# React.js

FSMVÜ Computer Eng. Dr. Öğr. Üyesi Ali NİZAM

# React JS



- Component based js framework the build on encapsulated components that manage their own state, then compose them to make complex UIs

# Environment Setup



- The name npm (Node Package Manager) stems from when npm first was created as a package manager for Node.js.
- All npm packages are defined in files called package.json.
- The content of package.json must be written in JSON.
- At least two fields must be present in the definition file: name and version.

# Installing



- npm init
- npm install --save browserify babelify
- npm install --save react react-dom
- npm install -g create-react-app
- Or

- npx create-react-app my-app

- npm is a tool mainly used to install packages.
- npx is a tool to execute packages.
- What does it means? That if you want to execute a package without installing it on your computer and then launch it you can use npx directly.
- Speaking of react , create-react-app is an utility to bootstrap a react project: if you use it with npx ( npx create-react-app my-app ) you will have your my-app project in place without the need to install create-react-app itself ( which will need another passage: npm install create-react-app and then create-react-app my-app) ^^

https://www.freecodecamp.org/forum/t/difference-between-npm-and-npx-in-react/265124

# Start application



- C:\Users\Your Name\myfirstreact>npm start

# npm



- Software Package Manager
- The name **npm** (Node Package Manager) stems from when npm first was created as a package manager for Node.js.
- All **npm** packages are defined in files called **package.json**.
- The content of package.json must be written in **JSON**.
- At least two fields must be present in the definition file: **name** and **version**.

# Managing Dependencies



- npm can manage dependencies.
- npm can (in one command line) install all the dependencies of a project.
- Dependencies are also defined in package.json.

# Component Design



- Your components must always be capitalized
- Inside render , you output the JSX you want to return
- To evaluate an expression, wrap it inside { }
- You can call a component directly by wrapping its name in < and > characters

# JSX



- In order to make its component API easier to use, React has its own syntax called JSX, which combines JavaScript and HTML. Let's take a look at updating our sample code to use JSX by copying the following code into hello-react.html and refreshing our browser:

# Babeljs.io



- Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments. Here are the main things Babel can do for you:
  - Transform syntax
  - Polyfill features that are missing in your target environment (through @babel/polyfill)
  - Source code transformations (codemods)
  - And more! (check out these videos for inspiration)
- Babel can convert JSX syntax! Check out our React preset to get started. Use it together with the babel-sublime package to bring syntax highlighting to a whole new level.

# React DOM Render



- The method ReactDom.render() is used to render (display) HTML elements:

- Example
- Hello World!

- <script type="text/babel">
- ReactDOM.render(
- <h1>Hello React!</h1>,
- document.getElementById(**'id01'**));
- </script>

# Basic Component



```
class App extends React.Component {
    render() {
        return (
            <h1>Hello Word</h1>
        )
    }
}
```

# Basic Example



```
import React from 'react'
import ReactDOM from 'react-dom';
import App from './js/components/App';

// Render component into the DOM - only once per app
ReactDOM.render(
    <App />,
    document.getElementById('root')
);
```

# Props



- Props are arguments passed into React components.
- Props are passed to components via HTML attributes.
- React Props are read-only! You will get an error if you try to change their value.

# Basic Example



```
class NameText extends React.Component {
  constructor(props){
     super(props);

  }

  render() {
     return (
       <div>
         <h1>Hello </h1>
         <div> {this.props.isim} </div>
       </div>
     )
    }
}
```

```
class App extends React.Component {
    render() {
      return (
        <div>
         <NameText isim="Ahmet Ak"/>
        </div>
      )
    }
}

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

# Basic Example



```
class Card extends React.Component {
  render(){
    return (
      <div className="card">
        <p>{this.props.name}</p>
      </div>
    )
  }
}
```

```
class App extends React.Component {

    // fires before component is mounted
    constructor(props) {

        // makes this refer to this component
        super(props);

        // set local state
        this.state = {
            name: "Michael"
        };

