**SESSAO 2**

Aula 07

1. Iremos utilizar a ferramenta creat-react-app para realizarmos a configuração de nossos projetos.
2. Criar o projeto com : npx create-react-app projeto1 e será criado a pasta projeto1 com os arquivos do projeto. Caso queira instalar os arquivos sem criar a pasta, basta digitar : npx create-react-app . .

Aula 08

1. Vamos olhar o package.json as dependências instaladas, olhando as pastas e arquivos do projeto.
2. Teste o projeto inicializado digite : npm start

Aula 09

1. Execute o npm run build que será criado uma pasta no projeto na versão de deploy.

Aula 11

1. O react não precisa estar em toda a página.

Aula 12

1. Componentes funcionais
2. import logo from './logo.svg';
3. import './App.css';
4. function App() {
5. return (
6. <div *className*="App">
7. <header *className*="App-header">
8. <img *src*={logo} *className*="App-logo" *alt*="logo" />
9. <p>
10. Edit <code>src/App.js</code> and save to reload.
11. </p>
12. <a
13. *className*="App-link"
14. *href*="https://reactjs.org"
15. *target*="\_blank"
16. *rel*="noopener noreferrer"
17. >
18. Learn React
19. </a>
20. </header>
21. </div>
22. );
23. }
24. export default App;
25. e componentes de classe.
26. import logo from './logo.svg';
27. import './App.css';
28. import {Component} from 'react'
29. class App extends *Component*{
30. render(){
31. return (
32. <div *className*="App">
33. <header *className*="App-header">
34. <img *src*={logo} *className*="App-logo" *alt*="logo" />
35. <p>
36. Edit <code>src/App.js</code> and save to reload.
37. </p>
38. <a
39. *className*="App-link"
40. *href*="https://reactjs.org"
41. *target*="\_blank"
42. *rel*="noopener noreferrer"
43. >
44. Learn React
45. </a>
46. </header>
47. </div>
48. );
49. }
50. }
51. export default App;

Aula 13

1. Estado de componentes.
2. São basicamente dados que o componente utiliza, mas toda vez que o estado mudar dizemos para o react para renderizar novamente o componente.
3. Vamos criar um construct, no react o construct sempre recebe props como parâmetro.
4. Devemos chamar o super() pois estamos estendendo da classe mãe Component.
5. Utilizamos o this.state para passar o state do componente variáveis ou funções que iremos utilizar na página. Por exemplo nos criamos uma variável name que recebe um valor. Dentro do render() nós utilizamos um destruct para pegar a variável salva dentro do state e esse valor é passado para uma variável dentro do html.
6. Utilizamos o método onClick e passamos uma arrow function para ele de maneira que quando clicarmos na variável, irá aparecer no console <p> Clicado;
7. constructor(*props*){
8. *super*(props);
9. *this*.state = {
10. name: 'Claudisnei Bello'
11. };
12. }
13. render(){
14. //const name = this.state.name;
15. const {name} = *this*.state;
16. return (
17. <div *className*="App">
18. <header *className*="App-header">
19. <img *src*={logo} *className*="App-logo" *alt*="logo" />
20. <p *onClick*={()=>{console.log('<p> Clicado') }}>
21. {name};
22. </p>
23. Como estamos dentro de uma classe, é como criarmos um método dentro da classe para executar uma ação.
24. Vamos criar então a função HandlePClick e vamos passar o mesmo conteúdo que estava dentro de onClick.
25. Agora basta apagarmos o conteúdo de onClick e chamarmos o método da classe com this.HandlePClick. isso funcionará da mesma maneira.
26. HandlePClick(){
27. console.log('<p> Clicado');
28. }
29. render(){
30. //const name = this.state.name;
31. const {name} = *this*.state;
32. return (
33. <div *className*="App">
34. <header *className*="App-header">
35. <img *src*={logo} *className*="App-logo" *alt*="logo" />
36. <p *onClick*={*this*.HandlePClick}>
37. {name};
38. </p>
39. Para podermos utilizar as variáveis que estão sendo declaradas no construtor, dentro dos métodos, devemos byndar o método no construtor
40. constructor(*props*){
41. *super*(props);
42. *this*.HandlePClick = *this*.HandlePClick.bind(*this*);
43. *this*.state = {
44. name: 'Claudisnei Bello'
45. };
46. }
47. HandlePClick(){
48. const {name} = *this*.state;
49. console.log(`<p> clicado ${name}`);
50. Para alteramos o valor das variáveis no state temos que utilizar o método this.setState{}.
51. HandlePClick(){
52. //const {name} = this.state;
53. // console.log(`<p> clicado ${name}`);
54. *this*.setState({name: 'Débora Sobrinho'});
55. }
56. Vamos criar outro método porém agora como arrow function, pois desta forma não precisamos byndar o método no construtor.
57. Iremos criar um método para que ao clicarmos no link da página, o link seja alterado para outro.
58. Porém devemos passar para função o parâmetro event e dentro da função devemos chamar o event.preventDefault(), desta forma ao clicarmos no link ele não executara mais o que está na página, mas irá executar o que esta dentro da função.
59. Devemos pegar o counter no destruct do this.state e passar essar variável em qualquer lugar dentro da página.
60. HandleAClick=(*event*)=>{
61. event.preventDefault();
62. const {counter} = *this*.state;
63. *this*.setState({counter : counter + 1});
64. }
65. render(){
66. //const name = this.state.name;
67. const {name, counter} = *this*.state;
68. return (
69. <div *className*="App">
70. <header *className*="App-header">
71. <img *src*={logo} *className*="App-logo" *alt*="logo" />
72. <p *onClick*={*this*.HandlePClick}>
73. {name}{counter};
74. </p>
75. <a
76. Para melhorar podemos eliminar a utilização do construct com a class Fields.
77. Ela nos permite criar atributos sem precisar utilizar o construtor.
78. import logo from './logo.svg';
79. import './App.css';
80. import {Component} from 'react'
81. class App extends *Component*{
82. // constructor(props){
83. //  super(props);
84. //   this.HandlePClick = this.HandlePClick.bind(this);
85. //  this.state = {
86. state ={
87. name: 'Claudisnei Bello',
88. counter: 0
89. };
91. HandlePClick=()=>{
92. //const {name} = this.state;
93. // console.log(`<p> clicado ${name}`);
94. *this*.setState({name: 'Débora Sobrinho'});
95. }
96. HandleAClick=(*event*)=>{
97. event.preventDefault();
98. const {counter} = *this*.state;
99. *this*.setState({counter : counter + 1});
100. }
101. render(){
102. //const name = this.state.name;
103. const {name, counter} = *this*.state;
104. return (
105. <div *className*="App">
106. <header *className*="App-header">
107. <img *src*={logo} *className*="App-logo" *alt*="logo" />
108. <p *onClick*={*this*.HandlePClick}>
109. {name} {counter};
110. </p>
111. <a
112. *onClick*={*this*.HandleAClick}
113. *className*="App-link"
114. *href*="https://reactjs.org"
115. *target*="\_blank"
116. *rel*="noopener noreferrer"
117. >
118. Learn React
119. </a>
120. </header>
121. </div>
122. );
123. }
124. }
125. export default App;

