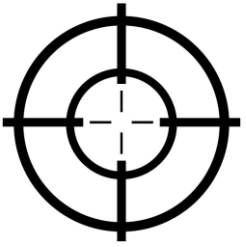


Sincronização e
integração com
back-end

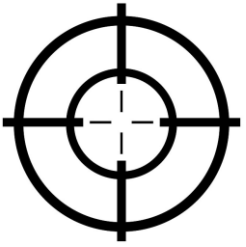
Sincronização de componentes



Em aplicações do tipo SPA é comum a existência de vários componentes, em execução simultânea.

Durante a execução simultânea, muitas vezes, a modificação de um dado tem que ser refletida em vários componentes, ou seja, estes componentes precisam ser sincronizados de alguma forma, a modificação do estado de um componente afeta o estado de outro componente.

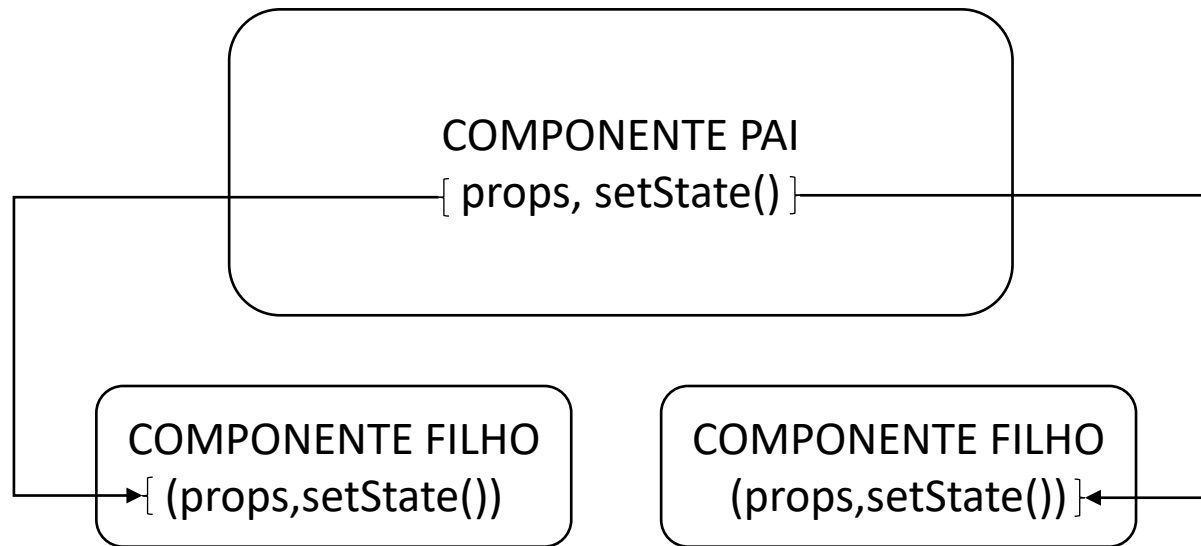
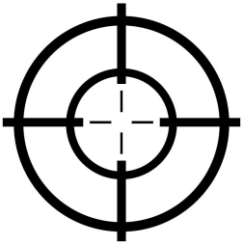
Sincronização de componentes



Em situações que envolve a sincronização de componentes a documentação da biblioteca react recomenda “elevar” o state (estado) dos componentes para um componente pai, desta forma o state torna-se “compartilhado”.

O componente pai passa a ser uma espécie de repositório onde a modificação de um dado altera seu próprio state. A alteração do state provoca uma nova renderização do componente pai, que também atualiza os componentes filhos atrelados a ele, gerando a sincronia desejada.

Sincronização de componentes



O componente pai compartilha com seus filhos propriedades e uma função, que modifica o seu próprio state.

A função compartilhada permite que cada componente filho atualize, diretamente, o state do componente pai. Isto gera uma nova renderização do componente pai e de seus filhos automaticamente, seguindo a regra do ciclo de vida.

```

class Tempo extends Component<props, state> {
  private conversor = new Conversor()
  constructor(props) { ...
  }

  obterHora(valorHora: string) { ...
  }

  obterMinuto(valorMinuto: string) { ...
  }

  obterSegundo(valorSegundo: string) { ...
  }

  render() { ...
  }
}

export default Tempo

```

```

ReactDOM.render(
  <React.StrictMode>
    <Tempo texto='Conversor de unidades de tempo'
      mensagem='Insira um valor para uma unidade de tempo' />
  </React.StrictMode>,
  document.getElementById('root')
);

