

REACT Docs - Polski

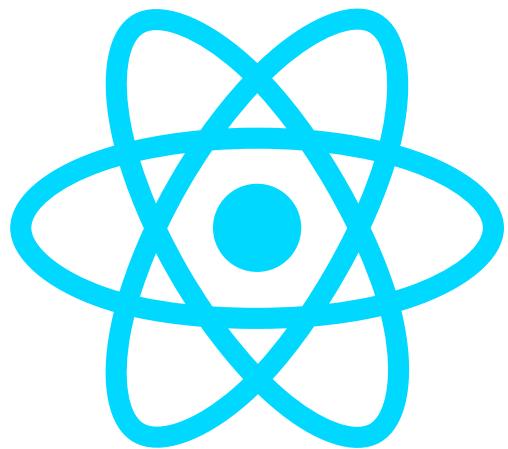


Table of contents

• Accessibility	5
• Add react /to /a /website	17
• Addons animation	22
• Addons create /fragment	29
• Addons perf	31
• Addons pure /render /mixin	35
• Addons shallow /compare	36
• Addons shallow /renderer	37
• Addons test /utils	39
• Addons two /way /binding /helpers	46
• Addons update	49
• Addons	52
• Cdn links	54
• Code splitting	55
• Codebase overview	61
• Components and /props	65
• Composition vs /inheritance	70
• Conditional rendering	74
• Context	79
• Create a /new /react /app	86
• Cross origin /errors	89
• Design principles	91
• Error boundaries	98
• Error decoder	103
• Faq ajax	104
• Faq build	107
• Faq functions	108
• Faq internals	115
• Faq state	116
• Faq structure	119
• Faq styling	121
• Faq versioning	123
• Forms	126
• Forwarding refs	132

• Fragments	135
• Getting started	138
• Handling events	142
• Hello world	145
• Higher order /components	147
• Hooks custom	156
• Hooks effect	162
• Hooks faq	174
• Hooks intro	196
• Hooks overview	201
• Hooks reference	208
• Hooks rules	226
• Hooks state	229
• How to /contribute	236
• Implementation notes	241
• Integrating with /other /libraries	259
• Introducing jsx	269
• Jsx in /depth	273
• Legacy context	282
• Legacy event /pooling	287
• Lifting state /up	288
• Lists and /keys	296
• Optimizing performance	301
• Portals	310
• React without /es6	313
• React without /jsx	318
• Reconciliation	319
• Reference dom /elements	324
• Reference events	329
• Reference glossary	337
• Reference javascript /environment /requirements	342
• Reference profiler	343
• Reference pure /render /mixin	346
• Reference react /component	348
• Reference react /dom /client	364
• Reference react /dom /server	366

• Reference react /dom	371
• Reference react	376
• Reference test /renderer	386
• Refs and /the /dom	392
• Release channels	398
• Render props	401
• Rendering elements	408
• State and /lifecycle	411
• Static type /checking	420
• Strict mode	427
• Testing environments	433
• Testing recipes	436
• Testing	447
• Thinking in /react	449
• Typechecking with /proptypes	455
• Uncontrolled components	460
• Web components	462

Dostępność

Czym jest dostępność?

Pojęcie dostępności stron internetowych (określanej również **a11y** od ang. *accessibility*) zostało zaprojektowane i stworzone z myślą o internecie przystępnym dla wszystkich. Wspieranie dostępności jest niezbędne, aby umożliwić technologiom asystującym poprawną interpretację stron.

React w pełni wspiera budowanie dostępnych dla wszystkich stron internetowych, często z wykorzystaniem standardowych technik HTML.

Standard oraz wytyczne

WCAG

[Web Content Accessibility Guidelines](#) dostarcza zbiór wytycznych, jak tworzyć poprawne oraz dostępne dla wszystkich strony internetowe.

Poniższe listy kontrolne WCAG zawierają przegląd:

- [Lista kontrolna WCAG stworzona przez Wuhcag](#)
- [Lista kontrolna WCAG stworzona przez WebAIM](#)
- [Lista kontrolna projektu A11Y](#)

WAI-ARIA

Dokument [Web Accessibility Initiative - Accessible Rich Internet Applications](#) zawiera listę technik pomagających w budowaniu w pełni dostępnych aplikacji javascriptowych.

Warto zaznaczyć, że wszystkie atrybuty HTML `aria-*` są w pełni wspierane przez JSX. Mimo że większość tagów oraz atrybutów DOM w Reakcie zapisujemy w notacji camelCase, te związane z dostępnością powinny być zapisywane z wykorzystaniem myślników (znanych również jako kebab-case, lisp-case itp.), ponieważ są one traktowane jak zwykłe atrybuty HTML.

```
<input  
  type="text"  
  aria-label={etykieta}  
  aria-required="true"  
  onChange={obserwatorZdarzenia}  
  value={wartoscPola}  
  name="imie"  
/>
```

Semantyczny HTML

Semantyczny HTML jest podstawą dostępności aplikacji webowych. Wykorzystując różne elementy HTML, które wzmacniają znaczenie informacji na naszych stronach, bardzo często możemy stworzyć w pełni dostępną stronę bez dodatkowego nakładu pracy.

- [Dokumentacja elementów HTML na MDN](#)

Czasem łamiemy zasady semantycznego HTML, kiedy dodajemy dodatkowy element `div` do naszego kodu JSX, aby uruchomić aplikację. Dzieje się tak zwłaszcza kiedy pracujemy z listami (``, `` czy `<dl>`) oraz tabelami `<table>`. W takim przypadkach powinniśmy korzystać z [fragmentów reactowych](#), które pozwalają na grupowanie elementów.

Przykład:

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

function ListItem({ item }) {
  return (
    <Fragment>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </Fragment>
  );
}

function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        <ListItem item={item} key={item.id} />
      ))}
    </dl>
  );
}
```

Możesz mapować kolekcje elementów do tablicy fragmentów, zupełnie jak w przypadku innych typów elementów:

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Fragmenty zawsze powinny mieć ustawioną wartość `key` podczas mapowania
        // kolekcji
        <Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </Fragment>
      ))}
    </dl>
  );
}
```

Jeśli nie chcesz przekazywać żadnych dodatkowych właściwości do Fragmentu, wówczas możesz użyć skróconej składni. Upewnij się, że wspomniany zapis jest wspierany przez używane przez ciebie środowisko.

```
function ListItem({ item }) {
  return (
    <>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </>
  );
}
```

Więcej informacji znajdziesz w [dokumentacji fragmentów](#).

Dostępne formularze

Nadawanie etykiet

Każdy element kontrolujący formularz, taki jak `input` czy `textarea`, powinien być etykietowany w przystępny sposób. Etykieta powinna dobrze opisywać pole tekstowe i być widoczna dla czytników ekranowych.

Poniższe zasoby opisują, jak robić to poprawnie:

- [W3C - etykietowanie elementów](#)
- [WebAim - etykietowanie elementów](#)
- [Grupa Paciello wyjaśnia przystępność nazw](#)

Chociaż tych standardowych praktyk HTML-owych można używać bezpośrednio w Reakcie, zauważ, że atrybut `for` jest w składni JSX zapisywany jako `htmlFor`:

```
<label htmlFor="imiePoleTekstowe">Imię:</label>
<input id="imiePoleTekstowe" type="text" name="imie"/>
```

Powiadamianie użytkownika o błędach

W sytuacji zgłoszenia błędu, komunikaty muszą być zrozumiałe dla wszystkich użytkowników. Poniższe linki pokazują, jak wyświetlić błędy w sposób zrozumiały dla czytników ekranowych.

- [W3C - notyfikowanie użytkowników](#)
- [WebAIM - walidacja formularzy](#)

Kontrola fokusa

Upewnij się, że twoją aplikację internetową można w pełni obsługiwać za pomocą samej klawiatury:

- [WebAIM - dostępność z wykorzystaniem klawiatury](#)

Fokus klawiaturowy a kontur

Fokus klawiaturowy odnosi się do bieżącego elementu w DOM, który został wybrany poprzez zdarzenia wywołane przez klawiaturę. Zazwyczaj oznaczany jest za pomocą konturu, podobnego do tego na obrazku poniżej:



Jeśli decydujesz się na usunięcie konturu, np. ustawiając właściwość CSS `outline: 0`, nie zapomnij zastąpić go inną implementacją konturu.

Przejście do treści

Zapewnij mechanizm umożliwiający użytkownikom pominięcie sekcji nawigacji na stronie, ponieważ ułatwia on i przyspiesza nawigowanie z wykorzystaniem klawiatury.

Łącza typu "Przejdź do treści" lub "Pomiń nawigację" to specjalne, ukryte linki nawigacyjne, które stają się widoczne tylko wtedy, gdy użytkownicy klawiatury wchodzą w interakcję ze stroną. Są bardzo łatwe w implementacji z wykorzystaniem wewnętrznych kotwic oraz odrobin stylowania:

- [WebAIM - łącza pomijające nawigację](#)

Używaj również elementów i punktów orientacyjnych, takich jak `<main>` i `<aside>`, aby rozgraniczyć sekcje strony. Dzięki nim technologie wspierające pozwalają użytkownikowi na szybkie przemieszczanie się między sekcjami.

Przeczytaj więcej o wykorzystaniu tych elementów w celu zwiększenia dostępności:

- [Dostępne punkty orientacyjne](#)

Programowe zarządzanie fokusem

Nasze aplikacje Reactowe nieustannie modyfikują HTML DOM w czasie działania, co chwilami prowadzi do utraty konturu aktywnego elementu lub ustawienia go na nieoczekiwany element. Aby to naprawić, musimy ręcznie ustawić fokus we właściwym miejscu. Przykładowo, jeśli użytkownik zamknie okno modalne, fokus mógłby zostać przeniesiony na przycisk, który to okno otworzył.

Dokumentacja MDN opisuje dokładniej, w jaki sposób możemy tworzyć [widżety javascriptowe z obsługą klawiatury](#).

W Reakcie, aby ustawić fokus, możemy posłużyć się mechanizmem [referencji do elementu DOM](#).

