REACT - INSTALACE

React byl od začátku navržen pro postupné osvojení a můžeme použít tolik Reactu, kolik chceme. Většina webových stránek nejsou a nemusí být single-page aplikace. S pár řádkami kódu a žádným stavebním nástrojem můžeme React vyzkoušet třeba v malé části našich webových stránek.

PŘIDÁNÍ REACTU POMOCÍ SCRIPT TAGŮ

Nejjednodušší cesta jak na stránku přidat React a začít jej používat je přidáním script tagů. To se ale hodí, když se React učíme nebo si chceme jen něco otestovat.

```
  <!-- ... další HTML ... -->

  <!-- Načtení Reactu -->
  <!-- Poznámka: pro produkci změňte "development.js" za "production.min.js". -->
  <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></script>
  <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script>

  <!-- naše vlastní scripty -->
</body>
```

PŘIDÁNÍ PODPORY JSX

Pokud chceme v našem kódu používat JSX, tak je nejjednodušší cesta jak je přidat použít následující script. To se ale hodí jen když si chceme JSX vyzkoušet. Pro produkci je to pomalé.

```
  <!-- ... další HTML ... -->

  <!-- Po přidání tohoto script tagu můžeme v našem kódu začít psát JSX -->
  <script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>

  <!-- ve scriptech, kde používáme JSX, musíme nastavit atribut type na "text/babel" -->
  <script src="./MujScript.js" type="text/babel"></script>
</body>
```

VYTVOŘENÍ REACT APLIKACE

Pokud nejsme zběhlý v technologiích jako je třeba Webpack, tak pro nás může být založení projektu a nastavení prostředí pro vývoj celkem složité. Proto existují různé nástroje, které nám s tím mohou pomoct.

Create React App - dobré pro single-page aplikace (hodí se i pro učení)
Next.js - dobré pro server-rendered weby s Node.js
Gatsby - dobré pro statické webové stránky s Reactem

REACT - JSX

JSX poskytuje syntaktický cukřík pro React.createElement(component, props, ...children).

```jsx
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

→

```jsx
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

ATRIBUTY ELEMENTŮ

V JSX se atributy píší camelCasem. Hodnotu jim nastavujeme jako string, nebo přes složené závorky.

```jsx
const element1 = (
  <a href="https://www.reactjs.org">
    link
  </a>
);

const element2 = (
  <img src={user.avatarUrl}></img>
);
```

REACT KOMPONENTY

V JSX můžeme kromě klasických elementů mít i komponenty. Ty musejí začínat velkým písmenem.

Props komponentám předáváme prostřednictvím atributů.

```jsx
// komponenta MyComponent s prop foo
// -komponenty bez potomků musí končit />
<MyComponent foo={1 + 2 + 3 + 4} />

// pokud pro prop neurčíme hodnotu, tak
// bude automaticky true
<MyTextBox autocomplete />

// pokud zadáme něco dovnitř otevíracího
// a uzavíracího tagu, tak se to do props
// předá pod názvem children
<MyButton kind="primary">
  Hello World!
</MyButton>

// pokud již máme props jako objekt, tak
// jej můžeme předat takto:
<Greeting {...props} />
```

VKLÁDÁNÍ HODNOT

Pokud chceme do JSX vložit nějakou hodnotu, tak to můžeme udělat pomocí složených závorek.

```jsx
<div>{user.name}</div>

// vložení hodnoty podle podmínky
<div>
  {isLoggedIn
    ? <LogoutButton/>
    : <LoginButton/>
  }
</div>

// můžeme využít i short-circuitingu
/* hodnoty false, true, undefined a null
se totiž nerenderují */
<div>
  {zpravy.length > 0 &&
    <h2>
      Máš {zpravy.length} zprávy.
    </h2>
  }
</div>

// vložení pole elementů
// (nesmíme zapomenount na atribut key)
const polozky = [
  <li key="1">React</li>,
  <li key="2">JavaScript</li>
];
<ul>
  {polozky}
</ul>
```

ROZDÍLY V ATRIBUTECH

V JSX se pár atributů píše trochu jinak než v klasickém HTML:

class - className, for - forHTML, a je tam ještě pár dalších rozdílů...

REACT - ELEMENTY

Elementy jsou nejmenším stavebním blokem React aplikací. Narozdíl od DOM elementů jsou React elementy prosté objekty a nejsou náročné na vytvoření. React DOM se stará o to, aby DOM odpovídal React elementům.

```
// vytvoření elementu s JSX
const element1 = <h1>Hello, world</h1>;

// vytvoření elementu bez JSX
const element2 = React.createElement(
  'h1',
  {},
  'Hello, world'
);
```

VYRENDEROVÁNÍ ELEMENTU

Pokud chceme do DOMu vyrenderovat nějaký element, tak si musíme určit do jakého elementu jej vyrenderujeme. Poté si vytvoříme root a pomocí metody render element do DOMu vyrenderujeme.

```
<!-- někde v HTML -->
<div id="root"></div>
```

```
const root = ReactDOM.createRoot(
  document.getElementById('root')
);
const element = <h1>Hello, world</h1>;
root.render(element);
```

React elementy jsou neměnné. Když vytvoříme element, nemůžeme změnit jeho potomky nebo atributy. Element je jako jeden snímek ve filmu: reprezentuje uživatelské rozhraní v určitém okamžiku. Vždy když zavoláme metodu render, tak se vytvoří nový element a ten který byl v DOMu vyrenderovaný dříve se odstraní.