    }
```

# Props new way



app/components/Note.jsx

```
import React from 'react';
export default ({task}) => <div>{task}</div>;
```

- Remember that this declaration is equivalent to:

```
import React from 'react';
export default (props) => <div>{props.task}</div>;
```

# React Prop Example



```
import React from 'react';
import ReactDOM from 'react-dom';

class Car extends React.Component {
  render() {
    return <h2>I am a {this.props.brand}!</h2>
  }
}

const myelement = <Car brand="Ford" />;

ReactDOM.render(myelement, document.getElementById('root'));
```

https://www.w3schools.com/react/showreact.asp?filename=demo2_react_props

# Function Type Dialog Example



```
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!" />

  );
}
```

# Sending a variable



If you have a variable to send, and not a string as in the example above, you just put the variable name inside curly brackets

```
class Car extends React.Component {
  render() {     return <h2>I am a {this.props.brand}!</h2>;
    }
}

class Garage extends React.Component {
  render() {
    const carname = "Ford";
    return (
      <div>
      <h1>Who lives in my garage?</h1>
      <Car brand={carname} />
      </div>
    );
  }
}

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

# Behavior



```
class Buttonify extends React.Component {
  render () {
   return (
    <div>
     <button type={this.props.behavior}></button>
    </div>
    );
  }
 }
```

```
ReactDOM.render(
<div>
<Buttonify behavior="submit">DO
SOMETHING GREAT!</Buttonify>
</div>,
destination
);
```

# Children



- this.props.children that represent chield elemens and can have one element, multiple elements, or none at all, its value is respectively a single child node, an array of child nodes or undefined
- Another definition :this.props.children does is that it is used to display whatever you include between the opening and closing tags when invoking a component.
- Possible usages are:
  - Grouping unknown number of similar elements into a parent element.
  - You don’t know elements ahead of the time.
  - The nested structure that needs a wrapper.

https://medium.com/javascript-in-plain-english/how-to-use-props-children-in-react-7d6ab5836c9d

# this.props.children



- <MovieBrowser>
- <Movie title="Message" />
- <Movie title="World" />
- </MovieBrowser>

# this.props.children



Example: to wrap our components in an extra div with a special class

```
export default class SomeComponent extends React.Component {
  render() {
    const childrenWithWrapperDiv = React.Children.map(this.props.children, child => {
      return (
        <div className="some-component-special-class">{child}</div>
      );
    });

    return (
      <div className="some-component">
        <p>This component has {React.Children.count(this.props.children)} children.</p>
        {childrenWithWrapperDiv}
      </div>
    );
  }
}
```

https://learn.co/lessons/react-this-props-children

# Component Composition



- http://jsfiddle.net/gh/gist/library/pure/9b4f8fe0de2fe7ecee52/

# Parameter passing



```
class Football extends React.Component {
  shoot(a) {
    alert(a);
  }
  render() {
    return (
      <button onClick={this.shoot.bind(this, "Goal")}>Take the shot!</button>
    );
  }
}
```

- Note on the second example: If you send arguments without using the bind method, (this.shoot(this, "Goal") instead of this.shoot.bind(this, "Goal")), the shoot function will be executed when the page is loaded instead of waiting for the button to be clicked.

https://www.w3schools.com/react/react_events.asp

# Map Function



- Rendering multiple element

- we loop through the numbers array using the JavaScript map() function. We return a <li> element for each item. Finally, we assign the resulting array of elements to listItems:

- const numbers = [1, 2, 3, 4, 5];
- const listItems = numbers.map((number) =>
- <li>{number}</li>
- );

https://en.reactjs.org/docs/lists-and-keys.html

# 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.

# Passing Arguments to Event Handlers



- Inside a loop it is common to want to pass an extra parameter to an event handler. For example, if id is the row ID, either of the following would work:

- <button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
- <button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>

# Bind method



- bind allows us to set the function context (first parameter) and arguments (following parameters). This gives us a technique known as partial application.

