat Politècnica de Catalunya (UPC)

React Basics

Outline

React's Main Concept

- Web UIs (User Interfaces) are defined with:
  1. **HTML & CSS**: markup.
  2. **Javascript**: data (model), behavior.
- Many approaches for building Web UIs such as Angular, Vue, Handlebars, etc. are **template engines**.
  - Template engines use their own language or DSL (Domain Specific Languages).
  - This lets you embed code inside markup language to create the mixing of markup and JS to build the UI.
- React does this reversed:
  - It uses **JSX**, which lets you write your markup language inside code.
  - Definitions of markup inside JSX look like regular HTML.

Template Engines

- Template engines end reinventing things like `if` and `for` and you have to learn a new language, e.g. an angular +2 template:

```
<!-- template file -->
<div *ngIf="!isLoggedIn">
  Please login
</div>
<ul>
  <li *ngFor="let item of todoList; let i = index">
    {{i}} {{item}}
  </li>
</ul>
```

```
// javascript file
const isLoggedIn = false;
const todoList = ["read a book", "walk with the dog"];
```

- The template language defines `for` and `if`, which in the angular template are `*ngIf` and `*ngFor`.
- The markup template can reference variables in JS:
  `isLoggedIn` and `todoList` are variables in a javascript file (model).

React's Main Features and Advantages

- Created initially by engineers from Facebook[1], now it is **open source**.
- Allows building **"composable"** user interfaces (UI):
  1. Components encapsulate **markup** and **Javascript**.
  2. Components might have **state** ("component data model") and state changes are explicit and predictable.
- Main advantages:
  1. Uses a **pure Javascript** approach (using the JSX syntax).
  2. Has a relatively small learning curve.
  3. Has a good community.
  4. It is relatively fast.
  5. With the same technology (concepts) allows creating **mobile native UIs**: react native.

[1] Note. Part of the following slides is based on the official documentation at: `https://reactjs.org/docs/getting-started.html`

Views with Vanilla JS: innerHTML

- We have the following markup:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
  </head>
  <body>
    <div id="app"></div>
  <script src="app.js"></script>
  </body>
</html>
```

- We wil create a view with vanilla JS using the div object of DOM and its innerHTML property.

```
document.getElementById('app').innerHTML = '<h1 id="title"> hello world <\h1>';
```

- You can run the app with a simple node-based web server like `live-server` (as npm package or VSCode extension).

Vanilla JS: DOM Objects

- Create a view using createElement:

```
const elem = document.createElement('h1');
elem.setAttribute('id','title');
elem.textContent = 'Hello world';
document.getElementById('app').appendChild(elem);
```

- Adding more elements (a span):

```
const elem = document.createElement('h1');
elem.setAttribute('id','title');
elem.textContent = 'Hello ';

const child = document.createElement('span');
child.textContent = 'world';
elem.appendChild(child);

document.getElementById('app').appendChild(elem);
```

- Is equivalent to adding to the div the following html:

```
<h1 id="title">Hello <span>world</span></h1>
```

innerHTML vs. createElement

- There are several advantages to manipulating objects instead of modifying innerHTML:
  1. When you append to or modify innerHTML, all the DOM nodes inside that element have to be re-parsed and recreated.
  2. Setting innerHTML will not automatically reattach event handlers to the new elements it creates, so you would have to keep track of them yourself and add them manually.
  3. Repeatedly re-parsing and creating elements tends to be slower.
- React uses **object manipulation**.

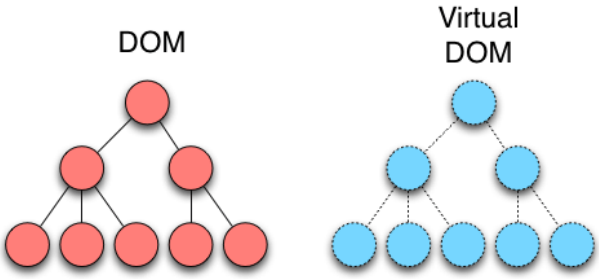
React's Virtual DOM

- React creates an in-memory copy of the DOM called **virtual DOM**:
  - The programmer makes changes in the virtual DOM.
  - React uses an algorithm to optimally synchronize the virtual DOM with the real DOM.
  - This process is called **reconciliation**.
  - Reconciliation computes the resulting differences and figures out the best way to batch the changes to the real DOM.
- The virtual DOM approach enables the **declarative API** of React.

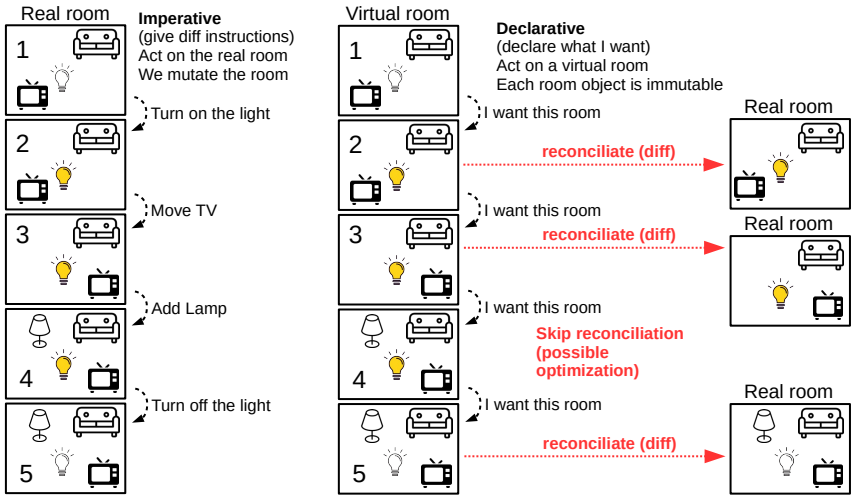
Declarative API

You tell React what state you want the UI to be in, and it makes sure the DOM matches that state.

- This allows the programmer to write code as if the entire page is rendered on each change.
- But React only renders subcomponents that actually change.



Room Analogy: Imperative, Declarative and Immutability



React Elements are Immutable

React elements are **immutable**, they are like screenshots.

- This is a convenient way of managing the state of the UI.
- If you change an element you have to render it again.

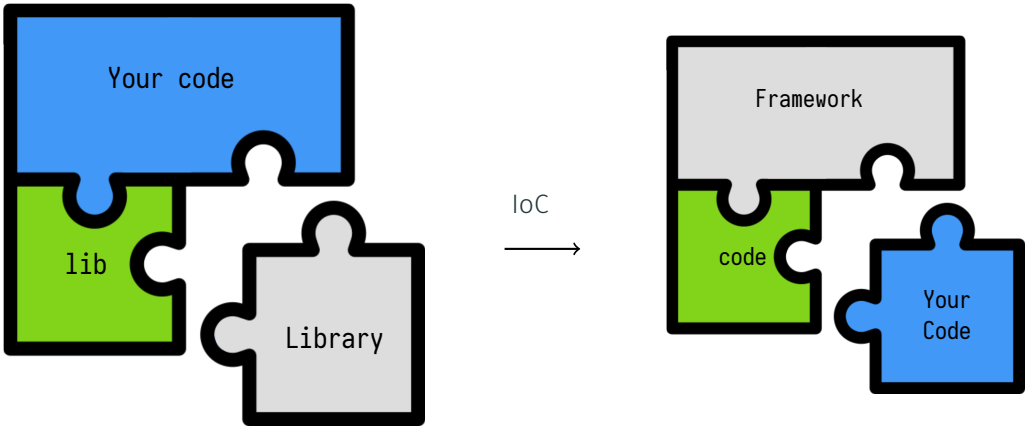
Example with vanilla JS:

```
const elem = document.getElementById('app');
function clock(){
    const now = new Date().toLocaleTimeString();
    elem.textContent = now;
    }
setInterval(clock,1000);
```

With a React element:

