REACT – The complete guide - Maximilian

React is a library for creating highly reactive and superfast js based web applications.

JS runs in the browser.

React is a JS library for building user interfaces

React is writing custom HTML elements

React is all about writing components

Codepen.io – to create a workspace for writing html/css/js code

ReactDOM is for rendering the components to real DOM

In its basic form, a React component is just a function.

The syntax React uses is called jsx.

Html --

<div id="p1"> </div>

<div class="person">

<h1> Hello </h1>

<p> age: 29 </o>

</div>

<div id='p2'> <h3> Romeo </h3> </div>

--

Css –

.person {

color: red

}

---

Js –

function Person(){

return (

<div class=”person”>

<h1> Max </h1>

<p> Your age: 28 </p>

</div>

);

}

ReactDOM.render(<Person />, document.querySelector('#p1'));

Output:

# Max

Your age: 28

# Hello

age: 29

### Romeo

Html –

<div id="p1"> </div>

<div class="person">

<h1> Hello </h1>

<p> age: 29 </o>

</div>

<div id='p2'> <h3> Romeo </h3> </div>

--

Css --

.person {

color: red

}

.person1 {

color: green

}

--

function Person(props){

return (

<div class="person1" >

<h1> {props.name} </h1>

<p> {props.age} </p>

</div>

);

}

ReactDOM.render(<Person name="Jaison" age="45" />, document.querySelector('#p1'));

---

# Jaison

45

# Hello

age: 29

### Romeo

Html –

<div id='app'</div>

--

Css –

.person {

color: red

}

.person1 {

color: orange

}

---

function Person(props){

return (

<div class="person1" >

<h1> {props.name} </h1>

<p> {props.age} </p>

</div>

);

}

var app = (<div> <Person name="Jaison" age="45" /><Person name="Sonia" age="39" /> </div>)

ReactDOM.render(app, document.querySelector('#app'));

---

Output:

# Jaison

45

# Sonia

39

In 3 ways, the component is written. 1) by function and render. 2) by function with parameter and render with argument. 3) create function with parameter and define the property, declare variable with function property definition, render with variable. With this approach, can create single page applications.

Angular an Vue are alternatives to React. JQuery is more about traversing the DOM and targeting the DOM elements.

Single page and multi page applications.

In a single page application, the root component manages the entire page.

Redux is a third party library

Jsbin.com is a web editor

When ‘this’ keyword is used with arrow functions, it always keeps its context.

Classes are blueprints for js objects. Properties are like variables attached to objects, methods are like functions attached to objects.

‘spread’ operator is used to split up array elements or object properties.

‘rest’ is used to merge a list of function arguments into an array

‘destructuring’ is extract array elements or object properties and store them in variables.

‘jsx’ is html in js syntax

Dependency management tool: npm or yarn. Bundler is Webpack. Native js compiler is Babel

Npm install –g create-react-app

Create-react-app my-app –scripts-version 1.1.5

Cd my-app // npm start

The nodes-module holds all the dependencies and sub-dependences of the project

The ‘public’ folder is surfed by the web server

Manifest.json is the metadata about the application

React.createElement(‘h1’,{ object} / null,’Hello’)

--

import React, { Component } from 'react';

import './App.css';

class App extends Component {

render() {

return (

// <div className="App">

// <h1> Hi, I am a React App!!</h1>

// </div>

React.createElement('div',{className:'App'},React.createElement('h1',null,"From React"))

);

}

}

export default App;

---

In JSX, use className. Class is reserved in JS

JSX elements are not HTML. React library provides it.

A component is just a function that returns a jsx (html)

Components & JSX Cheat Sheet

Components are the **core building block of React apps**. Actually, React really is just a library for creating components in its core.

A typical React app therefore could be depicted as a **component tree** - having one root component ("App") and then an potentially infinite amount of nested child components.

Each component needs to return/ render some **JSX** code - it defines which HTML code React should render to the real DOM in the end.

**JSX is NOT HTML** but it looks a lot like it. Differences can be seen when looking closely though (for example className in JSX vs class in "normal HTML"). JSX is just syntactic sugar for JavaScript, allowing you to write HTMLish code instead of nested React.createElement(...) calls.

When creating components, you have the choice between**two different ways:**

1. **Functional components** (also referred to as "presentational", "dumb" or "stateless" components - more about this later in the course) => const cmp = () => { return <div>some JSX</div> } (using ES6 arrow functions as shown here is recommended but optional)
2. **class-based components** (also referred to as "containers", "smart" or "stateful" components) => class Cmp extends Component { render () { return <div>some JSX</div> } }

We'll of course dive into the difference throughout this course, you can already note that you should use 1) as often as possible though. It's the best-practice.

Children refers to any element between the opening and closing tags of the component

‘porps’ are object giving access to all the attributes given in the component.

‘state’ is a property defined in a class which extends Component. ‘state’ is managed from inside a component. ‘state’ is a js object.

Props & State

props  and state  are **CORE concepts** of React. Actually, only changes in props  and/ or state  trigger React to re-render your components and potentially update the DOM in the browser (a detailed look at how React checks whether to really touch the real DOM is provided in section 6).

**Props**

props  allow you to pass data from a parent (wrapping) component to a child (embedded) component.

**Example:**

**AllPosts Component:**

1. const posts = () => {
2. return (
3. <div>
4. <Post title="My first Post" />
5. </div>
6. );
7. }

Here, title  is the custom property (prop ) set up on the custom Post  component. We basically replicate the default HTML attribute behavior we already know (e.g. <input type="text">  informs the browser about how to handle that input).

**Post Component:**

1. const post = (props) => {
2. return (
3. <div>
4. <h1>{props.title}</h1>
5. </div>
6. );
7. }

The Post  component receives the props  argument. You can of course name this argument whatever you want - it's your function definition, React doesn't care! But React will pass one argument to your component function => An object, which contains all properties you set up on <Post ... /> .

{props.title}  then dynamically outputs the title  property of the props  object - which is available since we set the title  property inside AllPosts  component (see above).

**State**

Whilst props allow you to pass data down the component tree (and hence trigger an UI update), state is used to change the component, well, state from within. Changes to state also trigger an UI update.

**Example:**

**NewPost Component:**

1. class NewPost extends Component { // state can only be accessed in class-based components!
2. state = {
3. counter: 1
4. };
6. render () { // Needs to be implemented in class-based components! Needs to return some JSX!
7. return (
8. <div>{this.state.counter}</div>
9. );
10. }
11. }

Here, the NewPost  component contains state . Only class-based components can define and use state . You can of course pass the state  down to functional components, but these then can't directly edit it.

state  simply is a property of the component class, you have to call it state  though - the name is not optional. You can then access it via this.state  in your class JSX code (which you return in the required render()  method).

Whenever state  changes (taught over the next lectures), the component will re-render and reflect the new state. The difference to props  is, that this happens within one and the same component - you don't receive new data (props ) from outside!

To Which Events Can You Listen?

In the last lecture, we saw that you can react to the onClick event - but to which other events can you listen? You can find a list of supported events here: <https://reactjs.org/docs/events.html#supported-events>

#### **Clipboard Events**

Event names:

1. onCopy onCut onPaste

Properties:

1. DOMDataTransfer clipboardData

Composition Events

Event names:

1. onCompositionEnd onCompositionStart onCompositionUpdate

Properties:

1. string data

Keyboard Events

Event names:

1. onKeyDown onKeyPress onKeyUp

Properties:

1. boolean altKey
2. number charCode
3. boolean ctrlKey
4. boolean getModifierState(key)
5. string key
6. number keyCode
7. string locale
8. number location
9. boolean metaKey
10. boolean repeat
11. boolean shiftKey
12. number which

Focus Events

Event names:

1. onFocus onBlur

These focus events work on all elements in the React DOM, not just form elements.

Properties:

1. DOMEventTarget relatedTarget

Form Events

Event names:

1. onChange onInput onInvalid onSubmit

For more information about the onChange event, see [Forms](https://reactjs.org/docs/forms.html).

Mouse Events

Event names:

1. onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit
2. onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave
3. onMouseMove onMouseOut onMouseOver onMouseUp

The onMouseEnter and onMouseLeave events propagate from the element being left to the one being entered instead of ordinary bubbling and do not have a capture phase.

Properties:

1. boolean altKey
2. number button
3. number buttons
4. number clientX
5. number clientY
6. boolean ctrlKey
7. boolean getModifierState(key)
8. boolean metaKey
9. number pageX
10. number pageY
11. DOMEventTarget relatedTarget
12. number screenX
13. number screenY
14. boolean shiftKey

Selection Events

Event names:

1. onSelect

Touch Events

Event names:

1. onTouchCancel onTouchEnd onTouchMove onTouchStart

Properties:

1. boolean altKey
2. DOMTouchList changedTouches
3. boolean ctrlKey
4. boolean getModifierState(key)
5. boolean metaKey
6. boolean shiftKey
7. DOMTouchList targetTouches
8. DOMTouchList touches

UI Events

Event names:

1. onScroll

Properties:

1. number detail
2. DOMAbstractView view

Wheel Events

Event names:

1. onWheel

Properties:

1. number deltaMode
2. number deltaX
3. number deltaY
4. number deltaZ

Media Events

Event names:

1. onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted
2. onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay
3. onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend
4. onTimeUpdate onVolumeChange onWaiting

Image Events

Event names:

1. onLoad onError

Animation Events

Event names:

1. onAnimationStart onAnimationEnd onAnimationIteration

Properties:

1. string animationName
2. string pseudoElement
3. float elapsedTime

Transition Events

Event names:

1. onTransitionEnd

Properties:

1. string propertyName
2. string pseudoElement
3. float elapsedTime

Other Events

Event names:

1. onToggle

----

This.getState({}) – applicable only in class –state

From React 16.8, react ‘hooks’ can be used to manage ‘state’ from functions

React hooks are a collection of functions which can be used inside functional components.

‘useState’ is the most important React hook.

‘stateful’ components are called smart or container components. Others are stateless presentational or dumb components.

2 ways of styling: .css file, inline styling.

Ek.js

import React from 'react'

const ek = () => {

return <p> Hello EK</p>

}

export default ek

--

App.js

Import Ek from ‘./Person/Ek’

----

Person.js

//props

import React from 'react'

const person = (props) =>{

return <p> Hello I am {props.name} {props.age} years old {props.children} </p>

}

export default person

---

App.js

import React, { Component } from 'react';

import './App.css';

import Person from './Person/Person';

import Ek from './Person/Ek'

class App extends Component {

render() {

return (

<div className="App">

<h1> Hi, I am a React App!!</h1>

<Person name= "Jaison" age = "45"/>

<Person name="Sonia" age= "39"> Facebook </Person>

<Ek />

</div>

// React.createElement('div',{className:'App'},React.createElement('h1',null,"From React"))

);

}

}

export default App;

---

//state App.js

import React, { Component } from 'react';

import './App.css';

import Person from './Person/Person';

import Ek from './Person/Ek'

class App extends Component {

state = {

persons: [

{name:'Max',age:28},

{name:'Manu', age:29},

{name:'Stephanie',age:27}

]

}

render() {

return (

<div className="App">

<h1> Hi, I am a React App!!</h1>

<Person name= {this.state.persons[0].name} age = {this.state.persons[0].age}/>

<Person name={this.state.persons[1].name} age={this.state.persons[1].age}> Facebook </Person>

<Person name={this.state.persons[2].name} age={this.state.persons[2].age}> Whatsapp </Person>

<Ek />

</div>

// React.createElement('div',{className:'App'},React.createElement('h1',null,"From React"))

);

}

}

export default App;

---

App.js

//eventHandler

import React, { Component } from 'react';

import './App.css';

import Person from './Person/Person';

import Ek from './Person/Ek'

class App extends Component {

state = {

persons: [

{name:'Max',age:28},

{name:'Manu', age:29},

{name:'Stephanie',age:27}

]

}

clickHandler = () => console.log("I am clicked")

render() {

return (

<div className="App">

<h1> Hi, I am a React App!!</h1>

<Person name= {this.state.persons[0].name} age = {this.state.persons[0].age}/>

<Person name={this.state.persons[1].name} age={this.state.persons[1].age}> Facebook </Person>

<Person name={this.state.persons[2].name} age={this.state.persons[2].age}> Whatsapp </Person>

<Ek />

<button onClick={this.clickHandler}>Click</button>

</div>

// React.createElement('div',{className:'App'},React.createElement('h1',null,"From React"))

);

}

}

export default App;

---

App.js //setState

import React, { Component } from 'react';

import './App.css';

import Person from './Person/Person';

import Ek from './Person/Ek'

class App extends Component {

state = {

persons: [

{name:'Max',age:28},

{name:'Manu', age:29},

{name:'Stephanie',age:27}

]

}

clickHandler = () => {

this.setState({persons: [

{ name: 'Maxmilian', age: 28 },

{ name: 'Manu', age: 29 },

{ name: 'Stephanie', age: 22 }

]})

}

render() {

return (

<div className="App">

<h1> Hi, I am a React App!!</h1>

<Person name= {this.state.persons[0].name} age = {this.state.persons[0].age}/>

<Person name={this.state.persons[1].name} age={this.state.persons[1].age}> Facebook </Person>

<Person name={this.state.persons[2].name} age={this.state.persons[2].age}> Whatsapp </Person>

<Ek />

<button onClick={this.clickHandler}>Click</button>

</div>

// React.createElement('div',{className:'App'},React.createElement('h1',null,"From React"))

);

}

}

export default App;

---

‘useState’ always returns 2 elements, exactly 2 elements!. The first element which returns always is the ‘current state’. 2nd return element is the function that updates the current state.