Aula 15 – Array dentro do State

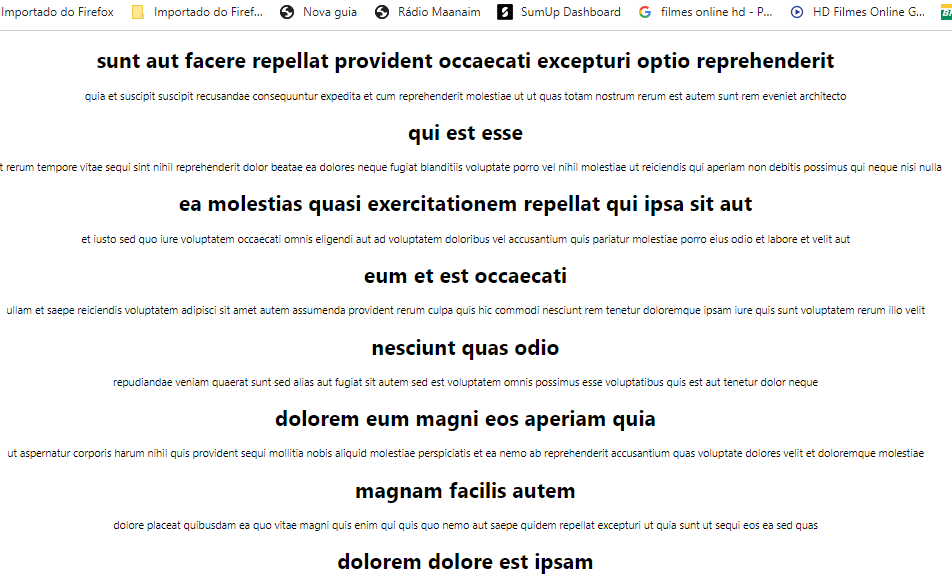
1. Vamos fazer algumas alterações, iremos apagar os métodos que foram criados dentro da classe App e todo conteúdo que está dentro da div App.
2. mport './App.css';
3. import {Component} from 'react'
4. class App extends *Component*{
6. state ={
7. name: 'Claudisnei Bello',
8. counter: 0
9. };
11. render(){
12. const {name, counter} = *this*.state;
13. return (
14. <div *className*="App">
16. </div>
17. );
18. }
19. }
20. Vamos criar um array dentro do objeto state do Component.
21. Vamos passar alguns dados para o array e em seguinda no Render() vamos criar uma variável post pegando essa informação do state.
22. import './App.css';
23. import {Component} from 'react'
24. class App extends *Component*{
26. state ={
27. posts:[
28. {
29. id: 1,
30. title: 'Título 1',
31. body: 'Corpo 1'
32. },
33. {
34. id: 2,
35. title: 'Título 2',
36. body: 'Corpo 2'
37. },
38. {
39. id: 3,
40. title: 'Título 3',
41. body: 'Corpo 3'
42. }
44. ]
45. };
47. render(){
48. const {posts} = *this*.state;
49. return (
50. <div *className*="App">
52. </div>
53. );
54. }
55. }
56. export default App;
57. Iremos utilizar o método map do Array.prototype.map() para retornar o que quisermos dentro de um array em Javascript.
58. Então dentro do root App vamos utilizar o método map de arrays na variável post passando como parâmetro post=> <h1>{post.title}</h1>} isso irá renderizar na tela todos os títulos que estão no array posts. Mas irá ocorrer um erro no console sobre Key porque não estamos fornecendo uma chave única para os itens do array, para corrigir este problema, basta adicionarmos no <h1 key={post.id}> e o erro será corrigido.
59. render(){
60. const {posts} = *this*.state;
61. return (
62. <div *className*="App">
63. {posts.map(*post*=> <h1 *key*={post.id}>{post.title}</h1>)}
64. </div>
65. );
66. }
67. Se quisermos mostrar na tela mais de uma linha de informação basta inserir todo <h1> dentro de parênteses.
68. Porem para funcionar temos que adicionar todas as linhas dentro de um único root ou seja tudo terá que estar dentro de uma div e o key que estava no h1 deverá ser colocado dentro da div.