```
const app = document.getElementById('app');
function clock(){
    const now = new Date().toLocaleTimeString();
    const elem = React.createElement('span',null,now);
    ReactDOM.render(elem,app);
    }
setInterval(clock,1000);
```

Inversion of Control (IoC)



IoC inverts the flow of the application.

Using React i

We have to include the react library:

```
<!DOCTYPE html>
<html lang="en">

  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
  </head>

  <body>
    <div id="root"></div>
    <script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
    <script src="app.js"></script>
  </body>

</html>
```

Using React ii

- To insert the object using React:

```
// This creates an element in the React virtual DOM
const elem = React.createElement('h1', {id:'title'}, 'Hello world');
// This renders the element as a child of app
ReactDOM.render(elem,document.getElementById('app'));
```

- `elem` contains a *React element*.
- Objects with more children:

```
const elem = React.createElement(
  'h1',
  {id:'title'},
  'Hello ',
  React.createElement('span',null,'world')
  );
ReactDOM.render(elem,document.getElementById('root'));
```

JSX Concept

- Writting JS to create objects is still cumbersome.
- React defines JSX (Javascript XML) which is a **superset of JS** that allows you to write what looks like HTML markup in your JavaScript code.
- JSX is ”like” a mix of HTML and JavaScript.
- Example of JSX:

```
1  //app.jsx
2  const elem = <h1>Hello <span>world</span></h1>;
3  ReactDOM.render(elem,document.getElementById('app'));
```

- JSX is not understandable by browsers: need to compile.

JSX Compilation

- After compilation (or "transcompilation"), JSX expressions become regular Javascript.
- Compile online: `https://babeljs.io/repl`
- Compiled version:

```
// const elem = <h1>Hello <span>world</span></h1>;

'use strict';
var elem = React.createElement(
  'h1',
  null,
  'Hello ',
  React.createElement(
    'span',
    null,
    'world'
  )
);
ReactDOM.render(elem, document.getElementById('app'));
```

Essential Aspects to Learn in React

- Virtual DOM / Reconciliation ☑
- JSX (JavaScript + HTML) ☑
- Components ☐
- Properties ☐
- State management/ One-way binding ☐

Outline

Compile JSX with Babel i

- We need to transform JSX code into plain JavaScript.
- You can use several tools to do this compilation.
- A popular choice is Babel, to install it locally in our project:

```
$ mkdir myapp; cd myapp
myapp$ npm init -y
myapp$ npm install babel-cli live-server
myapp$ npx babel --version
```

- Babel uses a *"plugin"* for each transformation.
- Defines *"presets"* which are a group of plugins.

Compile JSX with Babel ii

- Presets can include other presets, we will install:

  `babel-preset-env` which installs:

  `es2015, es2016, es2017` and `latest`.

  `babel-preset-react` includes:

  `babel-plugin-transform-react-jsx`.

```
myapp$ npm install babel-preset-react babel-preset-env
```

- We create src/app.js (contains the source JSX):

```
1  console.log('App.js is running');
2  var template = <p> This is JSX from app.js </p>;
3  var appRoot = document.getElementById('app');
4  ReactDOM.render(template,appRoot);
```

- To compile (./app.js contains compiled JS):

```
myapp$ npx babel src/app.js --out-file public/app.js --presets=env,react
```

Compile JSX with Babel iii

- To follow changes (in background) of the source:

```
myapp$ babel src/app.js --out-file public/app.js --presets=env,react --watch
```

- To run the app (with `index.html` also in the public dir):

```
myapp/public$ npx live-server
```

- You can also configure babel via .babelrc (recommended):

```
{
  "plugins": ["transform-react-jsx"]
}
```

Compile JSX with Babel iv

- JSX views can be transformed for being rendered in several engines like browsers (DOM), mobiles (React native) and some more.
- The default pragma is `React.createElement`.
- This means that all JSX will turn into calls to the DOM.

```
{
  "plugins": [
    ["transform-react-jsx", {
      "pragma": "dom" // default pragma is React.createElement
    }]
  ]
}
```

Create an App with **create-react-app**

- React developers have created a very useful tool called **create-react-app**.
- This tool creates the configuration and boilerplate for react projects.
- You can create a react application with:

```
$ npm install create-react-app -g
$ create-react-app otherapp
```

- Or with **npx** (if the command is not installed, npx downloads it from the cloud):

```
$ npx create-react-app otherapp
```

- Any of the previous commands create a working directory with the template of a React project.
- If you want to develop your app with **Typescript** you can use **--typescript** at the end of the previous command.

Modules & Configuration of a `create-react-app`

The modules and configurations installed include:

1. **React libraries**: components and dom
2. **Babel**: for compiling ES5+ and React
3. **Webpack**:
   - loads babel
   - uglifies
   - creates a bundle (single file `bundle.js`)
   - Tree shaking (include only used functions from libs)
   - Configures source maps (for debug, show source code not the bundle)
   - ESM (ECMA script modules with import)
   - Etc.
4. **Webpack-dev-server**: development server that follows changes (hot reload)
5. Packages for CSS processing, test for testing, a .gitignore and some more...

More info at: `https://facebook.github.io/create-react-app/docs/getting-started`

Project Structure of our **create-react-app**

The `src` directory contains the main script which is `index.js` and `index.js` loads `App.js`

```
/otherapp
├── package.json
├── package-lock.json
├── README.md
├── /node_modules
├── /public
│   ├── index.html
│   └── ...
└── /src
    ├── index.js   (jsx)
    ├── App.js   (jsx)
    └── ...
```

Running our Application

- **Important.** Make sure that you have a `.gitignore` file in your project[2] that contains `node_modules`.
- If you have VSCode open and you run `npm start` you might get an error:
  `ENOSPC: System limit for number of file watchers reached.`
- You should increase the watchers in Linux with:

```
$ echo fs.inotify.max_user_watches=524288 | sudo tee -a /etc/sysctl.conf && sudo sysctl -p
```

- Next, modify `index.js` as follows:

```
1  import React from 'react';
2  import ReactDom from 'react-dom';
3
4  const elem = <h1> Hello World </h1>;
5
6  ReactDom.render(elem, document.getElementById('root'));
```

- Finally, you can run the project with:

```
otherapp$ npm start
```

[2] This is to avoid making your editor (e.g. VSCode) go crazy with the number of files to keep track of.

Outline

JS Expressions & Statements

- Expressions:
  - Any unit of code that can be evaluated to a value is an expression.
  - Since expressions produce values, they can appear anywhere in a program where JavaScript expects a value such as the arguments of a function invocation.
- Statements:
  - A statement is an instruction to perform a specific action.
  - Such actions include:
    - Creating and/or assigning a variable or a function.
    - Looping through an array of elements (for).
    - Evaluating code based on a specific condition (if), ...
  - JavaScript programs are actually a sequence of statements.
- For functions expression or statement depends on the context[3]:

```
1  const f1 = function () { };      // f1 assigned to a function expression
2  const f2 = function foo() { };   // f2 assigned to a named function expression
3  function foo() { };              // function declaration (this is a statement and it is hoisted)
```

[3] The compiler, depending on the context, determines if the code is an expression or a statement.

JS Expressions Inside JSX

- In JSX, you can put any valid **JavaScript expression** inside curly braces:
  - **JS expressions**: variables, numbers, strings, booleans, arrays, object properties, function calls `()`, ternary expressions `?` and conditionals with `&&` and `||`.
  - **JS statements**: `if`, `for`, function definitions, assignations, ... **are forbidden** inside {} in JSX.