Aby móc skorzystać z tego mechanizmu, w kodzie JSX wybranego komponentu tworzymy referencję `ref` do elementu :

```

class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // Stwórz referencję do elementu DOM
    this.textInput= React.createRef();
  }
  render() {
    // Użyj funkcji zwrotnej `ref`, aby zapisać referencję do pola tekstowego
    // we właściwości instancji komponentu (na przykład this.textInput).
    return (
      <input
        type="text"
        ref={this.textInput}
      />
    );
  }
}

```

Wówczas, w razie potrzeby, możemy przenieść fokus na inny element naszego komponentu:

```

focus() {
  // Jawne przeniesienie fokusa na pole tekstowe przy użyciu natywnego interfejsu
  // DOM
  // Uwaga: korzystamy z "current", aby uzyskać dostęp do węzła DOM
  this.textInput.current.focus();
}

```

Czasami komponent nadrzedny musi ustawić fokus na element komponentu podległego. Możemy to zrobić poprzez [przesłanie referencji "w górę" do komponentu nadrzednego](#) za pomocą specjalnej właściwości nadanej komponentowi podległemu.

```

function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.inputElement = React.createRef();
  }
  render() {
    return (
      <CustomTextInput inputRef={this.inputElement} />
    );
  }
}

// Teraz możesz ręcznie ustawać fokus, kiedy to potrzebne.
this.inputElement.current.focus();

```

Kiedy używasz [HOC-a](#) do rozszerzenia komponentów, zaleca się [przekazanie referencji](#) do opakowanego komponentu przy użyciu funkcji `forwardRef`, która wbudowana jest w Reacta. Jeśli wybrany HOC z którejś zewnętrznej biblioteki nie implementuje takiego przekierowania, można użyć powyższego wzorca jako wyjście awaryjne.

Doskonałym przykładem zarządzania fokusem jest biblioteka [react-aria-modal](#). Jest to stosunkowo rzadki przykład w pełni dostępnego okna modalnego. Nie tylko ustawia początkowy fokus na przycisku zamykającym okno (uniemożliwiając tym samym użytkownikowi klawiatury przypadkowe aktywowanie akcji akceptującej) i zatrzymuje fokus klawiaturowy wewnątrz okna, lecz dodatkowo po zamknięciu przywraca fokus z powrotem na element, który zainicjował otwarcie okna.

Uwaga:

Chociaż jest to bardzo ważna technika zapewniająca dostępność, należy stosować jej z umiarem. Użyj jej, aby skorygować naturalną "drogę" fokusa w aplikacji, ale nie próbuj przewidzieć, jak użytkownicy będą chcieli korzystać z aplikacji i nie wymuszaj własnej "drogi".

Zdarzenia myszy i wskaźnika

Upewnij się, że wszystkie funkcje dostępne dla korzystających z myszy lub wskaźnika są również osiągalne za pomocą samej klawiatury. Poleganie na samych urządzeniach wskazujących prowadzi często do sytuacji, w których użytkownicy klawiatury nie mogą w ogóle korzystać z aplikacji.

Aby to zilustrować, spójrzmy na przykład zepsutej dostępności spowodowanej obsługą wyłącznie zdarzenia kliknięcia. Dotyczy to sytuacji, w której użytkownik może zamknąć otwarty "dymek" poprzez kliknięcie gdzieś poza nim.

Select an option

Load the option

Remove the option

Zazwyczaj jest to implementowane poprzez nasłuchiwanie zdarzenia `click` w obiekcie `window`, które zamkna dymek:

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

    this.state = { isOpen: false };
    this.toggleContainer = React.createRef();

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onClickOutsideHandler = this.onClickOutsideHandler.bind(this);
  }
```

```

componentDidMount() {
  window.addEventListener('click', this.onClickOutsideHandler);
}

componentWillUnmount() {
  window.removeEventListener('click', this.onClickOutsideHandler);
}

onClickHandler() {
  this.setState(currentState => ({
    isOpen: !currentState.isOpen
  }));
}

onClickOutsideHandler(event) {
  if (this.state.isOpen && !this.toggleContainer.current.contains(event.target))
  {
    this.setState({ isOpen: false });
  }
}

render() {
  return (
    <div ref={this.toggleContainer}>
      <button onClick={this.onClickHandler}>Wybierz opcję</button>
      {this.state.isOpen && (
        <ul>
          <li>Opcja 1</li>
          <li>Opcja 2</li>
          <li>Opcja 3</li>
        </ul>
      )}
    </div>
  );
}
}

```

Powyższy przykład działa poprawnie dla użytkowników korzystających ze wskaźników, takich jak np. mysz. Jednakże, obsługiwanie za pomocą samej klawiatury prowadzi do problemu przy przechodzeniu do następnego elementu listy za pomocą tabulatora. Dzieje się tak, ponieważ obiekt `window` nigdy nie otrzymuje zdarzenia `click`. Może to doprowadzić do uniemożliwienia użytkownikom korzystania z aplikacji.

Select an option

Load the option

Remove the option

Ta sama funkcjonalność może zostać uzyskana poprzez użycie odpowiednich procedur obsługi zdarzeń, takich jak `onBlur` i `onFocus`:

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

    this.state = { isOpen: false };
    this.timeOutId = null;

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onBlurHandler = this.onBlurHandler.bind(this);
    this.onFocusHandler = this.onFocusHandler.bind(this);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  // Zamykamy dymek w kolejnym cyklu za pomocą funkcji setTimeout.
  // Jest to konieczne, ponieważ musimy najpierw sprawdzić,
  // czy inny potomek elementu otrzymał fokus, jako że
  // zdarzenie onBlur wywołuje się przed ustawieniem fokusa
  // na innym elemencie.
  onBlurHandler() {
    this.timeOutId = setTimeout(() => {
      this.setState({
        isOpen: false
      });
    }, 100);
  }

  // Jeśli potomek otrzyma fokus, nie zamykaj dymku.
  onFocusHandler() {
    clearTimeout(this.timeOutId);
  }

  render() {
    // React wspiera nas w przesyłaniu fokusa
    // do rodzica.
    return (
      <div onBlur={this.onBlurHandler}
            onFocus={this.onFocusHandler}>
        <button onClick={this.onClickHandler}
                  aria-haspopup="true"
                  aria-expanded={this.state.isOpen}>
          Wybierz opcję
        </button>
        {this.state.isOpen && (
          <ul>
            <li>Opcja 1</li>
            <li>Opcja 2</li>
            <li>Opcja 3</li>
          </ul>
        )}
      </div>
    );
  }
}
```

Ten kod udostępnia funkcje użytkownikom zarówno urządzeń wskaźnikowych, jak i klawiatury. Zwróć także uwagę na właściwości `aria-` * dodane w celu obsłużenia czytników ekranu. Dla uproszczenia kodu nie zostały zaimplementowane zdarzenia klawiaturowe pozwalające na interakcję z dymkiem za pomocą klawiszy strzałek.

Select an option

Load the option

Remove the option

Jest to tylko jeden przykład z wielu przypadków, w których poleganie jedynie na zdarzeniach wskaźnika i myszy możemy uniemożliwić poruszanie się po aplikacji użytkownikom korzystającym z samej klawiatury. Każdorazowe testowanie aplikacji za pomocą klawiatury pozwala na sprawne wyszukiwanie problemów, którym można zaradzić poprzez dodanie obsługi zdarzeń klawiaturowych.

Bardziej złożone widżety

Bardziej złożone scenariusze niekoniecznie muszą być mniej dostępne dla użytkowników. Dostępność najłatwiej osiągnąć poprzez trzymanie się jak najbliżej wzorców znanych z natywnego HTML-a. Nawet najbardziej złożony widżet może być przyjazny dla użytkownika.

Wymagamy tutaj znajomości standardu ARIA, m.in. [ról](#) oraz [stanów i właściwości](#). Są to "skrzynki narzędziowe" wypełnione atrybutami HTML, które są w pełni obsługiwane przez JSX i umożliwiają nam tworzenie w pełni dostępnych, wysoce funkcjonalnych komponentów reactowych.

Każdy typ widżetu ma określone wzorce i zarówno użytkownicy, jak i przeglądarki oczekują, że będzie działał w określony sposób.

- [Dobre praktyki WAI-ARIA - Wzorce projektowe i widżety](#)
- [Heydon Pickering - ARIA w praktyce](#)
- [Inclusive Components](#)

Inne punkty do rozważenia

Ustawianie języka

Jawnie wskaż ludzki język tekstów zamieszczonych na stronie, ponieważ oprogramowanie czytnika ekranu używa go do wyboru prawidłowych ustawień głosu:

- [WebAIM - Język dokumentu](#)

Ustawienie tytułu dokumentu

Ustaw znacznik `<title>` dokumentu tak, by poprawnie opisywał zawartość strony. Ułatwia to użytkownikowi zrozumienie bieżącego kontekstu strony.

- [WCAG - Dlaczego tytuł dokumentu jest wymagany](#)

W Reakcie tytuł dokumentu możemy ustawić za pomocą [reactowego komponentu tytułu dokumentu](#).

Kontrast kolorów

Upewnij się, że wszystkie teksty na twojej stronie mają wystarczający kontrast kolorów, aby pozostały mak-symalnie czytelne dla użytkowników o słabym wzroku:

- [WCAG - Dlaczego wymagane jest zachowanie kontrastu kolorów](#)
- [Wszystko o kontraste kolorów i dlaczego warto się nad nim zastanowić](#)
- [A11yProject - Czym jest kontrast kolorów](#)

Ręczne obliczanie odpowiednich kombinacji kolorów dla wszystkich przypadków na swojej stronie internetowej może być nudne. Zamiast tego możesz użyć [Colorable do przeliczenia całej palety kolorów dla zachowania dobrej dostępności aplikacji](#).

Jeśli chcesz rozszerzyć możliwości testowania kontrastu, możesz użyć następujących narzędzi:

- [WebAIM - Sprawdzanie kontrastu kolorów](#)
- [The Paciello Group - Analizator kontrastu kolorów](#)

Narzędzia do tworzenia oraz testowania

Istnieje wiele narzędzi, których możemy użyć, aby pomóc sobie przy tworzeniu przystępnych aplikacji internetowych.

Klawiatura

Zdecydowanie najłatwiejszą i jedną z najważniejszych kontroli jest sprawdzenie, czy poruszanie się po całej stronie jest możliwe z wykorzystaniem wyłącznie klawiatury. Instrukcja sprawdzenia aplikacji:

1. Odłącz mysz od komputera.
2. Używaj wyłącznie klawiszy `Tab` oraz `Shift + Tab` do przeglądania strony.
3. Używaj klawisza `Enter` do aktywowania elementów.
4. W razie potrzeby używaj klawiszy strzałek do interakcji z niektórymi elementami, takimi jak menu i listy rozwijane.

Pomoc przy tworzeniu

Część testów dostępności możemy wykonać bezpośrednio w naszym kodzie JSX. Często kontrole dostępności dla ról, stanów i właściwości ARIA są wbudowane w środowisko IDE obsługujące JSX. Dodatkowo, mamy do dyspozycji również inne narzędzia:

eslint-plugin-jsx-a11y

Wtyczka [eslint-plugin-jsx-a11y] (<https://github.com/evcohen/eslint-plugin-jsx-a11y>) dla narzędzia ESLint informuje o problemach z dostępnością w twoim kodzie JSX. Wiele środowisk IDE umożliwia integrację ostrzeżeń o zgłoszonych problemach z dostępnością bezpośrednio z narzędziami do analizy kodu i oknami edytorów.