FRAGMENTY

Často v Reactu potřebujeme, aby komponenty vraceli více elementů najednou, i když musí vrátit jen jeden element. Fragmenty nám umožňují je obalit bez přidání dalších elementů.

```
<React.Fragment>
  <ChildA/>
  <ChildB/>
  <ChildC/>
</React.Fragment>
```

```
// zkrácená syntaxe pro Fragment
// - nemůžeme ale nastavit atribut key
<>
  <ChildA/>
  <ChildB/>
  <ChildC/>
</>
```

REACT - KOMPONENTY

Koncepčně jsou komponenty stejné jako JavaScript funkce. Berou nějaký vstup (props) a vracejí React elementy, které popisují co se má zobrazit.

VYTVOŘENÍ KOMPONENTY

Komponentu můžeme vytvořit pomocí funkce, nebo pomocí třídy.

```
function Welcome() {
  // vrátí se element, který se zobrazí
  return <h1>Hello</h1>;
}
```

```
class Welcome extends React.Component {
  // metoda render slouží k zobrazení
  // komponenty
  render() {
    return <h1>Hello</h1>;
  }
}
```

RENDEROVÁNÍ

React elementy mohou kromě klasických elementů reprezentovat také komponenty.

```
// props předáváme přes atributy
const element = <Welcome name="Sara" />;
```

PROPS

Pokud při vytváření elementu komponenty předáme prostřednictvím atributů nějaké hodnoty, tak se komponentě předají v podobě objektu (říká se jim props). Props jsou neměnné.

```
// function komponenty berou props jako
// parametr
function Welcome1(props) {
  return <h1>Hello, {props.name}</h1>;
}

// v class komponentách máme k props
// přístup pomocí this.props
class Welcome extends React.Component {
  render() {
    const name = this.props.name;
    return <h1>Hello, {name}</h1>;
  }
}
```

STATE

Pokud chceme v komponentě uchovávat nějaké hodnoty, které se mohou měnit a mají vliv na UI, tak je můžeme uložit jako state.

```
// u class komponenty definujeme state v
// jejím konstruktoru
class Clock extends React.Component {
  constructor(props) {
    // pokud máme vlastní konstruktor,
    // tak musíme props předat nahoru
    super(props);
    // vytvoření statu
    this.state = {date: new Date()};
  }
  // ...

// ve function komponentách jde state
// použít také, ale musíme použít hook
```

State neměníme přímo, ale pomocí metody setState.

```
// bere objekt se změněnými hodnotami
this.setState({ date: new Date() });

// React nemusí změnit state hned, proto
// máme možnost metodě setState předat
// funkci, a měnit hodnoty pomocí ní
// - funkce bere aktuální state a props
this.setState((state, props) => ({
  counter: state.counter + props.inc
}));
```