```
const aUser = {
  name: 'Joe',
  surname: 'Doll'
}
function getFullName(user){
   return user.name+' '+user.surname;
}
const elem = <h1> Hello {getFullName(aUser)} </h1>;
```

- Note. The reason being that React uses the fact that JS expressions can be passed as parameters of functions.

Conditional Rendering

If one JSX expression results in **undefined**, **null** or **false** then, nothing is rendered.

- E.g. don't show if user under 18:

```
<h1> { user.age && ( user.age >= 18 && user.name || "child" ) } </h1>
```

- We can use the ternary operator:

```
<h1> {user.name ? user.name : 'Anonymous' } </h1>
```

JSX inside JS

- You can use JSX (expressions that start with "<") inside JS:
  - Assign JSX to variables.
  - Accept JSX as function arguments.
  - Return JSX from functions.
  - Obviously, use JSX expressions inside `if`s and `for`s.

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {getFullName(user)}</h1>;
  }
  return <h1>Hello, Stranger</h1>;
}
```

Close All Tags

- In HTML you can leave certain elements not closed:

```
<img src="myfig">
```

```
<input type="text">
```

- The JSX compiler needs that you **close all the tags**:

```
<img src="myfig" />
```

```
<input type="text"></input>
```

Attributes

1. `class` is is a JS reserved word so JSX uses `className` instead:
   - In HTML:

```
<div class="active"></div>
```

   - In JSX becomes:

```
<div className="active"></div>
```

2. HTML dashes are replaced by camel case, for example, `tab-index` in html is replaced by `tabIndex` in JSX:
   - You can use quotes to specify string literals as attributes.

```
const element = <div tabIndex="0"></div>
```

   - Can use curly braces to embed a JS expression in an attribute:

```
const element = <img src={user.avatarUrl}></img>;
```

Comments in JSX

- To use comments in your JSX code, you need to use JavaScript.
- You can use regular /* Block Comments */, but they need to be wrapped in curly braces:

```
{/* A JSX comment */}
```

- Same goes for multiline comments:

```
{/*
  Multi
  line
  comment
*/}
```

- This a little annoying but React does not plan to create ”native” JSX comments.

Children with JSX

- Use parenthesis for multiple lines:

```
const name = "Joe";
const elem = (
<h1> Hello {name} </h1>
<h2> 2+7 is {2+7} </h2>
);
```

- However, the previous JSX expression **is not correct**.
- You cannot have JSX expressions with two elements at the same level.
- We can fix this with a container element like a div:

```
var name = "Joe";
const elem = (
<div>
  <h1> Hello {name} </h1>
  <h2> 2+7 is {2+7} </h2>
</div>
);
```

Parenthesis Best Practices

- Parenthesis are not compulsory but are good practice.
- Where Javascript inserts the semi-colon?

```
function f(){
return
  <p>hello
    <span> world </span>
  </p>
}
```

++Prof The previous used not to compile because it was equivalent to:

```
function f(){
return;
  <p>hello <span> world </span> </p>
}
```

But now compiles and works as expected!!!

Arrays

- Arrays are rendered by default when included in JSX.
- Example (you can put numbers, strings and bools):

```
{[1,99,"hi",false,null]}
```

- Is equivalent to:

```
{1}{99}{"hi"}{false}{null}
```

- Arrays can have JSX expressions as their items.
- From React 16.3 arrays can be used as another way of rendering multiple elements at the same level:

```
const elem = (
  [
    <p> a </p>,
    <p> b </p>
  ]
)
```

Keys

- The previous JSX produces a warning in the console.
- The problem is that:

  When DOM object has many children, if React cannot clearly identify these children, then, React cannot optimize the reconciliation process.
- Adding a **key** that uniquely identifies each object helps React:

```
const elem = ([
  <p key="a3f1"> a </p>,
  <p key="45cb"> b </p>
]);
```

- By default React uses the index of the array as key but this can produce performance penalties when doing reordering or removing items.

Lists with `map()`

- The `map()` function is typically used to create lists:

```
const names = ['Joe','Maria','Peter'];
elem = (
<ul>
  {names.map((name,index) => <li key={index}>{name}</li>)}
</ul>
);
```

- Notice that we assign the **key** to the name's index of the array.
- This removes the warning but does not solve potential performance problems.
- In general, this **key** has to be assigned to a stable ID (e.g. the ID of the data in the database).

Fragments

A common pattern in React is for a component to return multiple elements.

Fragments let you group a list of children without adding extra nodes to the DOM (no need div).

```
// Long syntax
(
<React.Fragment>
  <ChildA />
  <ChildB />
  <ChildC />
</React.Fragment>
)
```

There is a short syntax: <> </>

https://reactjs.org/docs/fragments.html

Keyed Fragments

```jsx
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Without the 'key', React will fire a key warning
        <React.Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </React.Fragment>
      ))}
    </dl>
  );
}
```

The key is the only attribute that can be passed to Fragment (at the moment in version 16).

Outline

Components

- The power of React comes, of course, not from the tags described in the HTML spec, but from the user-created components.
- Components allow to split the UI in **reusable pieces**.
- Then, we can **reason** about each component (piece) **independently**.
- Conceptually, components are like JS functions:
  - Accept arbitrary inputs called **props**.
  - Return JSX (React elements describing the UI).
- We have two basic types:
  - **Functional components**
  - **Class components**

Functional Components

- The simplest way to define a component is:

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

- Such components are called **functional components** because they are literally Javascript functions.
- You can also use arrow functions:

```
const Welcome = props => <h1>Hello, {props.name}</h1>
```

Rendering a Component i

- Previously, we only encountered React elements that represent DOM tags.
- However, elements can also represent user-defined components (notice that starts with a capital letter):

```
<Welcome name="Sara" />

// JSX will translate it to:
// Welcome(name: "Sara");
```

- When React sees an element representing a user-defined component:
  - Passes JSX attributes in a single object.
  - This object is called ”props”.

Rendering a Component ii

- E.g., the following code renders ”Hello, Sara” on the page:

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

const element = <Welcome name="Sara" />
ReactDOM.render( element, document.getElementById('root'))
```

- Note. If you are using VSCode you can install an extension to create the previous boilerplate code.
  ”ES7 React/Redux/GraphQL/React-Native snippets”:
  - rfc: react functional component
  - rcc: react class component
  - rce: react class component with module
  - rconst: constructor for react
  - ...

Composing Components

- Typically, new React apps have a single App component at the very top.
- Components can refer to other components in their output:

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

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  )
}

ReactDOM.render( <App />, document.getElementById('root') )
```

Import & Export Components

- Move component code to src/components/Welcome.js:

```
import React from 'react';

export default function Welcome(props) {
  return <h1>Hello, {props.name}</h1>
}
```

- We exported the component, need to import in src/index.js:

```
import React from 'react';
import ReactDom from 'react-dom';
import Welcome from './components/Welcome';

ReactDOM.render(
  <div>
    <Welcome name="Sara" />
    <Welcome name="Eva" />
  </div>,
  document.getElementById('root')
);
```

- Note. Starting from React 17 it is not necessary to import React to use JSX because the new JSX transform automatically imports the necessary react/jsx-runtime functions.

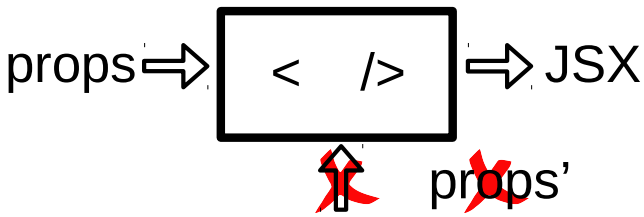
Props are Read-Only

- A component must never modify its own props.
- The component has to be a ”pure function” of props:
  1. Does not change the inputs.
  2. Always produces the same final result for the same inputs.

```
// pure
function sum(a, b) {
return a + b
}

//impure
function withdraw(account, amount) {
account.total -= amount
}
```

Props are Read-Only ii



- React is pretty flexible but it has a single strict rule:
  - All React components must act like **pure functions with respect to their props**.
  - React components that need to change their output over time in response to user actions, network responses, etc. **use state**.