```

```

obterHora(valorHora: string) {
  let convertido = this.conversor.obterNumero(valorHora)
  let valorMinuto = this.conversor.multiplicar(convertido)
  let valorSegundo = this.conversor.multiplicar(valorMinuto)
  this.setState({
    hora: convertido,
    minuto: valorMinuto,
    segundo: valorSegundo
  })
}

```

Sincronização de componentes



```

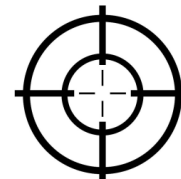
obterHora(valorHora: string) {
  let convertido = this.conversor.obterNumero(valorHora)
  let valorMinuto = this.conversor.multiplicar(convertido)
  let valorSegundo = this.conversor.multiplicar(valorMinuto)
  this.setState({
    hora: convertido,
    minuto: valorMinuto,
    segundo: valorSegundo
  })
}

export default class Conversor {
  private converterNumero(valor: string): number { ...
  }
  public obterNumero(valor: string): string { ...
  }
  public multiplicar(valorHora: string): string { ...
  }
  public dividir(valorHora: string): string { ...
  }
}

public obterNumero(valor: string): string {
  let numero = this.converterNumero(valor)
  let conversao = ''
  if (!Number.isNaN(numero)) {
    conversao = numero.toString()
  }
  return conversao
}

```

Sincronização de componentes



```
<Entrada funcao={this.obterHora} valor={this.state.hora} unidade="Hora" />
```

```
type props = {  
  unidade: string,  
  valor: string,  
  funcao: Function  
}
```

```
class Entrada extends Component<props> {  
  constructor(props) { ...  
  }
```

```
    bloquerSubmissao(evento) { ...  
    }
```

```
    obterEntrada(evento) { ...  
    }
```

```
    render() { ...  
    }
```

```
}
```

```
export default Entrada
```

```
class Tempo extends Component<props, state> {  
  private conversor = new Conversor()  
  constructor(props) { ...  
  }
```

```
    obterHora(valorHora: string) { ...  
    }
```

```
    obterMinuto(valorMinuto: string) { ...  
    }
```

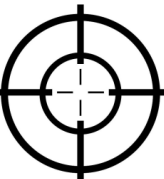
```
    obterSegundo(valorSegundo: string) { ...  
    }
```

```
    render() { ...  
    }
```

```
}
```

```
export default Tempo
```

Sincronização de componentes



```

class Entrada extends Component<props> {
  constructor(props) {
    super(props)
    this.obterEntrada = this.obterEntrada.bind(this)
    this.bloquerSubmissao = this.bloquerSubmissao.bind(this)
  }

  obterEntrada(evento) {
    let valor = evento.target.value
    this.props.funcao(valor)
  }

  render() {
    return (
      <form onSubmit={this.bloquerSubmissao}>
        <div className="input-field">
          <span>{this.props.unidade}: </span><br />
          <input value={this.props.valor} onChange={this.obterEntrada}>
        </div>
      </form>
    )
  }
}

```

```

class Tempo extends Component<props, state> {
  private conversor = new Conversor()
  constructor(props) { ...
  }

  obterHora(valorHora: string) { ...
  }

  obterMinuto(valorMinuto: string) { ...
  }

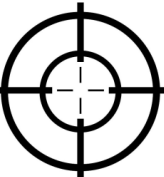
  obterSegundo(valorSegundo: string) { ...
  }

  render() { ...
  }

  export default Tempo
}

```

Sincronização de componentes



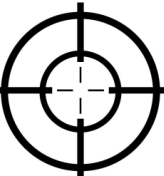
O componente pai passa funções/métodos como “props” para o componente filho!

O método permite ao componente filho alterar o “state” do pai!

```
class Tempo extends Component<props, state> {  
  private conversor = new Conversor()  
  constructor(props) { ...  
  }  
  
  obterHora(valorHora: string) { ...  
  }  
  
  obterMinuto(valorMinuto: string) { ...  
  }  
  
  obterSegundo(valorSegundo: string) { ...  
  }  
  
  render() { ...  
  }  
}  
export default Tempo
```

```
<div className="col s4">  
  <Entrada funcao={this.obterHora} valor={this.state.hora} unidade="Hora" />  
</div>  
<div className="col s4">  
  <Entrada funcao={this.obterMinuto} valor={this.state.minuto} unidade="Minuto" />  
</div>  
<div className="col s4">  
  <Entrada funcao={this.obterSegundo} valor={this.state.segundo} unidade="Segundo" />  
</div>
```

