互联网应用开发技术

Web Application Development

第4课
WEB前端-REACT简介

Episode Four

React Tutorials

陈昊鹏
chen-hp@sjtu.edu.cn

Web Application Development

React

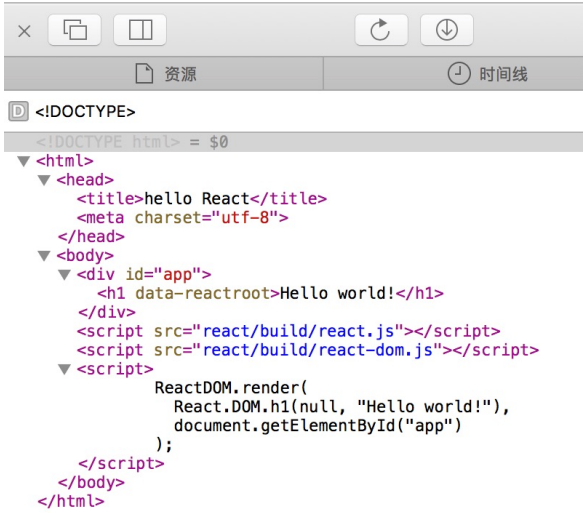
- Download
  - https://github.com/facebook/react/

- From
  - React Quick Start
  - https://reactjs.org/docs/hello-world.htm
  - React in IntelliJ IDEA
  - https://www.jetbrains.com/help/idea/react.html
  - React项目文件结构解析
  - https://my.oschina.net/korabear/blog/1815170

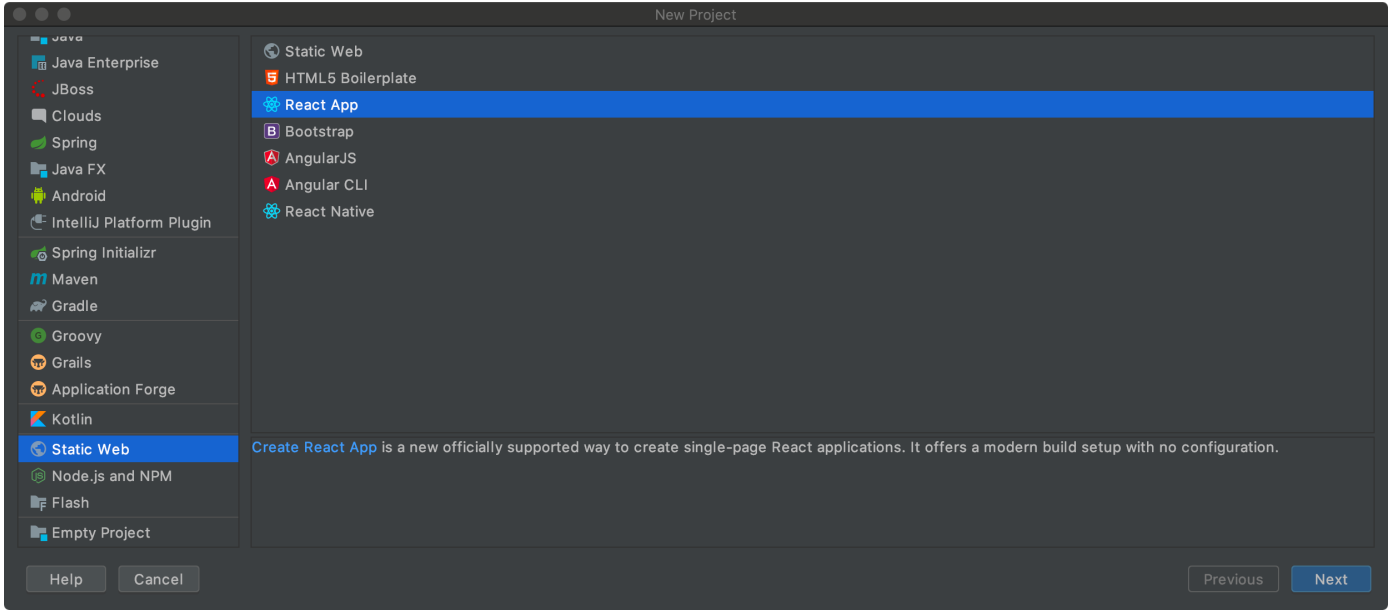
Hello world


```
<!DOCTYPE html>
<html>
 <head>
  <title>hello React</title>
  <meta charset="utf-8">
 </head>
 <body>
  <div id="app">
   <!-- my app renders here -->
  </div>
  <script src="react/build/react.js"></script>
  <script src="react/build/react-dom.js"></script>
  <script>
   ReactDOM.render(
    React.DOM.h1(null, "Hello world!"),
    document.getElementById("app")
   );
  </script>
 </body>
</html>
```