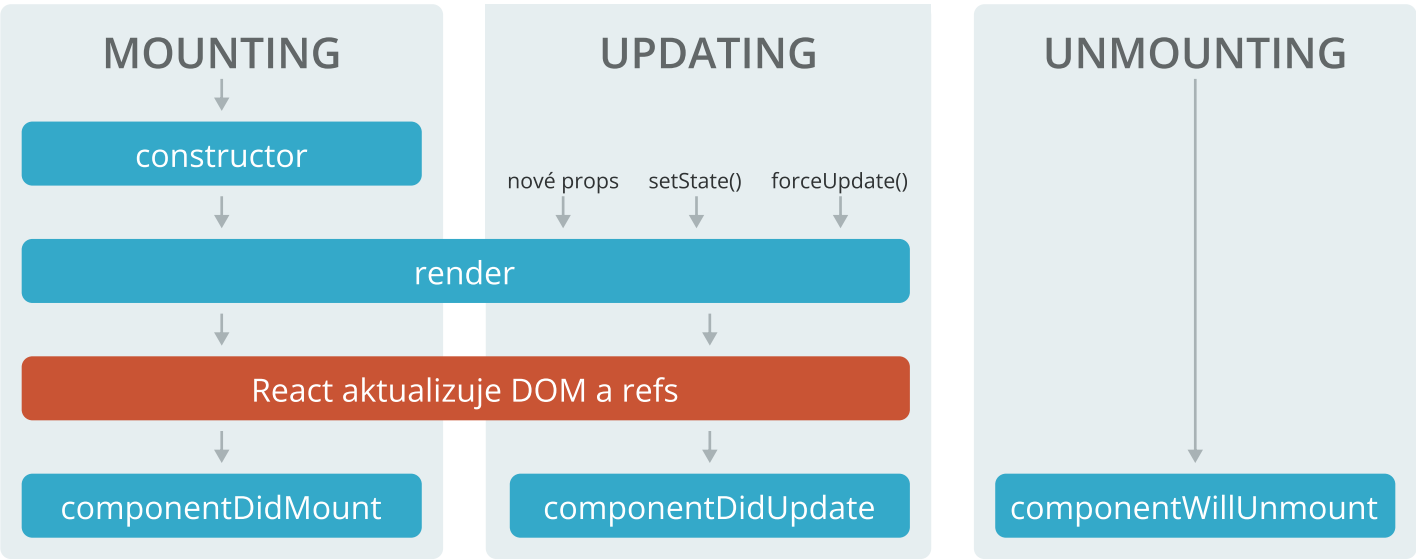
REACT - LIFECYCLE METODY

Class komponenty mají metody, které vývojářům umožňují aktualizovat stav aplikace a odrážet změny do UI před/po klíčových událostech.

FÁZE KOMPONENTY

Komponenta prochází těmito třemi fázemi:

- mounting - komponenta je poprvé vyrenderována do DOMu
- updating - komponenta dostane nové props nebo state a znova se vyrenderuje do DOMu
- unmounting - komponenta se odstraní z DOMu



constructor

V konstruktoru komponenty inicializujeme state, nebo třeba bindujeme metody. Je zavolán než se komponenta přidá do DOMu.

render

Slouží k vykreslení komponenty. Vrací element, který se má vyrenderovat do DOMu.

shouldComponentUpdate

Touto metodou můžeme Reactu říct, jestli se má komponenta aktualizovat nebo ne. Pokud vrátíme true: ano, když false: ne.

componentDidMount

Tato metoda se zavolá hned, jak se komponenta mountne. Je to dobré místo, kde například začít načítat nějaká data.

componentDidUpdate

Tato metoda se zavolá po aktualizaci komponenty. Jako parametr může brát předešlé props, předešlý state a snapshot.

componentWillUnmount

Tato metoda se zavolá před unmountnutím komponenty. Dobré pro zrušení časovačů, atd...

Je jich ještě víc, ale tyto jsou pravděpodobně nejpoužívanější.

REACT - ŘÍZENÍ EVENTŮ

Řízení eventů v Reactu je podobné jako řízení eventů u klasických DOM elementů. Je zde ale pár odlišností v syntaxi.

PŘIDÁNÍ EVENT LISTENERU

Event listenery se v reactu přidávají pomocí event atributů. Tyto atributy jsou v JSX psány camelCasem a můžeme jim nastavit jako hodnotu nějakou metodu komponenty. U class komponent je potřeba metody používané pro zpracovávání eventů bindovat.

```
class LoggingButton extends React.Component {
  constructor(props) {
    super(props);

    // když metodu handleClick předáváme, aby se později zavolala, tak v ní nebude klíčové slovo
    // this odkazovat na komponentu - tímto řádkem se to vyřeší (vytvoří se nová funkce, ve které
    // bude klíčové slovo this vždy odkazovat na komponentu)
    // - u function komponent tento problém řešit nemusíme, ale u class komponent ano
    this.handleClick = this.handleClick.bind(this);

    // - další možnost by byla vytvářet v render metodě namísto bindování novou funkci, to ale
    // není úplně ideální, protože by se tato funkce vytvářela při každém renderování
  }

  handleClick() {
    console.log("tlačítko bylo kliknuto");
  }

  render() {
    // atributem onClick se předává, která funkce se má po kliknutí na element zavolat
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

SYNTHETIC EVENT

V Reactu se do funkcí, které zpracovávají eventy, předává instance SyntheticEventu namísto nativního eventu. Má stejné rozhraní jako nativní event a obsahuje metody jako stopPropagation nebo preventDefault. Rozdíl je, že eventy fungují identicky ve všech prohlížečích.

```
function handleSubmit(e) {
  // SyntheticEvent má stejné rozhraní jako nativní Event
  e.preventDefault();
}
```

REACT - FORMULÁŘE

Defaultní akce při odeslání formuláře je přesměrovat uživatele na jinou stránku. V Reactu se nám ale často hodí zpracovávat formuláře jinak.

```
function handleSubmit(e) {
  // formulář se neodešle a uživatel nebude přesměrován na jinou stránku
  e.preventDefault();
}
```

CONTROLLED KOMPONENTY

V HTML mají input elementy jako je input, textarea, nebo select vlastní stav (anglicky state). V Reactu má komponenta také vlastní state. Tyto dva staty můžeme zkombinovat a nechat tak state komponenty být jediným zdrojem pravdy.

Pokud má input stejnou hodnotu jako state a při jeho změně se state mění, jedná se o controlled komponentu.