Sincronização de componentes





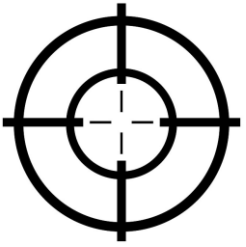
Sincronização de componentes

```
obterHora(valorHora: string) {  
  let convertido = this.conversor.obterNumero(valorHora)  
  let valorMinuto = this.conversor.multiplicar(convertido)  
  let valorSegundo = this.conversor.multiplicar(valorMinuto)  
  this.setState({  
    hora: convertido,  
    minuto: valorMinuto,  
    segundo: valorSegundo  
  })  
}
```

```
<div className="col s4">  
  <Entrada funcao={this.obterHora} valor={this.state.hora} unidade="Hora" />  
</div>  
<div className="col s4">  
  <Entrada funcao={this.obterMinuto} valor={this.state.minuto} unidade="Minuto" />  
</div>  
<div className="col s4">  
  <Entrada funcao={this.obterSegundo} valor={this.state.segundo} unidade="Segundo" />  
</div>
```

Quando o componente filho utilizar a função, enviada como propriedade, ele irá alterar o “status” do componente pai e será renderizado novamente, pelo ciclo de vida!

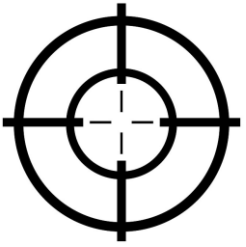
Integração com back-end



O back-end refere-se a partes de um aplicativo ou código de um software que permitem seu funcionamento, mas não podem ser acessadas por um usuário ou cliente. A maioria dos dados, processamento e lógica operacional são manipulados ou armazenados no back-end.

O objetivo da biblioteca react é auxiliar e facilitar o desenvolvimento da parte visual e, ainda que ocorra alguma programação, ela depende do back-end para produzir uma aplicação completa.

Integração com back-end



Para comunicação entre o aplicativo react (front-end) e o back-end é necessário usar um software intermediário, como uma API de comunicação. Uma API comum é a Fetch.

A API Fetch fornece uma interface JavaScript para acessar e manipular a comunicação via protocolo HTTP. Com ela pode-se buscar recursos em um servidor, de forma assíncrona, através da rede.

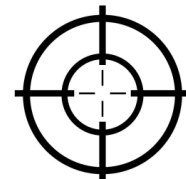
Pode-se utilizar a API para buscar ou enviar dados ao back-end. Para isso, deve-se atribuir parâmetros a função, que realiza os procedimentos de busca ou envio.

```
export enum URI {  
  CLIENTES = 'http://localhost:5555/clientes',  
  DELETAR_CLIENTE = 'http://localhost:5555/cliente/excluir',  
  CADASTRAR_CLIENTE = 'http://localhost:5555/cliente/cadastro'  
}  
  
export default class BuscadorClientes implements Buscador {  
  public async buscar() {  
    let json = await fetch(URI.CLIENTES).then(resposta => resposta.json())  
    return json  
  }  
}
```


The diagram illustrates the flow of data from the `URI` enum to the `fetch` function call. A line connects the `URI` enum definition to the `URI.CLIENTES` property access in the `fetch` function. Another line connects the `fetch` function call to the `resposta` parameter in the `then` function, which then points to the `resposta.json()` method call.

O resultado da função `fetch` precisa ser “resolvido”. Para o contexto da `fetch` “resolver” significa passar uma função para ela, que irá tratar a resposta obtida pela requisição.

Integração com back-end

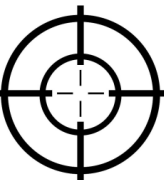


```
export enum URI {  
  CLIENTES = 'http://localhost:5555/clientes',  
  DELETAR_CLIENTE = 'http://localhost:5555/cliente/excluir',  
  CADASTRAR_CLIENTE = 'http://localhost:5555/cliente/cadastro'  
}  
  
class CadastradorCliente implements Cadastrador {  
  cadastrar(objeto: Object): void {  
    fetch(URI.CADASTRAR_CLIENTE, {  
      method: 'POST',  
      headers: {  
        'Content-Type': 'application/json'  
      },  
      body: JSON.stringify(objeto)  
    })  
  }  
}  
  
export default CadastradorCliente
```



Com a mesma função pode-se, também, enviar dados. Mas, é necessário fornecer informações sobre o método HTTP, headers e body da requisição.