App.js

//useState

import React, { useState } from 'react';

import './App.css';

import Person from './Person/Person';

import Ek from './Person/Ek'

const app = () => {

const [currentState, updateState] = useState({

persons: [

{name:'Max',age:28},

{name:'Manu', age:29},

{name:'Stephanie',age:27}

]})

const clickHandler = () => {

updateState({persons: [

{ name: 'Maxmilian', age: 28 },

{ name: 'Manu', age: 29 },

{ name: 'Stephanie Max', age: 22 }

]})

}

return (

<div className="App">

<h1> Hi, I am a React App!!</h1>

<Person name={currentState.persons[0].name} age={currentState.persons[0].age}/>

<Person name={currentState.persons[1].name} age={currentState.persons[1].age}> Facebook </Person>

<Person name={currentState.persons[2].name} age={currentState.persons[2].age}> Whatsapp </Person>

<Ek />

<button onClick={clickHandler}>Click</button>

</div>

// React.createElement('div',{className:'App'},React.createElement('h1',null,"From React"))

);

}

export default app;

---

eventHandler.bind(this,’Name’) – to pass arguments.

---

App.js

// passing method reference to other objects

import React, { Component } from 'react';

import './App.css';

import Person from './Person/Person';

import Ek from './Person/Ek'

class App extends Component {

state = {

persons: [

{ name: 'Max', age: 28 },

{ name: 'Manu', age: 29 },

{ name: 'Stephanie', age: 27 }

]

}

clickHandler = () => {

this.setState({

persons: [

{ name: 'Maxmilian', age: 28 },

{ name: 'Manu', age: 29 },

{ name: 'Stephanie', age: 22 }

]

})

}

render() {

return (

<div className="App">

<h1> Hi, I am a React App!!</h1>

<Person name={this.state.persons[0].name} age={this.state.persons[0].age} />

<Person name={this.state.persons[1].name} age={this.state.persons[1].age}> Facebook </Person>

<Person click={this.clickHandler} name={this.state.persons[2].name} age={this.state.persons[2].age}> Whatsapp </Person>

<Ek />

<button onClick={this.clickHandler}>Click</button>

</div>

// React.createElement('div',{className:'App'},React.createElement('h1',null,"From React"))

);

}

}

export default App;

---

Person.js

//props

import React from 'react'

const person = props =>{

return <p onClick={props.click}> Hello I am {props.name} {props.age} years old {props.children} </p>

}

export default person

---

App.js

// passing as argument using bind and method call

import React, { Component } from 'react';

import './App.css';

import Person from './Person/Person';

import Ek from './Person/Ek'

class App extends Component {

state = {

persons: [

{ name: 'Max', age: 28 },

{ name: 'Manu', age: 29 },

{ name: 'Stephanie', age: 27 }

]

}

clickHandler = (newName) => {

this.setState({

persons: [

{ name: newName, age: 28 },

{ name: 'Manu', age: 29 },

{ name: 'Stephanie', age: 22 }

]

})

}

render() {

return (

<div className="App">

<h1> Hi, I am a React App!!</h1>

<Person name={this.state.persons[0].name} age={this.state.persons[0].age} />

<Person name={this.state.persons[1].name} age={this.state.persons[1].age}> Facebook </Person>

<Person click= {() => this.clickHandler('Sonia')} name={this.state.persons[2].name} age={this.state.persons[2].age}> Whatsapp </Person>

<Ek />

<button onClick={this.clickHandler.bind(this,'Jaison')}>Click</button>

</div>

// React.createElement('div',{className:'App'},React.createElement('h1',null,"From React"))

);

}

}

export default App;

---

App.js

// onChange with an event object passed

import React, { Component } from 'react';

import './App.css';

import Person from './Person/Person';

import Ek from './Person/Ek'

class App extends Component {

state = {

persons: [

{ name: 'Max', age: 28 },

{ name: 'Manu', age: 29 },

{ name: 'Stephanie', age: 27 }

]

}

clickHandler = (newName) => {

this.setState({

persons: [

{ name: newName, age: 28 },

{ name: 'Manu', age: 29 },

{ name: 'Stephanie', age: 22 }

]

})

}

changeName = (event) => {

this.setState({

persons: [

{ name: 'Maximilan', age: 28 },

{ name: event.target.value, age: 29 },

{ name: 'Stephanie', age: 22 }

]

})

}

render() {

return (

<div className="App">

<h1> Hi, I am a React App!!</h1>

<Person name={this.state.persons[0].name} age={this.state.persons[0].age} />

<Person changed={this.changeName} name={this.state.persons[1].name} age={this.state.persons[1].age}> Facebook </Person>

<Person click= {() => this.clickHandler('Sonia')} name={this.state.persons[2].name} age={this.state.persons[2].age}> Whatsapp </Person>

<Ek />

<button onClick={this.clickHandler.bind(this,'Jaison')}>Click</button>

</div>

// React.createElement('div',{className:'App'},React.createElement('h1',null,"From React"))

);

}

}

export default App;

---

Person.js

import React from 'react'

const person = props =>{

return (

<div>

<p onClick={props.click}> Hello I am {props.name} {props.age} years old {props.children} </p>

<input type="text" onChange={props.changed} value={props.name} />

</div>

)

}

export default person

---

‘inline’ style is defined inside the ‘render()’ method in the class, which can be used in the element declared under ‘return’

Single curly braces {} used to input dynamic content.

Useful Resources & Links

* create-react-app: <https://github.com/facebookincubator/create-react-app>
* Introducing JSX: <https://reactjs.org/docs/introducing-jsx.html>
* Rendering Elements: <https://reactjs.org/docs/rendering-elements.html>
* Components & Props: <https://reactjs.org/docs/components-and-props.html>
* Listenable Events: <https://reactjs.org/docs/events.html>

Javascript expressions can be used inside jsx using single curly braces { expressions }.

Jsx code start with the return statement in the ‘render()’ method in the component. Inside jsx, if condition cannot be used. Instead, ternary operator can be used.

App.js

// Using map function to return array elements in jsx

render() {

let p = null

if(this.state.personOnoff) {

p = (

<div>

{

this.state.persons.map(person => {

return (

<Person name={person.name}

age={person.age} />

)

})

---

App.js

//removing an element from array by click on the text

removeHandler = (personIndex) => {

const list = this.state.persons

list.splice(personIndex,1)

this.setState({persons:list})

}

render() {

let p = null

if(this.state.personOnoff) {

p = (

<div>

{

this.state.persons.map((person,index) => {

return (

<Person click={() => this.removeHandler(index)} name={person.name}

age={person.age} />

)

})

---

newArray = Array.slice() – without arguments simply copies the array and returns a new array.

newArray = […oldArray] – also can be used to copy an array (using spread operator)

using a ‘key’ as a property inside the component is sometimes advisable

Useful Resources & Links

* Conditional Rendering: <https://reactjs.org/docs/conditional-rendering.html>
* Lists & Keys: <https://reactjs.org/docs/lists-and-keys.html>

Dynamic styles: first define multiple styles inside a .css file as classes, then define a variable and assign classes as array values then join them. Apply the variable conditionally. Inside the component, assign the variable against the component ‘style className’.

Radium – package to install style selector. Npm install –save radium. (save us used to save package into project .json folder). This allows to use pseudo selectors in ‘in-style’ selectors.

Import Radium from ‘radium’

Export default Radium(app)

App.js

render () {

const style = {

backgroundColor: 'green',

color: 'white',

font: 'inherit',

border: '1px solid blue',

padding: '8px',

cursor: 'pointer',

':hover': { //hover

backgroundColor: 'lightgreen',

color: 'black'

}

};

let persons = null;

if ( this.state.showPersons ) {

persons = (

<div>

{this.state.persons.map( ( person, index ) => {

return <Person

click={() => this.deletePersonHandler( index )}

name={person.name}

age={person.age}

key={person.id}

changed={( event ) => this.nameChangedHandler( event, person.id )} />

} )}

</div>

);

style.backgroundColor = 'red';

style[':hover'] = { //using hover

backgroundColor: 'salmon',

color: 'black'

};

}

const classes = [];

if ( this.state.persons.length <= 2 ) {

classes.push( 'red' ); // classes = ['red']

}

if ( this.state.persons.length <= 1 ) {

classes.push( 'bold' ); // classes = ['red', 'bold']

}

return (

<StyleRoot>

<div className="App">

<h1>Hi, I'm a React App</h1>

<p className={classes.join( ' ' )}>This is really working!</p>

<button

style={style}

onClick={this.togglePersonsHandler}>Toggle Persons</button>

{persons}

</div>

</StyleRoot>

);

// return React.createElement('div', {className: 'App'}, React.createElement('h1', null, 'Does this work now?'));

}

}

export default Radium( App );

Userinput.js

const inputText = (props) => {

const inputStyle = {

border: '8px solid red',

padding: '15px',

text\_align: 'center',

'@media (min-width:500px)': {

width:'450px'

}

}

---

App.js

import Radium, { StyleRoot } from 'radium'

render() {

const styl = {

backgroundColor:'green',

color: 'red'

}

return (

<StyleRoot>

<div className="App">

<header className="App-header">

<img src={logo} className="App-logo" alt="logo" />

<h1 className="App-title">Welcome to React</h1>

</header>

<p className="App-intro">

To get started, edit <code>src/App.js</code> and save to reload.

</p>

<p style={styl}> Hellooo</p>

<UserInput changed = {this.changeHandler} currentName={this.state.username}/>

<UserOutput userName = {this.state.username}/>

<UserOutput userName={this.state.username}/>

<UserOutput userName = "Sonia"/>

</div>

</StyleRoot>

);

---

Styled-components.com

Tagged templates use `` (tow backticks) at the end

Npm install –save styled-components

Import styled from ‘styled-components’

App.js

import React, { Component } from 'react';

import styled from 'styled-components';

import './App.css';

import Person from './Person/Person';

const StyledButton = styled.button`

background-color: ${props => props.alt ? 'red' : 'green'};

color: white;

font: inherit;

border: 1px solid blue;

padding: 8px;

cursor: pointer;

&:hover {

background-color: ${props => props.alt ? 'salmon' : 'lightgreen'};

color: black;

}

`;

class App extends Component {

state = {

persons: [

{ id: 'asfa1', name: 'Max', age: 28 },

{ id: 'vasdf1', name: 'Manu', age: 29 },

{ id: 'asdf11', name: 'Stephanie', age: 26 }

],

otherState: 'some other value',

showPersons: false

};

nameChangedHandler = (event, id) => {

const personIndex = this.state.persons.findIndex(p => {

return p.id === id;

});

const person = {

...this.state.persons[personIndex]

};

// const person = Object.assign({}, this.state.persons[personIndex]);

person.name = event.target.value;

const persons = [...this.state.persons];

persons[personIndex] = person;

this.setState({ persons: persons });

};

deletePersonHandler = personIndex => {

// const persons = this.state.persons.slice();

const persons = [...this.state.persons];

persons.splice(personIndex, 1);

this.setState({ persons: persons });

};

togglePersonsHandler = () => {

const doesShow = this.state.showPersons;

this.setState({ showPersons: !doesShow });

};

render() {

const style = {

backgroundColor: 'green',

color: 'white',

font: 'inherit',

border: '1px solid blue',

padding: '8px',

cursor: 'pointer',

':hover': {

backgroundColor: 'lightgreen',

color: 'black'

}

};

let persons = null;

if (this.state.showPersons) {

persons = (

<div>

{this.state.persons.map((person, index) => {

return (

<Person

click={() => this.deletePersonHandler(index)}

name={person.name}

age={person.age}

key={person.id}

changed={event => this.nameChangedHandler(event, person.id)}

/>

);

})}

</div>

);

// style.backgroundColor = 'red';

// style[':hover'] = {

// backgroundColor: 'salmon',

// color: 'black'

// };

}

const classes = [];

if (this.state.persons.length <= 2) {

classes.push('red'); // classes = ['red']

}

if (this.state.persons.length <= 1) {

classes.push('bold'); // classes = ['red', 'bold']

}

return (

<div className="App">

<h1>Hi, I'm a React App</h1>

<p className={classes.join(' ')}>This is really working!</p>

<StyledButton alt={this.state.showPersons} onClick={this.togglePersonsHandler}>

Toggle Persons

</StyledButton>

{persons}

</div>

);

// return React.createElement('div', {className: 'App'}, React.createElement('h1', null, 'Does this work now?'));

}

}

export default App;

---

More on CSS Modules

**CSS Modules** are a relatively new concept (you can dive super-deep into them here: <https://github.com/css-modules/css-modules>). With CSS modules, you can write normal CSS code and make sure, that it only applies to a given component.

It's not using magic for that, instead it'll simply **automatically generate unique CSS class names** for you. And by importing a JS object and assigning classes from there, you use these dynamically generated, unique names. So the imported JS object simply exposes some properties which hold the generated CSS class names as values.