```
class NameForm extends React.Component {
  /* ... */

  handleChange(e) {
    // když se změní input, tak se změní
    // i state
    this.setState(
      {value: e.target.value}
    );
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>Jméno:</label>
        <input type="text"
          value={this.state.value}
          onChange={this.handleChange}/>
        <input type="submit"
          value="Odeslat"/>
      </form>
    );
  }
}
```

ZPRACOVÁNÍ VÍCE INPUTŮ

Nemusíme mít pro kontrolování každého inputu samostatnou metodu. Můžeme mít třeba něco takového:

```
handleInputChange(e) {
  const target = e.target;
  const value = (
    target.type === 'checkbox'
    ? target.checked
    : target.value
  );
  const name = target.name;

  this.setState({
    [name]: value
  });
}
```

ATRIBUT VALUE

Některým inputům se v HTML nastavují hodnoty pomocí atributu value a některým jiným způsobem. Inputu textarea se například v HTML nastavuje hodnota dovnitř uzavíracího a otevíracího tagu. V Reactu se hodnota inputu textarea nastavuje pomocí atributu value. Stejné je to například i u selectu.

```
<textarea
  value={this.state.value}
  onChange={this.handleChange}
/>
```

REACT - UNCONTROLLED KOMPONENTY

Ve většině případů je doporučeno použít controlled komponenty. U controlled komponent jsou data řízena React komponentou. Alternativa jsou uncontrolled komponenty, kde jsou data řízena samotným DOMem.

VYTVOŘENÍ UNCONTROLLED KOMPONENTY

Když máme uncontrolled komponenty, tak k získávání jejich hodnoty používáme refs. Hodnotu tedy uchovává input element, není uložena ve statu.

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }

  handleSubmit(e) {
    alert('Bylo zadáno jméno: ' + this.input.current.value);
    e.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Jméno:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Odeslat" />
      </form>
    );
  }
}
```

NASTAVENÍ DEFAULTNÍ HODNOTY

Pokud chceme inputu nastavit defaultní hodnotu, tak to můžeme udělat pomocí atributu defaultValue. Tento atribut se nebude aktualizovat. Pro checkbox elementy existuje atribut defaultChecked. Inputy select a texarea podporují atribut defaultValue.

```
<input
  defaultValue="Bob"
  type="text"
  ref={this.input}
/>
```

REACT - SEZNAMY A KEYS

Pokud chceme vyrenderovat pole elementů, tak můžeme. Uděláme to v JSX jeho předáním pomocí složených závorek.

```
<ul>{listItems}</ul>
```

KEYS

Pokud renderujeme pole elementů, tak musíme každému elementu přidat atribut key s unikátní hodnotou. Tento atribut pomáhá Reactu určovat které položky byly změněny, přidány, nebo odstraněny.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

Atribut key slouží Reactu jako hint, ale nepředává se komponentě jako prop.

Pokud se pořadí elementů může měnit, tak není doporučováno používat jako key indexy. Může to negativně ovlivnit výkon a způsobit komplikace se statem komponenty.

REACT - REFS

Refs slouží k udržení odkazu na nějaký element nebo komponentu. Můžeme je použít pro řízení focusu, výběr textu, nebo třeba při integrování s knihovnami třetích stran. Neměli bychom je ale nadužívat.

VYTVOŘENÍ REF

Vytvořit ref můžeme pomocí funkce React.createRef a nastavením atributu ref na nějaký element.

```
class MyComponent extends React.Component
{
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={this.myRef} />;
  }
}
```

PŘÍSTUP K ELEMENTU REF

Poté co se elementu v render metodě předá atribut ref, tak k němu máme přes ref přístup.

Pokud ref použijeme na normální HTML element, tak bude ref odkazovat na DOM element. Pokud ji použijeme na class komponentu, tak ref bude odkazovat na instanci této komponenty. Na function komponenty se ref použít nedá, protože nemají instance.

```
const node = this.myRef.current;
```

CALLBACK REF

Další cesta jak použít ref je nastavit funkci, která jako argument přijímá DOM element nebo instanci komponenty. Tu si potom můžeme uložit třeba někde do proměnné.

```
class TextInput extends React.Component
{
  constructor(props) {
    super(props);

    this.textInput = null;

    this.setTextInputRef = element => {
      this.textInput = element;
    };
  }

  render() {
    return (
      <div>
        <input
          type="text"
          ref={this.setTextInputRef}
        />
      </div>
    );
  }
}
```

PŘEPOSÍLÁNÍ REF

Pokud u některé komponenty chceme ref poslat k nějakému jejímu potomkovi, můžeme ji vytvořit pomocí funkce forwardRef.

```
const FancyButton =
React.forwardRef((props, ref) => (
  <button ref={ref} className="special">
    {props.children}
  </button>
));
```

REACT - PORTÁLY

Portály nám poskytují renderovat potomky do elementu, který existuje mimo DOM hierarchii komponenty.