Create React App ma tę wtyczkę domyślnie zainstalowaną z aktywnymi niektórymi regułami. Jeśli chcesz włączyć dodatkowe reguły dotyczące dostępności, możesz utworzyć plik `.eslintrc` w katalogu głównym swojego projektu z następującą zawartością:

```
{
  "extends": ["react-app", "plugin:jsx-a11y/recommended"],
  "plugins": ["jsx-a11y"]
}
```

Testowanie dostępności w przeglądarce

Istnieje wiele narzędzi, które umożliwiają przeprowadzanie kontroli dostępności na stronach internetowych bezpośrednio w przeglądarce. Używaj ich w połączeniu z innymi narzędziami wymienionymi tutaj, aby jak najlepiej przygotować swój kod HTML.

aXe, aXe-core oraz react-axe

Deque Systems oferuje [aXe-core](#) do automatycznych i kompleksowych testów dostępności aplikacji. Moduł ten obejmuje integracje z Selenium.

The Accessibility Engine (lub inaczej aXe) jest rozszerzeniem przeglądarkowego inspektora dostępności, zbudowanym na bazie [aXe-core](#).

Możesz również użyć modułu [@axe-core/react](#), aby zgłosić luki dotyczące dostępności bezpośrednio do konсолi, podczas rozwoju aplikacji i debugowania.

WebAIM WAVE

[Web Accessibility Evaluation Tool](#) jest kolejną wtyczką pomagającą w testowaniu dostępności.

Inspektory dostępności i Accessibility Tree

The Accessibility Tree jest podzbiorem drzewa DOM. Zawiera wszystkie dostępne dla technologii wspomagających obiekty, odpowiadające każdemu elementowi modelu DOM.

W niektórych przeglądarkach możemy łatwo wyświetlić informacje o dostępności dla każdego elementu w drzewie DOM:

- [Korzystanie z Inspektora dostępności w Firefoksie](#)
- [Korzystanie z Inspektora dostępności w Chromie](#)
- [Korzystanie z Inspektora dostępności w OS X Safari](#)

Czytniki ekranowe

Testowanie za pomocą czytnika ekranu powinno stanowić część testów dostępności. Należy pamiętać, że kombinacje przeglądarka/czytnik ekranu mają bardzo duże znaczenie. Zaleca się testowanie aplikacji w przeglądarce rekomendowanej do wybranego czytnika ekranu.

Często używane czytniki ekranu

NVDA w Firefoxie

[NonVisual Desktop Access](#) (lub NVDA) to czytnik ekranu systemu Windows o otwartym kodzie źródłowym, który jest szeroko stosowany.

Zapoznaj się z następującymi poradnikami opisującymi, jak najlepiej wykorzystać NVDA:

- [WebAIM - Korzystanie z NVDA do oceny dostępności stron](#)
- [Deque - NVDA: Skróty klawiszowe](#)

VoiceOver w Safari

VoiceOver to zintegrowany czytnik ekranu na urządzeniach Apple.

Zapoznaj się z następującymi przewodnikami dotyczącymi aktywacji i korzystania z VoiceOver:

- [WebAIM - Korzystanie z VoiceOver do oceny dostępności stron internetowych](#)
- [Deque - VoiceOver dla OS X: Skróty klawiszowe](#)
- [Deque - VoiceOver dla iOS: Skróty](#)

JAWS w Internet Explorer

[Job Access With Speech](#) (lub JAWS) jest popularnym czytnikiem ekranu w systemie Windows.

Zapoznaj się z następującymi poradnikami, jak najlepiej korzystać z JAWS:

- [WebAIM - Korzystanie z JAWS do oceny dostępności stron internetowych](#)
- [Deque - JAWS: Skróty klawiszowe](#)

Inne czytniki ekranowe

ChromeVox w Google Chrome

[ChromeVox] (<https://www.chromevox.com/>) jest zintegrowanym czytnikiem ekranu na Chromebookach i jest dostępny [jako rozszerzenie] (<https://chrome.google.com/webstore/detail/chromevox/kgejglhpjiefp-pelplmljglcjbhoiplfn?hl=pl>) dla Google Chrome.

Zapoznaj się z następującymi poradnikami opisującymi, jak najlepiej korzystać z ChromeVox:

- [System operacyjny Chrome - Pomoc: Korzystanie z wbudowanego czytnika ekranu](#)
- [ChromeVox - klasyczne skróty klawiszowe](#)

[Go to TOC](#)

Dodaj Reacta do swojej strony

Używaj Reacta wybiórczo w miarę potrzeb.

Biblioteka React została zaprojektowana z myślą o stopniowym wdrażaniu. Dzięki temu zawarte w niej rozwiązania możesz stosować wybiórczo w zależności od potrzeb. Być może chcesz jedynie dodać trochę interaktywności do istniejącej strony. Komponenty reactowe świetnie się do tego nadają.

Większość stron internetowych nie jest i nie musi być aplikacjami jednostronиковymi (ang. *single-page apps*). Wypróbuj Reacta na niewielkiej części swojej strony za pomocą **kilku linii kodu i bez narzędzi komplikacji**. Następnie możesz stopniowo zwiększać jego wykorzystanie lub poprzedzić na kilku dynamicznych widżetach.

- [Dodaj Reacta w mgnieniu oka](#)
- Opcjonalnie: [Wypróbuj Reacta z JSX](#) (bundler nie jest wymagany!)

Dodaj Reacta w mgnieniu oka

W tej sekcji pokażemy, jak dodać komponent reactowy do strony internetowej. Możesz to zrobić korzystając z własnej strony internetowej lub stworzyć nowy plik HTML.

Nie będziemy instalowali żadnych zależności ani korzystali ze skomplikowanych narzędzi -- **aby ukończyć ten rozdział, będziesz potrzebować jedynie połączenia internetowego i odrobiny wolnego czasu**.

Opcjonalnie: [Pobierz cały przykład \(2KB po kompresji\)](#)

Krok 1: Dodaj kontener do HTML

Najpierw otwórz stronę internetową, którą chcesz edytować. Dodaj pusty znacznik `<div>`, aby określić miejsce, w którym React będzie wyświetlał treść. Na przykład:

```
<!-- ... istniejący kod HTML ... -->
<div id="like_button_container"></div>
<!-- ... istniejący kod HTML ... -->
```

Przypisaliśmy temu znacznikowi `<div>` unikalny atrybut `id`. Pozwoli nam to na późniejsze odnalezienie kodu za pomocą JavaScripta oraz na wyświetlenie w nim komponentu reactowego.

Wskazówka

"Kontenery" `<div>` można umieszczać **w dowolnych miejscach** wewnątrz znacznika `<body>`. Dopuszczalna liczba niezależnych kontenerów na stronie jest nieograniczona. Zazwyczaj są one puste -- React i tak zastąpi istniejącą zawartość w kontenerze.

Krok 2: Dodaj znaczniki `<script>` [{#step-2-add-the-script-tags}](#)

Następnie dodaj trzy znaczniki `<script>` do strony HTML tuż przed zamykającym znacznikiem `</body>`:

```
<!-- ... pozostała część kodu HTML ... -->
<!-- Załaduj Reacta. -->
<!-- Uwaga: podczas wdrażania aplikacji do środowiska produkcyjnego, zamień
"development.js" na "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>

<!-- Załaduj nasz komponent reactowy. -->
<script src="like_button.js"></script>

</body>
```

Pierwsze dwa znaczniki wczytają Reacta. Trzeci załadowuje kod twojego komponentu.

Krok 3: Stwórz komponent reactowy

Stwórz plik `like_button.js` w folderze, obok pliku HTML ze stroną.

Otwórz [szablon startowy](#) i wklej jego zawartość do utworzonego pliku.

Wskazówka

Kod ten definiuje komponent reactowy o nazwie `LikeButton`. Nie przejmuj się, jeśli jeszcze czegoś nie rozumiesz na tym etapie -- podstawy Reacta wyjaśnimy później w naszym [praktycznym samouczku](#) i w rozdziale pt. "Główne idee". Na razie jednak skupmy się na wyświetleniu tego na ekranie!

Pod kodem z [szablonu startowego](#) w pliku `like_button.js` dodaj trzy linijki:

```
// ... wklejony kod startowy ...
const domContainer = document.querySelector('#like_button_container');
const root = ReactDOM.createRoot(domContainer);
root.render(e(LikeButton));
```

Te trzy linijki kodu są odpowiedzialne za znalezienie naszego elementu `<div>` z pierwszego kroku, stworzenie w nim aplikacji reactowej i wyświetlenie przycisku `"Like"` pochodzącego z wklejonego komponentu.

To wszystko!

Nie ma kroku czwartego. **Właśnie udało ci się dodać do strony pierwszy komponent reactowy.**

Zapoznaj się z kolejnymi rozdziałami, aby uzyskać więcej informacji na temat integrowania strony z Reactem.

[Zobacz przykładowy kod źródłowy](#)[Pobierz cały przykład \(2KB po kompresji\)](#)

Wskazówka: Wykorzystuj komponenty wielokrotnie

Często zdarza się, że komponent trzeba wyświetlić na stronie kilka razy. Oto przykład, który trzykrotnie wyświetla przycisk "Like" i przekazuje do niego dane:

[Zobacz przykładowy kod źródłowy](#)[Pobierz cały przykład \(2KB po kompresji\)](#)

Wskazówka

Ta strategia jest szczególnie przydatna, gdy części strony oparte na Reakcie są od siebie odizolowane. W kodzie reactowym łatwiej jest zamiast tego korzystać z [kompozycji komponentów](#).

Wskazówka: Minifikuj JavaScript dla potrzeb środowiska produkcyjnego

Przed wdrożeniem strony do środowiska produkcyjnego, pamiętaj, że niezminifikowany kod javascriptowy może znacznie spowolnić działanie twojej strony.

Jeżeli twoje skrypty są już zminifikowane i jeśli wdrożony dokument HTML ładuje wersję Reacta z końcówką `production.min.js` w nazwie, wówczas Twoja aplikacja będzie gotowa do użytku w środowisku produkcyjnym:

```
<script src="https://unpkg.com/react@18/umd/react.production.min.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.production.min.js"
crossorigin></script>
```

Jeżeli nie masz jeszcze skonfigurowanej minifikacji skryptów, [oto jeden ze sposobów, aby to zrobić](#).