**Example:**

**In Post.css File**

1. .Post {
2. color: red;
3. }

**In Post Component File**

1. import classes from './Post.css';
3. const post = () => (
4. <div className={classes.Post}>...</div>
5. );

Here, classes.Post  refers to an automatically generated Post  property on the imported classes  object. That property will in the end simply hold a value like Post\_\_Post\_\_ah5\_1 .

So your .Post  class was automatically transformed to a different class (Post\_\_Post\_\_ah5\_1 ) which is unique across the application. You also can't use it accidentally in other components because you don't know the generated string! You can only access it through the classes  object. And if you import the CSS file (in the same way) in another component, the classes  object there will hold a Post  property which yields a **different** (!) CSS class name. Hence it's scoped to a given component.

By the way, if you somehow also want to define a global (i.e. un-transformed) CSS class in such a .css  file, you can prefix the selector with :global .

**Example:**

:global .Post { ... }

Now you can use className="Post"  anywhere in your app and receive that styling.

Useful Resources & Links

* Using CSS Modules in create-react-app Projects: <https://medium.com/nulogy/how-to-use-css-modules-with-create-react-app-9e44bec2b5c2>
* More information about CSS Modules: <https://github.com/css-modules/css-modules>

---

Add react developer tools to the chrome webstore

Useful Resources & Links

* Error Boundaries: <https://reactjs.org/docs/error-boundaries.html>
* Chrome Devtool Debugging: <https://developers.google.com/web/tools/chrome-devtools/javascript/>

Restructure the src folder:

* 1. Components (for individual components)
  2. Assets (for images …)
  3. Container (for App.js, .css)
  4. Cockpit (Cockpit.js)
  5. App.js

A presentational component is a functional component that does not manage state.

Component life-cycle is only available in class-based components

setState causes re-rendering.

‘usEffect’ is not a life-cycle hook. It’s a react hook. It is written in the function body, which takes a function which will run for every render() cycle.

useEffect runs for every update on the component where it is defined. Its component mount and component update in one place.

usEffect has an ‘array’ as its 2nd argument.

## **Lifecycle of Components**

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.

The three phases are: **Mounting**, **Updating**, and **Unmounting**.

## **Mounting**

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

1. constructor()
2. getDerivedStateFromProps()
3. render()
4. componentDidMount()

The render() method is required and will always be called, the others are optional and will be called if you define them.

### **constructor**

The constructor() method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state and other initial values.

The constructor() method is called with the props, as arguments, and you should always start by calling the super(props) before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (React.Component).

### **getDerivedStateFromProps**

The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM.

This is the natural place to set the state object based on the initial props.

It takes state as an argument, and returns an object with changes to the state.

The example below starts with the favorite color being "red", but the getDerivedStateFromProps() method updates the favorite color based on the favcol attribute:

### **render**

The render() method is required, and is the method that actually outputs the HTML to the DOM.

### **componentDidMount**

The componentDidMount() method is called after the component is rendered.

This is where you run statements that requires that the component is already placed in the DOM.

## **Updating**

The next phase in the lifecycle is when a component is updated.

A component is updated whenever there is a change in the component's state or props.

React has five built-in methods that gets called, in this order, when a component is updated:

1. getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. componentDidUpdate()

The render() method is required and will always be called, the others are optional and will be called if you define them.

### **getDerivedStateFromProps**

Also at updates the getDerivedStateFromProps method is called. This is the first method that is called when a component gets updated.

This is still the natural place to set the state object based on the initial props.

### **shouldComponentUpdate**

In the shouldComponentUpdate() method you can return a Boolean value that specifies whether React should continue with the rendering or not.

The default value is true.

### **render**

The render() method is of course called when a component gets updated, it has to re-render the HTML to the DOM, with the new changes.

### **getSnapshotBeforeUpdate**

In the getSnapshotBeforeUpdate() method you have access to the props and state before the update, meaning that even after the update, you can check what the values were before the update.

If the getSnapshotBeforeUpdate() method is present, you should also include the componentDidUpdate() method, otherwise you will get an error.

### **componentDidUpdate**

The componentDidUpdate method is called after the component is updated in the DOM.

## **Unmounting**

The next phase in the lifecycle is when a component is removed from the DOM, or unmounting as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

* componentWillUnmount()

### **componentWillUnmount**

The componentWillUnmount method is called when the component is about to be removed from the DOM.

### Ract component life cycle: The Component Lifecycle

Each component has several “lifecycle methods” that you can override to run code at particular times in the process. **You can use**[**this lifecycle diagram**](https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/)**as a cheat sheet.** In the list below, commonly used lifecycle methods are marked as **bold**. The rest of them exist for relatively rare use cases.

#### **Mounting**

These methods are called in the following order when an instance of a component is being created and inserted into the DOM:

* [**constructor()**](https://reactjs.org/docs/react-component.html#constructor)
* [static getDerivedStateFromProps()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromprops)
* [**render()**](https://reactjs.org/docs/react-component.html#render)
* [**componentDidMount()**](https://reactjs.org/docs/react-component.html#componentdidmount)

**Note:**

These methods are considered legacy and you should [avoid them](https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html) in new code:

* [UNSAFE\_componentWillMount()](https://reactjs.org/docs/react-component.html#unsafe_componentwillmount)

#### **Updating**

An update can be caused by changes to props or state. These methods are called in the following order when a component is being re-rendered:

* [static getDerivedStateFromProps()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromprops)
* [shouldComponentUpdate()](https://reactjs.org/docs/react-component.html#shouldcomponentupdate)
* [**render()**](https://reactjs.org/docs/react-component.html#render)
* [getSnapshotBeforeUpdate()](https://reactjs.org/docs/react-component.html#getsnapshotbeforeupdate)
* [**componentDidUpdate()**](https://reactjs.org/docs/react-component.html#componentdidupdate)

**Note:**

These methods are considered legacy and you should [avoid them](https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html) in new code:

* [UNSAFE\_componentWillUpdate()](https://reactjs.org/docs/react-component.html#unsafe_componentwillupdate)
* [UNSAFE\_componentWillReceiveProps()](https://reactjs.org/docs/react-component.html#unsafe_componentwillreceiveprops)

#### **Unmounting**

This method is called when a component is being removed from the DOM:

* [**componentWillUnmount()**](https://reactjs.org/docs/react-component.html#componentwillunmount)

#### **Error Handling**

These methods are called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.

* [static getDerivedStateFromError()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromerror)
* [componentDidCatch()](https://reactjs.org/docs/react-component.html#componentdidcatch)

### Other APIs

Each component also provides some other APIs:

* [setState()](https://reactjs.org/docs/react-component.html#setstate)
* [forceUpdate()](https://reactjs.org/docs/react-component.html#forceupdate)

### Class Properties

* [defaultProps](https://reactjs.org/docs/react-component.html#defaultprops)
* [displayName](https://reactjs.org/docs/react-component.html#displayname)

### Instance Properties

* [props](https://reactjs.org/docs/react-component.html#props)
* [state](https://reactjs.org/docs/react-component.html#state)

Export default React.memo(person) // react does memoization of the component.

React maintains old virtual dom and the rerendered virtual dom

An array [] of components with ‘key’ as property can be used to return an array of different components in a render() method.

React.Fragment – can also be used in place for returning multiple elements. Can create another element as children and wrap the element to return them elements inside the render.(HOC higher order components)

Npm install prop-types

React.createRef()

To use in functional components, import useRef, then useRef()

‘ref’ is a property like ‘key’ understood by ‘react’ components.

‘useEffect’ runs after every render cycle.

React.createContext() // AuthContext.Provider()

Import useContext allows to get access to contexts from functional components

Static contextType = AuthContext – is for class components

Useful Resources & Links

Useful Resources:

* More on useEffect(): <https://reactjs.org/docs/hooks-effect.html>
* State & Lifecycle: <https://reactjs.org/docs/state-and-lifecycle.html>
* PropTypes: <https://reactjs.org/docs/typechecking-with-proptypes.html>
* Higher Order Components: <https://reactjs.org/docs/higher-order-components.html>
* Refs: <https://reactjs.org/docs/refs-and-the-dom.html>

Both React and ReactDOM are available over a CDN.

<script crossorigin src="https://unpkg.com/react@17/umd/react.development.js"></script>

<script crossorigin src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"></script>

The versions above are only meant for development, and are not suitable for production. Minified and optimized production versions of React are available at:

<script crossorigin src="https://unpkg.com/react@17/umd/react.production.min.js"></script>

<script crossorigin src="https://unpkg.com/react-dom@17/umd/react-dom.production.min.js"></script>

---

JSX produces React “elements”.

You can put any valid [JavaScript expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Expressions) inside the curly braces in JSX.

After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.

This means that you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions:

Don’t put quotes around curly braces when embedding a JavaScript expression in an attribute. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

Since JSX is closer to JavaScript than to HTML, React DOM uses camelCase property naming convention instead of HTML attribute names.

By default, React DOM [escapes](https://stackoverflow.com/questions/7381974/which-characters-need-to-be-escaped-on-html) any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that’s not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent [XSS (cross-site-scripting)](https://en.wikipedia.org/wiki/Cross-site_scripting) attacks.

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!'

);

--

Elements are what components are “made of”, and we encourage you to read this section before jumping ahead.

React elements are [immutable](https://en.wikipedia.org/wiki/Immutable_object). Once you create an element, you can’t change its children or attributes. An element is like a single frame in a movie: it represents the UI at a certain point in time.

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

**All React components must act like pure functions with respect to their props.**

State allows React components to change their output over time in response to user actions, network responses, and anything else, without violating this rule.

Class components should always call the base constructor with props.

componentDidMount() // componentWillUnmount()

setInterval(), clearInterval()

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

‘state’ is not accessible to any component other than the one that owns and sets it.

* With JSX you pass a function as the event handler, rather than a string.

Generally, if you refer to a method without () after it, such as onClick={this.handleClick}, you should bind that method.

e.preventDefault();

You can build collections of elements and [include them in JSX](https://reactjs.org/docs/introducing-jsx.html#embedding-expressions-in-jsx) using curly braces {}. We assign the new array returned by map() to the variable.

A “key” is a special string attribute you need to include when creating lists of elements.

function NumberList(props) {

const numbers = props.numbers;

const listItems = numbers.map((number) =>

<li>{number}</li>

);

return (

<ul>{listItems}</ul>

);

}

const numbers = [1, 2, 3, 4, 5];

ReactDOM.render(

<NumberList numbers={numbers} />,

document.getElementById('root')

);

---

## Keys

Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity:

When you don’t have stable IDs for rendered items, you may use the item index as a key as a last resort:

const todoItems = todos.map((todo, index) =>

// Only do this if items have no stable IDs

<li key={index}>

{todo.text}

</li>

);

 If you choose not to assign an explicit key to list items then React will default to using indexes as keys.

function ListItem(props) {

// Correct! There is no need to specify the key here:

return <li>{props.value}</li>;

}

function NumberList(props) {

const numbers = props.numbers;

const listItems = numbers.map((number) =>

// Correct! Key should be specified inside the array.

<ListItem key={number.toString()} value={number} />

);

return (

<ul>

{listItems}

</ul>

);

}

const numbers = [1, 2, 3, 4, 5];

ReactDOM.render(

<NumberList numbers={numbers} />,

document.getElementById('root')

);

---

A good rule of thumb is that elements inside the map() call need keys.

Forms:

In React, because form elements naturally keep some internal state.

 In React, mutable state is typically kept in the state property of components, and only updated with [setState()](https://reactjs.org/docs/react-component.html#setstate).

An input form element whose value is controlled by React is called a “controlled component”.

You can pass an array into the value attribute, allowing you to select multiple options in a select tag:

<select multiple={true} value={['B', 'C']}>

Because its value is read-only, file is an **uncontrolled** component in React

Refs provide a way to access DOM nodes or React elements created in the render method.

### When to Use Refs

There are a few good use cases for refs:

* Managing focus, text selection, or media playback.
* Triggering imperative animations.
* Integrating with third-party DOM libraries.

Avoid using refs for anything that can be done declaratively.

If you’re looking for a complete solution including validation, keeping track of the visited fields, and handling form submission, [Formik](https://jaredpalmer.com/formik) is one of the popular choices.

Formik takes care of the repetitive and annoying stuff—keeping track of values/errors/visited fields, orchestrating validation, and handling submission—so you don't have to. This means you spend less time wiring up state and change handlers and more time focusing on your business logic.

Shared state between 2 react components:

* Parent to Child
  1. Props
  2. Instance Methods
* Child to Parent
  1. Callback Functions
  2. Event Bubbling
* Sibling to Sibling
  1. Parent Component
* Any to Any
  1. Observer Pattern
  2. Global Variables
  3. Context

Write functions that update the ‘state’ of a component and share that function with other components

In React, sharing state is accomplished by moving it up to the closest common ancestor of the components that need it. This is called “lifting state up”.

React has a powerful composition model, and we recommend using composition instead of inheritance to reuse code between components.