Integração com back-end



```

class Clientes extends Component<{}, state> {
  constructor(props) { ...
  }

  public buscarClientes() {
    let buscadorClientes = new BuscadorClientes()
    const clientes = buscadorClientes.buscar()
    clientes.then(clientes => {
      this.setState({ clientes })
    })
  }

  public excluirRemoto(idCliente: string) { ...
  }

  public excluirLocal(id: string, e: any) { ...
  }

  componentDidMount(){ ...
  }

  render() { ...
  }
}

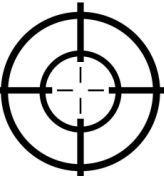
export default Clientes

export default class BuscadorClientes implements Buscador {
  public async buscar() {
    let json = await fetch(URI.CLIENTES).then(resposta => resposta.json())
    return json
  }
}

```

Objeto do buscador, aqui, é buscar um json com as informações de uma lista de clientes, que será atribuída ao state do componente.

Integração com back-end



```

class Clientes extends Component<{}, state> {
  constructor(props) { ...
  }

  public buscarClientes() { ...
  }

  public excluirRemoto(idCliente: string) {
    let removedor = new RemovedorCliente()
    removedor.remover({ id: idCliente })
  }

  public excluirLocal(id: string, e: any) { ...
  }

  componentDidMount(){ ...
  }

  render() { ...
  }
}
export default Clientes

```

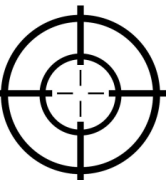
Neste exemplo, o objeto do tipo removedor utiliza a função fetch para enviar ao back-end um json, com informações de um cliente que precisa ser excluído da base de dados da aplicação.

```

export default class RemovedorCliente implements RemovedorRemoto {
  public remover(objeto: Object): void {
    let json = { id: objeto['id'] }
    fetch(URI.DELETAR_CLIENTE, {
      method: 'DELETE',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(json)
    })
  }
}

```

Integração com back-end

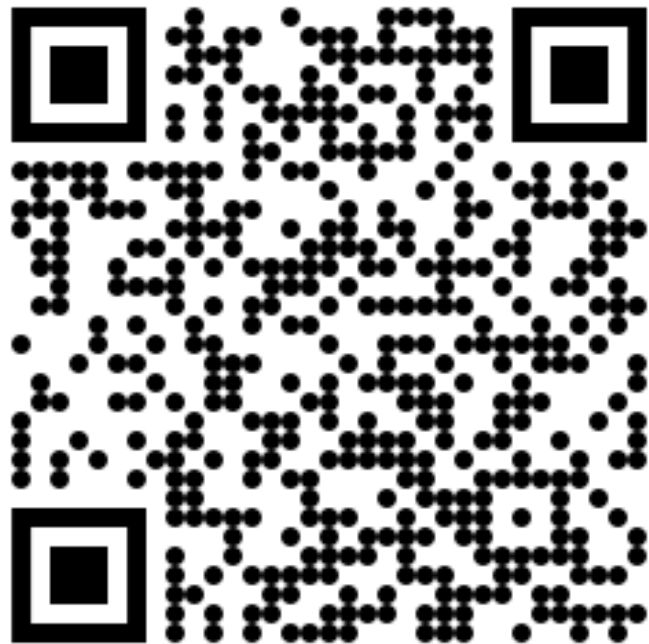


Formulário

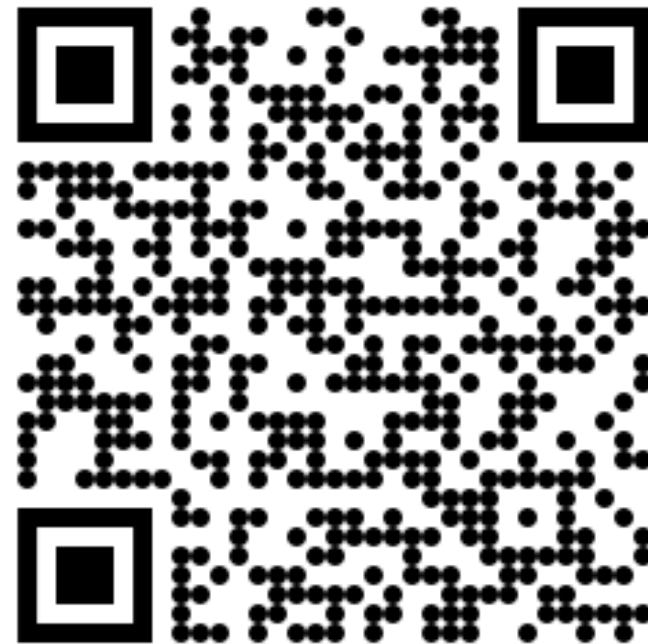


<https://github.com/gerson-pn>

[/react-synchronized-components](#)



[/react-integration-class](#)



Formulário – JDK 11



<https://github.com/gerson-pn/react-integration-class-jdk11>