Aula 16 – Life cicle methods part 1

1. Se quisermos que ocorra alguma coisa com o componente assim que ele for montado, podemos utilizar o método de ciclo de vida, componentDidMount(), é um método como o render()
2. Vamos adicionar um this.setState dentro deste método e dentro do this.setState nós iremos adicionar o array posts com algumas alterações nos dados para podermos visualizar as alterações ocorrendo na tela assim que o componente for montado irá ocorrer a alteração dos dados do state.
3. Iremos adicionar o this.setState dentro de um setTimeOut() passando um tempo de 5 segundos, ou seja assim que passa 5 segundos a alteração ocorrerá.
4. Iremos criar dentro do state uma variável chamada counter com valor 0.
5. import './App.css';
6. import {Component} from 'react'
7. class App extends *Component*{
9. state ={
10. counter: 0,
11. posts:[
12. {
13. id: 1,
14. title: 'Título 1',
15. body: 'Corpo 1'
16. },
17. {
18. id: 2,
19. title: 'Título 2',
20. body: 'Corpo 2'
21. },
22. {
23. id: 3,
24. title: 'Título 3',
25. body: 'Corpo 3'
26. }
28. ]
29. };
30. //variavel criada para zerar o timeout
31. timeoutUpdate = null;
32. //esse é um life cicle metods
33. componentDidMount(){
34. *this*.handleTimeout();
36. }
37. //é um life cicle que recebe o estado anterior ou props states
38. componentDidUpdate(){
39. *this*.handleTimeout();
40. }
41. //para apagar o lixo e não dar erro no navegador
42. componentWillUnmount(){
43. //zera o time para dar erro na página quando ocorrer alteraçoes
44. clearTimeout(*this*.timeoutUpdate);
45. }
46. //função criada para atualizar o state do componente
47. handleTimeout=()=>{
48. const {posts, counter} = *this*.state;
49. posts[0].title = 'o título mudou';
50. *this*.timeoutUpdate = setTimeout(() => {
51. *this*.setState({
52. posts,
53. counter: counter + 1
54. })
55. }, 1000);
56. }
58. render(){
59. const {posts, counter} = *this*.state;
60. return (
61. <div *className*="App">
62. <h1>{counter}</h1>
63. {posts.map(*post*=> (
64. <div *key*={post.id}>
65. <h1>{post.title}</h1>
66. <p>{post.body}</p>
67. </div>
68. ))}
69. </div>
70. );
71. }
72. }
73. export default App;