**Pure components**

A **React component** can be considered **pure** if it renders the same output for the same state and props. ... **Pure components** have some performance improvements and render optimizations since **React** implements the shouldComponentUpdate() method for them with a shallow comparison for props and state

---------------------------

Planning a React App

1. Component tree / structure
2. Application state (Data)
3. Components vs containers

CSS modules stylesheet ([name].module.css)

CSS Modules allows the scoping of CSS by automatically creating a unique classname of the format [filename]\\_[classname]\\_\\_[hash]

import styles from './Button.module.css'; // Import css modules stylesheet as styles

containers are ‘stateful’ components using ‘class’ keyword or functional using ‘useState’.

Presentational components don’t manage ‘state’

Npm install –save prop-types used for validation – can be used in functional components also

‘reduce’ is a built-in array function, which transforms an array into something else.

----

when using classes, ‘props’ not to be declared. It can be used using ‘this’ operator. this.props.<property>

CSS Modules are convenient for components that are placed in separate files.

The CSS inside a module is available only for the component that imported it, and you do not have to worry about name conflicts.

## **What is Sass**

Sass is a CSS pre-processor.

Sass files are executed on the server and sends CSS to the browser.

Server will provide some api end points, to which React can send requests.

Jsonplaceholder.typicode.com – online rest api for testing

Axios is a 3rd party js library. This can manage http requests and responses..

Npm install axios –save

Axios.get(‘url’)

Axios.post(‘url’,data)

Axios.delete(‘url’)

Axios.interceptors.request.use(request => request, error => Promise.reject(error))

Axios.interceptors.response.use(response => response, error => Promise.reject(error))

Removing interceptors: You learned how to add an interceptor, getting rid of one is also easy. Simply store the reference to the interceptor in a variable and call eject  with that reference as an argument, to remove it (more

Axios.defaults.baseURL = ‘url’

Let instance = axios.create({baseURL:’url’})

Console.firbase.google.com

<https://jjburger-1c5ca.firebaseio.com/>

**Axios interceptors** are functions that **Axios** calls for every request. You can use **interceptors** to transform the request before **Axios** sends it, or transform the response before **Axios** returns the response to your code.

Axios.interceptor.request.eject(intereceptor)

Axios.interceptor.response.eject(intereceptor)

Routing is being able to show different pages to the user.

In SPA’s, rerender page with different parts

Npm install react-router react-router-dom –save

(router for the logic, dom for the render)its not from Facebook. It’s the defacto standard for routing in React.

react-router vs react-router-dom

We installed both react-router  and react-router-dom . **Technically, only react-router-dom  is required for web development**. It wraps react-router  and therefore uses it as a dependency.

Import {BrowserRouter} from ‘react-router-dom’

Browserrouter is used to wrap the portion of the component which uses the routing. To use this, use it in index.js as <BrowserRouter> <Ap /> </BrowserRouter>

import {Route, Link} from 'react-router-dom'

<Route path='/' render={() => <h1>Helloow</h1>} />

<Route path='/' component={Posts} />

‘reloading’ a page will cause lose of ‘state’. To avoid this, use Link

<li> <Link to="/"> Home</Link> </li>

<li> <Link to={{pathname: '/new-post', hash:'#mybtn'}}> New Post</Link> </li>

{ withRouter } is a higher order component, which wraps the exporting component. This is used to get routing related props

To create relative path: pathname= this.props.match.url + ‘/<name>’. All other paths are absolute paths.

Absolute vs Relative Paths (Article)

You learned about <Link> , you learned about the to  property it uses.

The path you can use in to can be either **absolute** or **relative**.

#### **Absolute Paths**

By default, if you just enter to="/some-path"  or to="some-path" , that's an **absolute path**.

**Absolute path** means that it's**always appended right after your domain**. Therefore, both syntaxes (with and without leading slash) lead to example.com/some-path .

#### **Relative Paths**

Sometimes, you might want to create a relative path instead. This is especially useful, if your component is already loaded given a specific path (e.g. posts ) and you then want to append something to that existing path (so that you, for example, get /posts/new ).

If you're on a component loaded via /posts , to="new"  would lead to example.com/new , **NOT** example.com/posts/new .

To change this behavior, you have to find out which path you're on and add the new fragment to that existing path. You can do that with the url  property of props.match :

<Link to={props.match.url + '/new'}>  will lead to example.com/posts/new  when placing this link in a component loaded on /posts . If you'd use the same <Link>  in a component loaded via /all-posts , the link would point to /all-posts/new .

‘NavLink’ to provide extra styling to the ‘active link’

Pass ‘Route’ parameters using path=‘/:<id>’

Parsing Query Parameters & the Fragment

You learned how to extract route parameters (=> :id  etc).

But how do you extract **search** (also referred to as "**query**") **parameters**(=> ?something=somevalue  at the end of the URL)? How do you extract the **fragment** (=> #something  at the end of the URL)?

#### **Query Params:**

You can pass them easily like this:

<Link to="/my-path?start=5">Go to Start</Link>

or

1. <Link
2. to={‌{
3. pathname: '/my-path',
4. search: '?start=5'
5. }}
6. >Go to Start</Link>

React router makes it easy to get access to the search string: props.location.search .

But that will only give you something like ?start=5

You probably want to get the key-value pair, without the ?  and the = . Here's a snippet which allows you to easily extract that information:

1. componentDidMount() {
2. const query = new URLSearchParams(this.props.location.search);
3. for (let param of query.entries()) {
4. console.log(param); // yields ['start', '5']
5. }
6. }

URLSearchParams  is a built-in object, shipping with vanilla JavaScript. It returns an object, which exposes the entries()  method. entries()  returns an Iterator - basically a construct which can be used in a for...of...  loop (as shown above).

When looping through query.entries() , you get **arrays** where the first element is the **key name** (e.g. start ) and the second element is the assigned **value** (e.g. 5 ).

#### **Fragment:**

You can pass it easily like this:

<Link to="/my-path#start-position">Go to Start</Link>

or

1. <Link
2. to={‌{
3. pathname: '/my-path',
4. hash: 'start-position'
5. }}
6. >Go to Start</Link>

React router makes it easy to extract the fragment. You can simply access props.location.hash .

Use <Switch> </Switch> to load one Route at a time. Order of Route’s are important when using Switch. It loads only the first matching Route.

Navigation is all about moving in a stack of pages.

Use- this.props.history.push({{pathname: ‘/’+id}}) or simply

This.props.history.push(‘/’+id) // this is mostly done when a given operation is finished.

**React router does** partial matching, so /users partially matches /users/create , so it **would** incorrectly return the Users route again! The **exact** param disables the partial matching for a route and makes sure that it only returns the route if the path is an **EXACT** match to the current url.

Creating dynamic route: <Route path={this.props.match.url + ‘/:id’} component={component-name} />

‘Redirect’ component to redirect the pages.

<Redirect from=’/’ to=’/post’ />

If this is specified out of ‘<Switch>’ then the ‘from’ property cannot be specified.

Navigage history using – this.props.history.push(‘/page’) or this.props.history.replace(‘/post’)

Guard is a conditional check using ternary operator on property state if the user has authorization to access a page.

The technique of downloading only what you need is called code splitting or lazy loading. Use an asynchronous function which imports the file containing the component and returns it, which is loaded conditionally.

From React 16.6, ‘lazy’ method allows to load components asynchronously. (only when needed or renderd)

Import {Suspense} from ‘react’

Const Posts = React.lazy(()=> import(‘./Folder/component))

<Router path=’/posts/’, render={()=>(<Suspense fallback={<div> loading .. </div>} <Posts /> </Suspense>)}

In React, always load index.html (even at 404 error), then parse to the right page.

If the app is loaded from a sub directory, set the <BrowserRouter basename=’/my-app’>

Only the components directly added to the component property of the ‘Route’ will get the ‘props’ objects in the browser(history, location, match). Inorder to get these objects in other components, use ‘withRouter’ wrapping around the component export.

encodeURIComponent() – to encode the data which is sent.

const query = new URLSearchParams(this.props.location.search) – to search for items in the URL

Useful Resources & Links

* Validate.js (you may import its functionality into your React projects): <https://validatejs.org/>
* Get more ideas about potential validation approaches: <https://react.rocks/tag/Validation>

Alternatives to the manual approach taken in this course:

* react-validation package: <https://www.npmjs.com/package/react-validation>
* formsy-react package: <https://github.com/christianalfoni/formsy-react>

**Redux:** is a standalone 3rd party library to make ‘state’ management

Redux stores ‘state’ in a global variable?

Component send actions to reducer, reducer execute the action and update the state. Components which subscribed to the state will receive the updated state.

Npm install –save redux

To run node js, rum command as: node <file.js>

---

// Redux in action

// Redux in action

const redux = require('redux') // creating a redux instance

const initialState = { // creating a state

counter:0

}

const rootReducer = function (state=initialState, action) { // creating a reducer

if(action.type === 'INC\_COUNTER'){

return { // creating a copy of current state to update. Never mutate the current state

...state,

counter : state.counter + 1

}

}

if(action.type === 'ADD\_COUNTER'){

return {

...state,

counter : state.counter + action.value

}

}

return state

}

const store = redux.createStore(rootReducer) // creating a store attached with a reducer

store.dispatch({type: 'INC\_COUNTER'} ) // the argument is object where type is mandatory

store.dispatch({ type: 'ADD\_COUNTER', value: 10 })

console.log(store.getState()) // getting back the current state

store.subscribe(() => { // subscribing to store

console.log(store.getState())

})

const store = redux.createStore(rootReducer) // creating a store attached with a reducer

console.log(store.getState()) // getting back the current state

running the file and output:

node redux\_basics.js

{ counter: 11 }

The store.subscribe(function) – the function is executed whenever the store is updated.

---

Never mutate any data in the store. Always act immutably.

Npm install –save react-redux // to hook up redux store into react app.

Import {createStore) from ‘redux’

Import {Provier} from ‘react-redux’ // a helper component which will help in inject the ‘store’ in to ‘react’ components.

Connect is a function which returns a higher order component.it specifies the actions we want to dispatch and the state which we want to get.

To add a new element to an array in ‘state’ management, use array.concat instead of array.push

The array.filter() – returns a new array

Immutable Update Patterns

Immutable Update Patterns on reduxjs.org: https://redux.js.org/recipes/structuring-reducers/immutable-update-patterns/

Updating Nested Objects

The key to updating nested data is that every level of nesting must be copied and updated appropriately. This is often a difficult concept for those learning Redux, and there are some specific problems that frequently occur when trying to update nested objects. These lead to accidental direct mutation, and should be avoided.

Common Mistake #1: New variables that point to the same objects

Defining a new variable does not create a new actual object - it only creates another reference to the same object. An example of this error would be:

function updateNestedState(state, action) {

let nestedState = state.nestedState;

// ERROR: this directly modifies the existing object reference - don't do this!

nestedState.nestedField = action.data;

return {

...state,

nestedState

};

}

This function does correctly return a shallow copy of the top-level state object, but because the nestedState variable was still pointing at the existing object, the state was directly mutated.

Common Mistake #2: Only making a shallow copy of one level

Another common version of this error looks like this:

function updateNestedState(state, action) {

// Problem: this only does a shallow copy!

let newState = {...state};

// ERROR: nestedState is still the same object!

newState.nestedState.nestedField = action.data;

return newState;

}

Doing a shallow copy of the top level is not sufficient - the nestedState object should be copied as well.

Correct Approach: Copying All Levels of Nested Data

Unfortunately, the process of correctly applying immutable updates to deeply nested state can easily become verbose and hard to read. Here's what an example of updating state.first.second[someId].fourth might look like:

function updateVeryNestedField(state, action) {

return {

...state,

first : {

...state.first,

second : {

...state.first.second,

[action.someId] : {

...state.first.second[action.someId],

fourth : action.someValue

}

}

}

}

}

Obviously, each layer of nesting makes this harder to read, and gives more chances to make mistakes. This is one of several reasons why you are encouraged to keep your state flattened, and compose reducers as much as possible.

Inserting and Removing Items in Arrays

Normally, a Javascript array's contents are modified using mutative functions like push, unshift, and splice. Since we don't want to mutate state directly in reducers, those should normally be avoided. Because of that, you might see "insert" or "remove" behavior written like this:

function insertItem(array, action) {

return [

...array.slice(0, action.index),

action.item,

...array.slice(action.index)

]

}

function removeItem(array, action) {

return [

...array.slice(0, action.index),

...array.slice(action.index + 1)

];

}

However, remember that the key is that the original in-memory reference is not modified. As long as we make a copy first, we can safely mutate the copy. Note that this is true for both arrays and objects, but nested values still must be updated using the same rules.

This means that we could also write the insert and remove functions like this:

function insertItem(array, action) {

let newArray = array.slice();

newArray.splice(action.index, 0, action.item);

return newArray;

}

function removeItem(array, action) {

let newArray = array.slice();

newArray.splice(action.index, 1);

return newArray;

}

The remove function could also be implemented as:

function removeItem(array, action) {

return array.filter( (item, index) => index !== action.index);

}

Updating an Item in an Array

Updating one item in an array can be accomplished by using Array.map, returning a new value for the item we want to update, and returning the existing values for all other items:

function updateObjectInArray(array, action) {

return array.map( (item, index) => {

if(index !== action.index) {

// This isn't the item we care about - keep it as-is

return item;

}

// Otherwise, this is the one we want - return an updated value

return {

...item,

...action.item

};

});

}

Immutable Update Utility Libraries

Because writing immutable update code can become tedious, there are a number of utility libraries that try to abstract out the process. These libraries vary in APIs and usage, but all try to provide a shorter and more succinct way of writing these updates. Some, like dot-prop-immutable, take string paths for commands:

state = dotProp.set(state, `todos.${index}.complete`, true)

Others, like immutability-helper (a fork of the now-deprecated React Immutability Helpers addon), use nested values and helper functions:

var collection = [1, 2, {a: [12, 17, 15]}];

var newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13, 14]]}}});

They can provide a useful alternative to writing manual immutable update logic.