# bind example



```
class Football extends React.Component {
  constructor(props) {
    super(props)
    this.shoot = this.shoot.bind(this)
  }
  shoot() {
    alert(this);
    /*
    Thanks to the binding in the constructor function,
    the 'this' keyword now refers to the component object
    */
  }
  render() {
    return (
      <button onClick={this.shoot}>Take the shot!</button>
    );
  }
}
ReactDOM.render(<Football />, document.getElementById('root'));
```

# this keyword and bind



- For methods in React, the this keyword should represent the component that owns the method.

- That is why you should use arrow functions. With arrow functions, this will always represent the object that defined the arrow function.

# Event Handler Example



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

- is slightly different in React:

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

# Event handling example



```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Click me
    </a>
  );
}
```

# Component lifecycle events

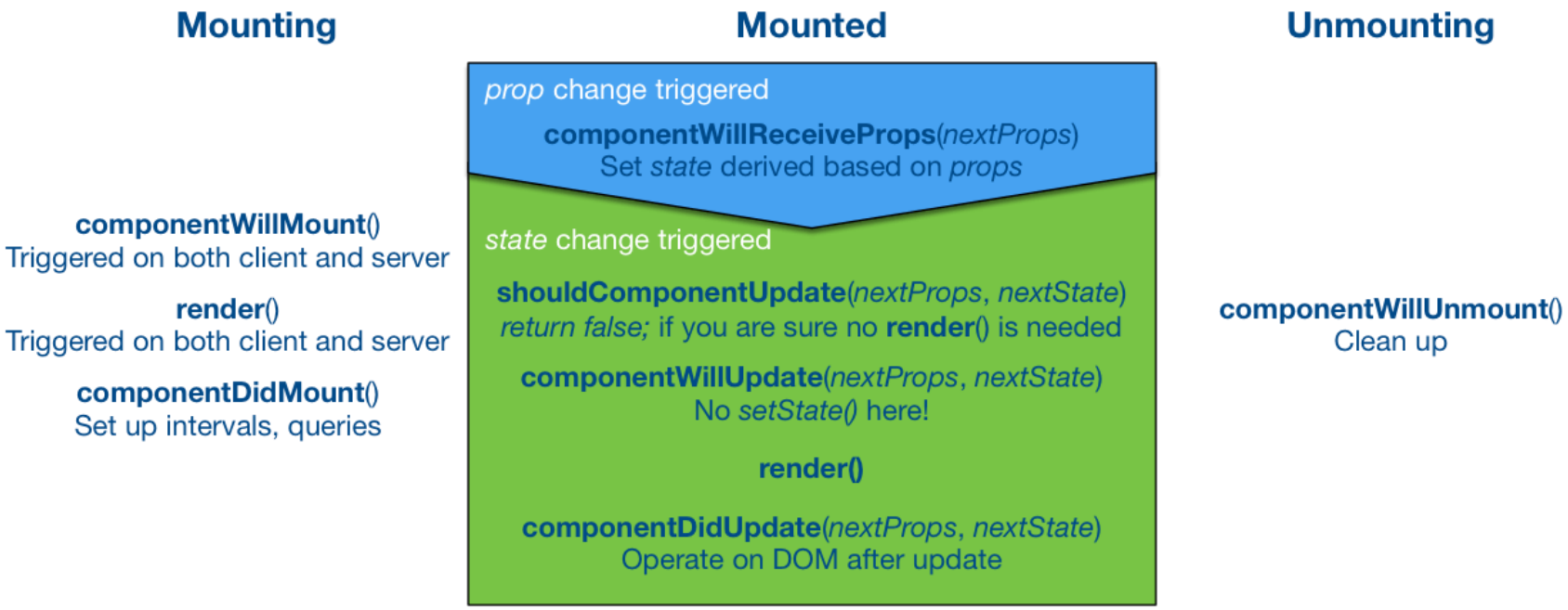


- Component lifecycle events  component in the preceding code:
  - componentWillReceiveProps
  - shouldComponentUpdate
  - componentDidUpdate

- http://jsfiddle.net/gh/gist/library/pure/c709657dc3584cb5962b/

# Process Model






https://survivejs.com/react/introduction/

# Mounting



- This process of creating instances and DOM nodes corresponding to React components, and inserting them into the DOM, is called mounting.
  - let foo = React.createElement(FooComponent);
  - It's currently not anywhere on the page, i.e. it is not a DOM element, doesn't exist anywhere in the DOM tree
  - tell React to "mount" it into a DOM container by calling
  - ReactDOM.render(foo, domContainer);

https://stackoverflow.com/questions/31556450/what-is-mounting-in-react-js

# e.stopPropagation()



- The idea of this is to tell the DOM to stop bubbling events. In short, we'll avoid triggering possible other events elsewhere in the structure if we delete a note.