Aula 17 – Life cicle methods part 2

**Aula 18 – Data fething – buscando dados de fora do aplicativo**

1. vamos realizar algumas alterações no código.
2. import './App.css';
3. import {Component} from 'react'
4. class App extends *Component*{
6. state ={
7. posts:[
8. {
9. id: 1,
10. title: 'Título 1',
11. body: 'Corpo 1'
12. },
13. {
14. id: 2,
15. title: 'Título 2',
16. body: 'Corpo 2'
17. },
18. {
19. id: 3,
20. title: 'Título 3',
21. body: 'Corpo 3'
22. }
24. ]
25. };
27. //esse é um life cicle metods
28. componentDidMount(){
30. }
31. //é um life cicle que recebe o estado anterior ou props states
32. componentDidUpdate(){
34. }
35. //para apagar o lixo e não dar erro no navegador
36. componentWillUnmount(){
38. }
39. render(){
40. const {posts} = *this*.state;
41. return (
42. <div *className*="App">
44. {posts.map(*post*=> (
45. <div *key*={post.id}>
46. <h1>{post.title}</h1>
47. <p>{post.body}</p>
48. </div>
49. ))}
50. </div>
51. );
52. }
53. }
54. export default App;
55. iremos utilizar o método componentDidMount() para carregar dados no app, buscando os dados em uma API. Para isso vamos utilizar a Api https://jsonplaceholder.typicode.com/posts que possui 100 posts..
56. Para fazermos uma requisição nós utilizamos a fetch api do navegador, mas poderíamos utilizar o axios ou outro qualquer.
57. Dentro do método componentDiMount() iremos chamar o fetch passando a url da api. Sendo que isso retorna uma primisse com uma resposta que devererá ser convertida para json que irá retornar outra promisse com uma resposta em json. Iremos utilizar o setState para setar a resposta no array posts.
58. //esse é um life cicle metods
59. componentDidMount(){
60. fetch('https://jsonplaceholder.typicode.com/posts')
61. .then(*response*=> response.json())
62. .then(*posts* => *this*.setState({posts}))
63. }
64. A resposta na tela ficará assim:
65. 
66. Ou seja carregou os dados da api.
67. Vamos fazer algumas alterações, iremos criar uma função assíncrona que irá realizar a requisição e salvará os dados em uma variável.
68. loadPosts = async () =>{
69. const postResponse = fetch('https://jsonplaceholder.typicode.com/posts');
70. }
71. Vamos aproveitar e fazer a requisição das fotos também.
72. Para isso iremos utilizar o método do Javascript Promise.
73. loadPosts = async () =>{
74. const postResponse = fetch('https://jsonplaceholder.typicode.com/posts');
75. const [posts] = await *Promise*.all([postResponse]);
76. const postJson = await posts.json();
77. *this*.setState({posts: postJson});
78. }
79. Iremos passar essa função loadPosts dentro do método componentDiMount() para carregar os posts.
80. //esse é um life cicle metods
81. componentDidMount(){
82. *this*.loadPosts();
83. }
84. **Melhorando o CSS**
85. Na div interna do .map iremos criar um className = post-content e na div externa iremos criar outra className = posts.
86. No arquivo App.css iremos apagar tudo. Vamos dar uma olhada nos estilos globais. Iremos zerar a margem o pading e o box-sizing: border-box.
87. \*{
88. margin: 0;
89. padding: 0;
90. box-sizing: border-box;
91. }
92. No arquivo App.css
93. Termos na classe .post um display = grid um grid-template-columns: repeate(auto-fill, minimax(280px, 1fr));
94. O diplay tipo grid mostra o conteúdo em linhas e colunas, o grid-template-columns, será repetido para todos os elementos realizando um auto-fill e para os elementos ele terá um tamanho mínimo de 280px porém se não couber ele usará um fragmento (1fr).
95. Voltando no html do app, vamos criar um section criando um className de container e colocando todo o conteúdo dentro deste section.
96. Voltando ao arquivo App.css iremos estilizar a classe container vamos configurar uma altura mínima como 100 da view na horizontal ou seja min-height: 100vh; a cor de fundo como cinza ou seja background: #eee.
97. No arquivo App.js vamos criar dentro do html dentro do map outra div com um className post e copiar o conteúdo da div de className= post-content para dentro dela.
98. Voltando ao arquivo App.css na classe post nós iremos adicionar a cor de fundo branca ou background: #fff.
99. Na classe .posts vamos adicionar um espaçamento vertical e horizontal entre cada post com gap: 30px;
100. Na classe .post vamos criar um efeito de sombra na parte externa da caixa para melhorar a visualização do post com um box-shadow 0 0 10px rgba( 0, 0, 0, 0.1).
101. Na classe post-content que é onde está o conteúdo iremos aplicar um padding dentro do post que pode ser 20px.
102. *.container*{
103. min-height: 100vh;
104. background: #eee;
105. }
106. *.posts*{
107. display: grid;
108. grid-template-columns: repeat(*auto-fill*, minmax(280px, 1fr));
109. gap: 30px;
110. }
111. *.post*{
112. background: #fff;
113. box-shadow: 0 0 10px rgba(0 0 0  1);
114. }
115. *.post-content*{
116. padding:20px;
117. }
118. **Colocando as imagens.**
119. Vamos colocar as imagens dentro dos posts para isso iremos na função loadPosts do arquivo App.js e iremos criar uma variável chamada photosResponse fazendo um fetch pegando o link das fotos. Iremos resolver o photosResponse colocando ele dentro do argumento da Promise.all, mas temos que criar outra variável chamada fotos que ficará no array.
120. Criando também uma variável chamada photosJson que irá receber as fotos no formato json.
121. Porem o link de fotos é muito maior que o de posts por isso iremos utilizar um zip para unir dois arrays.
122. Vamos criar então outro array chamado postsAndPhotos e nele iremos passar o postsJson.map uma vez que ele é o menor dos dois. Iremos utilizar dois atributos deste array que será o post e o index. O índice do post será utilizado para pegar a foto de mesmo index do outro array.
123. loadPosts = async () =>{
124. //utilizamos o fetch para fazer uma requisição e será esperado um response
125. //neste caso faremos uma requisição para os posts e outra para as fotos
126. const postResponse = fetch('https://jsonplaceholder.typicode.com/posts');
127. const photosResponse = fetch('https://jsonplaceholder.typicode.com/photos');
128. //criamos dois arrays que irão receber os conteúdos na ordem dos responses
129. //que são obtidos através da função Promise que pega o retorno do response
130. //e passa para cada variavel separada
131. const [posts, photos] = await *Promise*.all([postResponse, photosResponse]);
132. //aqui criamos um objeto que receberá a conversão do array para
133. //o formato json
134. const postJson = await posts.json();
135. const photosJson = await photos.json();
136. //fazendo a união entre dois arrays utilizando o map para escolher
137. /\*quais atributos do primeiro array serão utilizados, sendo que
138. iremos retornar o post e utilizando a função cover nós pegamos do
139. segundo array o atributo que queremos \*/
140. const postsAndPhotos = postJson.map((*post*,*index*)=>{
141. return { ...post, cover: photosJson[index].url }
142. })
143. //susbstituimos o postJson que tinha somente os posts pelo postsAndPhotos
144. *this*.setState({posts: postsAndPhotos});
145. }
146. Agora nós temos outro elemento que é o cover que possui a foto.
147. Vamos incluir dentro da div post uma imagem que ficará acima do texto da div post-content, essa imagem terá um alt.
148. Para isso basta adicionar a tag img com um src recebendo a variável post.cover e o alt recebendo o post.title.
149. <section *className*='container'>
150. <div *className*="App, posts">
151. {posts.map(*post*=> (
152. <div *className*='post'>
153. <img *src*={post.cover} *alt*={post.title} />
154. <div *key*={post.id} *className*='post-content'>
155. <h1>{post.title}</h1>
156. <p>{post.body}</p>
157. </div>
158. </div>
159. ))}
160. </div>
161. </section>
162. Vamos voltar ao arquivo App.css e iremos estilizar a img dentro do post criando uma largura máxima de 100%.
163. *.post* img{
164. max-width: 100%;
165. }
166. Vamos também determinar que a largura máxima seja de 1200px, a margem será 0 nas laterais e automático em cima e em baixo.
167. *.container*{
168. max-width: 1200px;
169. margin: 0 auto;
170. min-height: 100vh;
171. background: #eee;
172. padding: 30px;
173. }
174. Vamos também criar uma transição de forma que ao passar o mouse sobre o post ele aumente ligeiramente o tamanho.
175. *.post*{
176. background: #fff;
177. box-shadow: 0 0 10px rgba(0 0 0  1);
178. transition: transform 100ms ease-in-out;
179. }
180. *.post:hover*{
181. transform: scale(1.05);
182. }

