**React**

* React is a javascript library for building user interfaces.
* It runs in the browser not in server so it is fast.
* Jquery is all about traversing the DOM and targeting the elements in the dom.
* ***Jsx: Writing html in javascript***.
* ***ReactDOM***: which converts jsx into dom.

Creating a react app:

**create-react-app react-complete-guide --scripts-version 1.1.5**

**rfce**

npx create-react-app my-app

npm install --save react-router-dom

npm install --save redux

npm install --save react-redux

npm install --save redux-thunk

npm install @material-ui/core

npm i @material-ui/icons -s

npm install axios –save

npm install --save enzyme

npm install --save enzyme react-test-renderer enzyme-adapter-react-16

**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:

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)

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.

**Props**: A special feature of react which helps us to pass data(attributes and its values ) from one component to other component

**Props Children**: props.children mainly used to get the data between the tags(<Person>Data</Person>)

**state**: A special variable or property which can only used by any component extends Component from React and state is managed inside a component.

state = {

persons:[

{name:'vishal',age:24},

{name:'vissu',age:25},

{name:'max',age:31}

]

}

If state changes (any data in the state variable changes) it will lead react to re-render our DOM or update the DOM and that's the special thing about it and it only works on that state property. It basically analyzes the code it already rendered to the DOM and the code it would now render if it were to re-render everything and then it updates the existing DOM in all the places where it needs to update it to reflect your new state and props.

**Note**: When state changes only one field then after saving react won’t change the entire state, it just overrides what has changed and rest remains as it is.

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!

 switchNameHandler = ()=>{

    // console.log('was clicked');

    // Dont do these this.state.persons[0].name = 'vishal palla';

    this.setState({

      persons:[

        {name:'vishal palla',age:24},

        {name:'vissu',age:25},

        {name:'max',age:33}

      ]

    })

  }

A convention to give this a name like the following Switch Name handler maybe.

Now the first part Switch Name is totally up to you but you typically use handler here to indicate that this is a method you're not actively calling but you're assigning as an event handler.

So Switch Name handler now should be a function. Now if you just said equal, right now it is just the same syntax as for the state property but if we assign a function as a value here it becomes a method basically. It still is a property you could say but a property which holds a function which can be executed.

**React Hooks**

React hooks feature looks like and React hooks is really just the name of it, it's basically a collection of functions exposed to you by React which you can use in functional components and therefore, we'll have to convert this component here into a functional one.

Note:

**container** or **stateful** or **smart** component for the stateful components.

**Presentational** or **stateless** or **dumb** component for the functional components.

You can pass methods also as props so that you can call a method which might change the state in another component which doesn't have direct access to the state and which shouldn't have direct access to the state.

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

        click={this.switchNameHandler}>My Hobbies:Climbing trees</Person>

const person = (props)=>{

  return (

  <p onClick={props.click}>I'am {props.name}! and iam {props.age} old, {props.children}</p>

  )

};

npm install --save radium

npm run eject

{

                loader: require.resolve('css-loader'),

                options: {

                  importLoaders: 1,

                  modules:true,

                  localIdentName: '[name]\_\_[local]\_\_[hash:base64:5]'

                },

              }

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.

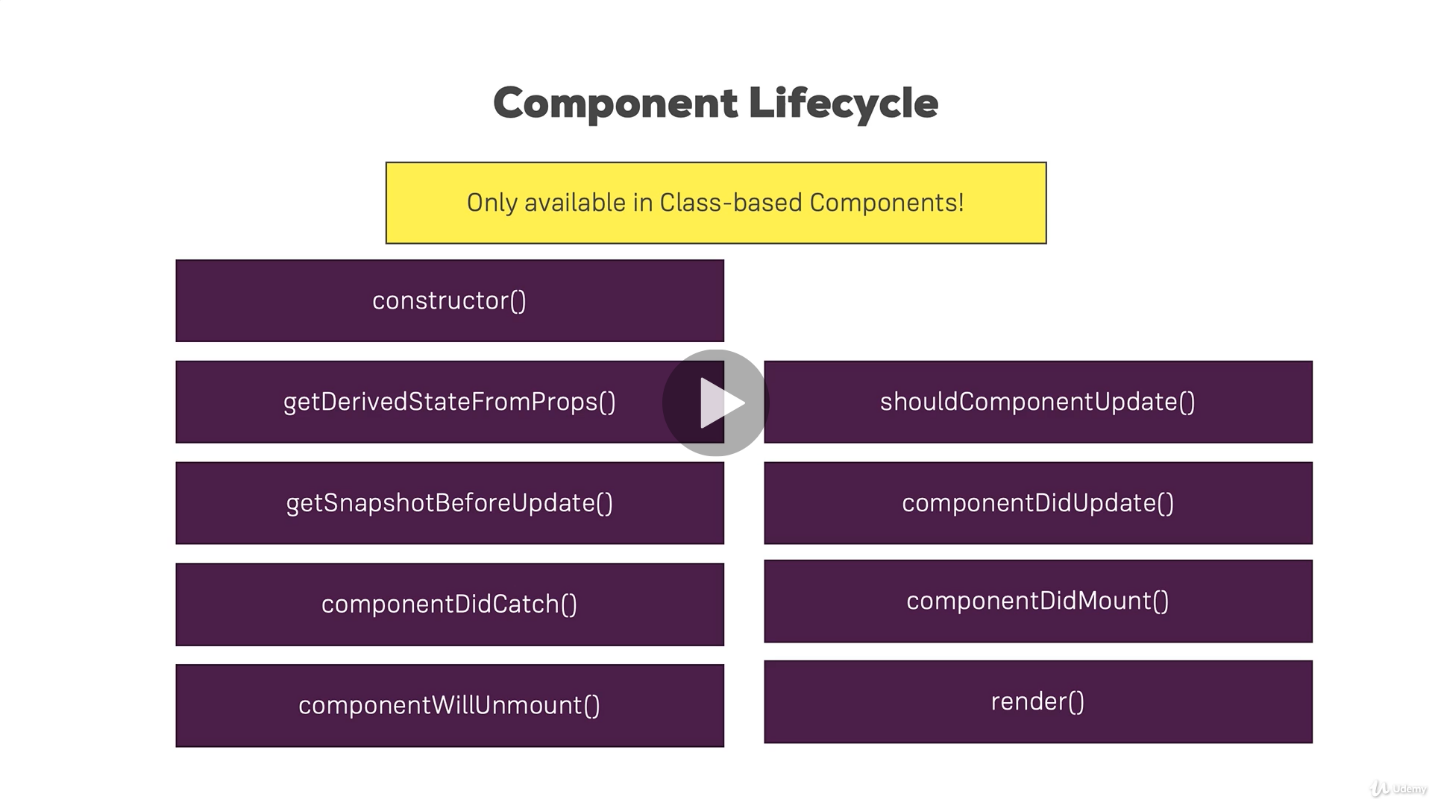
By the way, if you somehow also want to define a global (i.e. un-transformed) CSS class in such a .css  file, you can prefix the selector with :global .