Hello world!



Hello World


New Project
Java Enterprise
JBoss
Clouds
Spring
Java FX
Android
IntelliJ Platform Plugin
Spring Initializr
Maven
Gradle
Groovy
Grails
Application Forge
Kotlin
Static Web
Node.js and NPM
Flash
Empty Project
Static Web
HTML5 Boilerplate
React App
Bootstrap
AngularJS
Angular CLI
React Native
Create React App is a new officially supported way to create single-page React applications. It offers a modern build setup with no configuration.
Help
Cancel
Previous
Next

Hello World

REliable, INtelligent & Scalable Systems

```
se122react ~/IdeaProjects/se122react
  .idea
  node_modules library root
  public
    favicon.ico
    index.html
    logo192.png
    logo512.png
    manifest.json
    robots.txt
  src
    App.css
    App.js
    App.test.js
    Game.js
    index.css
    index.js
    logo.svg
    serviceWorker.js
    setupTests.js
  .gitignore
  package.json
  package-lock.json
  README.md
  se122react.iml
External Libraries
Scratches and Consoles
```

index.js | App.js | Game.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import Game from './Game';

import * as serviceWorker from './serviceWorker';

ReactDOM.render(<App />, document.getElementById( elementId: 'root'));

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

index.js | App.js | Game.js

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
      <h1> Hello World! </h1>
  );
}



export default App;
```

React App

localhost:3000

Hello World!

Demo 1

Introducing JSX

- Consider this variable declaration:

```
const element = <h1>Hello, world!</h1>;
```

  - This funny tag syntax is neither a string nor HTML.
  - It is called JSX, and it is a syntax extension to JavaScript.

```
function formatName(user) {
 return user.firstName + ' ' + user.lastName;
}
const user = {
 firstName: 'Harper',
 lastName: 'Perez'
};
const element = (
 <h1>
   Hello, {formatName(user)}!
 </h1>
);
ReactDOM.render(
 element,
 document.getElementById('root')
);
```

Hello, Harper Perez!

Demo 2

Introducing JSX

- **JSX is an Expression Too**
  - JSX can be used inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions:

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

Introducing JSX

- **Specifying Attributes with JSX**
  - You may use quotes to specify string literals as attributes:

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

  - You may also use curly braces to embed a JavaScript expression in an attribute:

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

- **Specifying Children with JSX**
  - If a tag is empty, you may close it immediately with />, like XML:

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

  - JSX tags may contain children:

```
const element = (
  <div>
     <h1>Hello!</h1>
     <h2>Good to see you here.</h2>
  </div>
);
```

Introducing JSX

- **JSX Represents Objects**
  - Babel compiles JSX down to `React.createElement()` calls.
  - These two examples are identical:

```
const element = ( <h1 className="greeting"> Hello, world!</h1>);

const element = React.createElement(
  'h1', {className: 'greeting'},
  'Hello, world!'
);
```

  - React.createElement() performs a few checks to help you write bug-free code but essentially it creates an object like this:

```
// Note: this structure is simplified
const element = {
 type: 'h1',
 props: {
  className: 'greeting',
  children: 'Hello, world'
 }
};
```

Rendering Elements

- Elements are the smallest building blocks of React apps.

- An element describes what you want to see on the screen:

```
const element = <h1>Hello, world</h1>;
```

- Unlike browser DOM elements, React elements are plain objects, and are cheap to create.
  - React DOM takes care of updating the DOM to match the React elements.

Rendering Elements

- **Rendering an Element into the DOM**
  - Let's say there is a somewhere in your HTML file:

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

  - We call this a "root" DOM node because everything inside it will be managed by React DOM.

  - To render a React element into a root DOM node, pass both to `ReactDOM.render()`:

```
const element = <h1>Hello, world</h1>; ReactDOM.render(element,
document.getElementById('root'));
```

Rendering Elements

- **Updating the Rendered Element**
  - React elements are immutable.
    - An element is like a single frame in a movie: it represents the UI at a certain point in time.
  - The only way to update the UI is to create a new element, and pass it to `ReactDOM.render()`.
  - Consider this ticking clock example:

```
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);
```

Demo 3

Rendering Elements

- **React Only Updates What's Necessary**
  - React DOM compares the element and its children to the previous one, and only applies the DOM updates necessary to bring the DOM to the desired state.
  - Even though we create an element describing the whole UI tree on every tick, only the text node whose contents has changed gets updated by React DOM.

Hello, world!

It is 12:26:46 PM.

Console Sources Network Timeline

```
▼<div id="root">
  ▼<div data-reactroot>
      <h1>Hello, world!</h1>
    ▼<h2>
        <!-- react-text: 4 -->
        "It is "
        <!-- /react-text -->
        <!-- react-text: 5 -->
        "12:26:46 PM"
        <!-- /react-text -->
        <!-- react-text: 6 -->
        "."
        <!-- /react-text -->
      </h2>
    </div>
  </div>
```

Components and Props

- Components let you split the UI into
  - independent, reusable pieces,
  - and think about each piece in isolation.

- Conceptually, components are like JavaScript functions.
  - They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

- **Functional and Class Components**
  - The simplest way to define a component is to write a JavaScript function:

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

  - We call such components "functional" because they are literally JavaScript functions.

  - You can also use an ES6 class to define a component:

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

  - The above two components are equivalent from React's point of view.

- **Rendering a Component**
  - Elements can also represent user-defined components:

```
const element = <Welcome name="Sara" />;
```

  - When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object. We call this object "props".

  - For example, this 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')
);
```

Hello, Sara

Components and Props

- **Composing Components**
  - 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')
);
```

Hello, Sara

Hello, Cahal

Hello, Edite

Demo 4

Components and Props

- **Extracting Components**
  - Don't be afraid to split components into smaller components.
  - For example, consider this Comment component:

```
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>
 );
}
```

Demo 5

- **Extracting Components**
  - This component can be tricky to change because of all the nesting, and it is also hard to reuse individual parts of it.
  - Let's extract a few components from it.
  - First, we will extract Avatar:

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

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

Components and Props

- **Extracting Components**
  - We can now simplify Comment a tiny bit:

```
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>
 );
}
```

Components and Props

- **Extracting Components**
  - Next, we will extract a UserInfo component that renders an Avatar next to the user's name:

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

  - 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>
  );
}
```

- **Extracting Components**
  - Apply Comment:

```
const comment = {
  date: new Date(),
  text: 'I hope you enjoy learning React!',
  author: {
    name: 'Hello Kitty',
    avatarUrl: 'http://placekitten.com/g/64/64',
  },
};

ReactDOM.render(
   <Comment
     date={comment.date}
     text={comment.text}
     author={comment.author}  />,
   document.getElementById('root')
);
```

Hello Kitty
I hope you enjoy learning React!
2018/3/11

Demo 6

- **Props are Read-Only**
  - Whether you declare a component as a function or a class, it must never modify its own props.
  - Consider this sum function:

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

  - Such functions are called "pure" because they do not attempt to change their inputs, and always return the same result for the same inputs.
  - In contrast, this function is impure because it changes its own input:

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

  - React is pretty flexible but it has a single strict rule:
  - **All React components must act like pure functions with respect to their props.**

State and Lifecycle

- **State and Lifecycle**
  - we will learn how to make the Clock component truly reusable and encapsulated. It will set up its own timer and update itself every second.
  - We can start by encapsulating how the clock looks:

```
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);
```

Demo 7

State and Lifecycle

- However, it misses a crucial requirement:
  - the fact that the Clock sets up a timer and updates the UI every second should be an implementation detail of the Clock.
  - 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.
  - Local state is exactly that: a feature available only to classes.

