React

Component, DOM, Virtual DOM, událost, props, state, hook, propagace stavu

Joseph Soukup – P4

Komponenty a Props

- Komponenty nám umožňují rozdělit UI do nezávislých, znovupoužitelných částí
- Konceptuálně jsou komponenty jako Javascript funkce
- Akceptují libovolné vstupy (props) a vrací React elementy popisující, co by mělo být zobrazováno na obrazovce

Funkční a Class Komponenty

- Nejjednodušší způsob definování komponenty je Javascript funkce
  - ```
    function Welcome(props) {
      return <h1>Hello, {props.name}</h1>;
    }
    ```
  - Tato funkce je validní komponenta, protože přijímá props objekt s daty a vrací React element
  - Tyto komponenty se nazývají funkční komponenty, protože jsou doslova Javascript funkce
- Můžete také použít ES6 třídu
  - ```
    class Welcome extends React.Component {
      render() {
        return <h1>Hello, {this.props.name}</h1>;
      }
    }
    ```
- Z pohledu Reactu jsou tyto dvě komponenty identické

Renderování komponenty

- Elementy mohou být reprezentovány jako uživatelem definované komponenty
  - ```
    const element = <Welcome name="Sara" />;
    ```
- Když React uvidí element, který je reprezentován jako uživatelem definovaná komponenta, předá jí atributy a následníky (children) jako jeden objekt => props
- Například toto renderuje „Hello, Sara“:
  - ```
    function Welcome(props) {
      return <h1>Hello, {props.name}</h1>;
    }

    const element = <Welcome name="Sara" />;
    ReactDOM.render(
      element,
      document.getElementById('root')
    );
    ```

Skládání komponent

- Komponenty mohou odkazovat na jiné komponenty v jejich výstupu
- V React aplikaci běžně popsány jako komponenty:
  - Tlačítko
  - Formulář
  - Dialog
- Například pokud chceme vytvořit aplikace s mnoho pozdravy

-
```jsx
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}

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

Vytahování komponent

- Tato komponenta přijímá autora, text a datum
  -
```jsx
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```
- Zaprvé vytáhneme Avatara:
  -
```jsx
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />
  );
}
```

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

- Zadruhé vytáhneme UserInfo

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}
```

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Props jsou Readn-Only

- Pokaždé co se definuje komponenta jako funkce nebo třída, nikdy nesmí upravovat své vlastní props:

```
function sum(a, b) {
  return a + b;
}
```

- Takové funkce nazýváme „pure“ protože se nesnaží změnit svůj vstup a vždycky vrací stejný výsledek pro stejné vstupy
- Tato funkce se nazývá „impure“ protože mění svůj vstup:

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

- Všechny React komponenty se musí chovat jako „pure“ funkce s respektováním svých props

DOM (Document Object Model)

- Je programovací interface pro HTML a XML dokumenty
- Reprezentuje stránku tak, aby program mohl změnit strukturu, styl a obsah dokumentu
- DOM reprezentuje dokument jako uzly a objekty, tím se programovací jazyky spojí se stránkou

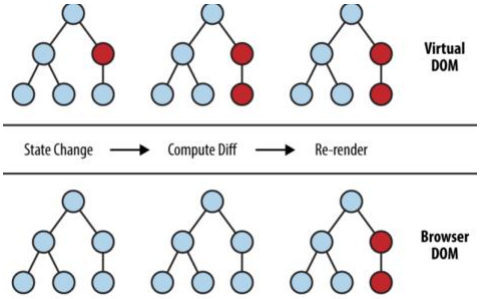
DOM + Javascript

- Dom je klíčovým pro Javascript
- DOM zprostředkovává všechny komponenty z ostatních jazyků jsou implementovány do Javascriptu pomocí DOM, bez DOM by nemohl mít přístup k HTML a XML prvkům
- API = DOM + JS
- Celý obsah stránky je uložen v DOM a přistoupit k němu lze pomocí Javascriptu
- DOM je nezávislý na programovacím jazyce

ReactDOM

- render()
  - ```
    ReactDOM.render(element, container[, callback])
    ```
  - Renderuje React elementu do DOM do containeru a vrací referenci komponentě

Virtual DOM

- Je koncept, kde stav UI nejdříve virtuálně uchová v paměti a pokud bude stejný jako je námi určený stav, tak se synchronizuje s knihovnou DOM
- Optimalizuje rychlost
- 