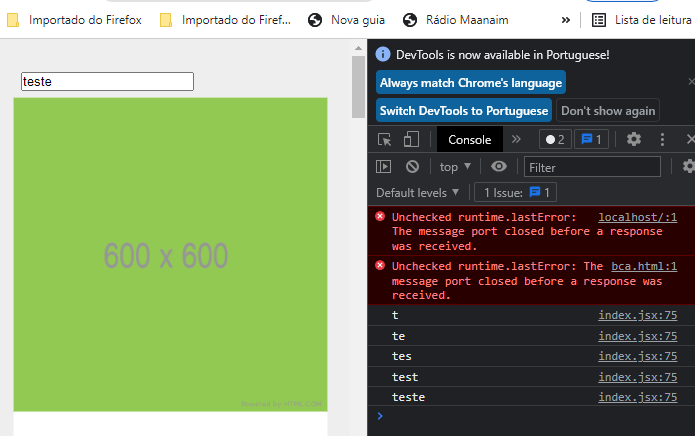
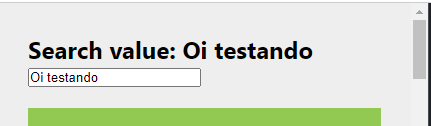
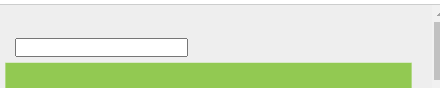
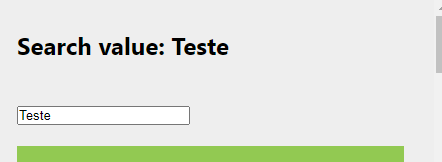
**Aula 19 – Props e organizando os componentes part 1.**