State and Lifecycle

- **Converting a Function to a Class**
  - You can convert a functional component like Clock to a class in five steps:

1. Create an ES6 class, with the same name, that extends React.Component.
2. Add a single empty method to it called render().
3. Move the body of the function into the render() method.
4. Replace props with this.props in the render() body.
5. Delete the remaining empty function declaration.

```
class Clock extends React.Component {
 render() {
   return (
     <div>
       <h1>Hello, world!</h1>
       <h2>
         It is {this.props.date.toLocaleTimeString()}.
       </h2>
     </div>
   );
 }
}
```

Demo 8

- **Adding Local State to a Class**
  - We will move the date from props to state in three steps:

1. Replace this.props.date with this.state.date in the render() method:

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

State and Lifecycle

- **Adding Local State to a Class**
  - We will move the date from props to state in three steps:

  2. Add a class constructor that assigns the initial this.state:

```
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>
  );
 }
}
```

- **Adding Local State to a Class**
  - We will move the date from props to state in three steps:

3. Remove the date prop from the <Clock /> element:

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

State and Lifecycle

- **Adding Lifecycle Methods to a Class**
  - We want to set up a timer whenever the Clock is rendered to the DOM for the first time. This is called "mounting" in React.
  - We also want to clear that timer whenever the DOM produced by the Clock is removed. This is called "unmounting" in React.

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

  - These methods are called "lifecycle hooks".

Demo 8

- **Adding Lifecycle Methods to a Class**
  - The componentDidMount() hook runs after the component output has been rendered to the DOM. This is a good place to set up a timer:

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

  - Note how we save the timer ID right on this.
  - If you don't use something in render(), it shouldn't be in the state.
  - We will tear down the timer in the componentWillUnmount() lifecycle hook:

```
componentWillUnmount() {
 clearInterval(this.timerID);
}
```

State and Lifecycle

- **Adding Lifecycle Methods to a Class**
  - Finally, we will implement a method called tick() that the Clock component will run every second.
  - It will use this.setState() to schedule updates to the component local state:

```
class Clock extends React.Component {
 constructor(props) {…}
 componentDidMount() {…}
 componentWillUnmount(){…}

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

 render() {…}
}

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

- **Using State Correctly**
  - There are three things you should know about setState().
  - **Do Not Modify State Directly**
  - For example, this will not re-render a component:

```
// Wrong
this.state.comment = 'Hello';
```

  - Instead, use setState():

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

  - The only place where you can assign this.state is the constructor.

- **Using State Correctly**
  - **State Updates May Be Asynchronous**
  - React may batch multiple setState() calls into a single update for performance.
  - Because this.props and this.state may be updated asynchronously, you should not rely on their values for calculating the next state.
  - For example, this code may fail to update the counter:

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

  - To fix it, use a second form of setState() that accepts a function rather than an object.:

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

  - We used an arrow function above, but it also works with regular functions:

```
// Correct
this.setState(function(state, props) {
  return {
   counter: state.counter + props.increment
  };
});
```

State and Lifecycle

- **Using State Correctly**
  - **State Updates are Merged**
  - When you call setState(), React merges the object you provide into the current state.
  - For example, your state may contain several independent variables:

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

  - Then you can update them independently with separate setState() calls:

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

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

  - The merging is shallow, so this.setState({comments}) leaves this.state.posts intact, but completely replaces this.state.comments.

- **The Data Flows Down**
  - Neither parent nor child components can know if a certain component is stateful or stateless, and they shouldn't care whether it is defined as a function or a class.
  - This is why state is often called local or encapsulated. It is not accessible to any component other than the one that owns and sets it.
  - A component may choose to pass its state down as props to its child components:

```
<h2>It is {this.state.date.toLocaleTimeString()}.</h2>
```

  - This also works for user-defined components:

```
<FormattedDate date={this.state.date} />
```

  - The FormattedDate component would receive the date in its props and wouldn't know whether it came from the Clock's state, from the Clock's props, or was typed by hand:

```
function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}.</h2>;
}
```

State and Lifecycle

- **The Data Flows Down**
  - To show that all components are truly isolated, we can create an App component that renders three <Clock>s:

```
function App() {
 return (
   <div>
     <Clock />
     <Clock />
     <Clock />
   </div>
 );
}
ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Hello, world!

It is 下午7:29:36.

Hello, world!

It is 下午7:29:36.

Hello, world!

It is 下午7:29:36.

Demo 9

Handling Events

- Handling events with React elements is very similar to handling 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>
```

  - is slightly different in React:

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

Handling Events

- Another difference is that you cannot return false to prevent default behavior in React.
  - You must call preventDefault explicitly.
  - For example, with plain HTML, to prevent the default link behavior of opening a new page, you can write:

```
<a href="#"
   onclick="console.log('The link was clicked.');
            return false">
   Click me
</a>
```

  - In React, this could instead be:

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

Handling Events

- When you define a component using an ES6 class, a common pattern is for an event handler to be a method on the class.
- For example, this Toggle component renders a button that lets the user toggle between “ON” and “OFF” states:

```
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') );
```

Demo 10

Thinking in React

- **Start With A Mock**

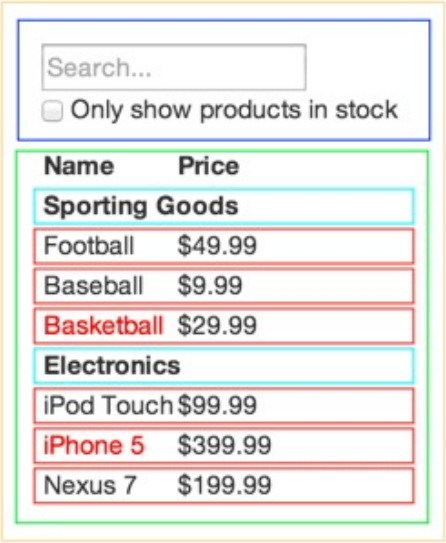
```
[
  {category: "Sporting Goods", price: "$49.99", stocked: true, name: "Football"},
  {category: "Sporting Goods", price: "$9.99", stocked: true, name: "Baseball"},
  {category: "Sporting Goods", price: "$29.99", stocked: false, name: "Basketball"},
  {category: "Electronics", price: "$99.99", stocked: true, name: "iPod Touch"},
  {category: "Electronics", price: "$399.99", stocked: false, name: "iPhone 5"},
  {category: "Electronics", price: "$199.99", stocked: true, name: "Nexus 7"}
];
```

Search...

☐ Only show products in stock

Name	Price
Sporting Goods	
Football	$49.99
Baseball	$9.99
Basketball	$29.99
Electronics	
iPod Touch	$99.99
iPhone 5	$399.99
Nexus 7	$199.99

- **Step 1: Break The UI Into A Component Hierarchy**
  - Since you're often displaying a JSON data model to a user, you'll find that if your model was built correctly, your UI (and therefore your component structure) will map nicely.
  - **FilterableProductTable (orange):** contai ns the entirety of the example
  - **SearchBar (blue):** receives all *user input*
  - **ProductTable (green):** displays and filters the *data collection* based on *user input*
  - **ProductCategoryRow (turquoise):** displa ys a heading for each *category*
  - **ProductRow (red):** displays a row for each *product*



Thinking in React

- **Step 2: Build A Static Version in React**
  - To build a static version of your app that renders your data model, you'll want to build components that reuse other components and pass data using *props*.
  - *props* are a way of passing data from parent to child.
  - If you're familiar with the concept of *state*, **don't use state at all** to build this static version.