Immutable Data#Immutable Update Utilities section of the Redux Addons Catalog.

Import \* as actions from ‘./filename’ – imports everything from that file and store as an object in the variable ‘actions’

Import { combineReducers } from ‘redux’

Const rootReducer = combineReducer({

Ctr: counterReducer,

Res: resultReducer

})

Redux is to manage the state wen the application is live, not like a persistant database.

Mostly client state, like user authentication, filter sets are used by the Redux

Conainers are connected to Redux, not the presentational components

Useful Resources & Links

Redux Docs: https://redux.js.org/

Core Concepts: https://redux.js.org/introduction/core-concepts

Actions: https://redux.js.org/basics/actions

Reducers: https://redux.js.org/basics/reducers

Redux FAQs: <https://redux.js.org/faq>

**props** are used to pass data from a parent component to a child component in React and they are the main mechanism for component communication.

The Provider should wrap everything inside the ‘index.js’

Middleware: is a function or code hook into a process which gets executed a part of the process without stopping it.

Import {applyMiddleware} from ‘redux’

// index.js  
import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

import registerServiceWorker from './registerServiceWorker';

import {createStore, applyMiddleware} from 'redux'

import {Provider} from 'react-redux'

import reducer from './store/store'

const logger = store => {

return next => {

return action => {

console.log('[Middleware dispatching]',action)

const result = next(action)

console.log('[Middlware] next state', store.getState())

return result

}

}

}

const store = createStore(reducer, applyMiddleware(logger))

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

registerServiceWorker();

---

Add redux-devtools to the browser extension.

Import {compose} from ‘redux’

const composeEnhancers = window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_ || compose;

const store = createStore(reducer, composeEnhancers(applyMiddleware(logger)))

---

We can execute asynchronous code with the help of action creators

Action creators are just a function which creates an action

Redux-thunk which helps create a function that returns a function which dispatch an action.

Npm install –save redux-thunk

Import thunk from ‘redux-thunk’ // it’s a default export, so no curly braces needed

Const store = createStores(reducer,applyMiddleware(thunk))

// synchronous function

export const saveResult = (res) => {

return{

type: 'STORE\_RESULT',

result: res

}

}

// Redux thunk is executing the asynchronous function

export const storeResult = (res) =>{

return dispatch => {

setTimeout(

dispatch(saveResult(res)

),2000)

}

}

---

Reducers are the only place which updates state.

In the asynchronous code, along with dispatch, we can pas getState as an argument to get the state

Useful Resources & Links

Middleware: https://redux.js.org/advanced/middleware/

redux-thunk package: https://github.com/gaearon/redux-thunk

Async Actions: <https://redux.js.org/advanced/async-actions>

#### Jiji

CSS Syntax

A CSS rule-set consists of a selector and a declaration block:

The selector points to the HTML element you want to style.

The declaration block contains one or more declarations separated by semicolons.

Each declaration includes a CSS property name and a value, separated by a colon.

Multiple CSS declarations are separated with semicolons, and declaration blocks are surrounded by curly braces.

CSS Selectors

CSS selectors are used to "find" (or select) the HTML elements you want to style.

We can divide CSS selectors into five categories:

• Simple selectors (select elements based on name, id, class)

• Combinator selectors (select elements based on a specific relationship between them)

• Pseudo-class selectors (select elements based on a certain state)

• Pseudo-elements selectors (select and style a part of an element)

• Attribute selectors (select elements based on an attribute or attribute value)

The universal selector (\*) selects all HTML elements on the page.

All CSS Simple Selectors

Selector Example Example description

.class

.intro Selects all elements with class="intro"

#id

#firstname Selects the element with id="firstname"

\*

\* Selects all elements

element

p Selects all <p> elements

element,element,..

div, p Selects all <div> elements and all <p> elements

There are three ways of inserting a style sheet:

• External CSS

• Internal CSS

• Inline CSS

<link rel="stylesheet" href="mystyle.css">

The internal style is defined inside the <style> element, inside the head section.

<head>

<style>

body {

background-color: linen;

}

h1 {

color: maroon;

margin-left: 40px;

}

</style>

</head>

To use inline styles, add the style attribute to the relevant element. The style attribute can contain any CSS property.

<h1 style="color:blue;text-align:center;">This is a heading</h1>

If some properties have been defined for the same selector (element) in different style sheets, the value from the last read style sheet will be used.

However, if the internal style is defined before the link to the external style sheet, the external style will be used.

Css comment style: /\* This is a single-line comment \*/

Html comment style: <! -- sfdsf -->

Colors are specified using predefined color names, or RGB, HEX, HSL, RGBA, HSLA values.

RGBA color values are an extension of RGB color values with an alpha channel - which specifies the opacity for a color.

HEX Value

In CSS, a color can be specified using a hexadecimal value in the form:

#rrggbb 0-9 a-f

HSL Value

In CSS, a color can be specified using hue, saturation, and lightness (HSL) in the form:

hsl(hue, saturation, lightness)

Hue is a degree on the color wheel from 0 to 360. 0 is red, 120 is green, and 240 is blue.

Saturation is a percentage value, 0% means a shade of gray, and 100% is the full color.

Lightness is also a percentage, 0% is black, 50% is neither light or dark, 100% is white

The alpha parameter is a number between 0.0 (fully transparent) and 1.0 (not transparent at all):

The CSS background properties are used to define the background effects for elements.

background-color

background-image

background-repeat

background-attachment

background-position

The background-image property specifies an image to use as the background of an element.

body {

background-image: url("paper.gif");

}

If the image above is repeated only horizontally (background-repeat: repeat-x;), the background will look better:

Example

body {

background-image: url("gradient\_bg.png");

background-repeat: repeat-x; //no-repeat

background-position: right top;

background-attachment: scroll; // fixed

}

The background-attachment property specifies whether the background image should scroll or be fixed (will not scroll with the rest of the page)

To shorten the code, it is also possible to specify all the background properties in one single property. This is called a shorthand property.

Instead of writing:

body {

background-color: #ffffff;

background-image: url("img\_tree.png");

background-repeat: no-repeat;

background-position: right top;

}

You can use the shorthand property background:

Example

Use the shorthand property to set the background properties in one declaration:

body {

background: #ffffff url("img\_tree.png") no-repeat right top;

}

All CSS Background Properties

Property Description

background

Sets all the background properties in one declaration

background-attachment

Sets whether a background image is fixed or scrolls with the rest of the page

background-clip

Specifies the painting area of the background

background-color

Sets the background color of an element

background-image

Sets the background image for an element

background-origin

Specifies where the background image(s) is/are positioned

background-position

Sets the starting position of a background image

background-repeat

Sets how a background image will be repeated

background-size

Specifies the size of the background image(s)

The CSS border properties allow you to specify the style, width, and color of an element's border.

The border-style property specifies what kind of border to display.

The following values are allowed:

• dotted - Defines a dotted border

• dashed - Defines a dashed border

• solid - Defines a solid border

• double - Defines a double border

• groove - Defines a 3D grooved border. The effect depends on the border-color value

• ridge - Defines a 3D ridged border. The effect depends on the border-color value

• inset - Defines a 3D inset border. The effect depends on the border-color value

• outset - Defines a 3D outset border. The effect depends on the border-color value

• none - Defines no border

• hidden - Defines a hidden border

The border-style property can have from one to four values (for the top border, right border, bottom border, and the left border).

Border-style, border-width px, medium, thick, border-color

p {

border-top-style: dotted;

border-right-style: solid;

border-bottom-style: dotted;

border-left-style: solid;

}

The border property is a shorthand property for the following individual border properties:

• border-width

• border-style (required)

• border-color

p {

border: 5px solid red;

}

p {

border-left: 6px solid red;

background-color: lightgrey;

}

CSS Rounded Borders

The border-radius property is used to add rounded borders to an element:

Normal border

Round border

Rounder border

Roundest border

Example

p {

border: 2px solid red;

border-radius: 5px;

}

All CSS Border Properties

Property Description

border Sets all the border properties in one declaration

border-bottom Sets all the bottom border properties in one declaration

border-bottom-color Sets the color of the bottom border

border-bottom-style Sets the style of the bottom border

border-bottom-width Sets the width of the bottom border

border-color Sets the color of the four borders

border-left Sets all the left border properties in one declaration

border-left-color Sets the color of the left border

border-left-style Sets the style of the left border

border-left-width Sets the width of the left border

border-radius Sets all the four border-\*-radius properties for rounded corners

border-right Sets all the right border properties in one declaration

border-right-color Sets the color of the right border

border-right-style Sets the style of the right border

border-right-width Sets the width of the right border

border-style Sets the style of the four borders

border-top Sets all the top border properties in one declaration

border-top-color Sets the color of the top border

border-top-style Sets the style of the top border

border-top-width Sets the width of the top border

border-width Sets the width of the four borders

CSS Margins

The CSS margin properties are used to create space around elements, outside of any defined borders.

With CSS, you have full control over the margins. There are properties for setting the margin for each side of an element (top, right, bottom, and left).

Margin - Individual Sides

CSS has properties for specifying the margin for each side of an element:

margin-top

margin-right

margin-bottom

margin-left

All the margin properties can have the following values:

auto - the browser calculates the margin

length - specifies a margin in px, pt, cm, etc.

% - specifies a margin in % of the width of the containing element

inherit - specifies that the margin should be inherited from the parent element

Tip: Negative values are allowed.

Example

Set different margins for all four sides of a <p> element:

p {

margin-top: 100px;

margin-bottom: 100px;

margin-right: 150px;

margin-left: 80px;

}

Margin Collapse

Top and bottom margins of elements are sometimes collapsed into a single margin that is equal to the largest of the two margins.

This does not happen on left and right margins! Only top and bottom margins!

All CSS Margin Properties

Property Description

margin A shorthand property for setting the margin properties in one declaration

margin-bottom Sets the bottom margin of an element

margin-left Sets the left margin of an element

margin-right Sets the right margin of an element

margin-top Sets the top margin of an element

CSS Padding

The CSS padding properties are used to generate space around an element's content, inside of any defined borders.

With CSS, you have full control over the padding. There are properties for setting the padding for each side of an element (top, right, bottom, and left).

Padding - Individual Sides

CSS has properties for specifying the padding for each side of an element:

padding-top

padding-right

padding-bottom

padding-left

Padding and Element Width

The CSS width property specifies the width of the element's content area. The content area is the portion inside the padding, border, and margin of an element (the box model).

So, if an element has a specified width, the padding added to that element will be added to the total width of the element. This is often an undesirable result.

Example

Here, the <div> element is given a width of 300px. However, the actual width of the <div> element will be 350px (300px + 25px of left padding + 25px of right padding):

div {

width: 300px;

padding: 25px;

}

All CSS Padding Properties

Property Description

padding A shorthand property for setting all the padding properties in one declaration

padding-bottom Sets the bottom padding of an element

padding-left Sets the left padding of an element

padding-right Sets the right padding of an element

padding-top Sets the top padding of an element

CSS Setting height and width

The height and width properties are used to set the height and width of an element.

The height and width properties do not include padding, borders, or margins. It sets the height/width of the area inside the padding, border, and margin of the element.

CSS height and width Values

The height and width properties may have the following values:

auto - This is default. The browser calculates the height and width

length - Defines the height/width in px, cm etc.

% - Defines the height/width in percent of the containing block

initial - Sets the height/width to its default value

inherit - The height/width will be inherited from its parent value

All CSS Dimension Properties

Property Description

height Sets the height of an element

max-height Sets the maximum height of an element

max-width Sets the maximum width of an element

min-height Sets the minimum height of an element

min-width Sets the minimum width of an element

width Sets the width of an element

Box properties:

Explanation of the different parts:

Content - The content of the box, where text and images appear

Padding - Clears an area around the content. The padding is transparent

Border - A border that goes around the padding and content

Margin - Clears an area outside the border. The margin is transparent

The box model allows us to add a border around elements, and to define space between elements.

Example

Demonstration of the box model:

div {

width: 300px;

border: 15px solid green;

padding: 50px;

margin: 20px;

}

Important: When you set the width and height properties of an element with CSS, you just set the width and height of the content area. To calculate the full size of an element, you must also add padding, borders and margins.

CSS Outline

An outline is a line that is drawn around elements, OUTSIDE the borders, to make the element "stand out".

CSS has the following outline properties:

outline-style

outline-color

outline-width

outline-offset

outline

The outline-width property specifies the width of the outline, and can have one of the following values:

thin (typically 1px)

medium (typically 3px)

thick (typically 5px)

A specific size (in px, pt, cm, em, etc)

outline-color: invert

CSS Outline - Shorthand property

The outline property is a shorthand property for setting the following individual outline properties:

outline-width

outline-style (required)

outline-color

The outline-offset property adds space between an outline and the edge/border of an element. The space between an element and its outline is transparent.