1. Vamos organizar o App.js em componentes.
2. Crie um pasta chamada components
3. Dentro desta pasta crie outra pasta com o nome do componente neste caso será PostCard.
4. Crie um arquivo chamado index.jsx pois será um arquivo html do tipo js.
5. Vamos digitar export pois teremos que importar o arquivo em outro local.
6. Vamos criar uma função arrow function chamada PostCard ou uma classe que retorna o método render(). Neste caso será uma função mesmo.
7. Vamos então recortar o conteúdo html que está dentro o .map no arquivo App.js e copiar para este arquivo index.jsx.
8. Mas temos que passar as variáveis que estão no html como uma props no parâmetro da função.
9. Vamos importar o componente PostCard dentro do App.js e vamos incluir este componente dentro de .map no componente <PostCard /> temos que inserir os atributos que serão as variáveis que serão utilizadas dentro do componente.
10. Ou podemos também utilizar somente o post como atributo pois lá no componente poderemos pegar os atributos dele uma vez que post é um array.
11. export const PostCard = ({*post*}) => {
12. //para pegarmos o post que está vindo para o componente podemos fazer
13. //de tres maneiras
14. //const post = props.post
15. //const {post} = props
16. //ou export const PostCard = ({post})
17. return(
18. <div *className*='post'>
19. <img *src*={post.cover} *alt*={post.title} />
20. <div *key*={post.id} *className*='post-content'>
21. <h1>{post.title}</h1>
22. <p>{post.body}</p>
23. </div>
24. </div>
25. );
26. }
27. O arquivo App.js ficará assim:
28. import './App.css';
29. import {Component} from 'react'
30. //importando o componente PostCart
31. import {PostCard} from '../src/Components/PostCard'
32. class App extends *Component*{
34. state ={
35. posts:[
36. ]
37. };
38. //esse é um life cicle metods
39. componentDidMount(){
40. *this*.loadPosts();
41. }
42. loadPosts = async () =>{
43. //utilizamos o fetch para fazer uma requisição e será esperado um response
44. //neste caso faremos uma requisição para os posts e outra para as fotos
45. const postResponse = fetch('https://jsonplaceholder.typicode.com/posts');
46. const photosResponse = fetch('https://jsonplaceholder.typicode.com/photos');
47. //criamos dois arrays que irão receber os conteúdos na ordem dos responses
48. //que são obtidos através da função Promise que pega o retorno do response
49. //e passa para cada variavel separada
50. const [posts, photos] = await *Promise*.all([postResponse, photosResponse]);
51. //aqui criamos um objeto que receberá a conversão do array para
52. //o formato json
53. const postJson = await posts.json();
54. const photosJson = await photos.json();
55. //fazendo a união entre dois arrays utilizando o map para escolher
56. /\*quais atributos do primeiro array serão utilizados, sendo que
57. iremos retornar o post e utilizando a função cover nós pegamos do
58. segundo array o atributo que queremos \*/
59. const postsAndPhotos = postJson.map((*post*,*index*)=>{
60. return { ...post, cover: photosJson[index].url }
61. })
62. //susbstituimos o postJson que tinha somente os posts pelo postsAndPhotos
63. *this*.setState({posts: postsAndPhotos});
64. }
65. //é um life cicle que recebe o estado anterior ou props states
66. componentDidUpdate(){
68. }
69. //para apagar o lixo e não dar erro no navegador
70. componentWillUnmount(){
72. }
73. render(){
74. const {posts} = *this*.state;
75. return (
76. <section *className*='container'>
77. <div *className*="App, posts">
78. {posts.map(*post*=> (
79. <*PostCard*
80. *post*={post}
81. />
82. ))}
83. </div>
84. </section>
86. );
87. }
88. }
89. export default App;
90. sempre que utilizamos um map temos que criar um key.
91. Então temos que retirar o key que está dentro do html do componente PostCard e inseri-lo dentro post.map no App.js..
92. Com as alterações o App.js ficará assim
93. return (
94. <section *className*='container'>
95. <div *className*="App, posts">
96. {posts.map(*post*=> (
97. <*PostCard*
98. *key*={post.id}
99. *post*={post}
100. />
101. ))}
102. </div>
103. </section>
105. );
106. E o componente PostCard ficará assim:
107. return(
108. <div *className*='post'>
109. <img *src*={post.cover} *alt*={post.title} />
110. <div *className*='post-content'>
111. <h1>{post.title}</h1>
112. <p>{post.body}</p>
113. </div>
114. </div>
115. );
116. **RESUMO**:
117. Os atributos que estão sendo passados para o componente são props e o estado (state) que está no componente raiz ou root ou seja do componente que está renderizando na página, este estado não volta para trás ele só vai para frente ou seja pode descer ou passar para os elementos filhos deste componente isso é o estado e é o estado que rege quando os elementos serão rerenderizados na página. E quando passo esse estado ou qualquer outra coisa para os componentes filhos o que eu tenho na verdade é uma props, então os atributos que estamos passando no componente root para os filhos são props.
118. Quando alteramos nosso estado, o render do componente root é chamado novamente e atualiza as informações nos componentes filhos.