Opcjonalnie: Wypróbuj Reacta z JSX

W przykładach powyżej opieramy się tylko na funkcjonalnościach, które są natywnie obsługiwane przez przeglądarki. Dlatego wywołaliśmy funkcję javascriptową, aby poinformować Reacta, co ma wyświetlić:

```
const e = React.createElement;

// Wyświetla przycisk "Lubię to!"
return e(
  'button',
  { onClick: () => this.setState({ liked: true }) },
  'Lubię to!'
);
```

Niemniej React oferuje również możliwość użycia składni JSX:

```
// Wyświetla przycisk "Lubię to!"
return (
  <button onClick={() => this.setState({ liked: true })}>
    Lubię to!
  </button>
);
```

Te dwa fragmenty kodu działają w ten sam sposób. Mimo iż **składnia JSX jest całkowicie opcjonalna**, wiele osób uważa, że jest pomocna przy pisaniu kodu dla UI -- zarówno w Reakcie, jak i z innymi bibliotekami.

Möżesz wypróbować składnię JSX korzystając z [tego konwertera online](#).

Szybki test składni JSX

Najszybszym sposobem na wypróbowanie składni JSX w projekcie jest dodanie następującego znacznika `<script>` do strony:

```
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
```

Od teraz możesz używać składni JSX wewnątrz dowolnego znacznika `<script>`, który zaopatrzony jest w atrybut `type="text/babel"`. Tutaj jest [przykład pliku HTML z JSX](#), który możesz pobrać i wypróbować.

Takie podejście jest dobre do nauki i tworzenia projektów na prezentacje. Powoduje to jednak spowolnienie działania strony i **nie jest odpowiednie dla środowiska produkcyjnego**. Kiedy już zdecydujesz, że możesz iść naprzód z materiałem, usuń ostatnio dodany znacznik `<script>` i atrybuty `type="text/babel"` z innych znaczników. W kolejnym podrozdziale skonfigurujemy preprocesor składni JSX, który będzie konwertował zawartość wszystkich znaczników `<script>` automatycznie.

Dodaj JSX do projektu

Dodanie JSX do projektu nie wymaga skomplikowanych narzędzi, takich jak bundler czy serwer deweloperski. Zasadniczo dodawanie JSX **przypomina dodawanie preprocesora CSS**. Jedynym wymaganiem jest posiadanie zainstalowanego na komputerze środowiska [Node.js](#).

Przejdź do folderu projektu i wywołaj dwie komendy:

- Krok 1:** Uruchom `npm init -y` (jeżeli wywołanie się nie powiedzie - tutaj opisano, jak obejść problem)
- Krok 2:** Uruchom `npm install babel-cli@6 babel-preset-react-app@3`

Wskazówka

Korzystamy tu z `npm` tylko do instalacji preprocesora JSX; nie będzie on potrzebny do niczego innego. Zarówno React, jak i kod aplikacji mogą pozostać bez zmian w znacznikach `<script>`.

Gratulacje! Właśnie udało ci się dodać do swojego projektu **konfigurację JSX gotową do wdrożenia do środowiska produkcyjnego**.

Uruchom preprocessora JSX

Utwórz folder o nazwie `src` i uruchom polecenie w terminalu:

```
npx babel --watch src --out-dir . --presets react-app/prod
```

Uwaga

`npx` to nie literówka -- jest to [narzędzie do uruchamiania pakietów, dostępne w npm 5.2+](#).

Jeśli wyświetli się komunikat o błędzie: "You have mistakenly installed the `babel` package" (pol. "Błędnie zainstalowano pakiet `babel`"), możliwe, że pominięty został [poprzedni krok](#). Wykonaj go w tym samym folderze i spróbuj ponownie.

Nie czekaj na jego zakończenie -- to polecenie uruchamia skrypt automatycznie śledzący zmiany w kodzie ze składnią JSX.

Jeśli utworzysz teraz plik o nazwie `src/like_button.jsx` przy użyciu [tego przykładowego kodu JSX](#), obserwator (ang. *watcher*) utworzy przetworzony plik o nazwie `like_button.js` zawierający kod JavaScriptowy zrozumiały dla przeglądarki. Gdy wyedytujesz plik źródłowy z kodem JSX, transformacja zostanie ponownie uruchomiona automatycznie.

Dodatkowo umożliwia to również korzystanie z nowych składni JavaScript, takich jak klasy, bez obawy o brak kompatybilności ze starszymi przeglądarkami. Narzędzie, którego właśnie użyliśmy, nazywa się Babel. Więcej na jego temat możesz dowiedzieć się z [dokumentacją](#).

Jeśli uważasz, że czujesz się już komfortowo z narzędziami do komplikacji i chcesz, by robiły za ciebie więcej, zajrzyj do [następnego rozdziału](#), w którym opisaliśmy niektóre z najbardziej popularnych i przystępnych narzędzi. Jeżeli nie -- cóż, znaczniki `script` też wystarczą!

Animation Add-Ons

Note:

`ReactTransitionGroup` and `ReactCSSTransitionGroup` have been moved to the `react-transition-group` package that is maintained by the community. Its 1.x branch is completely API-compatible with the existing addons. Please file bugs and feature requests in the [new repository](#).

The `ReactTransitionGroup` add-on component is a low-level API for animation, and `ReactCSSTransitionGroup` is an add-on component for easily implementing basic CSS animations and transitions.

High-level API: ReactCSSTransitionGroup

`ReactCSSTransitionGroup` is a high-level API based on `ReactTransitionGroup` and is an easy way to perform CSS transitions and animations when a React component enters or leaves the DOM. It's inspired by the excellent [ng-animate](#) library.

Importing

```
import ReactCSSTransitionGroup from 'react-transition-group'; // ES6
var ReactCSSTransitionGroup = require('react-transition-group'); // ES5 with npm
```

```
class TodoList extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: ['hello', 'world', 'click', 'me']};
    this.handleAdd = this.handleAdd.bind(this);
  }

  handleAdd() {
    const newItems = this.state.items.concat([
      prompt('Enter some text')
    ]);
    this.setState({items: newItems});
  }

  handleRemove(i) {
    let newItems = this.state.items.slice();
    newItems.splice(i, 1);
    this.setState({items: newItems});
  }

  render() {
    const items = this.state.items.map((item, i) => (
      <div key={i} onClick={() => this.handleRemove(i)}>
        {item}
      </div>
    ));
    return (
      <div>
```

```

        <button onClick={this.handleAdd}>Add Item</button>
        <ReactCSSTransitionGroup
          transitionName="example"
          transitionEnterTimeout={500}
          transitionLeaveTimeout={300}>
          {items}
        </ReactCSSTransitionGroup>
      </div>
    );
  }
}

```

Note:

You must provide the `key` attribute for all children of `ReactCSSTransitionGroup`, even when only rendering a single item. This is how React will determine which children have entered, left, or stayed.

In this component, when a new item is added to `ReactCSSTransitionGroup` it will get the `example-enter` CSS class and the `example-enter-active` CSS class added in the next tick. This is a convention based on the `transitionName` prop.

You can use these classes to trigger a CSS animation or transition. For example, try adding this CSS and adding a new list item:

```

.example-enter {
  opacity: 0.01;
}

.example-enter.example-enter-active {
  opacity: 1;
  transition: opacity 500ms ease-in;
}

.example-leave {
  opacity: 1;
}

.example-leave.example-leave-active {
  opacity: 0.01;
  transition: opacity 300ms ease-in;
}

```

You'll notice that animation durations need to be specified in both the CSS and the render method; this tells React when to remove the animation classes from the element and -- if it's leaving -- when to remove the element from the DOM.

Animate Initial Mounting

`ReactCSSTransitionGroup` provides the optional prop `transitionAppear`, to add an extra transition phase at the initial mount of the component. There is generally no transition phase at the initial mount as the default value of `transitionAppear` is `false`. The following is an example which passes the prop `transitionAppear` with the value `true`.

```

render() {
  return (
    <ReactCSSTransitionGroup
      transitionName="example"
      transitionAppear={true}
      transitionAppearTimeout={500}
      transitionEnter={false}
      transitionLeave={false}>
      <h1>Fading at Initial Mount</h1>
    </ReactCSSTransitionGroup>
  );
}

```

During the initial mount `ReactCSSTransitionGroup` will get the `example-appear` CSS class and the `example-appear-active` CSS class added in the next tick.

```

.example-appear {
  opacity: 0.01;
}

.example-appear.example-appear-active {
  opacity: 1;
  transition: opacity .5s ease-in;
}

```

At the initial mount, all children of the `ReactCSSTransitionGroup` will `appear` but not `enter`. However, all children later added to an existing `ReactCSSTransitionGroup` will `enter` but not `appear`.

Note:

The prop `transitionAppear` was added to `ReactCSSTransitionGroup` in version `0.13`. To maintain backwards compatibility, the default value is set to `false`.

However, the default values of `transitionEnter` and `transitionLeave` are `true` so you must specify `transitionEnterTimeout` and `transitionLeaveTimeout` by default. If you don't need either enter or leave animations, pass `transitionEnter={false}` or `transitionLeave={false}`.

Custom Classes

It is also possible to use custom class names for each of the steps in your transitions. Instead of passing a string into `transitionName` you can pass an object containing either the `enter` and `leave` class names, or an object containing the `enter`, `enter-active`, `leave-active`, and `leave` class names. If only the enter and leave classes are provided, the enter-active and leave-active classes will be determined by appending '-active' to the end of the class name. Here are two examples using custom classes:

```

// ...
<ReactCSSTransitionGroup
  transitionName={{ 
    enter: 'enter',
    enterActive: 'enterActive',
    leave: 'leave',
    leaveActive: 'leaveActive',
  }}

```

```

        appear: 'appear',
        appearActive: 'appearActive'
    } }>
{item}
</ReactCSSTransitionGroup>

<ReactCSSTransitionGroup
    transitionName={ {
        enter: 'enter',
        leave: 'leave',
        appear: 'appear'
    } }>
{item2}
</ReactCSSTransitionGroup>
// ...

```

Animation Group Must Be Mounted To Work

In order for it to apply transitions to its children, the `ReactCSSTransitionGroup` must already be mounted in the DOM or the prop `transitionAppear` must be set to `true`.