VYTVOŘENÍ PORTÁLU

Když vrátíme element v render metodě, tak se přidá do DOMu jako potomek nejbližšího předka. Někdy se ale hodí vložit potomka na jinou lokaci v DOMu. K tomu můžeme v render metodě použít funkci React.createPortal, které předáme potomky a node do které je chceme vyrenderovat.

```
render() {
  // `domNode` je jakákoliv validní DOM node
  return ReactDOM.createPortal(
    this.props.children,
    domNode
  );
}
```

Typické použití portálů je třeba pro dialogová okna nebo tooltipy.

EVENT BUBBLING PŘES PORTÁLY

I když mohou být portály kdekoliv v DOM tree, tak fungují stejně jako normální React potomci. Funkce jako je context fungují úplně stejně i když je potomek portál. Zahrnuje to i event bubbling. Event spuštěný v portálu propaguje k předkům v React tree, i když tyto elementy nejsou předky v DOM tree.

REACT - CONTEXT

Context umožňuje poslat dolů data bez jejich předávání po jednotlivých komponentách. Měli bychom jej ale používat střídmě, protože se potom mohou hůře znovupoužívat komponenty.

VYTVOŘENÍ CONTEXTU

Context můžeme vytvořit pomocí funkce React.createContext, které předáme defaultní hodnotu, kterou by měl context uchovávat.

```
const ThemeContext =
  React.createContext('light');

class App extends React.Component {
  render() {
    // Pokud chceme některým komponentám,
    // které context používají, nastavit
    // jinou než defaultní hodnotu, tak
    // je obalíme do Providera contextu
    return (
      <ThemeContext.Provider
        value="dark"
      >
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}
```

POUŽITÍ CONTEXTU

Pokud chceme v některé komponentě použít context, tak této komponentě musíme nastavit statickou vlastnost contextType na context, který chceme použít.

K hodnotě contextu se potom dostaneme pomocí this.context.

```
class ThemedButton extends React.Component
{
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context}/>;
  }
}
```

MĚNĚNÍ CONTEXTU

Pokud chceme v contextu něco měnit, tak si na to můžeme v contextu vytvořit funkci.

```
class App extends React.Component {
  constructor(props) {
    super(props);

    // funkce pro měnění statu (vlastně
    // contextu)
    this.toggleTheme = () => {
      this.setState(state => ({
        theme:
          state.theme === themes.dark
            ? themes.light
            : themes.dark
      }));
    };

    // state se nastavuje jako value
    // contextu
    this.state = {
      theme: themes.light,
      toggleTheme: this.toggleTheme
    };
  }

  render() {
    return (
      <ThemeContext.Provider
        value={this.state}
      >
        <Content />
      </ThemeContext.Provider>
    );
  }
}
```

CONTEXT.CONSUMER

Komponenta Context.Consumer bere funkci s hodnotou contextu. Lze ji použít namísto contextType.

```
<MyContext.Consumer>
  {value => /* ... */}
</MyContext.Consumer>
```

REACT - ERROR BOUNDARIES

Error Boundaries jsou React komponenty, které chytají JavaScript errory všude v jejich potomcích. Logují chyby a zobrazují chybové UI.

Error Boundaries chytají errory při renderování, v lifecycle metodách a v konstruktorech celého tree pod nimi. Nechytají error pro:

- event handlery
- asynchronní kód (např. setTimeout nebo requestAnimationFrame)
- server side renderování
- errory, které jsou vyhozeny v samotné Error Boundary komponentě

VYTVOŘENÍ ERROR BOUNDARY

Class komponenta se stane Error Boundary, když definuje alespoň jednu z těchto metod: static getDerivedStateFromError() nebo componentDidCatch(). Metodu getDerivedStateFromError můžeme k nastavení statu, když vznikne chyba, abychom mohli vyrenderovat chybovou zprávu. Metoda componentDidCatch slouží k logování chyb.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // aktualizujeme state a nastavíme tak že vznikla chyba
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // zde můžeme logovat errory (můžeme je třeba poslat do nějaké logovací služby)
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // Při vzniku chyby se zobrazí chybová zpráva
      return <h1>Something went wrong.</h1>;
    }

    // pokud nevznikla žádná chyba, tak se vyrenderují potomci Error Boundary
    return this.props.children;
  }
}
```

POUŽITÍ ERROR BOUNDARY

Error Boundary můžeme použít jako jakoukoliv jinou komponentu.

```
<ErrorBoundary> <MyWidget/> </ErrorBoundary>
```

REACT - HOC

Higher-Order komponenty (HOC) je technika, která se v Reactu používá pro znovupoužívání logiky komponenty. Jedná se o vzor, který vychází z kompoziční povahy Reactu.

VYTVOŘENÍ HIGHER-ORDER KOMPONENTY

HIgher-Order komponenta je funkce, která vezme komponentu a vrátí novou komponentu. Vrácená komponenta vlastně obaluje jinou komponentu a přidává jí nějakou funkcionalitu.