**Example:**

:global .Post { ... }

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

Graphical user interface

Description automatically generatedDiagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

import React,{Component} from 'react'

import Person from './Person/Person'

class Persons extends Component{

   state = {

     persons:[]

   }

    finalresult =()=>{

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

          return (

            <Person>

              <h1>name:{per.name.title +per.name.first+per.name.last }</h1>

              <h1>gender:{per.gender}</h1>

            </Person>

          )

        })}

   async componentDidMount(){

    const results = await fetch(`https://randomuser.me/api/`)

    const data = await results.json();

    this.setState({persons:data.results})

   }

  render(){

    let test = this.state.persons.map((per,index)=>{

      return (

        <Person>

          <h1>name:{per.name.title +per.name.first+per.name.last }</h1>

          <h1>gender:{per.gender}</h1>

        </Person>

      )

    })

    return(

      <div>

        This is persons component

        {test}

        {this.finalresult()}

        <Person>{this.state.persons.map((per,index)=>{

          return (

            <div>

              <h1>name:{per.name.title +per.name.first+per.name.last }</h1>

              <h1>gender:{per.gender}</h1>

            </div>

          )

        })}

        </Person>

      </div>

    )

  }

}

export default Persons

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 .

**There's no better or worse way of creating Link paths** - choose the one you need. Sometimes, you want to ensure that you always load the same path, no matter on which path you already are => Use absolute paths in this scenario.

Use relative paths if you want to navigate relative to your existing path.

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 .

**Burgerproject database**

use burgerproject;

select \* from address;

select \* from customer c join address a on c.address\_id = a.id;

select \* from orderburger where orderburger.customer\_id=1;

create table address(id BIGINT primary key auto\_increment,

street varchar(100) not null,zipcode varchar(100),

country varchar(20))auto\_increment=100;

insert into address(street,zipcode,country)

values('23-135/2','531116','India'),('53-35/2','530002','India');

create table customer(id BIGINT primary key auto\_increment,customer varchar(100) not null,email varchar(100),

phonenumber varchar(20),address\_id BIGINT,

FOREIGN KEY(address\_id) references address(id))auto\_increment=1;

insert into customer(customer,email,phonenumber,address\_id)

values('vishal','vishalpalla27@gmail.com','8143486643',100),

('ananya','ananya@gmail.com','9603091962',101);

create table orderburger(id BIGINT primary key auto\_increment,

ingredients varchar(100) not null,

price int not null,

delivery varchar(20) not null,

customer\_id bigint,

FOREIGN KEY(customer\_id) references customer(id)

)auto\_increment=2000;

insert into orderburger(ingredients,price,delivery,customer\_id)

values('salad,bacon,chesse,meat',20,'fastest',1),

('chesse,meat',15,'moderate',2)

CREATE DEFINER=`root`@`localhost` PROCEDURE `placeOrder`(customer varchar(100),email varchar(100),country varchar(100),

street varchar(100),zipcode varchar(100),delivery varchar(100),price FLOAT,food varchar(100))

BEGIN

DECLARE cus\_id int ;

DECLARE add\_id int;

SELECT c.id INTO cus\_id from customer c where c.customer = customer;

IF(CUS\_ID IS NULL)

THEN

INSERT INTO address(street,zipcode,country)

SELECT street,zipcode,country;

SELECT a.id INTO add\_id from address a order by id desc limit 1;

INSERT INTO customer(customer,email,address\_id)

SELECT customer,email,add\_id;

SELECT c.id INTO cus\_id from customer c order by id desc limit 1;

END IF;

INSERT INTO orderburger(ingredients,price,delivery,customer\_id)

SELECT food,price,delivery,cus\_id ;

SELECT 'success';

END

Diagram

Description automatically generated