Smaller Components i

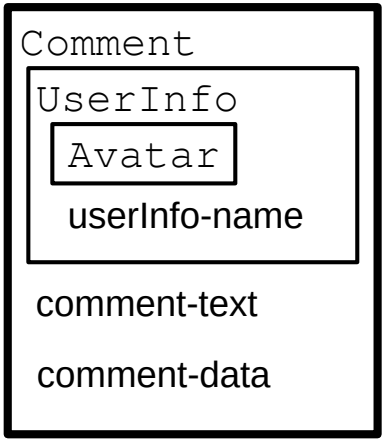
<Comment/> accepts author (an object), text (a string), and date (a date) as props, and describes a comment:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
          <img className="Avatar" src={props.author.avatarUrl}
               alt={props.author.name} />
          <div className="UserInfo-name"> {props.author.name} </div>
      </div>
      <div className="Comment-text"> {props.text} </div>
      <div className="Comment-date"> {formatDate(props.date)} </div>
    </div>  )
}
```

However, it can be tricky to change or reuse because of all the nesting.

Smaller Components ii

- Don't be afraid to split components into smaller components.
- Palettes of reusable components are very useful in large apps.
- Let's **extract** a few components from it.



Smaller Components iii

First, we will extract Avatar:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
          <img className="Avatar" src={props.author.avatarUrl}
               alt={props.author.name} />
          <div className="UserInfo-name"> {props.author.name} </div>
      </div>
      <div className="Comment-text"> {props.text} </div>
      <div className="Comment-date"> {formatDate(props.date)} </div>
    </div>  )
}
```

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />
  )
}
```

Notice that:

- The Avatar doesn't need to know that it is being rendered inside a Comment.
- This is why we have given its prop a more generic name: user rather than author.

Smaller Components iv

- It is recommended naming props from the component's own point of view rather than the context in which it is being used.

Smaller Components v

We can now simplify a little <Comment/>:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  )
}
```

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />
  )
}
```

Smaller Components vi

Next, we will extract a <UserInfo/> component:

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  )
}
```

<UserInfo/> renders an <Avatar/> next to the user's name.

This lets us simplify <Comment/> even further:

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  )
}
```

Candidates to be **reusable** components are:

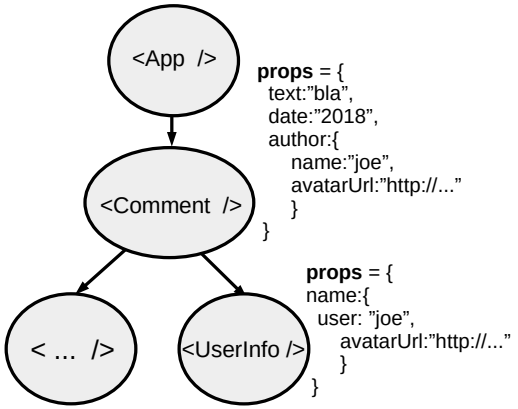
1. UI parts that are used several times: Button, Panel, Avatar, etc.
2. Complex parts: App, FeedStory, Comment, etc.

Components & Props Overview

```
<Comment text="bla" date="2018" author={{ name:"joe",avatarUrl:"http://..."}} />
```

```
<UserInfo user={{ name:"joe",avatarUrl:"http://..."}} />
```

- A functional component is a pure function of props.
- Each component decides the name of its ”attributes” that it will receive in the props object.
- Props flow from parent to children (top-down).
- For a children to re-render anything different it has to be re-rendered by its father with different props.
- Something is missing: **how to auto re-render?** for this, we use **state**.



Outline

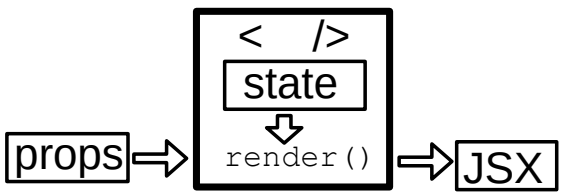
State i

- **state** is what allows you to create components that are **dynamic** and **interactive**.
- At this moment, we can define **state** as a plain Javascript **object** that allows you keep track of a component's data.
- The **state** of a component can **change**.
  - **state** changes can be based on user response, new messages from server-side, network response, or anything.
  - Component **state** is expected to be **private** to the component and controlled by the same component.
  - To make changes to a component's **state**, you have to make them **inside the component**.

State ii

state can "only"[4] be created for **class components**.

- `render()` is a pure function of **state** and **props**.
- When state changes, the component is **automatically re-rendered** by React.



[4] Now, we have also **React Hooks** which allow to manage state using functional components (we will see hooks later).

Class Components

Class components are defined using the ES6 class syntax:

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

```
import React from 'react'

class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>
  }
}
```

Note. If the constructor is not specified in a class definition, by default Javascript calls the constructor of the father with `super()`.

Class components have some additional features[5].

[5] Like `state` and `lifecycle hooks`

Example a Clock

- By now, the only way we now to update the UI is to re-render with different **props**:

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );

  ReactDOM.render( element, document.getElementById('root'));
}

setInterval(tick, 1000);
```

- But... we want a truly **reusable** and **encapsulated** clock component.

Encapsulating <Clock />

```
function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  ReactDOM.render( <Clock date={new Date()} />, document.getElementById('root'));
}

setInterval(tick, 1000);
```

- In the typical React dataflow, **props** are the **only way that parent components interact with their children**.
- So we followed this strategy of re-rendering with different props.
- However, we miss a crucial requirement:
  Timer set up and UI update should be **an implementation detail** of <Clock />.

State in <Clock /> i

- Ideally we want to write this once and have the Clock update itself:

```
ReactDOM.render( <Clock />, document.getElementById('root'));
```

- To implement this, we need to add `state` to the Clock component.
- `state` is similar to `props`, but it is private and fully controlled by the component.
- We have to use class components to include `state` and then, we move the date from `props` to `state`.

State in <Clock /> ii

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
ReactDOM.render( <Clock />, document.getElementById('root'));
```

With VSCode you can use an extension to automatically convert a functional component to a class component:

”React pure to class” (run its command from menu)

Next question: **how to update the clock?**

Setting the State in <Clock /> i

We can encapsulate the `tick()` function inside the class component.

Then, in the `constructor` we call `tick()` each second with `setInterval` to set the state:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
    setInterval(this.tick, 1000);
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
 tick() { this.state = { date: new Date() }; }
}
ReactDOM.render( <Clock />, document.getElementById('root'));
```

Setting the State in <Clock /> ii

The problem is that React needs to be informed about the state change.

For this purpose a function called **setState** is used:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
    setInterval(this.tick, 1000);

  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
 tick() { this.setState({ date: new Date() }); }
}
ReactDOM.render( <Clock />, document.getElementById('root'));
```

Review: Javascript Scope and Javascript Context

Variables available to functions are:

1. Function arguments (parameters).
2. Inner defined variables.

```
function f(a){
  let b = 2;
  console.log(a, b)
  }
f(1); // 1 2
```

   But variables defined in child functions are not available to father functions.

3. Variables in **scope**.
4. **Context** (this).

Review: Javascript Scope

- **Scopes** are composed in general of many variables.
- Scopes are lexical which means that available variables are determined by where the function is defined (inside of which other functions).