```
class ProductCategoryRow extends React.Component {
  render() {
    const category = this.props.category;
    return (
      <tr>
        <th colSpan="2">
          {category}
        </th>
      </tr>
    );
  }
}
```

Demo 11

- **Step 2: Build A Static Version in React**

```
class ProductRow extends React.Component {
  render() {
    const product = this.props.product;
    const name = product.stocked ?      product.name :
      <span style={{color: 'red’}}>
        {product.name}
      </span>;
    return (
      <tr>
        <td>{name}</td>
        <td>{product.price}</td>
      </tr>
    );
  }
}
```

Thinking in React

- **Step 2: Build A Static Version in React**

```
class ProductTable extends React.Component {
  render() {
    const rows = [];
    let lastCategory = null;
    this.props.products.forEach((product) => {
      if (product.category !== lastCategory) {
        rows.push(
          <ProductCategoryRow
           category={product.category}
           key={product.category} />
        );
      }
      rows.push(
        <ProductRow
          product={product}
          key={product.name} />
      );
     lastCategory = product.category;
    });
    return (
      <table>
        <thead>
          <tr>
            <th>Name</th>
            <th>Price</th>
          </tr>
        </thead>
        <tbody>{rows}</tbody>
      </table>
    );
  }
}
```

- **Step 2: Build A Static Version in React**

```
class SearchBar extends React.Component {
  render() {
    return (
      <form>
        <input type="text" placeholder="Search..." />
        <p>
          <input type="checkbox" />
          {' '}
          Only show products in stock
        </p>
      </form>
    );
  }
}
```

Thinking in React

- **Step 2: Build A Static Version in React**

```
class FilterableProductTable extends React.Component {
  render() {
    return (
      <div>
        <SearchBar />
        <ProductTable products={this.props.products} />
      </div>
    );
  }
}

const PRODUCTS = [
  {category: 'Sporting Goods', price: '$49.99', stocked: true, name: 'Football'},
  {category: 'Sporting Goods', price: '$9.99', stocked: true, name: 'Baseball’},
  {category: 'Sporting Goods', price: '$29.99', stocked: false, name: 'Basketball’},
  {category: 'Electronics', price: '$99.99', stocked: true, name: 'iPod Touch’},
  {category: 'Electronics', price: '$399.99', stocked: false, name: 'iPhone 5’},
  {category: 'Electronics', price: '$199.99', stocked: true, name: 'Nexus 7’}
];

ReactDOM.render(
  <FilterableProductTable products={PRODUCTS} />,
  document.getElementById('container’)
);
```

- **Step 3: Identify The Minimal (but complete) Representation Of UI State**
  - Think of all of the pieces of data in our example application. We have:
    - The original list of products
    - The search text the user has entered
    - The value of the checkbox
    - The filtered list of products
  - Let's go through each one and figure out which one is state. Simply ask three questions about each piece of data:
    - Is it passed in from a parent via props? If so, it probably isn't state.
    - Does it remain unchanged over time? If so, it probably isn't state.
    - Can you compute it based on any other state or props in your component? If so, it isn't state.
  - So finally, our state is:
    - The search text the user has entered
    - The value of the checkbox

- **Step 4: Identify Where Your State Should Live**
  - Remember: React is all about one-way data flow down the component hierarchy. It may not be immediately clear which component should own what state.
  - For each piece of state in your application:
    - Identify every component that renders something based on that state.
    - Find a common owner component (a single component above all the components that need the state in the hierarchy).
    - Either the common owner or another component higher up in the hierarchy should own the state.
    - If you can't find a component where it makes sense to own the state, create a new component simply for holding the state and add it somewhere in the hierarchy above the common owner component.

- **Step 4: Identify Where Your State Should Live**
  - Let's run through this strategy for our application:
    - ProductTable needs to filter the product list based on state and SearchBar needs to display the search text and checked state.
    - The common owner component is FilterableProductTable.
    - It conceptually makes sense for the filter text and checked value to live in FilterableProductTable
  - Cool, so we've decided that our state lives in FilterableProductTable.
    - First, add an instance property this.state = {filterText: '', inStockOnly: false} to FilterableProductTable's constructor to reflect the initial state of your application.
    - Then, pass filterText and inStockOnly to ProductTable and SearchBar as a prop.
    - Finally, use these props to filter the rows in ProductTable and set the values of the form fields in SearchBar.

- **Step 4: Identify Where Your State Should Live**