The example below would **not** work, because the `ReactCSSTransitionGroup` is being mounted along with the new item, instead of the new item being mounted within it. Compare this to the [Getting Started](#) section above to see the difference.

```

render() {
  const items = this.state.items.map((item, i) =>
    <div key={item} onClick={() => this.handleRemove(i)}>
      <ReactCSSTransitionGroup transitionName="example">
        {item}
      </ReactCSSTransitionGroup>
    </div>
  );
  return (
    <div>
      <button onClick={this.handleAdd}>Add Item</button>
      {items}
    </div>
  );
}

```

Animating One or Zero Items

In the example above, we rendered a list of items into `ReactCSSTransitionGroup`. However, the children of `ReactCSSTransitionGroup` can also be one or zero items. This makes it possible to animate a single element entering or leaving. Similarly, you can animate a new element replacing the current element. For example, we can implement a simple image carousel like this:

```

import ReactCSSTransitionGroup from 'react-transition-group';

function ImageCarousel(props) {
  return (
    <div>
      <ReactCSSTransitionGroup
        transitionName="carousel"
        transitionEnterTimeout={300}

```

```

        transitionLeaveTimeout={300}>
          <img src={props.imageSrc} key={props.imageSrc} />
        </ReactCSSTransitionGroup>
      </div>
    );
}

```

Disabling Animations

You can disable animating `enter` or `leave` animations if you want. For example, sometimes you may want an `enter` animation and no `leave` animation, but `ReactCSSTransitionGroup` waits for an animation to complete before removing your DOM node. You can add `transitionEnter={false}` or `transitionLeave={false}` props to `ReactCSSTransitionGroup` to disable these animations.

Note:

When using `ReactCSSTransitionGroup`, there's no way for your components to be notified when a transition has ended or to perform any more complex logic around animation. If you want more fine-grained control, you can use the lower-level `ReactTransitionGroup` API which provides the hooks you need to do custom transitions.

Low-level API: `ReactTransitionGroup`

Importing

```

import ReactTransitionGroup from 'react-addons-transition-group' // ES6
var ReactTransitionGroup = require('react-addons-transition-group') // ES5 with
npm

```

`ReactTransitionGroup` is the basis for animations. When children are declaratively added or removed from it (as in the [example above](#)), special lifecycle methods are called on them.

- `componentWillAppear()`
- `componentDidAppear()`
- `componentWillEnter()`
- `componentDidEnter()`
- `componentWillLeave()`
- `componentDidLeave()`

Rendering a Different Component

`ReactTransitionGroup` renders as a `span` by default. You can change this behavior by providing a `component` prop. For example, here's how you would render a ``:

```

<ReactTransitionGroup component="ul">
  {/* ... */}
</ReactTransitionGroup>

```

Any additional, user-defined, properties will become properties of the rendered component. For example, here's how you would render a `` with CSS class:

```
<ReactTransitionGroup component="ul" className="animated-list">
  {/* ... */}
</ReactTransitionGroup>
```

Every DOM component that React can render is available for use. However, `component` does not need to be a DOM component. It can be any React component you want; even ones you've written yourself! Just write `component={List}` and your component will receive `this.props.children`.

Rendering a Single Child

People often use `ReactTransitionGroup` to animate mounting and unmounting of a single child such as a collapsible panel. Normally `ReactTransitionGroup` wraps all its children in a `span` (or a custom `component` as described above). This is because any React component has to return a single root element, and `ReactTransitionGroup` is no exception to this rule.

However if you only need to render a single child inside `ReactTransitionGroup`, you can completely avoid wrapping it in a `` or any other DOM component. To do this, create a custom component that renders the first child passed to it directly:

```
function FirstChild(props) {
  const childrenArray = React.Children.toArray(props.children);
  return childrenArray[0] || null;
}
```

Now you can specify `FirstChild` as the `component` prop in `<ReactTransitionGroup>` props and avoid any wrappers in the result DOM:

```
<ReactTransitionGroup component={FirstChild}>
  {someCondition ? <MyComponent /> : null}
</ReactTransitionGroup>
```

This only works when you are animating a single child in and out, such as a collapsible panel. This approach wouldn't work when animating multiple children or replacing the single child with another child, such as an image carousel. For an image carousel, while the current image is animating out, another image will animate in, so `<ReactTransitionGroup>` needs to give them a common DOM parent. You can't avoid the wrapper for multiple children, but you can customize the wrapper with the `component` prop as described above.

Reference

`componentWillAppear()` **#componentwillappear**

```
componentWillAppear(callback)
```

This is called at the same time as `componentDidMount()` for components that are initially mounted in a `TransitionGroup`. It will block other animations from occurring until `callback` is called. It is only called on the initial render of a `TransitionGroup`.

`componentDidAppear()` **{#componentdidappear}**

`componentDidAppear()`

This is called after the `callback` function that was passed to `componentWillAppear` is called.

`componentWillEnter()` **{#componentwillenter}**

`componentWillEnter(callback)`

This is called at the same time as `componentDidMount()` for components added to an existing `TransitionGroup`. It will block other animations from occurring until `callback` is called. It will not be called on the initial render of a `TransitionGroup`.

`componentDidEnter()` **{#componentdidenter}**

`componentDidEnter()`

This is called after the `callback` function that was passed to `componentWillEnter()` is called.

`componentWillLeave()` **{#componentwillleave}**

`componentWillLeave(callback)`

This is called when the child has been removed from the `ReactTransitionGroup`. Though the child has been removed, `ReactTransitionGroup` will keep it in the DOM until `callback` is called.

`componentDidLeave()` **{#componentdidleave}**

`componentDidLeave()`

This is called when the `willLeave` `callback` is called (at the same time as `componentWillUnmount()`).

[Go to TOC](#)

Fragmenty z kluczami

Uwaga:

Punkt wejścia `React.addons` jest przestarzały od wersji Reacta 15.5. Mamy teraz natywne wsparcie dla fragmentów, o których możesz przeczytać [tutaj](#).

Importowanie

```
import createFragment from 'react-addons-create-fragment'; // ES6
var createFragment = require('react-addons-create-fragment'); // ES5 z npm
```

Informacje ogólne

W większości przypadków możesz użyć właściwości `key`, aby określić klucze w elementach zwracanych z metody `render`. Problem pojawia się jednak w sytuacji, gdy masz dwa zestawy komponentów potomnych, które musisz zamienić miejscami. Nie ma sposobu, aby umieścić klucz na każdym zestawie bez dodania elementu opakowującego.

Przykładowo, masz następujący komponent:

```
function Swapper(props) {
  let children;
  if (props.swapped) {
    children = [props.rightChildren, props.leftChildren];
  } else {
    children = [props.leftChildren, props.rightChildren];
  }
  return <div>{children}</div>;
}
```

Gdy zmienisz wartość właściwości `swapped` na inną, komponenty potomne odmontują się i ponownie zamontują, ponieważ żadna z grup potomków nie ma ustawionego klucza.

Aby rozwiązać ten problem, możesz użyć dodatku `createFragment`, dzięki któremu możesz przekazać klucze do zestawów komponentów potomnych.

```
Array<ReactNode> createFragment(object children) {#arrayreactnode-createfragmentobject-children}
```

Zamiast tworzyć tablice, piszemy:

```
import createFragment from 'react-addons-create-fragment';

function Swapper(props) {
  let children;
  if (props.swapped) {
    children = createFragment({
```

```
        right: props.rightChildren,
        left: props.leftChildren
    });
} else {
    children = createFragment({
        left: props.leftChildren,
        right: props.rightChildren
    });
}
return <div>{children}</div>;
}
```

Klucze przekazanego obiektu (czyli `left` i `right`) są używane jako klucze dla całej grupy potomków, a kolejność kluczów obiektu jest używana do określenia kolejności renderowanych grup. Dzięki tej zmianie, obydwa zestawy elementów zostaną odpowiednio uporządkowane w DOM bez odmontowywania.

Wartość zwracaną funkcji `createFragment` należy traktować jako obiekt nieprzezroczysty; możesz użyć funkcji `React.Children` do iterowania fragmentu, ale nie zaglądaj bezpośrednio do niego. Zauważ również, że polegamy tu na silniku JavaScript zachowującym kolejność wyliczania obiektów, która nie jest gwarantowana przez specyfikację, ale jest implementowana przez wszystkie główne przeglądarki i maszyny wirtualne dla obiektów z kluczami nienumerycznymi.

Performance Tools

Note:

As of React 16, `react-addons-perf` is not supported. Please use [your browser's profiling tools](#) to get insight into which components re-render.

Importing

```
import Perf from 'react-addons-perf'; // ES6
var Perf = require('react-addons-perf'); // ES5 with npm
```

Overview

React is usually quite fast out of the box. However, in situations where you need to squeeze every ounce of performance out of your app, it provides a `shouldComponentUpdate()` method where you can add optimization hints to React's diff algorithm.

In addition to giving you an overview of your app's overall performance, `Perf` is a profiling tool that tells you exactly where you need to put these methods.

See these articles for an introduction to React performance tooling:

- ["How to Benchmark React Components"](#)
- ["Performance Engineering with React"](#)
- ["A Deep Dive into React Perf Debugging"](#)

Development vs. Production Builds

If you're benchmarking or seeing performance problems in your React apps, make sure you're testing with the [minified production build](#). The development build includes extra warnings that are helpful when building your apps, but it is slower due to the extra bookkeeping it does.

However, the perf tools described on this page only work when using the development build of React. Therefore, the profiler only serves to indicate the *relatively* expensive parts of your app.

Using Perf

The `Perf` object can be used with React in development mode only. You should not include this bundle when building your app for production.

Getting Measurements

- `start()`
- `stop()`

- `getLastMeasurements()`

Printing Results

The following methods use the measurements returned by `Perf.getLastMeasurements()` to pretty-print the result.

- `printInclusive()`
 - `printExclusive()`
 - `printWasted()`
 - `printOperations()`
 - `printDOM()`
-

Reference

`start() {#start}`

`stop() {#stop}`

```
Perf.start()  
// ...  
Perf.stop()
```

Start/stop the measurement. The React operations in-between are recorded for analyses below. Operations that took an insignificant amount of time are ignored.

After stopping, you will need `Perf.getLastMeasurements()` to get the measurements.

`getLastMeasurements() {#getlastmeasurements}`

```
Perf.getLastMeasurements()
```

Get the opaque data structure describing measurements from the last start-stop session. You can save it and pass it to the other print methods in `Perf` to analyze past measurements.

Note

Don't rely on the exact format of the return value because it may change in minor releases. We will update the documentation if the return value format becomes a supported part of the public API.

`printInclusive() {#printinclusive}`

```
Perf.printInclusive(measurements)
```

Prints the overall time taken. When no arguments are passed, `printInclusive` defaults to all the measurements from the last recording. This prints a nicely formatted table in the console, like so:

(index)	Owner > component	Inclusive time (ms)	Instances
0	"<root> > App"	15.31	37
1	"App > Box"	6.47	37
2	"Box > Box2"	1.96	37

printExclusive() {#printexclusive}

`Perf.printExclusive(measurements)`

"Exclusive" times don't include the times taken to mount the components: processing props, calling `componentWillMount` and `componentDidMount`, etc.