All CSS Outline Properties

Property Description

outline A shorthand property for setting outline-width, outline-style, and outline-color in one declaration

outline-color Sets the color of an outline

outline-offset Specifies the space between an outline and the edge or border of an element

outline-style Sets the style of an outline

outline-width Sets the width of an outline

text-color

text-align

Text Direction

The direction and unicode-bidi properties can be used to change the text direction of an element:

Example

p {

direction: rtl;

unicode-bidi: bidi-override;

}

Vertical Alignment

The vertical-align property sets the vertical alignment of an element. top, middle, bottom

text-decoration: overline, underline, line-through

text-transfrom: uppercase, lowercalse, capitalize

The text-indent property is used to specify the indentation of the first line of a text:

text-indent letter-spacing word-spacing white-space: nowrap

The line-height property is used to specify the space between lines:

The text-shadow property adds shadow to text.

h1 {

text-shadow: 2px 2px 5px red;

}

All CSS Text Properties

Property Description

color Sets the color of text

direction Specifies the text direction/writing direction

letter-spacing Increases or decreases the space between characters in a text

line-height Sets the line height

text-align Specifies the horizontal alignment of text

text-decoration Specifies the decoration added to text

text-indent Specifies the indentation of the first line in a text-block

text-shadow Specifies the shadow effect added to text

text-transform Controls the capitalization of text

text-overflow Specifies how overflowed content that is not displayed should be signaled to the user

unicode-bidi Used together with the direction property to set or return whether the text should be overridden to support multiple languages in the same document

vertical-align Sets the vertical alignment of an element

white-space Specifies how white-space inside an element is handled

word-spacing Increases or decreases the space between words in a text

CSS Font Families

In CSS, there are two types of font family names:

generic family - a group of font families with a similar look (like "Serif" or "Monospace")

font family - a specific font family (like "Times New Roman" or "Arial")

Generic family Font family Description

Serif Times New Roman

Georgia Serif fonts have small lines at the ends on some characters

Sans-serif Arial

Verdana "Sans" means without - these fonts do not have the lines at the ends of characters

Monospace Courier New

Lucida Console All monospace characters have the same width

.serif {

font-family: "Times New Roman", Times, serif;

}

.sansserif {

font-family: Arial, Helvetica, sans-serif;

}

.monospace {

font-family: "Lucida Console", Courier, monospace;

}

The font-style property is mostly used to specify italic text.

This property has three values:

normal - The text is shown normally

italic - The text is shown in italics

oblique - The text is "leaning"

font-weight

The font-variant property specifies whether or not a text should be displayed in a small-caps font. normal, small-caps

The font-size value can be an absolute, or relative size.

Absolute size:

Sets the text to a specified size

Does not allow a user to change the text size in all browsers (bad for accessibility reasons)

Absolute size is useful when the physical size of the output is known

Relative size:

Sets the size relative to surrounding elements

Allows a user to change the text size in browsers

To allow users to resize the text (in the browser menu), many developers use em instead of pixels.

1em is equal to the current font size. The default text size in browsers is 16px. So, the default size of 1em is 16px.

h1 {

font-size: 2.5em; /\* 40px/16=2.5em \*/

}

The text size can be set with a vw unit, which means the "viewport width".

Access Google fonts:

<head>

<link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Sofia">

<style>

body {

font-family: "Sofia";

font-size: 22px;

}

</style>

</head>

The font property is a shorthand property for:

font-style

font-variant

font-weight

font-size/line-height

font-family

Note: The font-size and font-family values are required. If one of the other values is missing, their default value are used.

All CSS Font Properties

Property Description

font Sets all the font properties in one declaration

font-family Specifies the font family for text

font-size Specifies the font size of text

font-style Specifies the font style for text

font-variant Specifies whether or not a text should be displayed in a small-caps font

font-weight Specifies the weight of a font

Bootstrap icons

<!DOCTYPE html>

<html>

<head>

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

</head>

<body>

<i class="glyphicon glyphicon-cloud"></i>

<i class="glyphicon glyphicon-remove"></i>

<i class="glyphicon glyphicon-user"></i>

<i class="glyphicon glyphicon-envelope"></i>

<i class="glyphicon glyphicon-thumbs-up"></i>

</body>

</html>

AwesomeFont

<!DOCTYPE html>

<html>

<head>

<title>Font Awesome Icons</title>

<meta name="viewport" content="width=device-width, initial-scale=1">

<script src="https://kit.fontawesome.com/a076d05399.js"></script>

<!--Get your own code at fontawesome.com-->

</head>

<body>

<p>Some Font Awesome icons:</p>

<i class="fas fa-cloud"></i>

<i class="fas fa-heart"></i>

<i class="fas fa-car"></i>

<i class="fas fa-file"></i>

<i class="fas fa-bars"></i>

<p>Styled Font Awesome icons (size and color):</p>

<i class="fas fa-cloud" style="font-size:24px;"></i>

<i class="fas fa-cloud" style="font-size:36px;"></i>

<i class="fas fa-cloud" style="font-size:48px;color:red;"></i>

<i class="fas fa-cloud" style="font-size:60px;color:lightblue;"></i>

</body>

</html>

Google icons

The four links states are:

a:link - a normal, unvisited link

a:visited - a link the user has visited

a:hover - a link when the user mouses over it

a:active - a link the moment it is clicked

/\* unvisited link \*/

a:link {

color: red;

}

The list-style-type property specifies the type of list item marker.

circle, square, upper-roman, lower-alpha

list-style-image: url('sqpurple.gif');

ul.a {

list-style-position: outside;

}

ul.b {

list-style-position: inside;

}

Remove Default Settings

The list-style-type:none property can also be used to remove the markers/bullets. Note that the list also has default margin and padding. To remove this, add margin:0 and padding:0 to <ul> or <ol>:

ul {

list-style-type: none;

margin: 0;

padding: 0;

}

List - Shorthand property

The list-style property is a shorthand property. It is used to set all the list properties in one declaration:

Example

ul {

list-style: square inside url("sqpurple.gif");

}

When using the shorthand property, the order of the property values are:

list-style-type (if a list-style-image is specified, the value of this property will be displayed if the image for some reason cannot be displayed)

list-style-position (specifies whether the list-item markers should appear inside or outside the content flow)

list-style-image (specifies an image as the list item marker)

If one of the property values above are missing, the default value for the missing property will be inserted, if any.

All CSS List Properties

Property Description

list-style Sets all the properties for a list in one declaration

list-style-image Specifies an image as the list-item marker

list-style-position Specifies the position of the list-item markers (bullet points)

list-style-type Specifies the type of list-item marker

table, th, td {

border: 1px solid black;

}

table {

border-collapse: collapse;

}

table {

width: 100%;

}

th {

height: 70px;

}

td {

height: 50px;

text-align: left

vertical-align: bottom;

}

tr:nth-child(even) {background-color: #f2f2f2;}

tr:hover {background-color: #f5f5f5;}

table {

width:100px

height: 50px

}

A responsive table will display a horizontal scroll bar if the screen is too small to display the full content:

Add a container element (like <div>) with overflow-x:auto around the <table> element to make it responsive:

Example

<div style="overflow-x:auto;">

<table>

... table content ...

</table>

</div>

The display property is the most important CSS property for controlling layout.

The display property specifies if/how an element is displayed.

Every HTML element has a default display value depending on what type of element it is. The default display value for most elements is block or inline.

Block-level Elements

A block-level element always starts on a new line and takes up the full width available (stretches out to the left and right as far as it can).

The <div> element is a block-level element.

Examples of block-level elements:

<div>

<h1> - <h6>

<p>

<form>

<header>

<footer>

<section>

Inline Elements

An inline element does not start on a new line and only takes up as much width as necessary.

This is an inline <span> element inside a paragraph.

Examples of inline elements:

<span>

<a>

<img>

display: none; is commonly used with JavaScript to hide and show elements without deleting and recreating them.

visibility: hiddn // takes up space, but will be unvisible

The position Property

The position property specifies the type of positioning method used for an element.

There are five different position values:

static

relative

fixed

absolute

sticky

Elements are then positioned using the top, bottom, left, and right properties. However, these properties will not work unless the position property is set first. They also work differently depending on the position value.

div.relative {

position: relative;

width: 400px;

height: 200px;

border: 3px solid #73AD21;

}

div.absolute {

position: absolute;

top: 80px;

right: 0;

width: 200px;

height: 100px;

border: 3px solid #73AD21;

}

div.sticky {

position: -webkit-sticky;

position: sticky;

top: 0;

padding: 5px;

background-color: #cae8ca;

border: 2px solid #4CAF50;

}

In this example, the sticky element sticks to the top of the page (top: 0), when you reach its scroll position.

When elements are positioned, they can overlap other elements.

The z-index property specifies the stack order of an element (which element should be placed in front of, or behind, the others).

img {

position: absolute;

left: 0px;

top: 0px;

z-index: -1; // -1 will be sent behind

}

All CSS Positioning Properties

Property Description

bottom Sets the bottom margin edge for a positioned box

clip Clips an absolutely positioned element

left Sets the left margin edge for a positioned box

position Specifies the type of positioning for an element

right Sets the right margin edge for a positioned box

top Sets the top margin edge for a positioned box

z-index Sets the stack order of an element

The CSS overflow property controls what happens to content that is too big to fit into an area.

The overflow property has the following values:

visible - Default. The overflow is not clipped. The content renders outside the element's box

hidden - The overflow is clipped, and the rest of the content will be invisible

scroll - The overflow is clipped, and a scrollbar is added to see the rest of the content

auto - Similar to scroll, but it adds scrollbars only when necessary

div {

overflow-x: hidden; /\* Hide horizontal scrollbar \*/

overflow-y: scroll; /\* Add vertical scrollbar \*/

}

All CSS Overflow Properties

Property Description

overflow Specifies what happens if content overflows an element's box

overflow-x Specifies what to do with the left/right edges of the content if it overflows the element's content area

overflow-y Specifies what to do with the top/bottom edges of the content if it overflows the element's content area

The float Property

The float property is used for positioning and formatting content e.g. let an image float left to the text in a container.

The float property can have one of the following values:

left - The element floats to the left of its container

right - The element floats to the right of its container

none - The element does not float (will be displayed just where it occurs in the text). This is default

inherit - The element inherits the float value of its parent

If you set box-sizing: border-box; on an element, padding and border are included in the width and height:

The @media rule, introduced in CSS2, made it possible to define different style rules for different media types.

Media queries can be used to check many things, such as:

width and height of the viewport

width and height of the device

orientation (is the tablet/phone in landscape or portrait mode?)

resolution

Using media queries are a popular technique for delivering a tailored style sheet to desktops, laptops, tablets, and mobile phones (such as iPhone and Android phones).

CSS3 Media Types

Value Description

all Used for all media type devices

print Used for printers

screen Used for computer screens, tablets, smart-phones etc.

speech Used for screenreaders that "reads" the page out loud

@media screen and (min-width: 480px) {

body {

background-color: lightgreen;

}

}

<meta name="viewport" content="width=device-width, initial-scale=1.0">

The viewport is the user's visible area of a web page.

CSS Selectors

In CSS, selectors are patterns used to select the element(s) you want to style.

Use our CSS Selector Tester to demonstrate the different selectors.