```
let b = 10;
let c = 3;
function f(a) {
    let b = 2;
    console.log(a, b, c);
  }
f(1); // 1 2 3
```

- Another example:

```
let c = 3;
function g(d) {
    let e = 4;
    function f(a) {
        let b = 2;
        console.log(a, b, c, d, e);
      }
    f(9);
  }
g(1); // 9 2 3 1 4
```

- **Prof++** b is in the scope of g? the answer is no, father doesn't have access to children inner variables.

Review: Javascript Context i

- **Context** is just a single variable called `this`.
- The context (variable `this`) is set **when the function is CALLED**.
- Exactly, is the object on the left of the funcion when called.
- If there is not such a left object then the global object is implicitly used (which can be `global` in nodejs or `window` in browsers).

Review: Javascript Context ii

- Example:

```
let c = 3;
let z = {
    b: 2,
    f(a) { console.log(a, c, this); }
}

z.f(1); // 1 3 {z}
let g = z.f; // defines the variable g and assigns it to z.f
g(1); // 1 3 {global}
```

Review: Javascript Context iii

- Consider slight modification:

```
let c = 3;
let z = {
    b: 2,
    f(a) { console.log(a, c, this.b); /* replace this by this.b */ }
}

z.f(1); // 1 3 2
let g = z.f;
g(1); // 1 3 undefined
```

Review: Binding i

- We can fix the previous code with binding:

```
g = g.bind(z);  // g is re-assigned to a version of itself with this = z
g(1);           // 1 3 2
```

- Another example:

```
let c = 3;
let z = {
        b: 2,
        f(a) {
            function k() { console.log(a, c, this.b); }
            k();
        }
}

z.f(1); // 1 3 undefined
        // "this" inside f is {z}
        // "this" inside k is {global}
```

Review: Binding ii

- Finally, we can fix it with binding:

```
let c = 3;
let z = {
          b: 2,
          f(a) {
            function k() { console.log(a, c, this.b); }
            k = k.bind(z);
            k();
          }
}

z.f(1); // 1 3 2
```

Review: Bind to **this** i

- The previous example could broke if we use another name for our object:

```
let c = 3;
let z = {
          b: 2,
          f(a) {
            function k() { console.log(a, c, this.b); }
            k = k.bind(z);
            k();
          }
}

let y = z;
z = null;
y.f(1); // 1 3 undefined
```

Review: Bind to **this** ii

- Better way is to bind to **this**:

```
let c = 3;
let z = {
          b: 2,
          f(a) {
            function k(){ console.log(a, c, this.b); }
            k = k.bind(this);
            k();
          }
}

let y = z;
z = null;
y.f(1); // 1 3 2
```

Review: Context of Arrow Functions

- Arrow functions do not have context.
- They get the `this` variable from the **scope** (lexical).
- Taking this into account, we can re-write the previous example as follows:

```
let c = 3;
let z = {
          b: 2,
          f(a) {
            k = () => { console.log(a, c, this.b); }
            k();
          }
}

z.f(1); // 1 3 2
```

Re-contextualize our `tick` Function

- So the problem is that `this.tick` is extracted from its object to be passed to `setInterval` and thus, it losses its context (`this` variable).
- We can re-contextualize it with `bind()`:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
    setInterval(this.tick.bind(this), 1000);
  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
 tick() { this.setState({ date: new Date() }); }
}
ReactDOM.render( <Clock />, document.getElementById('root'));
```

Arrow Functions in Class Properties

New syntax:

```
class A {
  static color = "red";
  counter = 0;

  handleClick = () => {
    this.counter++;
  }

  handleLongClick() {
    this.counter++;
  }
}
```

Compiled to ES6:

```
class A {
  constructor() {
    this.counter = 0;

    this.handleClick = () => {
      this.counter++;
    };
  }

  handleLongClick() {
    this.counter++;
  }
}
A.color = "red";
```

CRA has the new syntax of arrow functions in class properties activated by default.

React and Arrow Functions in Class Properties

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
    setInterval(this.tick, 1000);
  }
  render() {
    return ( <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div> );
  }
 tick = () => { this.setState({ date: new Date() }); }
}
ReactDOM.render( <Clock />, document.getElementById('root'));
```

However:

- Setting the interval in the `constructor` is not elegant.
- We should do as less work as possible before rendering, so `constructor` is not a good place to put logic.
- The timer should be activated once the component is rendered (and the user is seeing something).

++Prof Advanced About Arrow Functions in Class Properties

- There are some issues with arrow functions as class properties[6].
- Arrow functions created like this ARE NOT in the object prototype.
- This causes several problems:
  - Mockability: if you want to mock or spy a class method the easiest way is to do it at the prototype.
  - Inheritance: derived classes do not get methods that are not in the prototype.
  - Performance: this is probably the worst, since the arrow functions are not in the prototype, they are created PER object.
- Maybe is better to just bind!

[6] Check https://medium.com/@charpeni/arrow-functions-in-class-properties-might-not-be-as-great-as-we-think-3b3551c440b1

Component Mounting i

- When an instance of a component is first created and inserted into the DOM (called **mounting**) the following functions in the class component are called:
  1. `constructor()`
  2. `static getDerivedStateFromProps() // rarely used`
  3. `render()`
  4. `componentDidMount()`

Component Mounting ii

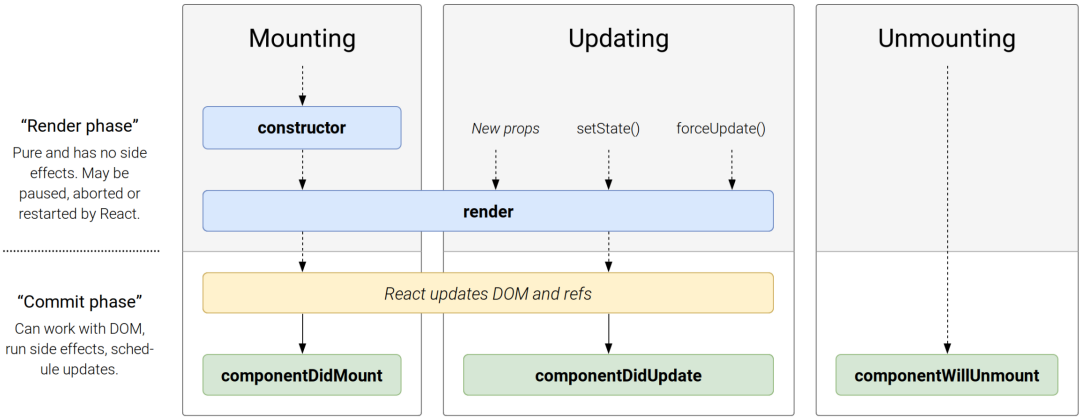
1. `constructor()` is used to initialize state and bind methods.
2. `getDerivedStateFromProps()` is used in rare cases.
   - This is for components where the state depends on changes in props over time.
   - For example, it might be handy for implementing a `<Transition>` component that compares its previous and next children to decide which of them to animate in and out.

Component Mounting iii

3. `render()` function to render the component and should be **pure** for `state` and `props`:
   - Does not modify component `state`.
   - It returns the same result each time it's invoked with the same `state` and `props`.
   - Does not directly interact with the browser.
   - Keeping `render()` pure makes components easier to reason about them.

4. `componentDidMount()` is where you put non pure code, e.g. if you need to interact with the browser or consume an API.

Typical Lifecycle Methods



Source: `http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram` The previous methods are also called "lifecycle hooks"

Timer for <Clock/>

```
import React from 'react';

export default class Clock extends React.Component {

    constructor(props) {
        super(props);
        this.state = {date: new Date()};
    }

    render() {
        return (
        <div>
          <h1>Hello, world!</h1>
          <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
        </div>
        );
    }

    tick = () => this.setState({ date: new Date() });

    componentDidMount() {
        setInterval(this.tick, 1000);
    }
}
```

Stop Timer of <Clock/>

```
class Clock extends React.Component {

  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }

  tick = () => this.setState({ date: new Date() });

  componentDidMount() { this.timerID = setInterval(this.tick, 1000); }

  componentWillUnmount() { clearInterval(this.timerID); } // Be polite and stop the timer ;-)
}
```

Coloured <Clock/>

We can create components that render with **props + state**:

```
class Clock extends React.Component {

  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2 style={{ color: this.props.color }}>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }

  tick = () => this.setState({ date: new Date() });

  componentDidMount() { this.timerID = setInterval(this.tick, 1000); }

  componentWillUnmount() { clearInterval(this.timerID); } // Be polite and stop the timer ;-)
}
```

Using State Correctly: Don’t Modify State Directly

1. **Don’t modify state directly**
   - For example, this will not re-render a component:

```
// Wrong
this.state.comment = 'Hello';
// Correct
this.setState({comment: 'Hello'});
```

   - The only place where you can assign directly `this.state` is the `constructor()`.
   - In any other place use `setState()`

2. State updates are merged
   - React merges the object you provide into the current state when you call `setState()`
   - For example, your state may contain several independent variables:

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}

```

   - Then you can update `posts` and `comments` independently with separate `setState()` calls.