(index)	Component c...	Total inclus...	Exclusive m...	Exclusive r...	Mount time ...	Render time...	Instances
0	"App"	15.31	1.76	1.23	0.04	0.03	37

printWasted() {#printwasted}

`Perf.printWasted(measurements)`

The most useful part of the profiler.

"Wasted" time is spent on components that didn't actually render anything, e.g. the render stayed the same, so the DOM wasn't touched.

(index)	Owner > component	Wasted time (ms)	Instances
0	"<root> > App"	15.317999947001226	37
1	"App > Box"	6.479999952716753	37
2	"Box > Box2"	1.9630000460892916	37

printOperations() {#printoperations}

`Perf.printOperations(measurements)`

Prints the underlying DOM manipulations, e.g. "set innerHTML" and "remove".

(index)	data-reactid	type	args
0	"0"	"set innerHTML"	"<div data-reactid=".0\..."

printDOM() {#printdom}

`Perf.printDOM(measurements)`

This method has been renamed to `printOperations()`. Currently `printDOM()` still exists as an alias but it prints a deprecation warning and will eventually be removed.

[Go to TOC](#)

PureRenderMixin

Note:

`PureRenderMixin` is a legacy add-on. Use `React.PureComponent` instead.

Importing

```
import PureRenderMixin from 'react-addons-pure-render-mixin'; // ES6
var PureRenderMixin = require('react-addons-pure-render-mixin'); // ES5 with npm
```

Overview

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
const createReactClass = require('create-react-class');

createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Under the hood, the mixin implements `shouldComponentUpdate`, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

[Go to TOC](#)

Shallow Compare

Note:

`shallowCompare` is a legacy add-on. Use `React.memo` or `React.PureComponent` instead.

Importing

```
import shallowCompare from 'react-addons-shallow-compare'; // ES6
var shallowCompare = require('react-addons-shallow-compare'); // ES5 with npm
```

Overview

Before `React.PureComponent` was introduced, `shallowCompare` was commonly used to achieve the same functionality as `PureRenderMixin` while using ES6 classes with React.

If your React component's render function is "pure" (in other words, it renders the same result given the same props and state), you can use this helper function for a performance boost in some cases.

Example:

```
export class SampleComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

`shallowCompare` performs a shallow equality check on the current `props` and `nextProps` objects as well as the current `state` and `nextState` objects.

It does this by iterating on the keys of the objects being compared and returning true when the values of a key in each object are not strictly equal.

`shallowCompare` returns `true` if the shallow comparison for props or state fails and therefore the component should update.

`shallowCompare` returns `false` if the shallow comparison for props and state both pass and therefore the component does not need to update.

[Go to TOC](#)

Shallow Renderer

Importowanie

```
import ShallowRenderer from 'react-test-renderer/shallow'; // ES6
var ShallowRenderer = require('react-test-renderer/shallow'); // ES5 z
zainstalowanym npm
```

Ogólne informacje

Podczas pisania testów jednostkowych dla kodu reactowego przydatne może okazać się renderowanie płytkie (ang. *shallow rendering*). Pozwala ono na wyrenderowanie komponentu "jeden poziom w głąb" i wykonanie asercji na zwróconym drzewie, bez obaw o efekty uboczne komponentów potomnych, które wcale nie są tworzone i renderowane. Proces ten nie wymaga obecności drzewa DOM.

Założmy, że mamy do czynienia z następującym komponentem:

```
function MyComponent() {
  return (
    <div>
      <span className="heading">Tytuł</span>
      <Subcomponent foo="bar" />
    </div>
  );
}
```

Można dla niego napisać taki oto test:

```
import ShallowRenderer from 'react-test-renderer/shallow';

// w teście:
const renderer = new ShallowRenderer();
renderer.render(<MyComponent />);
const result = renderer.getRenderOutput();

expect(result.type).toBe('div');
expect(result.props.children).toEqual([
  <span className="heading">Tytuł</span>,
  <Subcomponent foo="bar" />
]);
```

Testowanie płytkie posiada obecnie pewne ograniczenia, jak choćby brak wsparcia dla referencji (ang. *refs*) komponentów.

Uwaga:

Zalecamy również zapoznanie się z [interfejsem API do płytkiego renderowania](#) biblioteki Enzyme. Dostarcza ona tę samą funkcjonalność, jednak ma wygodniejszy i bardziej wysokopoziomowy interfejs.

Dokumentacja

`shallowRenderer.render() {#shallowrendererrender}`

`shallowRenderer` jest "miejscem", w którym można wyrenderować testowany komponent i otrzymać zwracaną przez niego strukturę.

`shallowRenderer.render()` w działaniu przypomina `root.render()`, jednak nie wymaga obecności drzewa DOM, a ponadto renderuje tylko jeden poziom struktury. Oznacza to, iż doskonale nadaje się do testowania komponentów w izolacji, bez brania pod uwagę implementacji komponentów potomnych.

`shallowRenderer.getRenderOutput() {#shallowrenderergetrenderoutput}`

Po wywołaniu `shallowRenderer.render()` należy użyć `shallowRenderer.getRenderOutput()`, aby otrzymać płasko wyrenderowaną strukturę.

Można na niej wykonywać dowolne asercje.

Narzędzia do testowania

Importowanie

```
import ReactTestUtils from 'react-dom/test-utils'; // ES6  
var ReactTestUtils = require('react-dom/test-utils'); // ES5 z zainstalowanym npm
```

Ogólne informacje

`ReactTestUtils` pozwala na łatwiejsze testowanie komponentów reactowych przy pomocy dowolnego frameworka. W Facebooku korzystamy do tego celu z biblioteki [Jest](#), która sprawia, że pisanie testów jest mniej kłopotliwe. Do nauki podstaw Jesta polecamy [samouczek dla Reacta](#), znajdujący się na oficjalnej stronie biblioteki.

Uwaga:

Zalecamy korzystanie z biblioteki [React Testing Library](#). Została ona stworzona w celu propagowania idei pisania testów, które używają komponentów podobnie jak potencjalny użytkownik aplikacji.

Dla Reacta w wersji 16 lub niższej polecamy bibliotekę [Enzyme](#), dzięki której w łatwy sposób można pisać asercje, a także zmieniać i przechodzić drzewo zwrócone przez komponenty reactowe.

- `act()`
- `mockComponent()`
- `isElement()`
- `isElementOfType()`
- `isDOMComponent()`
- `isCompositeComponent()`
- `isCompositeComponentWithType()`
- `findAllInRenderedTree()`
- `scryRenderedDOMComponentsWithClass()`
- `findRenderedDOMComponentWithClass()`
- `scryRenderedDOMComponentsWithTag()`
- `findRenderedDOMComponentWithTag()`
- `scryRenderedComponentsWithType()`
- `findRenderedComponentWithType()`
- `renderIntoDocument()`
- `Simulate`

Dokumentacja

act() {#act}

Aby przygotować dany komponent do testowania, należy renderujący i aktualizujący go kod "opakować" w wywołanie funkcji `act()`. Dzięki temu test zostanie uruchomiony w taki sposób, aby jak najwierniej odtworzyć zachowanie Reacta w przeglądarce.

Uwaga

Biblioteka `react-test-renderer` również udostępnia funkcję `act`, która działa w podobny sposób.

Dla przykładu, założymy, że napisaliśmy następujący komponent `Counter` (pol. *licznik*):

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 0};
    this.handleClick = this.handleClick.bind(this);
  }
  componentDidMount() {
    document.title = `Kliknięto ${this.state.count} razy`;
  }
  componentDidUpdate() {
    document.title = `Kliknięto ${this.state.count} razy`;
  }
  handleClick() {
    this.setState(state => ({
      count: state.count + 1,
    }));
  }
  render() {
    return (
      <div>
        <p>Kliknięto {this.state.count} razy</p>
        <button onClick={this.handleClick}>
          Kliknij mnie
        </button>
      </div>
    );
  }
}
```

W taki oto sposób moglibyśmy go przetestować:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
```

```

    document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('potrafi wyrenderować i aktualizować licznik', () => {
  // Testuje pierwsze renderowanie i metodę cyklu życia "componentDidMount"
  act(() => {
    ReactDOM.createRoot(container).render(<Counter />);
  });
  const button = container.querySelector('button');
  const label = container.querySelector('p');
  expect(label.textContent).toBe('Kliknięto 0 razy');
  expect(document.title).toBe('Kliknięto 0 razy');

  // Testuje drugie renderowanie i metodę cyklu życia "componentDidUpdate"
  act(() => {
    button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
  });
  expect(label.textContent).toBe('Kliknięto 1 razy');
  expect(document.title).toBe('Kliknięto 1 razy');
});

```

Zwróć uwagę, że przesyłanie zdarzeń DOM działa tylko wtedy, gdy kontener jest umieszczony w `document`. Aby uniknąć powtarzania szablonowego kodu, możesz użyć biblioteki pomocniczej, jak na przykład [React Testing Library](#).

W `przykładach` znajdziesz więcej szczegółów na temat zachowania funkcji `act()`, jak również przykłady jej użycia.

`mockComponent()` [{#mockcomponent}](#)

```

mockComponent(
  componentClass,
  [mockTagName]
)

```

Jeśli przekażesz do tej funkcji atrapę komponentu (ang. *mocked component*), zostanie ona wzbogacona o przydatne funkcje, które pozwolą na traktowanie jej jak sztucznego komponentu reactowego. Zamiast wyrenderować się zgodnie z implementacją, komponent stanie się zwykłym elementem `<div>` (lub innym, jeśli podamy wartość w parametrze `mockTagName`) renderującym przekazanych potomków.

Uwaga:

Funkcja `mockComponent()` jest przestarzała. Zamiast niej zalecamy używanie funkcji `jest.mock()`.

isElement() **{#iselement}**

```
isElement(element)
```

Zwraca `true`, jeśli argument `element` jest elementem reactowym.

isElementOfType() **{#iselementoftype}**

```
isElementOfType(  
  element,  
  componentClass  
)
```

Zwraca `true`, jeśli argument `element` jest elementem reactowym o klasie podanej jako `componentClass`.

isDOMComponent() **{#isdomcomponent}**

```
isDOMComponent(instance)
```

Zwraca `true`, jeśli argument `instance` jest standardowym komponentem DOM (np. `<div>` lub ``).

isCompositeComponent() **{#iscompositecomponent}**