# State Management



- The reason why is simple. React cannot notice the structure has changed and won't react accordingly (that is, trigger render()). To overcome this issue, we can implement our modification through React's own API. This makes it notice that the structure has changed. As a result it is able to render() as we might expect.

https://survivejs.com/react/getting-started/implementing-notes/

# React Form



- You can control the values of more than one input field by adding a name attribute to each element.
- When you initialize the state in the constructor, use the field names.
- To access the fields in the event handler use the event.target.name and event.target.value syntax.
- To update the state in the this.setState method, use square brackets [bracket notation] around the property name.

# React Form Example



```
class MyForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {     username: ‘’,   age: null };
  }
  myChangeHandler = (event) => {
    let nam = event.target.name;
    let val = event.target.value;
    this.setState({[nam]: val});
  }
  render() {
    return (
      <form>
      <h1>Hello {this.state.username} {this.state.age}</h1>
      <p>Enter your name:</p>    <input     type='text’   name='username’   onChange={this.myChangeHandler}      />
      <p>Enter your age:</p>    <input  type='text’  name='age’  onChange={this.myChangeHandler}  />
      </form>
    );
  }
}
```

# State Management



- Local (Component State)
- Global (Redux)

https://reactjs.org/docs/state-and-lifecycle.html

# The state of a component (local state)



- The state of a component is like the props which are passed to a component, a plain JavaScript object containing information that influences the way a component is rendered.
- In comparison, to the props, the state can be changed by the component itself by calling setState which will trigger a re-render of the component.
- The state API of React is really simple at all and doesn't add too much complexity to your application.
- Besides all other state management solutions, the component state is preferred to not be replaced at all because you should always keep your state as close to where it is needed to avoid unnecessary complexity.
- Because managing the state of an single input in a global state isn't what you are aiming for.

https://medium.com/dailyjs/comparison-of-state-management-solutions-for-react-2161a0b4af7b

# The state of a component



```
class App extends React.Component {

    constructor(props){
        super(props);

        this.state = {
            teams: data
        }
    }

    render(){
        ...
    }

}
```

https://ihatetomatoes.net/react-state-management-tutorial-do-you-really-need-redux-or-mobx/

# access the data from the state



```
render(){
    const {teams} = this.state;
    return(
        <ul>
            {
                teams.map(team => <li key={team.name}>{team.name}</li>)
            }
        </ul>
    )
}
```

# Local State



```
render() {
    return <button onClick={this.inc}>Click to update</button>
  }

  inc() {
    console.log('before: ' + this.state.test);
    this.setState({
      test: this.state.test+1
    });
    console.log('after: ' + this.state.test);
  }
```