Událost

- camelCase
- s JSX se funkce předává jako event handler a né jako string
- HTML:
  - ```
    <button onclick="activateLasers()">
      Activate Lasers
    </button>
    ```
- React:

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

- Nemůže se vrátit false pro zabránění defaultního chování v Reactu
- Musí se vracet preventDefault
- HTML:

```
<a href="#" onclick="console.log('The link was clicked.'); return false">
  Click me
</a>
```

- React:

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

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

- e je event (událost)
- React eventy nefungují stejně jako nativní eventy
- Není potřeba volat addEventListener
- Místo toho stačí poskytnout listener, když je prvek původně vykreslen
- Když se definuje komponenta pomocí ES6 class, tak běžné je, že event handler je metodou třídy

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

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

Předávání argumentů

- Například pokud by řádek měl svoje vlastní ID:

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

State

- React komponenty mají vestavěný **state** objekt

- State objekt je místo, kam ukládáte hodnoty vlastností, které patří komponentě
- Když se změní state objekt, tak se cely komponent načte znovu

Vytvoření state objektu

- State objekt je inicializovaný v konstruktoru
  - Specify the `state` object in the constructor method:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {brand: "Ford"};
  }
  render() {
    return (
      <div>
        <h1>My Car</h1>
      </div>
    );
  }
}
```

- State objekt může obsahovat několik vlastností:
  -

```
this.state = {
  brand: "Ford",
  model: "Mustang",
  color: "red",
  year: 1964
};
```

Používání state objektu

- Ke state se může přistupovat kdekoliv v komponentě pomocí „this.state.XXX“
  - Refer to the `state` object in the `render()` method:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
      </div>
    );
  }
}
```

Změna state objektu

- Pro změnu hodnoty state objektu se používá metoda „this.setState()“
- Když se změní hodnota v state objektu, tak se komponent načte znovu => výstup se změní podle nové hodnoty
  - Add a button with an `onClick` event that will change the color property:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  changeColor = () => {
    this.setState({color: "blue"});
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
        <button
          type="button"
          onClick={this.changeColor}
        >Change color</button>
      </div>
    );
  }
}
```

Hook

- Umožňuje použití state objektu a jiné React komponenty bez vytvoření vlastních tříd
- Hook jsou funkce, které „hook into“ (připojit se) do state objektu a životního cyklu funkčních komponent
- Nefunguje uvnitř funkcí
- Hooky jsou zpětně kompatibilní => neobsahují nějaké změny, které by mohly program rozbít

Kdy použít Hook

- Dříve pokud jste vytvořily komponentu a chtěli jste přidat state, tak se musela konvertovat do třídy
- Teď je možno přidat Hook do funkční komponenty

Pravidla

Pouze na nejvyšší úrovni

- Hooky se nesmí volat uvnitř smyček, podmínek nebo vnořených funkcí
- Z důvodu, aby se při znovunačtení komponenty volaly vždy ve stejném pořadí

Pouze z React funkcí

- Hooky se nemohou volat z klasických Javascript funkcí

Hooks State

- Nový způsob deklarování state objektu
- useState()
  - `const [count, setCount] = useState(0);`

Hooks Effect

- Effect Hook jsou stejné jako metody životního cyklu componentDidMount(), componentDidUpdate() a componentWillUnmount()
- useEffect()
- V React komponentách jsou dva typy Effect:
  - Effects bez Cleanupu:
    - Používá useEffect, který neblokuje aktualizace browseru
    - Logging
    - Síťové požadavky
  - Effect s Cleanupem
    - Například když jsme připojeni k nějakému externímu zdroji dat, je důležité vyčistit paměť

Vlastní Hooky

- Javascript funkce
- Jakákoliv funkce, co začíná „use“ a dodržuje pravidla hooku

Propagace stavu

- Pokud chceme z potomka změnit data u rodičovského elementu, tak v rodičovském elementu předáme metodu do potomka a v potomku tuto metodu použijeme a tím se změní data a rodičovském elementu

```
function App() {
  const [a, setA] = useState(4);
  const [b, setB] = useState(6);
  return (
    <div className="App">
      <p>{a + b}</p>
      <UpDown value={a} min={0} max={10} setValue={setA} />
      <UpDown value={b} min={0} max={10} setValue={setB} />
    </div>
  );
}
```