Selector Example Example description

.class .intro Selects all elements with class="intro"

.class1.class2 .name1.name2 Selects all elements with both name1 and name2 set within its class attribute

.class1 .class2 .name1 .name2 Selects all elements with name2 that is a descendant of an element with name1

#id #firstname Selects the element with id="firstname"

\* \* Selects all elements

element p Selects all <p> elements

element.class p.intro Selects all <p> elements with class="intro"

element,element div, p Selects all <div> elements and all <p> elements

element element div p Selects all <p> elements inside <div> elements

element>element div > p Selects all <p> elements where the parent is a <div> element

element+element div + p Selects all <p> elements that are placed immediately after <div> elements

element1~element2 p ~ ul Selects every <ul> element that are preceded by a <p> element

[attribute] [target] Selects all elements with a target attribute

[attribute=value] [target=\_blank] Selects all elements with target="\_blank"

[attribute~=value] [title~=flower] Selects all elements with a title attribute containing the word "flower"

[attribute|=value] [lang|=en] Selects all elements with a lang attribute value starting with "en"

[attribute^=value] a[href^="https"] Selects every <a> element whose href attribute value begins with "https"

[attribute$=value] a[href$=".pdf"] Selects every <a> element whose href attribute value ends with ".pdf"

[attribute\*=value] a[href\*="w3schools"] Selects every <a> element whose href attribute value contains the substring "w3schools"

:active a:active Selects the active link

::after p::after Insert something after the content of each <p> element

::before p::before Insert something before the content of each <p> element

:checked input:checked Selects every checked <input> element

:default input:default Selects the default <input> element

:disabled input:disabled Selects every disabled <input> element

:empty p:empty Selects every <p> element that has no children (including text nodes)

:enabled input:enabled Selects every enabled <input> element

:first-child p:first-child Selects every <p> element that is the first child of its parent

::first-letter p::first-letter Selects the first letter of every <p> element

::first-line p::first-line Selects the first line of every <p> element

:first-of-type p:first-of-type Selects every <p> element that is the first <p> element of its parent

:focus input:focus Selects the input element which has focus

:fullscreen :fullscreen Selects the element that is in full-screen mode

:hover a:hover Selects links on mouse over

:in-range input:in-range Selects input elements with a value within a specified range

:indeterminate input:indeterminate Selects input elements that are in an indeterminate state

:invalid input:invalid Selects all input elements with an invalid value

:lang(language) p:lang(it) Selects every <p> element with a lang attribute equal to "it" (Italian)

:last-child p:last-child Selects every <p> element that is the last child of its parent

:last-of-type p:last-of-type Selects every <p> element that is the last <p> element of its parent

:link a:link Selects all unvisited links

:not(selector) :not(p) Selects every element that is not a <p> element

:nth-child(n) p:nth-child(2) Selects every <p> element that is the second child of its parent

:nth-last-child(n) p:nth-last-child(2) Selects every <p> element that is the second child of its parent, counting from the last child

:nth-last-of-type(n) p:nth-last-of-type(2) Selects every <p> element that is the second <p> element of its parent, counting from the last child

:nth-of-type(n) p:nth-of-type(2) Selects every <p> element that is the second <p> element of its parent

:only-of-type p:only-of-type Selects every <p> element that is the only <p> element of its parent

:only-child p:only-child Selects every <p> element that is the only child of its parent

:optional input:optional Selects input elements with no "required" attribute

:out-of-range input:out-of-range Selects input elements with a value outside a specified range

::placeholder input::placeholder Selects input elements with the "placeholder" attribute specified

:read-only input:read-only Selects input elements with the "readonly" attribute specified

:read-write input:read-write Selects input elements with the "readonly" attribute NOT specified

:required input:required Selects input elements with the "required" attribute specified

:root :root Selects the document's root element

::selection ::selection Selects the portion of an element that is selected by a user

:target #news:target Selects the current active #news element (clicked on a URL containing that anchor name)

:valid input:valid Selects all input elements with a valid value

:visited a:visited Selects all visited links

Transition:

div {

width: 100px;

height: 100px;

background: red;

transition: width 2s;

}

div:hover {

width: 300px;

}

Transition:

div {

width: 100px;

height: 100px;

background: red;

transition: width 2s;

}

div:hover {

width: 300px;

}

The transition property is a shorthand property for:

transition-property

transition-duration

transition-timing-function

transition-delay

Property Values

Value Description

transition-property Specifies the name of the CSS property the transition effect is for

transition-duration Specifies how many seconds or milliseconds the transition effect takes to complete

transition-timing-function Specifies the speed curve of the transition effect

transition-delay Defines when the transition effect will start

initial Sets this property to its default value. Read about initial

inherit Inherits this property from its parent element. Read about inherit

input[type=text] {

width: 100px;

transition: width .35s ease-in-out;

}

input[type=text]:focus {

width: 250px;

}

CSS Properties

A

align-content

Specifies the alignment between the lines inside a flexible container when the items do not use all available space

align-items

Specifies the alignment for items inside a flexible container

align-self

Specifies the alignment for selected items inside a flexible container

all

Resets all properties (except unicode-bidi and direction)

animation

A shorthand property for all the animation-\* properties

animation-delay

Specifies a delay for the start of an animation

animation-direction

Specifies whether an animation should be played forwards, backwards or in alternate cycles

animation-duration

Specifies how long an animation should take to complete one cycle

animation-fill-mode

Specifies a style for the element when the animation is not playing (before it starts, after it ends, or both)

animation-iteration-count

Specifies the number of times an animation should be played

animation-name

Specifies a name for the @keyframes animation

animation-play-state

Specifies whether the animation is running or paused

animation-timing-function

Specifies the speed curve of an animation

B

backface-visibility

Defines whether or not the back face of an element should be visible when facing the user

background

A shorthand property for all the background-\* properties

background-attachment

Sets whether a background image scrolls with the rest of the page, or is fixed

background-blend-mode

Specifies the blending mode of each background layer (color/image)

background-clip

Defines how far the background (color or image) should extend within an element

background-color

Specifies the background color of an element

background-image

Specifies one or more background images for an element

background-origin

Specifies the origin position of a background image

background-position

Specifies the position of a background image

background-repeat

Sets if/how a background image will be repeated

background-size

Specifies the size of the background images

border

A shorthand property for border-width, border-style and border-color

border-bottom

A shorthand property for border-bottom-width, border-bottom-style and border-bottom-color

border-bottom-color

Sets the color of the bottom border

border-bottom-left-radius

Defines the radius of the border of the bottom-left corner

border-bottom-right-radius

Defines the radius of the border of the bottom-right corner

border-bottom-style

Sets the style of the bottom border

border-bottom-width

Sets the width of the bottom border

border-collapse

Sets whether table borders should collapse into a single border or be separated

border-color

Sets the color of the four borders

border-image

A shorthand property for all the border-image-\* properties

border-image-outset

Specifies the amount by which the border image area extends beyond the border box

border-image-repeat

Specifies whether the border image should be repeated, rounded or stretched

border-image-slice

Specifies how to slice the border image

border-image-source

Specifies the path to the image to be used as a border

border-image-width

Specifies the width of the border image

border-left

A shorthand property for all the border-left-\* properties

border-left-color

Sets the color of the left border

border-left-style

Sets the style of the left border

border-left-width

Sets the width of the left border

border-radius

A shorthand property for the four border-\*-radius properties

border-right

A shorthand property for all the border-right-\* properties

border-right-color

Sets the color of the right border

border-right-style

Sets the style of the right border

border-right-width

Sets the width of the right border

border-spacing

Sets the distance between the borders of adjacent cells

border-style

Sets the style of the four borders

border-top

A shorthand property for border-top-width, border-top-style and border-top-color

border-top-color

Sets the color of the top border

border-top-left-radius

Defines the radius of the border of the top-left corner

border-top-right-radius

Defines the radius of the border of the top-right corner

border-top-style

Sets the style of the top border

border-top-width

Sets the width of the top border

border-width

Sets the width of the four borders

bottom

Sets the elements position, from the bottom of its parent element

box-decoration-break

Sets the behavior of the background and border of an element at page-break, or, for in-line elements, at line-break.

box-shadow

Attaches one or more shadows to an element

box-sizing

Defines how the width and height of an element are calculated: should they include padding and borders, or not

break-after

Specifies whether or not a page-, column-, or region-break should occur after the specified element

break-before

Specifies whether or not a page-, column-, or region-break should occur before the specified element

break-inside

Specifies whether or not a page-, column-, or region-break should occur inside the specified element

C

caption-side

Specifies the placement of a table caption

caret-color

Specifies the color of the cursor (caret) in inputs, textareas, or any element that is editable

@charset

Specifies the character encoding used in the style sheet

clear

Specifies on which sides of an element floating elements are not allowed to float

clip

Clips an absolutely positioned element

color

Sets the color of text

column-count

Specifies the number of columns an element should be divided into

column-fill

Specifies how to fill columns, balanced or not

column-gap

Specifies the gap between the columns

column-rule

A shorthand property for all the column-rule-\* properties

column-rule-color

Specifies the color of the rule between columns

column-rule-style

Specifies the style of the rule between columns

column-rule-width

Specifies the width of the rule between columns

column-span

Specifies how many columns an element should span across

column-width

Specifies the column width

columns

A shorthand property for column-width and column-count

content

Used with the :before and :after pseudo-elements, to insert generated content

counter-increment

Increases or decreases the value of one or more CSS counters

counter-reset

Creates or resets one or more CSS counters

cursor

Specifies the mouse cursor to be displayed when pointing over an element

D

direction

Specifies the text direction/writing direction

display

Specifies how a certain HTML element should be displayed

E

empty-cells

Specifies whether or not to display borders and background on empty cells in a table

F

filter

Defines effects (e.g. blurring or color shifting) on an element before the element is displayed

flex

A shorthand property for the flex-grow, flex-shrink, and the flex-basis properties

flex-basis

Specifies the initial length of a flexible item

flex-direction

Specifies the direction of the flexible items

flex-flow

A shorthand property for the flex-direction and the flex-wrap properties

flex-grow

Specifies how much the item will grow relative to the rest

flex-shrink

Specifies how the item will shrink relative to the rest

flex-wrap

Specifies whether the flexible items should wrap or not

float

Specifies whether or not a box should float

font

A shorthand property for the font-style, font-variant, font-weight, font-size/line-height, and the font-family properties

@font-face

A rule that allows websites to download and use fonts other than the "web-safe" fonts

font-family

Specifies the font family for text

font-feature-settings

Allows control over advanced typographic features in OpenType fonts

@font-feature-values Allows authors to use a common name in font-variant-alternate for feature activated differently in OpenType

font-kerning

Controls the usage of the kerning information (how letters are spaced)

font-language-override Controls the usage of language-specific glyphs in a typeface

font-size

Specifies the font size of text

font-size-adjust

Preserves the readability of text when font fallback occurs

font-stretch

Selects a normal, condensed, or expanded face from a font family

font-style

Specifies the font style for text

font-synthesis Controls which missing typefaces (bold or italic) may be synthesized by the browser

font-variant

Specifies whether or not a text should be displayed in a small-caps font

font-variant-alternates Controls the usage of alternate glyphs associated to alternative names defined in @font-feature-values

font-variant-caps

Controls the usage of alternate glyphs for capital letters

font-variant-east-asian Controls the usage of alternate glyphs for East Asian scripts (e.g Japanese and Chinese)

font-variant-ligatures Controls which ligatures and contextual forms are used in textual content of the elements it applies to

font-variant-numeric Controls the usage of alternate glyphs for numbers, fractions, and ordinal markers

font-variant-position Controls the usage of alternate glyphs of smaller size positioned as superscript or subscript regarding the baseline of the font

font-weight

Specifies the weight of a font

G

grid

A shorthand property for the grid-template-rows, grid-template-columns, grid-template-areas, grid-auto-rows, grid-auto-columns, and the grid-auto-flow properties

grid-area

Either specifies a name for the grid item, or this property is a shorthand property for the grid-row-start, grid-column-start, grid-row-end, and grid-column-end properties

grid-auto-columns

Specifies a default column size

grid-auto-flow

Specifies how auto-placed items are inserted in the grid

grid-auto-rows

Specifies a default row size

grid-column

A shorthand property for the grid-column-start and the grid-column-end properties

grid-column-end

Specifies where to end the grid item

grid-column-gap

Specifies the size of the gap between columns

grid-column-start

Specifies where to start the grid item

grid-gap

A shorthand property for the grid-row-gap and grid-column-gap properties

grid-row

A shorthand property for the grid-row-start and the grid-row-end properties

grid-row-end

Specifies where to end the grid item

grid-row-gap

Specifies the size of the gap between rows

grid-row-start

Specifies where to start the grid item

grid-template

A shorthand property for the grid-template-rows, grid-template-columns and grid-areas properties

grid-template-areas

Specifies how to display columns and rows, using named grid items

grid-template-columns

Specifies the size of the columns, and how many columns in a grid layout

grid-template-rows

Specifies the size of the rows in a grid layout

H

hanging-punctuation

Specifies whether a punctuation character may be placed outside the line box

height

Sets the height of an element

hyphens

Sets how to split words to improve the layout of paragraphs

I

image-rendering Gives a hint to the browser about what aspects of an image are most important to preserve when the image is scaled

@import

Allows you to import a style sheet into another style sheet

isolation

Defines whether an element must create a new stacking content

J

justify-content

Specifies the alignment between the items inside a flexible container when the items do not use all available space

K

@keyframes

Specifies the animation code

L

left

Specifies the left position of a positioned element

letter-spacing

Increases or decreases the space between characters in a text

line-break Specifies how/if to break lines

line-height

Sets the line height

list-style

Sets all the properties for a list in one declaration

list-style-image

Specifies an image as the list-item marker

list-style-position

Specifies the position of the list-item markers (bullet points)

list-style-type

Specifies the type of list-item marker

M

margin

Sets all the margin properties in one declaration

margin-bottom

Sets the bottom margin of an element

margin-left

Sets the left margin of an element

margin-right

Sets the right margin of an element

margin-top

Sets the top margin of an element

mask Hides an element by masking or clipping the image at specific places

mask-type Specifies whether a mask element is used as a luminance or an alpha mask

max-height

Sets the maximum height of an element

max-width

Sets the maximum width of an element

@media

Sets the style rules for different media types/devices/sizes

min-height

Sets the minimum height of an element

min-width

Sets the minimum width of an element

mix-blend-mode

Specifies how an element's content should blend with its direct parent background

O

object-fit

Specifies how the contents of a replaced element should be fitted to the box established by its used height and width

object-position

Specifies the alignment of the replaced element inside its box

opacity

Sets the opacity level for an element

order

Sets the order of the flexible item, relative to the rest

orphans Sets the minimum number of lines that must be left at the bottom of a page when a page break occurs inside an element

outline

A shorthand property for the outline-width, outline-style, and the outline-color properties

outline-color

Sets the color of an outline

outline-offset

Offsets an outline, and draws it beyond the border edge

outline-style

Sets the style of an outline

outline-width

Sets the width of an outline

overflow

Specifies what happens if content overflows an element's box

overflow-wrap Specifies whether or not the browser may break lines within words in order to prevent overflow (when a string is too long to fit its containing box)

overflow-x

Specifies whether or not to clip the left/right edges of the content, if it overflows the element's content area

overflow-y

Specifies whether or not to clip the top/bottom edges of the content, if it overflows the element's content area