Or

```
this.setState((state) => ({ value: state.value + 1}));
```

# Redux

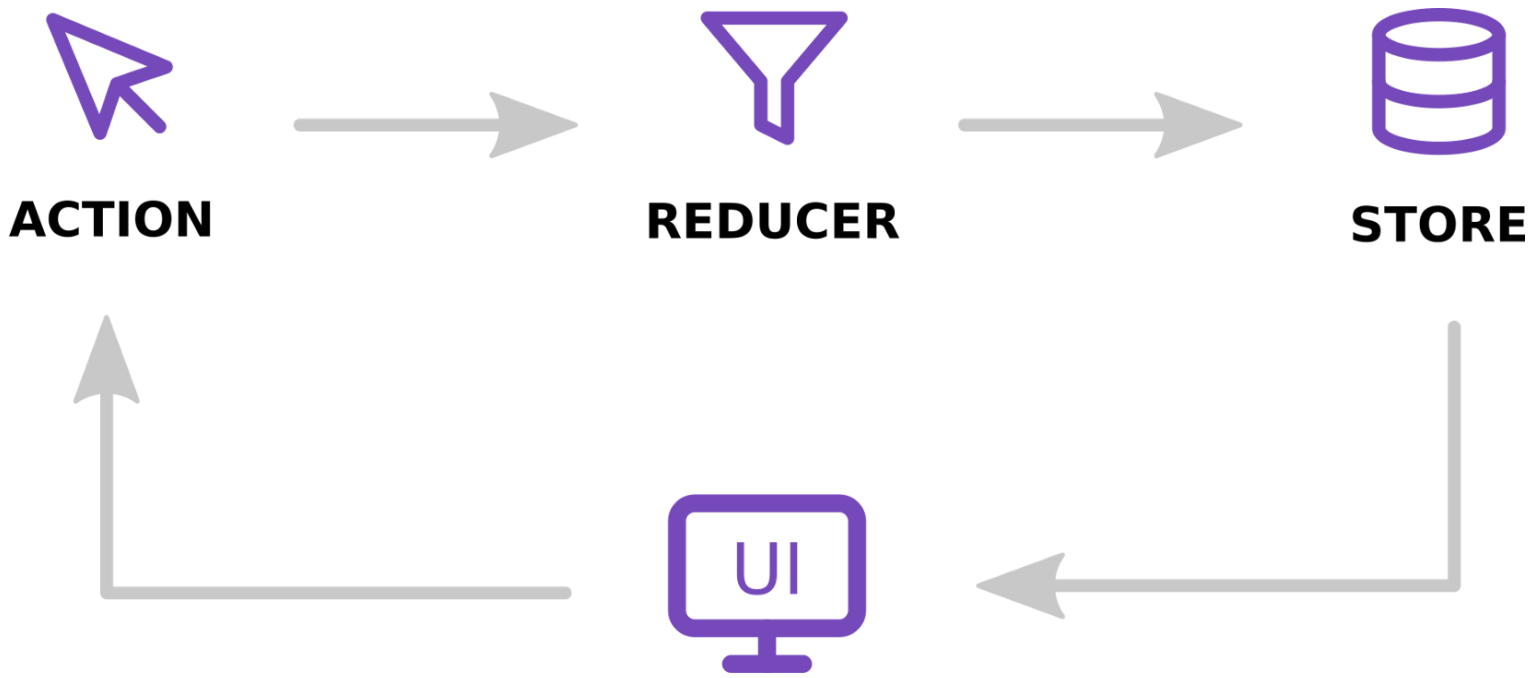


- Redux is a predictable state container for JavaScript apps.

https://redux.js.org/introduction/getting-started

# Redux






https://medium.com/javascript-in-plain-english/the-only-introduction-to-redux-and-react-redux-youll-ever-need-8ce5da9e53c6

# Basic Principles



- Single source of truth - The state for the entire application is in a single store.
- State is read only - The only way to change state is to dispatch an action which will result in the state changing.
- Changes are made with pure functions - These pure functions transform the state and are called reducers.

# Actions



- Actions are payloads of information that send data from your application to your store.
- They are the only source of information for the store.
- You send them to the store using store.dispatch().

```
const ADD_TODO = 'ADD_TODO'
{
  type: ADD_TODO,
  text: 'Build my first Redux app'
}
```

# Action Creators



- Action creators are exactly that—functions that create actions. It's easy to conflate the terms “action” and “action creator”, so do your best to use the proper term.