Using State Correctly: State Updates are Merged ii

```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({ posts: response.posts });
  });

  fetchComments().then(response => {
    this.setState({ comments: response.comments });
  });
}
```

- The merging is shallow:

  `this.setState(comments)` leaves `this.state.posts` intact, but completely replaces `this.state.comments`

3. State updates are, in general, asynchronous
   - Consider that you don't want to replace a part of the state but that you want to **set a new state based on a previous state or props**.
   - State updating is **asynchronous**, so, a previous `setState()` call might not be applied when you expect.
   - This is because React may batch multiple `setState()` calls into a single update for performance.

Using State Correctly: State Updates are Asynchronous ii

- Consider that `this.state.counter = 0` and we try increment the value using `setState()`:

```
// Wrong way

this.setState({
  counter: this.state.counter + 1 ,
});
// ...
this.setState({
  counter: this.state.counter + 1,
});

```

- Because `this.state` (and `this.props`) may be updated asynchronously you can end executing the second `setState()` from `this.state.counter = 0`.
- Then, you end with `state.counter = 1` (not 2, what is what you expect).
- In general, you should **not rely on `this.props` or `this.state`** for calculating the next `state`:
  - To fix this, `setState()` accepts a second form that uses a function rather than an object as parameter.

Using State Correctly: State Updates are Asynchronous iii

- That function will receive the previous **state** as the first argument, and the **props** at the time the update is applied as the second argument:

```
// Correct way
this.setState((prevState, props) => ({
  counter: prevState.counter + props.increment
}));

this.setState((prevState, props) => ({
  counter: prevState.counter + props.increment
}));

```

Using State Correctly: Use Immutable State

3. Always treat `this.state` as immutable.

```
this.state = {
    user: {name: "joe", age: 40 },
    hobbies: ["basket", "walk"]
}

function myFunctionIncorrect(name) {
    let user = this.state.user; // this is a reference, not a copy...
    user.name = name; // so this mutates the state!!!
    this.setState({user});
}

function myFunctionCorrect(name) {
    this.setState( prevState => {
          return {user: {...prevState, name}}; // we do not modify the previous state
    });
}
```

Outline

Consume APIs

The ideal place to consume APIs is after the component is mounted:

```
import React from 'react';
import axios from 'axios';

export default class UserList extends React.Component {
  state = {  // with CRA we can use class variables
    users: []
  }
  render() {
    return (
      <ul>
        { this.state.users.map(user => <li>{user.name}</li>)}
      </ul>
    )
  }
  componentDidMount() {
    axios.get("https://jsonplaceholder.typicode.com/users")
      .then(res => {
        const users = res.data;
        this.setState({ users });
      })
  }
}
```

Consume APIs ii

We can improve our code using `async`/`await` and the `id` from the data to set the key in the array of `li`'s:

```
import React from 'react';
import axios from 'axios';

export default class UserList extends React.Component {
  state = {
    users: []
  }
  render() {
    return (
      <ul>
        { this.state.users.map(user => <li key={user.id}>{user.name}</li>)}
      </ul>
    )
  }
  async componentDidMount() {
    const res = await axios.get("https://jsonplaceholder.typicode.com/users");
    const users = res.data;
    this.setState({ users });
  }
}
```

Consume Our Own API

- We can create mock an API server with `json-server`:

```
$ npm install json-server -g
$ json-server --watch data.json --port 4000
```

- Where `data.json` can be something like:

```
{
  "users": [
    { "id": 1, "name": "joe" },
    { "id": 2, "name": "elle" }
  ]
}
```

- Then, we can test the API with `curl`:

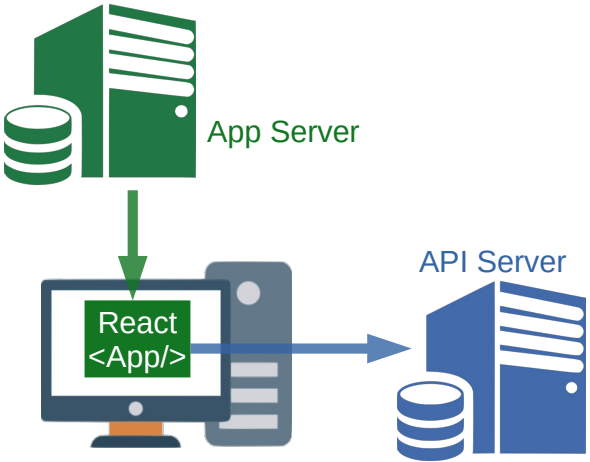
```
$ curl -X GET http://localhost:4000/users
$ curl -X POST --header "Content-Type: application/json" --data '{"id": 3, "name": "john"}' http://localhost:4000/users
$ curl -X DELETE localhost:3000/users/5
$ curl -X PUT --header "Content-Type: application/json" --data '{"id": 3, "name": "jane"}' http://localhost:4000/users/3
```

Consume Our Own API ii

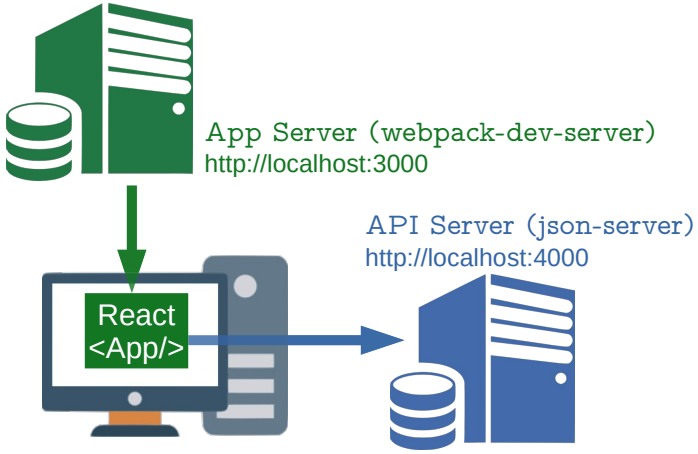
Finally, we just change the API server to our local instance:

```
export default class UserList extends React.Component {
  state = {
    users: []
  }
  render() {
    return (
      <ul>
        { this.state.users.map(user => <li key={user.id}>{user.name}</li>)}
      </ul>
    )
  }
  async componentDidMount() {
    const res = await axios.get("http://localhost:4000/users");
    const users = res.data;
    this.setState({ users });
  }
}
```

Consume Our Own API iii



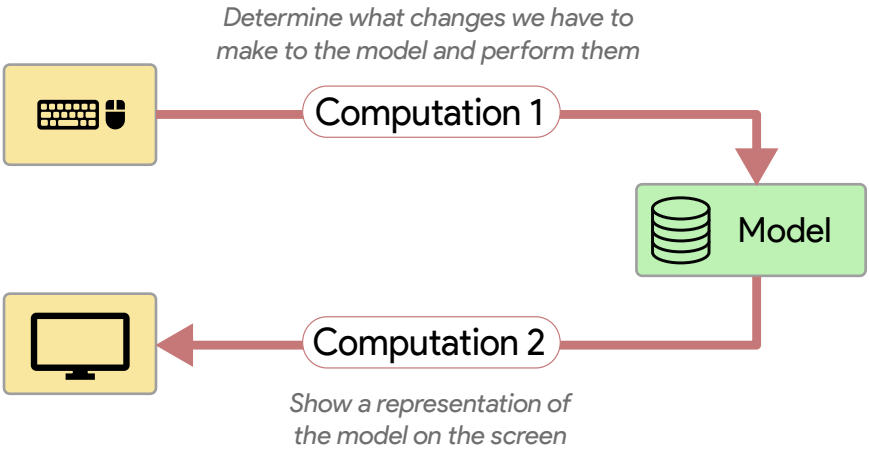
Consume Our Own API iv



Note that with some resource servers you might have problems because of **CORS**.