**Aula 20 – Props e organizando os componentes parte 2 e criando uma paginação dos posts.**

1. Para melhorar a organização dos nossos códigos, vamos criar uma pasta chamada útil e o arquivo load-posts.js.
2. Neste arquivo iremos recortar o conteúdo da função loadPosts do App.js e iremos passar para o arquivo load-posts.js
3. Vamos importar o arquivo load-posts para o App.js .
4. dentro do método loadPosts vamos criar um a const postAndPhotos e passar para ela o loadPosts que foi destrurado de load-posts na importação deste.
5. Arquivo App.js
6. //esse é um life cicle metods
7. async componentDidMount(){
8. await *this*.loadPosts();
9. }
10. loadPosts = async () =>{
11. const postsAndPhotos = await loadPosts();
12. //susbstituimos o postJson que tinha somente os posts pelo postsAndPhotos
13. *this*.setState({posts: postsAndPhotos});
14. }
15. Arquivo load-posts.js
16. export const loadPosts = async () =>{
17. //utilizamos o fetch para fazer uma requisição e será esperado um response
18. //neste caso faremos uma requisição para os posts e outra para as fotos
19. const postResponse = fetch('https://jsonplaceholder.typicode.com/posts');
20. const photosResponse = fetch('https://jsonplaceholder.typicode.com/photos');
21. //criamos dois arrays que irão receber os conteúdos na ordem dos responses
22. //que são obtidos através da função Promise que pega o retorno do response
23. //e passa para cada variavel separada
24. const [posts, photos] = await *Promise*.all([postResponse, photosResponse]);
25. //aqui criamos um objeto que receberá a conversão do array para
26. //o formato json
27. const postJson = await posts.json();
28. const photosJson = await photos.json();
29. //fazendo a união entre dois arrays utilizando o map para escolher
30. /\*quais atributos do primeiro array serão utilizados, sendo que
31. iremos retornar o post e utilizando a função cover nós pegamos do
32. segundo array o atributo que queremos \*/
33. const postsAndPhotos = postJson.map((*post*,*index*)=>{
34. return { ...post, cover: photosJson[index].url }
35. })
36. return postsAndPhotos;
37. }
38. Ainda vamos criar outro componente passando para ele todo o conteúdo html com excessão da section, do arquivo App.js para outro componente.
39. Vamos criar então um comonente Posts.jsx dentro de uma pasta chamada Posts.
40. //importando o componente PostCart
41. import {PostCard} from '../PostCard'
42. export const Posts = ({*posts*})=>{
43. return(
44. <div *className*="App, posts">
45. {posts.map(*post*=> (
46. <*PostCard*
47. *key*={post.id}
48. *post*={post}
49. />
50. ))}
51. </div>
52. )
53. }
54. Antes ainda vamos melhorar mais a organização criando uma pasta chamada template e outra chamada home, movendo o arquivo App.js, App.test.js e App.css. Renomeie para index.jsx, [Home.test.js](http://Home.test.js) e styles.css .
55. Crie outra pasta chamada styles e mova o arquivo index.css para lá renomeado como global-styles.css
56. Agora vamos fazer a paginação.
57. **Criando uma paginação com o state**.
58. Dentro do state crie outro array chamado allPosts que irá conter todos os posts.
59. Vamos na função loadPosts que carrega os posts no array e iremos adicionar o array allPosts carregando nele também o postsAndPhotos.
60. Porém iremos fazer um slice no array posts, utilizando o Array.prototype.slice() ou seja fatiar, nele nós passamos no índice a posição de inicio e fim.
61. Por isso vamos criar no state uma variável chamada page que terá a página que começara e a postPerPage que terá o número de posts em cada página.
62. Dentro de postAndPhotos.slice() inclua o page e o postsPerPage.
63. Crie outro método chamado loadMorePosts para carregar mais posts na página.
64. Porém teremos que criar algum meio do usuário clicar e passar para a outra página.
65. Por isso vamos criar uma nova pasta dentro de Components chamada Button e um novo componente chamado index.jsx.
66. import {Component} from 'react';
67. export class Button extends *Component*{
68. render(){
69. const {text, onClick} = *this*.props;
70. return(
71. <button *onClick*={onClick}>
72. {text}
73. </button>
74. )
75. }
76. }
77. Codigo do arquivo index.jsx de Home
78. import './styles.css';
79. import {Component} from 'react';
80. //importando os dados
81. import {loadPosts} from '../../utils/load-posts';
82. import { Posts } from '../../Components/Posts';
83. import { Button } from '../../Components/Button';
84. class Home extends *Component*{
86. state ={
87. posts:[],
88. allPosts:[],
89. page:0,
90. postsPerPage: 2
91. };
92. //esse é um life cicle metods
93. async componentDidMount(){
94. await *this*.loadPosts();
95. }
96. loadPosts = async () =>{
97. const postsAndPhotos = await loadPosts();
98. const {page, postsPerPage}= *this*.state;
99. //susbstituimos o postJson que tinha somente os posts pelo postsAndPhotos
100. *this*.setState({
101. posts: postsAndPhotos.slice(page,postsPerPage),
102. allPosts: postsAndPhotos
103. });
104. }
105. //carregando mais posts na página
106. loadMorePosts = () =>{
107. //carrega as variaveis que iremos precisar do state
108. const {
109. page,
110. postsPerPage,
111. allPosts,
112. posts
113. } = *this*.state;
114. //nextPage terá a página mais a quantidade de postes por página
115. const nextPage = page + postsPerPage;
116. //criando um novo array que terá somente os posts que farão parte
117. //da página que será mostrada
118. const nextPosts = allPosts.slice(nextPage, nextPage + postsPerPage)
119. //passamos para posts o array somente com os itens que serão mostrados
120. posts.push(...nextPosts);
121. //setamos agora no state o novo array de posts e a nova pagina
122. *this*.setState({posts, page: nextPage});
123. console.log(page, postsPerPage, nextPage, nextPage + postsPerPage);
124. }
125. render(){
126. const {posts} = *this*.state;
127. return (
128. <section *className*='container'>
129. <*Posts* *posts*={posts} />
130. <*Button*
131. *text*='Load more posts'
132. *onClick*={*this*.loadMorePosts}
133. />
134. </section>
136. );
137. }
138. }
139. export default Home;
140. no arquivo styles.css do componente Home o css está todo misturado, por isso temos que separar o código de cada componente, para isso vamos criar dentro das pastas dos componentes novos arquivos chamados index.jsx e style.css e vamos separar os conteúdos nesses arquivos.