- In Redux, action creators simply return an action:

```
function addTodo(text) {
  return {
    type: ADD_TODO,
    text
  }
}
```

# initiate a dispatch



- to actually initiate a dispatch, pass the result to the dispatch() function:

- dispatch(addTodo(text))

- Or

- **const** boundAddTodo = text => dispatch(addTodo(text))
- boundAddTodo(text)

# Reducers



- Reducers specify how the application's state changes in response to actions sent to the store.
- Remember that actions only describe what happened, but don't describe how the application's state changes.
- Note that a reducer is a pure function. It only computes the next state. It should be completely predictable: calling it with the same inputs many times should produce the same outputs. It shouldn't perform any side effects like API calls or router transitions. These should happen before an action is dispatched.

# Handling Actions



- Now that we've decided what our state object looks like, we're ready to write a reducer for it. The reducer is a pure function that takes the previous state and an action, and returns the next state.

- (previousState, action) => newState

# It's very important that the reducer stays pure



- It's very important that the reducer stays pure. Things you should never do inside a reducer:
  - Mutate its arguments;
  - Perform side effects like API calls and routing transitions;
  - Call non-pure functions, e.g. Date.now() or Math.random().

# Handle State



```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

# Or Object Spread operator



- spread (...) operator to copy enumerable properties from one object to another in a more succinct way.

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return { ...state, visibilityFilter: action.filter }
    default:
      return state
  }
}
```

# Basic example



```
import {createStore} from "redux";
const reducer=(state,action)=>{
    switch(action.type){
        case "ADD":
            state=state+action.payload;
            break;

    }
    return state;
}
const store =createStore(reducer,1);


store.subscribe(()=>{console.log(store.getState())})
store.dispatch(
    {
        type:"ADD",
        payload:10
    }
)
```

# Split Reducers



- Note that each of these reducers is managing its own part of the global state. The state parameter is different for every reducer, and corresponds to the part of the state it manages.

- This is already looking good! When the app is larger, we can split the reducers into separate files and keep them completely independent and managing different data domains.

# combineReducers



- Redux provides a utility called combineReducers() that does the same boilerplate logic that the todoApp above currently does

# combineReducers



```
import { combineReducers } from 'redux'

const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp
```

Note that this is equivalent to:

```
export default function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  }
}
```

# The Store



- The Store is the object that brings them together. The store has the following responsibilities:

- Holds application state;
- Allows access to state via getState();
- Allows state to be updated via dispatch(action);
- Registers listeners via subscribe(listener);
- Handles unregistering of listeners via the function returned by subscribe(listener).

# createStore



- **import** { createStore } **from** 'redux’
- **import** todoApp **from** './reducers’
- **const** store = createStore(todoApp)

# Data Flow



- You call store.dispatch(action).
- The Redux store calls the reducer function you gave it
- The root reducer may combine the output of multiple reducers into a single state tree.
- The Redux store saves the complete state tree returned by the root reducer.

https://redux.js.org/basics/data-flow

# store.dispatch(action).



- { type: 'LIKE_ARTICLE', articleId: 42 }
- { type: 'FETCH_USER_SUCCESS', response: { id: 3, name: 'Mary' } }
- { type: 'ADD_TODO', text: 'Read the Redux docs.' }

# The Redux store calls the reducer function you gave it.



- The store will pass two arguments to the reducer: the current state tree and the action. For example, in the todo app, the root reducer might receive something like this:
- a reducer is a pure function. It only computes the next state. It should be completely predictable: calling it with the same inputs many times should produce the same outputs. It shouldn't perform any side effects like API calls or router transitions. These should happen before an action is dispatched.

# React and Redux



- React let you describe UI as a function of state, and Redux emits state updates in response to actions.
- npm install --save react-redux

# Presentational and Container Components



| | Presentational Components | Container Components |
|---|---|---|
| **Purpose** | How things look (markup, styles) | How things work (data fetching, state updates) |
| **Aware of Redux** | No | Yes |
| **To read data** | Read data from props | Subscribe to Redux state |
| **To change data** | Invoke callbacks from props | Dispatch Redux actions |
| **Are written** | By hand | Usually generated by React Redux |

# Implementing Presentational Components



- These are all normal React components, so we won't examine them in detail.
- We write functional stateless components unless we need to use local state or the lifecycle methods.
- This doesn't mean that presentational components have to be functions—it's just easier to define them this way.
- If and when you need to **add local state**, lifecycle methods, or performance optimizations, you can convert them to classes.