Outline

Main Updating Loop Diagram



Handling Events i

- React events are very similar to events on DOM elements.
- There are some syntactic differences:
  - React events are named using camelCase, rather than lowercase.
  - With JSX you pass a function as the event handler, rather than a string.
- For example, the HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

- In React:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Prof++ en onClick podemos poner varias funciones separadas por comas.

Handling Events ii

- Always explicitly call `preventDefault()`, which in this case would follow the link:

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }
  return (
    <a href="#" onClick={handleClick}>  Click me  </a>
  );
}
```

- Here, `e` is a synthetic event[7] (don't need to worry about cross-browser compatibility).
- Exercise:

  Modify the `<ActionLink>` so that when we click the `<a>` link, we toggle the visualization/hidding of a `<Clock>`.

[7] In general, don't need `addEventListener()` to add listeners to a DOM element after it is created, just provide a listener when the element is initially rendered.

Event Handlers in Class Components

- Event handlers are class methods:

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};
    // This binding is necessary to make "this" work in the callback
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState(prevState => ({ isToggleOn: !prevState.isToggleOn }));
  }
  render() {
    return <button onClick={this.handleClick}> {this.state.isToggleOn ? 'ON' : 'OFF'} </button>
  }
}
ReactDOM.render( <Toggle />, document.getElementById('root') )
```

- Bind in a prop?

Public Class Fields Syntax

- If calling bind annoys you, you can use the experimental public class fields syntax:

```
class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  // Warning: this is *experimental* syntax.
  handleClick = () => {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

- This syntax is enabled by default when using `create-react-app`.

HTML Forms

- The default HTML form behavior is browsing to a new page when the user submits the form.
- If you want this behavior in React, it just works:

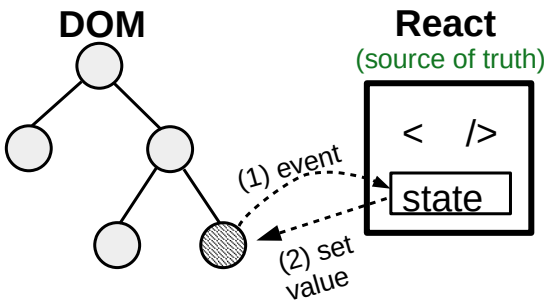
```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

- But, in most cases, it's convenient to avoid the default behaviour and have a JavaScript function that handles the **submission** and **state** of the form.
- HTML form elements such as `<input>`, `<textarea>`, and `<select>`:
  - Maintain their own internal state.
  - Update it based on user input.

Controlled Components

An input form element whose value is controlled by React state is called a **controlled component**.

We can write the previous form as a controlled component.



A Controlled Component

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
    this.setState({value: event.target.value});
  }
  handleSubmit(event) {
    event.preventDefault();
    alert('A name was submitted: ' + this.state.value);
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label> Name:
         <input type="text" value={this.state.value} onChange={this.handleChange}/>
        </label>
        <input type="submit" value="Submit" />
      </form>  );
  }
}
```

The textarea Tag

- In HTML, a `<textarea>` element defines its text by its children:

```
<textarea>
  Hello there, this is some text in a text area
</textarea>
```

- In React, a `<textarea>` uses a value attribute instead.
- This way, a form using a `<textarea>` can be written very similarly to `<input>`

Example: textarea

```
class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Please write an essay about your favorite DOM element.'
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
    this.setState({value: event.target.value});
  }
  handleSubmit(event) {
    event.preventDefault();
    alert('An essay was submitted: ' + this.state.value);
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>  Essay:
          <textarea value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>  )
  }
}
```

Debugging

When you see something wrong in the UI, you can use **React Developer Tools**:

- You can inspect the props and move up the tree until you find the component responsible for updating the state (lets you trace the bugs to their source).
- Right-click any node and choose ”Show Source” to jump to the render method in the Sources panel.
- Selected component instance is available as `$r` from the console.
- Right-click any props or state value to make it available as `$reactTemp0` (or similar) from the console.

The select Tag

- In HTML, `<select>` creates a drop-down list:

```
<select>
  <option value="grapefruit">Grapefruit</option>
  <option value="lime">Lime</option>
  <option selected value="coconut">Coconut</option>
  <option value="mango">Mango</option>
</select>
```

- Note that the Coconut option is initially selected.
- React, instead of using this selected attribute, uses a value attribute on the root select tag.
- This is more convenient in a controlled component because you only need to update it in one place.

Example: select Tag

```
class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'coconut'};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) { this.setState({value: event.target.value}); }
  handleSubmit(event) { event.preventDefault(); }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Pick your favorite flavor:
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">Grapefruit</option>
            <option value="lime">Lime</option>
            <option value="coconut">Coconut</option>
            <option value="mango">Mango</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Attribute **value**

- Overall, this makes it so that `<input type="text">`, `<textarea>`, and `<select>` all work very similarly with the value attribute.
- You can also allow multiple selection.
- To do so, you pass an array into the value attribute:

```
<select multiple={true} value={['coconout', 'lime']}>

// ...

<select/>
```

undefined value

- Specifying the value prop on a controlled component prevents the user from changing the input unless you desire so.
- However, if value is set to **undefined** then, the value is **automatically editable** (we turn the **input** uncontrolled).
- The following code demonstrates this:

```
ReactDOM.render(<input value="hi" />, mountNode);

setTimeout(function() {
  ReactDOM.render(<input value={undefined} />, mountNode);
}, 1000);
```

- The input is locked at first but becomes editable after a short delay.
- If you don't want that the value is editable be careful to not to set the value prop to **undefined**.

Uncontrolled Components i

- It can sometimes be tedious to use controlled components:
  - You need to write an event handler for every way your data can change.
  - Pipe all of the input state through a React component.
- This can become particularly annoying when you are converting a pre-existing codebase to React, or integrating a React application with a non-React library.
- In these situations, you might want to use **uncontrolled components**.
- Another situation in which you have to use an uncontrolled component is with the `file input Tag` because its value is read-only.

Uncontrolled Components ii

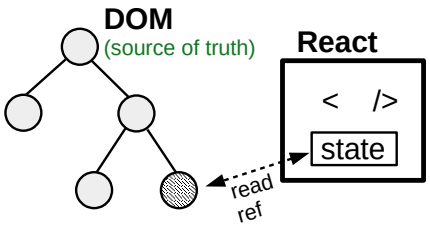
In uncontrolled components the form data is handled by the DOM itself (source of truth is the DOM).

- To write an uncontrolled component:
  1. You don't write an event handler for every state update.
  2. You use a `ref` to get form values from the DOM.
- For this, we use `React.createRef()`.

  Note. There is an old (and deprecated) method for creating refs:

```
1  <input ref={node => this.input = node} type="text"/>
```

  But it is deprecated because it creates a function per render.