P

padding

A shorthand property for all the padding-\* properties

padding-bottom

Sets the bottom padding of an element

padding-left

Sets the left padding of an element

padding-right

Sets the right padding of an element

padding-top

Sets the top padding of an element

page-break-after

Sets the page-break behavior after an element

page-break-before

Sets the page-break behavior before an element

page-break-inside

Sets the page-break behavior inside an element

perspective

Gives a 3D-positioned element some perspective

perspective-origin

Defines at which position the user is looking at the 3D-positioned element

pointer-events

Defines whether or not an element reacts to pointer events

position

Specifies the type of positioning method used for an element (static, relative, absolute or fixed)

Q

quotes

Sets the type of quotation marks for embedded quotations

R

resize

Defines if (and how) an element is resizable by the user

right

Specifies the right position of a positioned element

S

scroll-behavior

Specifies whether to smoothly animate the scroll position in a scrollable box, instead of a straight jump

T

tab-size

Specifies the width of a tab character

table-layout

Defines the algorithm used to lay out table cells, rows, and columns

text-align

Specifies the horizontal alignment of text

text-align-last

Describes how the last line of a block or a line right before a forced line break is aligned when text-align is "justify"

text-combine-upright Specifies the combination of multiple characters into the space of a single character

text-decoration

Specifies the decoration added to text

text-decoration-color

Specifies the color of the text-decoration

text-decoration-line

Specifies the type of line in a text-decoration

text-decoration-style

Specifies the style of the line in a text decoration

text-indent

Specifies the indentation of the first line in a text-block

text-justify

Specifies the justification method used when text-align is "justify"

text-orientation Defines the orientation of the text in a line

text-overflow

Specifies what should happen when text overflows the containing element

text-shadow

Adds shadow to text

text-transform

Controls the capitalization of text

text-underline-position Specifies the position of the underline which is set using the text-decoration property

top

Specifies the top position of a positioned element

transform

Applies a 2D or 3D transformation to an element

transform-origin

Allows you to change the position on transformed elements

transform-style

Specifies how nested elements are rendered in 3D space

transition

A shorthand property for all the transition-\* properties

transition-delay

Specifies when the transition effect will start

transition-duration

Specifies how many seconds or milliseconds a transition effect takes to complete

transition-property

Specifies the name of the CSS property the transition effect is for

transition-timing-function

Specifies the speed curve of the transition effect

U

unicode-bidi

Used together with the direction property to set or return whether the text should be overridden to support multiple languages in the same document

user-select

Specifies whether the text of an element can be selected

V

vertical-align

Sets the vertical alignment of an element

visibility

Specifies whether or not an element is visible

W

white-space

Specifies how white-space inside an element is handled

widows Sets the minimum number of lines that must be left at the top of a page when a page break occurs inside an element

width

Sets the width of an element

word-break

Specifies how words should break when reaching the end of a line

word-spacing

Increases or decreases the space between words in a text

word-wrap

Allows long, unbreakable words to be broken and wrap to the next line

writing-mode

Specifies whether lines of text are laid out horizontally or vertically

Z

z-index

Sets the stack order of a positioned element

---

CSS PropertiesOrder by Category

The following section contains a complete list of standard properties belonging to the latest CSS3 specifications. All the properties are listed alphabetically.

Property Description

align-content

Specifies the alignment of flexible container's items within the flex container.

align-items

Specifies the default alignment for items within the flex container.

align-self

Specifies the alignment for selected items within the flex container.

animation

Specifies the keyframe-based animations.

animation-delay

Specifies when the animation will start.

animation-direction

Specifies whether the animation should play in reverse on alternate cycles or not.

animation-duration

Specifies the number of seconds or milliseconds an animation should take to complete one cycle.

animation-fill-mode

Specifies how a CSS animation should apply styles to its target before and after it is executing.

animation-iteration-count

Specifies the number of times an animation cycle should be played before stopping.

animation-name

Specifies the name of @keyframes defined animations that should be applied to the selected element.

animation-play-state

Specifies whether the animation is running or paused.

animation-timing-function

Specifies how a CSS animation should progress over the duration of each cycle.

backface-visibility

Specifies whether or not the "back" side of a transformed element is visible when facing the user.

background

Defines a variety of background properties within one declaration.

background-attachment

Specify whether the background image is fixed in the viewport or scrolls.

background-clip

Specifies the painting area of the background.

background-color

Defines an element's background color.

background-image

Defines an element's background image.

background-origin

Specifies the positioning area of the background images.

background-position

Defines the origin of a background image.

background-repeat

Specify whether/how the background image is tiled.

background-size

Specifies the size of the background images.

border

Sets the width, style, and color for all four sides of an element's border.

border-bottom

Sets the width, style, and color of the bottom border of an element.

border-bottom-color

Sets the color of the bottom border of an element.

border-bottom-left-radius

Defines the shape of the bottom-left border corner of an element.

border-bottom-right-radius

Defines the shape of the bottom-right border corner of an element.

border-bottom-style

Sets the style of the bottom border of an element.

border-bottom-width

Sets the width of the bottom border of an element.

border-collapse

Specifies whether table cell borders are connected or separated.

border-color

Sets the color of the border on all the four sides of an element.

border-image

Specifies how an image is to be used in place of the border styles.

border-image-outset

Specifies the amount by which the border image area extends beyond the border box.

border-image-repeat

Specifies whether the image-border should be repeated, rounded or stretched.

border-image-slice

Specifies the inward offsets of the image-border.

border-image-source

Specifies the location of the image to be used as a border.

border-image-width

Specifies the width of the image-border.

border-left

Sets the width, style, and color of the left border of an element.

border-left-color

Sets the color of the left border of an element.

border-left-style

Sets the style of the left border of an element.

border-left-width

Sets the width of the left border of an element.

border-radius

Defines the shape of the border corners of an element.

border-right

Sets the width, style, and color of the right border of an element.

border-right-color

Sets the color of the right border of an element.

border-right-style

Sets the style of the right border of an element.

border-right-width

Sets the width of the right border of an element.

border-spacing

Sets the spacing between the borders of adjacent table cells.

border-style

Sets the style of the border on all the four sides of an element.

border-top

Sets the width, style, and color of the top border of an element.

border-top-color

Sets the color of the top border of an element.

border-top-left-radius

Defines the shape of the top-left border corner of an element.

border-top-right-radius

Defines the shape of the top-right border corner of an element.

border-top-style

Sets the style of the top border of an element.

border-top-width

Sets the width of the top border of an element.

border-width

Sets the width of the border on all the four sides of an element.

bottom

Specify the location of the bottom edge of the positioned element.

box-shadow

Applies one or more drop-shadows to the element's box.

box-sizing

Alter the default CSS box model.

caption-side

Specify the position of table's caption.

clear

Specifies the placement of an element in relation to floating elements.

clip

Defines the clipping region.

color

Specify the color of the text of an element.

column-count

Specifies the number of columns in a multi-column element.

column-fill

Specifies how columns will be filled.

column-gap

Specifies the gap between the columns in a multi-column element.

column-rule

Specifies a straight line, or "rule", to be drawn between each column in a multi-column element.

column-rule-color

Specifies the color of the rules drawn between columns in a multi-column layout.

column-rule-style

Specifies the style of the rule drawn between the columns in a multi-column layout.

column-rule-width

Specifies the width of the rule drawn between the columns in a multi-column layout.

column-span

Specifies how many columns an element spans across in a multi-column layout.

column-width

Specifies the optimal width of the columns in a multi-column element.

columns

A shorthand property for setting column-width and column-count properties.

content

Inserts generated content.

counter-increment

Increments one or more counter values.

counter-reset

Creates or resets one or more counters.

cursor

Specify the type of cursor.

direction

Define the text direction/writing direction.

display

Specifies how an element is displayed onscreen.

empty-cells

Show or hide borders and backgrounds of empty table cells.

flex

Specifies the components of a flexible length.

flex-basis

Specifies the initial main size of the flex item.

flex-direction

Specifies the direction of the flexible items.

flex-flow

A shorthand property for the flex-direction and the flex-wrap properties.

flex-grow

Specifies how the flex item will grow relative to the other items inside the flex container.

flex-shrink

Specifies how the flex item will shrink relative to the other items inside the flex container.

flex-wrap

Specifies whether the flexible items should wrap or not.

float

Specifies whether or not a box should float.

font

Defines a variety of font properties within one declaration.

font-family

Defines a list of fonts for element.

font-size

Defines the font size for the text.

font-size-adjust

Preserves the readability of text when font fallback occurs.

font-stretch

Selects a normal, condensed, or expanded face from a font.

font-style

Defines the font style for the text.

font-variant

Specify the font variant.

font-weight

Specify the font weight of the text.

height

Specify the height of an element.

justify-content

Specifies how flex items are aligned along the main axis of the flex container after any flexible lengths and auto margins have been resolved.

left

Specify the location of the left edge of the positioned element.

letter-spacing

Sets the extra spacing between letters.

line-height

Sets the height between lines of text.

list-style

Defines the display style for a list and list elements.

list-style-image

Specifies the image to be used as a list-item marker.

list-style-position

Specifies the position of the list-item marker.

list-style-type

Specifies the marker style for a list-item.

margin

Sets the margin on all four sides of the element.

margin-bottom

Sets the bottom margin of the element.

margin-left

Sets the left margin of the element.

margin-right

Sets the right margin of the element.

margin-top

Sets the top margin of the element.

max-height

Specify the maximum height of an element.

max-width

Specify the maximum width of an element.

min-height

Specify the minimum height of an element.

min-width

Specify the minimum width of an element.

opacity

Specifies the transparency of an element.

order

Specifies the order in which a flex items are displayed and laid out within a flex container.

outline

Sets the width, style, and color for all four sides of an element's outline.

outline-color

Sets the color of the outline.

outline-offset

Set the space between an outline and the border edge of an element.

outline-style

Sets a style for an outline.

outline-width

Sets the width of the outline.

overflow

Specifies the treatment of content that overflows the element's box.

overflow-x

Specifies the treatment of content that overflows the element's box horizontally.

overflow-y

Specifies the treatment of content that overflows the element's box vertically.

padding

Sets the padding on all four sides of the element.

padding-bottom

Sets the padding to the bottom side of an element.

padding-left

Sets the padding to the left side of an element.

padding-right

Sets the padding to the right side of an element.

padding-top

Sets the padding to the top side of an element.

page-break-after

Insert a page breaks after an element.

page-break-before

Insert a page breaks before an element.

page-break-inside

Insert a page breaks inside an element.

perspective

Defines the perspective from which all child elements of the object are viewed.

perspective-origin

Defines the origin (the vanishing point for the 3D space) for the perspective property.

position

Specifies how an element is positioned.

quotes

Specifies quotation marks for embedded quotations.

resize

Specifies whether or not an element is resizable by the user.

right

Specify the location of the right edge of the positioned element.

tab-size

Specifies the length of the tab character.

table-layout

Specifies a table layout algorithm.

text-align

Sets the horizontal alignment of inline content.

text-align-last

Specifies how the last line of a block or a line right before a forced line break is aligned when text-align is justify.

text-decoration

Specifies the decoration added to text.

text-decoration-color

Specifies the color of the text-decoration-line.

text-decoration-line

Specifies what kind of line decorations are added to the element.

text-decoration-style

Specifies the style of the lines specified by the text-decoration-line property

text-indent

Indent the first line of text.

text-justify

Specifies the justification method to use when the text-align property is set to justify.

text-overflow

Specifies how the text content will be displayed, when it overflows the block containers.

text-shadow

Applies one or more shadows to the text content of an element.

text-transform

Transforms the case of the text.

top

Specify the location of the top edge of the positioned element.

transform

Applies a 2D or 3D transformation to an element.

transform-origin

Defines the origin of transformation for an element.

transform-style

Specifies how nested elements are rendered in 3D space.

transition

Defines the transition between two states of an element.

transition-delay

Specifies when the transition effect will start.

transition-duration

Specifies the number of seconds or milliseconds a transition effect should take to complete.

transition-property

Specifies the names of the CSS properties to which a transition effect should be applied.

transition-timing-function

Specifies the speed curve of the transition effect.

vertical-align

Sets the vertical positioning of an element relative to the current text baseline.

visibility

Specifies whether or not an element is visible.

white-space

Specifies how white space inside the element is handled.

width

Specify the width of an element.

word-break

Specifies how to break lines within words.

word-spacing

Sets the spacing between words.

word-wrap

Specifies whether to break words when the content overflows the boundaries of its container.

z-index

Specifies a layering or stacking order for positioned elements.

----

https://www.w3.org/TR/CSS/#css

https://www.w3.org/TR/CSS22/

https://www.w3.org/TR/CSS21/propidx.html

https://www.w3.org/TR/2011/REC-CSS2-20110607/#minitoc

For JavaScript Tutorial: https://www.w3schools.com/js/default.asp

https://www.w3resource.com/css/CSS-values.php

cssjiji