```
class ProductTable extends React.Component {
  ……
  render() {
    const filterText = this.props.filterText;
    const inStockOnly = this.props.inStockOnly;
    ……
    this.props.products.forEach((product) => {
      if (product.name.indexOf(filterText) === -1)
      {
        return;
      }
      if (inStockOnly && !product.stocked)
      {
        return;
      }
      ……
```

- **Step 4: Identify Where Your State Should Live**

```
class SearchBar extends React.Component {
  render() {
    const filterText = this.props.filterText;
    const inStockOnly = this.props.inStockOnly;
    return (
      <form>
        <input
          type="text"
          placeholder="Search..."
          value={filterText} />
        <p>
          <input
            type="checkbox"
            checked={inStockOnly} />
          {' '}
          Only show products in stock
        </p>
      </form>
    );
  }
}
```

Thinking in React

- **Step 4: Identify Where Your State Should Live**

```
class FilterableProductTable extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      filterText: '',
      inStockOnly: false
    };
  }
  render() {
    return (
      <div>
        <SearchBar
          filterText={this.state.filterText}
          inStockOnly={this.state.inStockOnly}
        />
        <ProductTable
          products={this.props.products}
          filterText={this.state.filterText}
          inStockOnly={this.state.inStockOnly}
        />
      </div>
    );
  }
}
```

- **Step 5: Add Inverse Data Flow**
- If you try to type or check the box in the current version of the example, you'll see that React ignores your input.

- Let's think about what we want to happen.
  - We want to make sure that whenever the user changes the form, we update the state to reflect the user input.
  - Since components should only update their own state, FilterableProductTable will pass callbacks to SearchBar that will fire whenever the state should be updated.
  - We can use the onChange event on the inputs to be notified of it. The callbacks passed by FilterableProductTable will call setState(), and the app will be updated.

- **Step 5: Add Inverse Data Flow**

```
class SearchBar extends React.Component {
  constructor(props) {
    super(props);
    this.handleFilterTextChange =
         this.handleFilterTextChange.bind(this);
    this.handleInStockChange =
         this.handleInStockChange.bind(this);
  }
  handleFilterTextChange(e) {
    this.props.onFilterTextChange(e.target.value);
  }
  handleInStockChange(e) {
    this.props.onInStockChange(e.target.checked);
  }
```

```
render() {
    return (
      <form>
        <input
          type="text”
          placeholder="Search..."
          value={this.props.filterText}
          onChange={this.handleFilterTextChange}
        />
        <p>
          <input
            type="checkbox”
            checked={this.props.inStockOnly}
            onChange={this.handleInStockChange}
          />
          {' ‘}
          Only show products in stock
        </p>
      </form>
    );
  }
}
```

Thinking in React

- **Step 5: Add Inverse Data Flow**

```
class FilterableProductTable extends React.Component {
  constructor(props) {
    ……
    this.handleFilterTextChange =
         this.handleFilterTextChange.bind(this);
    this.handleInStockChange =
         this.handleInStockChange.bind(this);
  }
  handleFilterTextChange(filterText) {
    this.setState({      filterText: filterText    });
  }
  handleInStockChange(inStockOnly) {
    this.setState({      inStockOnly: inStockOnly    })
  }

  render() {
    return (
      <div>
        <SearchBar
          filterText={this.state.filterText}
          inStockOnly={this.state.inStockOnly}
          onFilterTextChange={this.handleFilterTextChange}
          onInStockChange={this.handleInStockChange}
        />
  ……
```

Demo 12

- *Web开发技术*
- *Web Application Development*

Thank You!