Uncontrolled Components iii

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input1 = React.createRef(); // React 16 notation
  }
  handleSubmit(event) {
    event.preventDefault();
    alert('A name was submitted: ' + this.input.current.value);
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={this.input1} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Default Values in Uncontrolled

- The value attribute on React form elements will override the value in the DOM.
- With an uncontrolled component, you often want React to specify the initial value, but leave subsequent updates uncontrolled.
- To handle this case, you can specify a `defaultValue` attribute instead of value.

```
<form onSubmit={this.handleSubmit}>
  <label>
    Name:
    <input
      defaultValue="Bob"
      type="text"
      ref={this.input} />
  </label>
  <input type="submit" value="Submit" />
</form>
```

`<input type="checkbox">` and `<input type="radio">` support `defaultChecked`

`<select>` and `<textarea>` support `defaultValue`.

The `file input` Tag

- In HTML, an `<input type="file">` lets the user choose one or more files.
- Then, these files can be uploaded to a server or manipulated by JavaScript via the File API.
- In React, an `<input type="file" />` is always an uncontrolled component because its value can only be set by a user, and not programmatically.
- The following example shows how to create a ref to the DOM node to access file(s) in a submit handler.

Example: **file input** Tag

```
class FileInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.fileInput = React.createRef();
  }

  handleSubmit(event) {
    event.preventDefault();
    alert(`Selected file - ${this.fileInput.current.files[0].name}`);
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label> Upload file:
          <input type="file" ref={this.fileInput} />
        </label>
        <br />
        <button type="submit">Submit</button>
      </form> )
  }
}
ReactDOM.render(  <FileInput />,  document.getElementById('root') )
```

We can use the prop **multiple** to allow selecting multiple files, and then, we have multiple objects in **files**.

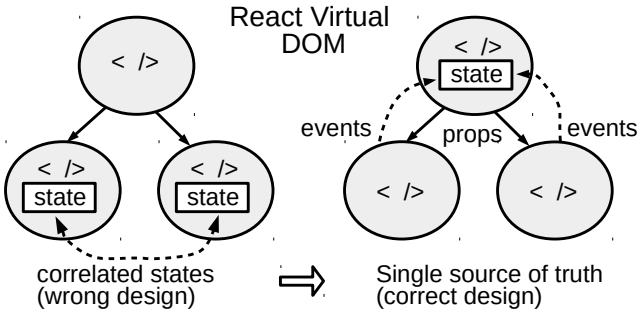
Outline

Lifting State Up

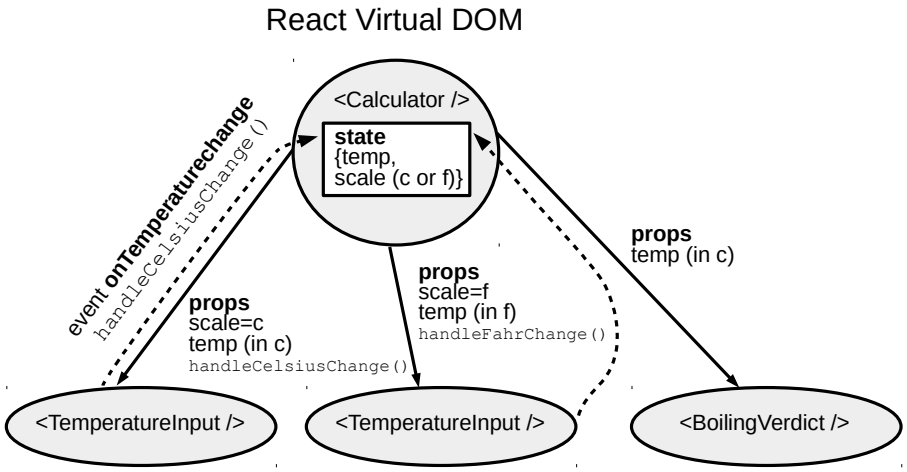
Often, several components need to reflect the same changing data.

The recommendation is **lifting the shared state up** to their closest common ancestor.

- The common ancestor needs to receive **events** from children to manage state.
- To do so, the ancestor will **pass event handlers** to these children.
- Finally, the children receive the state from the ancestor as a prop.
- A basic idea in React is: "**state up**, **props down**"



Temperatures Example



https://reactjs.org/docs/lifting-state-up.html

Temperatures Example Code

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrChange = this.handleFahrChange.bind(this);
    this.state = {temperature: '', scale: 'c'};
  }
  handleCelsiusChange(event){this.setState({scale: 'c', temperature: event.target.value});}
  handleFahrChange(event){this.setState({scale: 'f', temperature: event.target.value});}
  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) : temperature;
    const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) : temperature;
    return (<div>
          <TemperatureInput scale="c" temperature={celsius} onTemperatureChange={this.handleCelsiusChange} />
          <TemperatureInput scale="f" temperature={fahrenheit} onTemperatureChange={this.handleFahrChange} />
          <BoilingVerdict celsius={parseFloat(celsius)} />
      </div>);}
}
```

```
function TemperatureInput(props) {
  return (<div>
    <h1> {props.scale === 'c' ? 'Celsius' : 'Fahrenheit' } </h1>
    <input type="text" value={props.temperature} onChange={props.onTemperatureChange} />
  </div>);}
```

Single source of Truth

- There should be a single **source of truth** for any data that changes in a React application.
  - Instead of trying to sync the state between different components, you should rely on the top-down data flow (props).
  - If something can be derived from either props or state, it probably should not be in the state.
- **Dealing with state**:
  - Usually, the state is first added to the component that needs it for rendering.
  - Then, if other components also need it, you can lift it up to their closest common ancestor.
  - Additionally, you can implement any custom logic to reject or transform user input.

Outline

Composition Model

- React has a powerful composition model[8]:
  - It is recommended to always use composition to reuse code between components.
  - `props` and composition give you all the flexibility you need to customize a component's look and behavior.
  - `props` can be primitive values, React elements and functions.
- If you want to reuse non-UI functionality between components:
  - Extract this functionality into a module.
  - The components may import and use functionality without extending it.
- Next, we show how many situations can be solved with the composition model.

[8] Component inheritance hierarchies do not have any known use case.

Specialization

- Sometimes we think about components as being ”special cases” of other components.
- For example, we might say that a `<WelcomeDialog/>` is a special case of `<Dialog/>`.
- In React, this is also achieved by composition, more ”specific” component renders a more ”generic” one and configures it with `props`:

```
function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title"> {props.title} </h1>
      <p className="Dialog-message"> {props.message} </p>
    </FancyBorder>  ) }

function WelcomeDialog() {
  return  <Dialog title="Welcome" message="Thank you for visiting our spacecraft!" />;
}
```

Containment: **children** prop

- Some components don't know their children a priori.
- The recommended way to deal with this situation is to use the special `children` property:

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}

function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Welcome
      </h1>
      <p className="Dialog-message"> Thank you for visiting our spacecraft! </p>
    </FancyBorder>
  );
}
```

Your own **children** prop

- While this is less common, sometimes you might need multiple ”holes” in a component.
- In such cases you may come up with your own convention instead of using `children`:

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left"> {props.left} </div>
      <div className="SplitPane-right"> {props.right} </div>
    </div>
  );
}
function App() {
  return (
    <SplitPane
      left={ <Contacts /> }
      right={ <Chat /> } />
  )
}
```

Building a UI

- One of the many great parts of React is how it makes you think about apps as you build them.
- Imagine that we already have a JSON API and a mock from our designer.
- The steps to build the app:
  1. Break the UI into a component hierarchy.
  2. Build a static version (without state or interaction).
  3. Identify the minimal (but complete) representation of UI state.
  4. Identify where your state should live.
  5. Add the inverse data flow (”our computation 1”) as handlers passed in props.
- Try the example in:

  `https://reactjs.org/docs/thinking-in-react.html`