**Aula 21 - Trabalhando com inputs de texto.**

1. Iremos colocar um input para busca de posts baseados no conteúdo digitado no input.
2. Então vamos no arquivo index.jsx de Home e vamos incluir o input do tipo Search no interior da section container.
3. <section *className*='container'>
4. <input *type*='search' /> <br/><br/>
5. Nos eventos de formulário nós temos o evento sintético do react chamado onChange que precisa receber uma função que recebe o próprio evento que estamos trabalhando e neste evento temos as informações do evento como por exemplo o target do evento e dentro do target temos também o value.
6. Para visualizarmos isso vamor incluir dentro do input o evento onChange passando para a função dele um evento e chamando um console.log que irá mostrar o valor do target do evento no console do navegador.
7. <input
8. *onChange*={(*e*)=> console.log(e.target.value)}
9. *type*='search'
10. /> <br/><br/>
11. Podemos ver isso funcionando no console do navegador.
12. 
13. Porém nos precisamos inserir isso no state, para que a medida que digitarmos eu quero fazer uma ação com o resultado que teremos.
14. Por isso vamos criar um método chamado handleChange que receberá a função que irá receber um evento dentro do método faremos um destruct criando uma const value que irá receber o target do evento.
15. //método para realizar busca no input
16. handleChange = (e){
17. //destruct capturando o target do evento
18. const{value} = e.target;
20. }
21. Vamos passar esse método para o evento onChange do input.
22. <input
23. *onChange*={*this*.handleChange}
24. *type*='search'
25. /> <br/><br/>
26. Temos que incluir no state uma nova variável que será a serchValue que terá de inicio uma string vazia.
27. state ={
28. posts:[],
29. allPosts:[],
30. page:0,
31. postsPerPage: 10,
32. searchValue: ''
33. };
34. Agora no método handleChange nós iremos incluir o setState passando o value como novo valor da variável searchValue do state.
35. //método para realizar busca no input
36. handleChange = (*e*)=>{
37. //destruct capturando o target do evento
38. const{value} = e.target;
39. //passamos o novo valor de searchValue utilizando o setState
40. *this*.setState({searchValue: value});
41. }
42. Vamos inserir no value do input a variável searchValue que foi criada no state do componente.
43. <input
44. *onChange*={*this*.handleChange}
45. *type*='search'
46. *value*={searchValue}
47. /> <br/><br/>
48. Para isso temos também que incluir a serachValue no destruct que fazemos dentro do render()
49. render(){
50. const {posts, page, postsPerPage, allPosts, searchValue } = *this*.state;
51. A partir desse ponto podemos fazer várias coisas. Suponha que queira ver o conteúdo buscado na tela.
52. Vamos inserir antes do input um h1 com o texto ‘Search Value: {searchValue}, todo conteúdo digitado no input é passado para o h1.
53. 
54. Porém se não tivermos nenhuma busca não é o ideal mostrar o conteúdo do h1. Por isso vamos utilizar o que chamamos de avaliação de curto circuito no Javascript.
55. Que basicamente é a utilização de operadores lógicos, para isso vamos inserir dentro da section uma expressão Javascript perguntando se existe valor dentro de searchValue.
56. Para isso utilizamos dois sinais de interrogação na frente de searchValue(negado duas vezes é o mesmo que true), o que transformará searchValue em um booleando. Ou seja se !!searchValue for true significa que searchValue possui valor caso contrário se for false significa que não possui valor.
57. Nós iremos utilizar dois sinais && após a expressão lógica para que seja realizada alguma ação, neste caso iremos mostrar o h1.
58. <section *className*='container'>
59. {!!searchValue && (
60. <h1>Search value: {searchValue}</h1><br/><br/>
61. )}
62. Porém isso dará um erro pois estamos retornando mais de um elemento no jsx, para resolver esse erro basta utilizarmos uma div e colocarmos os elementos dentro ou um fragmento (<> elementos aqui </>)
63. <section *className*='container'>
64. {!!searchValue && (
65. <>
66. <h1>Search value: {searchValue}</h1><br/><br/>
67. </>
68. )}
69. Agora se não digitarmos nada no imput o valor não será mostrado na tela.
70. 
71. Mas ao digitarmos qualquer coisa o valor começa a aparecer na tela.
72. 
73. Outra coisa que faremos é retirar o botão de more posts quando realizamos a busca. Então iremos colocar dentro do container do botão uma condição.
74. Porém a condição será se não tivermos busca, ou seja !serachValue(negado uma vez), isso significa que não existe valor dentro de searchValue entao o botão irá aparecer.
75. Agora precisamos manipular os posts para que ao digitarmos no input apareça somente os posts que contenham o conteudo do input.
76. E para isso nos vamos criar outra variável chamada filteredPosts que receberá o conteúdo de posts.
77. Após isso vamos