# React Redux : Implementing Container Components



- Technically, a container component is just a React component that uses store.subscribe() to read a part of the Redux state tree and supply props to a presentational component it renders.
- generating container components with the React Redux library's connect() which provides many useful optimizations to prevent unnecessary re-renders

# Provider



- <Provider> to magically make the store available to all container components in the application without passing it explicitly. You only need to use it once when you render the root component:
- const store = createStore(todoApp)

- render(
- <Provider store={store}>
- <App />
- </Provider>,
- document.getElementById('root')
- )

# connect



- React Redux provides a connect function. We use this function whenever we want to “connect” a component to Redux, i.e. make a React component interact with our Redux store.
- Note: We almost never access the store directly. We just fire actions creators and all of the logic later is handled automatically by the connect function.
- It also takes in two optional arguments: *mapStateToProps* and *mapDispatchToProps*.

https://medium.com/javascript-in-plain-english/the-only-introduction-to-redux-and-react-redux-youll-ever-need-8ce5da9e53c6

# connect



```
import { connect } from 'react-redux'

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

# mapStateToProps



- describes how to transform the current Redux store state into the props you want to pass to a presentational component you are wrapping.

```
const mapStateToProps = state => {
  return {
    todos: getVisibleTodos(state.todos,
state.visibilityFilter)
  }
}
```

# mapDispatchToProps()



- mapDispatchToProps() that receives the dispatch() method and returns callback props that you want to inject into the presentational component.

```
const mapDispatchToProps = dispatch => {
  return {
    onTodoClick: id => {
      dispatch(toggleTodo(id))
    }
  }
}
```

# **Additional info**: Javascript bind is a technique used in the state management



```
this.x = 9;    // this refers to global "window" object here in the browser
var module = {
  x: 81,
  getX: function() { return this.x; }
};

module.getX(); // 81

var retrieveX = module.getX;
retrieveX();
// returns 9 - The function gets invoked at the global scope

// Create a new function with 'this' bound to module
// New programmers might confuse the
// global var x with module's property x
var boundGetX = retrieveX.bind(module);
boundGetX(); // 81
```

# Creating a bound function



- The simplest use of bind() is to make a function that, no matter how it is called, is called with a particular this value.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

# JavaScript | Spread Operator (…)



- Spread operator allows an iterable to expand in places where 0+ arguments are expected. It is mostly used in variable array where there is more than 1 values are expected.It allows us the privilege to obtain a list of parameters from an array. Syntax of Spread operator is same as Rest parameter but it works completely opposite of it.

# Call restfull API from React



```
import React, { Component } from 'react'

  class App extends Component {
    ...
    componentDidMount() {
      fetch('http://jsonplaceholder.typicode.com/users')
      .then(res => res.json())
      .then((data) => {
        this.setState({ contacts: data })
      })
      .catch(console.log)
    }
    ...
  }
```

https://pusher.com/tutorials/consume-restful-api-react

# Call restfull API from React



- fetch('http://jsonplaceholder.typicode.com/users') will make a GET request to the endpoint
- .then(res => res.json()) parses the output to JSON,
- .then((data) => {this.setState({ contacts: data })}) sets the value of our state to the output from the
- API call and finally .catch(console.log) logs any error we get to the console.

# Using bootstrap in React



- Use  bootstrap min

  npm install bootstrap

  - In component class

    import 'bootstrap/dist/css/bootstrap.min.css’;

- To use Bootstrap’s JavaScript components in your app

  npm install jquery popper.js

  - In component class

    import 'bootstrap/dist/css/bootstrap.min.css';
    import $ from 'jquery';
    import Popper from 'popper.js';
    import 'bootstrap/dist/js/bootstrap.bundle.min';

https://blog.logrocket.com/how-to-use-bootstrap-with-react-a354715d1121/

# Lab



- http://www.darrenbeck.co.uk/nodejs/react/reacttutorial-part1/
- https://pusher.com/tutorials/consume-restful-api-react