```
// funkce bere jako parametr komponentu
function withSubscription(WrappedComponent, selectData) {
  // ...a vrací novou komponentu...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(DataSource, props)
      };
    }

    componentDidMount() {
      DataSource.addChangeListener(this.handleChange);
    }

    componentWillUnmount() {
      DataSource.removeChangeListener(this.handleChange);
    }

    handleChange() {
      this.setState({
        data: selectData(DataSource, this.props)
      });
    }

    render() {
      // komponenta vrací komponentu kterou obaluje, a předává jí navíc prop data
      // - také jí předává ostatní props, které byly předány
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };
}
```

- HOC přidává komponentě funkcionalitu, ale očekává se, že navrácená komponenta bude mít stejné rozhraní jako ta obalená
- HOC nemůžeme aplikovat v render metodě, protože se vytváří nová komponenta (namísto updatnutí komponenty by se vytvořil nový subtree)
- pokud na komponentu aplikujeme HOC, tak si musíme uvědomit, že se ztratí její statické metody (komponenta je obalená)
- ref se při aplikování HOC nepředá (musíme použít React.createRef)

REACT - STRICT MODE

StrictMode je nástroj pro highlightování potenciálních problémů v aplikaci. Jako například Fragment, StrictMode také nerenderuje žádné viditelné UI. Aktivuje dodatečné kontroly a varování pro jeho potomky. Strict mode kontroly běží jen v development módu, nemají vliv na produkční build.

ZAPNUTÍ STRICT MÓDU

Strict Mode můžeme spustit pro jakoukoliv část naší aplikace. Stačí jen přidat komponentu React.StrictMode.

```
<React.StrictMode>
  <div>
    <ComponentOne />
    <ComponentTwo />
  </div>
</React.StrictMode>
```

S ČÍM STRICT MODE POMÁHÁ

Strict Mode pomáhá s těmito věcmi:

- identifikování komponent s nebezpečnými lifecycles
- varování o použití zastaralé string ref API
- varování o použití zastaralé findDOMNode
- detekce nečekaných vedlejších účinků
- detekce zastaralé context API
- zajištění znovupoužitelného statu

Další funkcionality mohou/mohly být přidány v budoucích verzích Reactu.

REACT - DĚLENÍ KÓDU

Většina React aplikací bude mít svůj kód zabalený pomocí nástrojů jako je Webpack, Rollup, nebo Browserify.

BUNDLING

Bundling je proces následování importnutých souborů a jejich slučování do jednoho souboru. K bundlování můžeme využít třeba Webpack. Pokud používáme nástroj jako je Create React App, tak je pro nás Webpack již nastavený a nemusíme si jej konfigurovat sami.

DYNAMIC IMPORTING

Bundling je super, ale jak aplikace roste, tak se zvyšuje načítání. Kvůli tomu nemusíme všechen kód slučovat do jednoho souboru, ale můžeme některý načíst pomocí dynamického importu. V Reactu máme k dispozici funkci React.lazy, která nám umožňuje vyrenderovat dynamický import jako komponentu. Tato funkce bere jako parametr funkci, která musí zavolat dynamický import. Tato funkce se zavolá při prvním vyrenderování komponenty.

Lazy komponenty by se měli renderovat uvnitř komponenty Suspense, která nám umožňuje nějak zobrazit že se lazy komponenty načítají.

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Načítám...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

Pokud chceme nějak ošetřit, když se module nepodaří z nějakého důvodu načíst, tak to můžeme provést pomocí Error Boundaries. Po vytvoření naší Error Boundary ji můžeme použít kdekoliv nad komponentou Suspense.

```
<MyErrorBoundary>
  <Suspense fallback={<div>Loading...</div>}>
    <OtherComponent />
  </Suspense>
</MyErrorBoundary>
```

REACT - OPTIMALIZACE VÝKONU

React interně používá několik technik pro minimalizování počtu drahých DOM operací potřebné pro aktualizaci UI. Pro spoustu aplikací vede použití Reactu k rychlému UI, bez provedení příliš velké práce k optimalizaci výkonu. Nicméně je tu několik cest, jak můžeme naši React aplikaci zrychlit.

POUŽITÍ PRODUKČNÍHO BUILDU

V development módu nám React přidává spoustu užitečných varování. Ty ale dělají React větším a pomalejším, takže pro produkci je určitě lepší použít produkční verzy.

Pokud si nejsme jistí, jestli je náš build proces nastaven správně, tak si můžeme nainstalovat React Developer Tools pro Chrome. Pokud navštívíme webovou stránku s Reactem v produkčním módu, tak bude mít ikona rozšíření tmavé pozadí, pokud v development módu, tak bude mít červené pozadí.

VIRTUALIZOVÁNÍ DLOUHÝCH SEZNAMŮ

Pokud naše aplikace renderuje dlouhé seznamy dat (stovky nebo tisíce řádků), tak je doporučeno použít techniku známou jako "windowing". Tato technika renderuje jen malou podmnožinu našich řádků a může dramaticky snížit čas, který se zabere renderováním komponenty a také počtu vytvořených nodes.

Knihovny react-window a react-virtualized jsou popularní windowing knihovny. Poskytují několik znovupoužitelných komponent pro zobrazení listů, gridů a tabulkových dat.

POUŽITÍ METODY shouldComponentUpdate

Metodu shouldComponentUpdate můžeme u class komponenty implementovat ke kontrolování, jestli se má komponenta aktualizovat nebo ne.

```
shouldComponentUpdate(nextProps,nextState)
{
  if (
    this.props.color !== nextProps.color
  ) {
    return true;
  }
  if (
    this.state.count !== nextState.count
  ) {
    return true;
  }
  return false;
}
```

Ve většině případů ale můžeme dědit od React.PureComponent. Je to stejné jako bychom měli metodu shouldComponentUpdate s porovnáváním minulého a aktuálního statu a props.

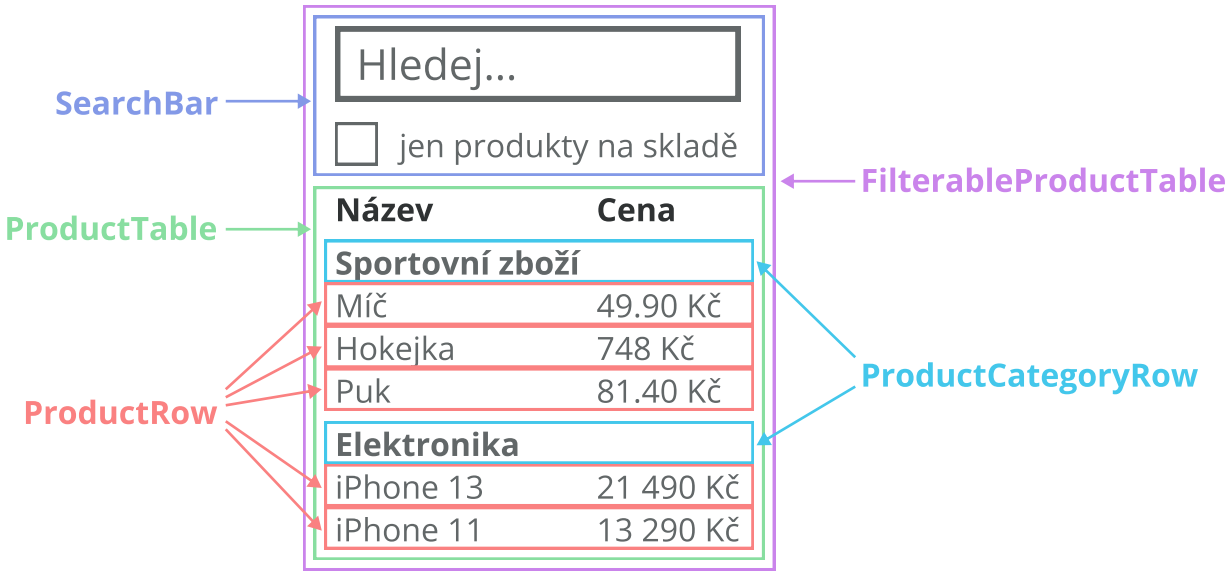
```
class CounterButton extends React.PureComponent
```

REACT - MYŠLENÍ V REACTU

Jedna ze skvělých částí Reactu je, jakým způsobem nás nutí přemýšlet o aplikaci když ji tvoříme.

1. ROZDĚLENÍ UI DO HIERARCHIE KOMPONENT

Rozdělíme si grafický návrh na části, které budou v aplikaci představovat komponenty.



2. VYTVOŘENÍ STATICKÉ VERZE V REACTU

Vytvoříme si v kódu UI bez žádné interaktivity. Používáme při tom jen props, state zatím ne. U jednodušších příkladech je jednodušší začít s horními komponentami a pokračovat směrem dolů (top-down). U složitějších příkladů může být jednodušší začít s dolními komponentami a pokračovat směrem nahoru (bottom-up).

3. IDENTIFIKOVÁNÍ MINIMÁLNÍ REPREZENTACE STATU

Identifikujeme minimální (ale kompletní) reprezentaci UI statu. Pokud jde něco odvodit z něčeho jiného, tak to být ve statu nemusí. Můžeme se sami sebe zeptat pár otázek na které když odpovímé ano, pravděpodobně to nepatří do statu.

- Je to předané od předka pomocí props?
- Zůstává to stále nezměněné?
- Dá se to odvodit podle jiných hodnot ve statu nebo props?

4. URČÍME KDE SE MÁ STATE NACHÁZET

React je celý o one-way data flow. Data putují jednou cestou směrem dolů. Podle toho si tedy musíme určit, kde se má nacházet state. Můžeme provést následující kroky:

- Indentifikování komponenty, která něco renderuje podle statu.
- Nalezení společného vlastníka (komponentu). Tato komponenta se bude nacházet nad všemi komponentami, které potřebují state.
- Nalezený vlastník nebo jiná komponenta nad vlastníkem by měla vlastnit state.
- Pokud nemůžeme najít vhodnou komponentu, ve které by bylo vhodné držet state, tak můžeme pro tento účel vytvořit novou.

5. PŘIDÁNÍ PŘEVRÁCENÉHO DATA FLOW

Předáme komponentám jako props funkce, které mohou volat pro měnění statu v jejich předcích.

REACT - HOOKS

Hooks nám umožňují použít state a další React funkce bez psaní class komponent. Jsou kompletně volitelné, 100% zpětně kompatibilní, a je jen na nás jestli je budeme používat.

PRAVIDLA PRO POUŽÍVÁNÍ HOOKS

Hooks jsou JavaScript funkce, ale musíme dodržovat dvě pravidla:

- Voláme je jen v nejvyšší úrovni function komponenty. Nevoláme je v cyklech, podmínkách nebo vnořených funkcích.
- Hooks voláme jen z function komponenty. Nevoláme je z klasických JavaScript funkcí. Ještě je můžeme volat z vlastní custom hook.

useState

Hook pro použití statu. Vrací state hodnotu a funkci, pomocí které tuto hodnotu můžeme měnit. Jako parametr můžeme předat defaultní hodnotu.

```
// hook useState vrací pole obsahující
// hodnotu a funkci k její změnění
// -můžeme použít destrukturovací syntaxi
const [value, setValue] = useState(0);
```

useEffect

Hook useEffect slouží ke stejnému účelu jako metody componentDidMount, componentDidUpdate a componentWillUnmount v class komponentách. Hodí se pro fetchování dat, nastavování odběrů a tak podobně.

Jako parametr předáváme funkci, která se bude volat po každém renderování (i po prvním).

```
useEffect(() => {
  document.title = `Počet: ${count}x`;
});
```

Pokud předaná funkce vrátí funkci, tak se tato funkce zavolá před dalším volání předané funkce nebo při unmountnutí komponenty.

```
useEffect(() => {
  const subscription = props.subscribe();
  return () => {
    // zde můžeme provést nějaký úklid
    subscription.unsubscribe();
  };
});
```

Pokud nechceme aby se předaná funkce volala po každém renderování, tak můžeme do useEffect předat také pole hodnot. Funkce se potom bude volat, když se v tomto poli změní hodnoty.

```
// tato funkce se spustí, jen když se
// změní hodnota props.source
useEffect(() => {
  const subscription = (
    props.source.subscribe()
  );
  return () => {
    subscription.unsubscribe();
  };
}, [props.source]);

// tato funkce se spustí jen jednou
useEffect(() => {console.log("1")}, []);
```

useContext

Tato hook bere context objekt a vrací jeho aktuální hodnotu.

```
const value = useContext(MyContext);
```

useRef

Pomocí hook useRef můžeme uložit hodnotu, kterou když změníme, tak se komponenta nebude rerenderovat.

```
const myRef = useRef(0);

// myRef jde používat jako normální ref
myRef.current = 1;
```

Nejvíce se refs používají k referencování elementů v HTML.

```
const inputEl = useRef(null);

return (
  <input ref={inputEl} type="text" />
);
```

useMemo

Hook useMemo můžeme použít, když máme nějakou drahou operaci, kterou nechceme provádět při každém renderování.

Jako parametr předáváme funkci a pole hodnot. Předaná funkce se spouští jen v případě, že se změní hodnoty v poli. Hodnotu kterou předaná funkce vrátí se pomocí hook useMemo uloží a vrací se při každém renderování.

```
const value = useMemo(() => {
  return vypocitejHodnotu(a, b);
}, [a, b]);
```

useCallback

Hook useCallback bere jako parametr funkci a pole hodnot. Funkce se pomocí useCallback uloží a bude se vracet při každém renderování namísto vytváření stále nové funkce. Funkce se vytvoří znovu jen v případě, že se změní některé z hodnot v předaném poli.

```
const funkce = useCallback(() => {
    doSomething(a, b);
}, [a, b]);
```

CUSTOM HOOKS

Někdy můžeme chtít znovu použít nějakou stateful logiku mezi více komponentami. K tomu si můžeme vytvořit vlastní hook.

Custom Hook můžeme vytvořit jako JavaScript funkci, kterou na začátku pojmenuje s textem "use". Uvnitř této funkce můžeme používat jiné hooks. Custom Hook nemusí mít žádnou specifickou strukturu, je na nás co bude přijímat za argumenty a co bude vracet (pokud bude něco vracet).

```
const useFetch = (url) => {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch(url)
      .then((res) => res.json())
      .then((data) => setData(data));
  }, [url]);

  return [data];
};

export default useFetch;
```