```
isCompositeComponent(instance)
```

Zwraca `true`, jeśli argument `instance` jest komponentem użytkownika, typu klasowego lub funkcyjnego.

isCompositeComponentWithType() **{#iscompositecomponentwithtype}**

```
isCompositeComponentWithType(  
  instance,  
  componentClass  
)
```

Zwraca `true`, jeśli argument `instance` jest komponentem o klasie podanej jako `componentClass`.

findAllInRenderedTree() **{#findallinrenderedtree}**

```
findAllInRenderedTree(  
  tree,  
  test  
)
```

Przeszukuje wszystkie komponenty w drzewie `tree` i zwraca te, dla których wywołanie funkcji `test(komponent)` daje `true`. Funkcja ta sama w sobie nie jest zbyt użyteczna, jednak jest podstawą dla innych narzędzi do testowania.

scryRenderedDOMComponentsWithClass() {#scryrendereddomcomponentswithclass}

```
scryRenderedDOMComponentsWithClass(  
  tree,  
  className  
)
```

Wyszukuje wszystkie elementy DOM w wyrenderowanym drzewie, których nazwa klasy CSS odpowiada wartości argumentu `className`.

findRenderedDOMComponentWithClass() {#findrendereddomcomponentwithclass}

```
findRenderedDOMComponentWithClass(  
  tree,  
  className  
)
```

Podobna w działaniu do `scryRenderedDOMComponentsWithClass()`, lecz oczekuje dokładnie jednego wyniku. W przypadku znalezienia innej liczby elementów rzuca wyjątek.

scryRenderedDOMComponentsWithTag() {#scryrendereddomcomponentswithtag}

```
scryRenderedDOMComponentsWithTag(  
  tree,  
  tagName  
)
```

Wyszukuje wszystkie elementy DOM w wyrenderowanym drzewie, których nazwa znacznika pasuje do wartości argumentu `tagName`.

findRenderedDOMComponentWithTag() {#findrendereddomcomponentwithtag}

```
findRenderedDOMComponentWithTag(  
  tree,  
  tagName  
)
```

Podobna w działaniu do `scryRenderedDOMComponentsWithTag()`, lecz oczekuje dokładnie jednego wyniku. W przypadku znalezienia innej liczby elementów rzuca wyjątek.

scryRenderedComponentsWithType() {#scryrenderedcomponentswithtype}

```
scryRenderedComponentsWithType(
  tree,
  componentClass
)
```

Wyszukuje wszystkie instancje komponentów, których typ jest równy argumentowi `componentClass`.

findRenderedComponentWithType() {#findrenderedcomponentwithtype}

```
findRenderedComponentWithType(
  tree,
  componentClass
)
```

Podobna w działaniu do `scryRenderedComponentsWithType()`, lecz oczekuje dokładnie jednego wyniku. W przypadku znalezienia innej liczby elementów rzuca wyjątek.

renderIntoDocument() {#renderintodocument}

```
renderIntoDocument(element)
```

Renderuje element reactowy do stworzonego w locie węzła drzewa DOM. **Ta funkcja działa tylko na drzewie DOM w ramach dokumentu.** Daje ten sam rezultat, co:

```
const domContainer = document.createElement('div');
ReactDOM.createRoot(domContainer).render(element);
```

Uwaga:

Zanim zimportujesz bibliotekę React w kodzie, w globalnym zakresie muszą być dostępne zmienne `window`, `window.document` oraz `window.document.createElement`. W przeciwnym wypadku React będzie "myślał", że nie ma dostępu do drzewa DOM, co spowoduje wyłączenie niektórych funkcji, np. `setState`.

Inne narzędzia

Simulate {#simulate}

```
Simulate.{eventName}(
  element,
  [eventData]
)
```

Symuluje przesłanie zdarzenia do węzła DOM, opcjonalnie dodając do niego dane zawarte w argumencie `eventData`.

Obiekt `Simulate` posiada odpowiednie metody dla każdego ze zdarzeń obsługiwanych przez Reacta.

Kliknięcie w element

```
// <button ref={(node) => this.button = node}>...</button>
const node = this.button;
ReactTestUtils.Simulate.click(node);
```

Zmiana wartości pola i wcisnięcie klawisza ENTER.

```
// <input ref={(node) => this.textInput = node} />
const node = this.textInput;
node.value = 'żyrafa';
ReactTestUtils.Simulate.change(node);
ReactTestUtils.Simulate.keyDown(node, {key: "Enter", keyCode: 13, which: 13});
```

Uwaga

Wszelkie właściwości dla zdarzenia (np. `keyCode`, `which` itp.) należy przekazać jawnie, ponieważ React nie dodaje ich automatycznie.

Two-way Binding Helpers

Note:

`LinkedStateMixin` is deprecated as of React v15. The recommendation is to explicitly set the value and change handler, instead of using `LinkedStateMixin`.

Importing

```
import LinkedStateMixin from 'react-addons-linked-state-mixin'; // ES6
var LinkedStateMixin = require('react-addons-linked-state-mixin'); // ES5 with npm
```

Overview

`LinkedStateMixin` is an easy way to express two-way binding with React.

In React, data flows one way: from owner to child. We think that this makes your app's code easier to understand. You can think of it as "one-way data binding."

However, there are lots of applications that require you to read some data and flow it back into your program. For example, when developing forms, you'll often want to update some React `state` when you receive user input. Or perhaps you want to perform layout in JavaScript and react to changes in some DOM element size.

In React, you would implement this by listening to a "change" event, read from your data source (usually the DOM) and call `setState()` on one of your components. "Closing the data flow loop" explicitly leads to more understandable and easier-to-maintain programs. See [our forms documentation](#) for more information.

Two-way binding -- implicitly enforcing that some value in the DOM is always consistent with some React `state` -- is concise and supports a wide variety of applications. We've provided `LinkedStateMixin`: syntactic sugar for setting up the common data flow loop pattern described above, or "linking" some data source to React `state`.

Note:

`LinkedStateMixin` is just a thin wrapper and convention around the `onChange / setState()` pattern. It doesn't fundamentally change how data flows in your React application.

LinkedStateMixin: Before and After

Here's a simple form example without using `LinkedStateMixin`:

```
var createReactClass = require('create-react-class');

var NoLink = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  handleChange: function(event) {
    this.setState({message: event.target.value});
  },
  render: function() {
    var message = this.state.message;
    return <input type="text" value={message} onChange={this.handleChange}>;
  }
});
```

This works really well and it's very clear how data is flowing, however, with a lot of form fields it could get a bit verbose. Let's use `LinkedStateMixin` to save us some typing:

```
var createReactClass = require('create-react-class');

var WithLink = createReactClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    return <input type="text" valueLink={this.linkState('message')} />;
  }
});
```

`LinkedStateMixin` adds a method to your React component called `linkState()`. `linkState()` returns a `valueLink` object which contains the current value of the React state and a callback to change it.

`valueLink` objects can be passed up and down the tree as props, so it's easy (and explicit) to set up two-way binding between a component deep in the hierarchy and state that lives higher in the hierarchy.

Note that checkboxes have a special behavior regarding their `value` attribute, which is the value that will be sent on form submit if the checkbox is checked (defaults to `on`). The `value` attribute is not updated when the checkbox is checked or unchecked. For checkboxes, you should use `checkedLink` instead of `valueLink`:

```
<input type="checkbox" checkedLink={this.linkState('booleanValue')} />
```

Under the Hood

There are two sides to `LinkedStateMixin`: the place where you create the `valueLink` instance and the place where you use it. To prove how simple `LinkedStateMixin` is, let's rewrite each side separately to be more explicit.

valueLink Without LinkedStateMixin

```
var createReactClass = require('create-react-class');

var WithoutMixin = createReactClass({
```

```

getInitialState: function() {
  return {message: 'Hello!'};
},
handleChange: function(newValue) {
  this.setState({message: newValue});
},
render: function() {
  var valueLink = {
    value: this.state.message,
    requestChange: this.handleChange
  };
  return <input type="text" valueLink={valueLink} />;
}
);

```

As you can see, `valueLink` objects are very simple objects that just have a `value` and `requestChange` prop. And `LinkedStateMixin` is similarly simple: it just populates those fields with a value from `this.state` and a callback that calls `this.setState()`.

LinkedStateMixin Without valueLink

```

var LinkedStateMixin = require('react-addons-linked-state-mixin');
var createReactClass = require('create-react-class');

var WithoutLink = createReactClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    var valueLink = this.linkState('message');
    var handleChange = function(e) {
      valueLink.requestChange(e.target.value);
    };
    return <input type="text" value={valueLink.value} onChange={handleChange} />;
  }
});

```

The `valueLink` prop is also quite simple. It simply handles the `onChange` event and calls `this.props.valueLink.requestChange()` and also uses `this.props.valueLink.value` instead of `this.props.value`. That's it!

Immutability Helpers

Note:

`update` is a legacy add-on. Use `immutability-helper` instead.

Importing

```
import update from 'react-addons-update'; // ES6
var update = require('react-addons-update'); // ES5 with npm
```

Overview

React lets you use whatever style of data management you want, including mutation. However, if you can use immutable data in performance-critical parts of your application it's easy to implement a fast `shouldComponentUpdate()` method to significantly speed up your app.

Dealing with immutable data in JavaScript is more difficult than in languages designed for it, like [Clojure](#). However, we've provided a simple immutability helper, `update()`, that makes dealing with this type of data much easier, *without* fundamentally changing how your data is represented. You can also take a look at Facebook's [Immutable-js](#) and the [Advanced Performance](#) section for more detail on Immutable-js.

The Main Idea

If you mutate data like this:

```
myData.x.y.z = 7;
// or...
myData.a.b.push(9);
```

You have no way of determining which data has changed since the previous copy has been overwritten. Instead, you need to create a new copy of `myData` and change only the parts of it that need to be changed. Then you can compare the old copy of `myData` with the new one in `shouldComponentUpdate()` using triple-equals:

```
const newData = deepCopy(myData);
newData.x.y.z = 7;
newData.a.b.push(9);
```

Unfortunately, deep copies are expensive, and sometimes impossible. You can alleviate this by only copying objects that need to be changed and by reusing the objects that haven't changed. Unfortunately, in today's JavaScript this can be cumbersome:

```
const newData = extend(myData, {
  x: extend(myData.x, {
    y: extend(myData.x.y, {z: 7}),
```

```
  },
  a: extend(myData.a, {b: myData.a.b.concat(9)})
});
```

While this is fairly performant (since it only makes a shallow copy of `log n` objects and reuses the rest), it's a big pain to write. Look at all the repetition! This is not only annoying, but also provides a large surface area for bugs.

update() {#update}

`update()` provides simple syntactic sugar around this pattern to make writing this code easier. This code becomes:

```
import update from 'react-addons-update';

const newData = update(myData, {
  x: {y: {z: {$set: 7}}},
  a: {b: {$push: [9]}}
});
```

While the syntax takes a little getting used to (though it's inspired by MongoDB's query language) there's no redundancy, it's statically analyzable and it's not much more typing than the mutative version.

The `$`-prefixed keys are called *commands*. The data structure they are "mutating" is called the *target*.

Available Commands

- `{$push: array} push()` all the items in `array` on the target.
- `{$unshift: array} unshift()` all the items in `array` on the target.
- `{$splice: array of arrays}` for each item in `arrays` call `splice()` on the target with the parameters provided by the item.
- `{$set: any}` replace the target entirely.
- `{$merge: object}` merge the keys of `object` with the target.
- `{$apply: function}` passes in the current value to the function and updates it with the new returned value.

Examples

Simple push

```
const initialArray = [1, 2, 3];
const newArray = update(initialArray, {$push: [4]}); // => [1, 2, 3, 4]
```

`initialArray` is still `[1, 2, 3]`.

Nested collections

```
const collection = [1, 2, {a: [12, 17, 15]}];
const newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13, 14]]}}});
// => [1, 2, {a: [12, 13, 14, 15]}]
```

This accesses `collection`'s index `2`, key `a`, and does a splice of one item starting from index `1` (to remove `17`) while inserting `13` and `14`.

Updating a value based on its current one

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {b: {$apply: function(x) {return x * 2;}}});
// => {a: 5, b: 6}
// This is equivalent, but gets verbose for deeply nested collections:
const newObj2 = update(obj, {b: {$set: obj.b * 2}});
```

(Shallow) Merge

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {$merge: {b: 6, c: 7}}); // => {a: 5, b: 6, c: 7}
```

Add-Ons

Note:

`React.addons` entry point is deprecated as of React v15.5. The add-ons have moved to separate modules, and some of them have been deprecated.

The React add-ons are a collection of useful utility modules for building React apps. **These should be considered experimental** and tend to change more often than the core.

- `createFragment`, to create a set of externally-keyed children.

The add-ons below are in the development (unminified) version of React only:

- `Perf`, a performance profiling tool for finding optimization opportunities.
- `ReactTestUtils`, simple helpers for writing test cases.

Legacy Add-ons

The add-ons below are considered legacy and their use is discouraged. They will keep working in observable future, but there is no further development.

- `PureRenderMixin`. Use `React.PureComponent` instead.
- `shallowCompare`, a helper function that performs a shallow comparison for props and state in a component to decide if a component should update. We recommend using `React.PureComponent` instead.
- `update`. Use `kolodny/immutability-helper` instead.
- `ReactDOMFactories`, pre-configured DOM factories to make React easier to use without JSX.

Deprecated Add-ons

- `LinkedStateMixin` has been deprecated.
- `TransitionGroup` and `CSSTransitionGroup` have been deprecated in favor of [their drop-in replacements](#).

Using React with Add-ons

You can install the add-ons individually from npm (e.g. `npm install react-addons-create-fragment`) and import them:

```
import createFragment from 'react-addons-create-fragment'; // ES6
var createFragment = require('react-addons-create-fragment'); // ES5 with npm
```

When using React 15 or earlier from a CDN, you can use `react-with-addons.js` instead of `react.js`:

```
<script src="https://unpkg.com/react@15/dist/react-with-addons.js"></script>
```

The add-ons will be available via the `React.addons` global (e.g. `React.addons.TestUtils`).

Linki do CDN-ów

Zarówno React, jak i ReactDOM są dostępne przez CDN.

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.development.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
```

Powyższe wersje przeznaczone są do uruchamiania na serwerach deweloperskich, nie nadają się natomiast na środowiska produkcyjne. Zminifikowane i zoptymalizowane wersje produkcyjne Reacta są dostępne pod adresem:

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.production.min.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.production.min.js"></script>
```

Aby załadować określoną wersję paczki `react` i `react-dom`, zastąp `18` odpowiednim numerem wersji.

Dlaczego atrybut `crossorigin`?

Jeśli korzystasz z Reacta z CDN, sugerujemy dodać atrybut `crossorigin` ustawiony na:

```
<script crossorigin src="... "></script>
```

Zalecamy również sprawdzenie, czy CDN, z którego korzystasz, zwraca nagłówek HTTP `Access-Control-Allow-Origin: *`:

	Headers	Preview	Response	Timing
x	Headers			
▼	General			
	Request URL: https://unpkg.com/react@18/umd/react.development.js			
	Request Method: GET			
	Status Code: 200			
	Remote Address: 12400:cb00:2048:1::6810			
	Referrer Policy: no-referrer-when-downgrade			
▼	Response Headers			
	access-control-allow-origin: *			
	cache-control: public, max-age=31536000			

`Access-Control-Allow-Origin: *`

Umożliwia to lepszą [obsługę błędów](#) w Reakcie 16 i późniejszych wersjach.

[Go to TOC](#)

Dzielenie kodu

Pakowanie

Większość reactowych aplikacji będzie "dołączała" swoje pliki poprzez narzędzia takie jak [Webpack](#), [Rollup](#) czy [Browserify](#). Pakowanie to proces śledzenia zimportowanych plików i łączenia ich w pojedynczy plik tzw. "bundle" (pol. *paczka*). Taka paczka może zostać umieszczona na stronie w celu załadowania całej aplikacji naraz.

Przykład

Kod aplikacji:

```
// app.js
import { add } from './math.js';

console.log(add(16, 26)); // 42
```

```
// math.js
export function add(a, b) {
  return a + b;
}
```

Paczka:

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

console.log(add(16, 26)); // 42
```

Uwaga:

Twoje paczki prawdopodobnie będą się znaczowo różnić.

Jeśli używasz [Create React App](#), [Next.js](#), [Gatsby](#) lub innego podobnego narzędzia, Webpack powinien być już skonfigurowany i gotowy do użytku.

Jeśli nie, musisz skonfigurować Webpacka samodzielnie. Przykłady znajdziesz w [przewodniku po instalacji](#) oraz w sekcji pt. "[Podstawowe informacje](#)" zawartych w dokumentacji Webpacka.

Dzielenie kodu

Tworzenie paczek jest świetne, ale wraz ze wzrostem objętości kodu twojej aplikacji, rośnie również objętość paczek. Zwłaszcza gdy dołączasz do projektu duże, zewnętrzne biblioteki. Musisz pilnować, aby twój pakiet nie był zbyt duży, ponieważ wtedy aplikacja będzie się dłużej ładowała.

Aby uniknąć problemu zbyt dużego pakietu, warto już na początku o tym pomyśleć i rozpocząć "dzielenie" swojej paczki. [Dzielenie kodu](#) to funkcja wspierana przez narzędzia takie jak [Webpack](#), [Rollup](#) oraz [Browsersify](#) (przez [factor-bundle](#)), które mogą tworzyć wiele pakietów doładowywanych dynamicznie w czasie wykonania kodu aplikacji.

Dzielenie kodu twojej aplikacji ułatwi ci użycie "leniwego ładowania" do wczytywania jedynie tych zasobów które są aktualnie wymagane przez użytkownika zasobów, co może znacznie poprawić wydajność twojej aplikacji. Mimo że nie zmniejszysz w ten sposób sumarycznej ilości kodu, unikniesz ładowania funkcjonalności zbędnych dla użytkownika w danym momencie, co przełoży się na mniejszą ilość kodu do pobrania na starcie aplikacji.

`import() {#import}`

Najlepszym sposobem na wprowadzenie podziału kodu do twojej aplikacji jest użycie dynamicznego wariantu funkcji `import()`.

Przed:

```
import { add } from './math';
console.log(add(16, 26));
```

Po:

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

Gdy Webpack natknie się na taką składnię, automatycznie zacznie dzielić kod w twojej aplikacji. Jeśli używasz Create React App, posiadasz już gotową konfigurację i możesz natychmiast [zacząć z niej korzystać](#). Jest ona również obsługiwana domyślnie przez [Next.js](#).

Jeśli konfigurujesz Webpacka samodzielnie, zalecamy przeczytać [przewodnik po dzieleniu kodu](#). Twoja konfiguracja Webpacka powinna wyglądać podobnie [do tej](#).

Kiedy używasz [Babeli](#), musisz się upewnić, że jest on w stanie sparsować składnię dynamicznego importu, ale jej nie przekształca w żaden sposób. W tym celu skorzystaj z pluginu [@babel/plugin-syntax-dynamic-import](#).

`React.lazy {#reactlazy}`

Funkcja `React.lazy` pozwala renderować dynamicznie importowane komponenty jak zwykłe komponenty.

Przed:

```
import OtherComponent from './OtherComponent';
```

Po:

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

Powyższy kod automatycznie załadowuje paczkę zawierającą `OtherComponent` podczas pierwszego renderowania komponentu.

`React.lazy` jako argument przyjmuje funkcję, która wywołuje dynamiczny `import()`. Musi ona zwrócić obiekt (`Promise`) (pol. *obietnicę*), która rozwiązuje się do modułu z eksportem domyślnym (`default`) zawierającym komponent reactowy.

"Leniwy" komponent powinien zostać wyrenderowany wewnętrz `Suspense`, dzięki któremu na czas ładowania możemy wyświetlić komponent zastępczy (np. wskaźnik ładowania).

```
import React, { Suspense } from 'react';
const OtherComponent = React.lazy(() => import('./OtherComponent'));

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

Właściwość `fallback` tego komponentu akceptuje dowolny element reactowy, który będzie wyświetlany w trakcie oczekiwania na załadowanie komponentu. Możesz umieścić komponent `Suspense` w dowolnym miejscu nad "leniwym" (ang. *lazy*) komponentem. Możesz nawet opakować kilka "leniwych" komponentów w jeden komponent `Suspense`.

```
import React, { Suspense } from 'react';
const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

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

Unikanie zastępników

Każdy komponent może zawiesić działanie na skutek renderowania, nawet taki, który już został wcześniej wyświetlony użytkownikowi. Aby zapewnić spójność treści na ekranie, w takim przypadku React musi ukryć jego poddrzewo elementów aż do najbliższej granicy `<Suspense>`. Może to jednak zdezorientować użytkownika.

Założymy, że mamy poniższy komponent zarządzający zmienianiem zakładek:

```
import React, { Suspense } from 'react';
import Tabs from './Tabs';
import Glimmer from './Glimmer';

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

function MyComponent() {
  const [tab, setTab] = React.useState('photos');

  function handleTabSelect(tab) {
    setTab(tab);
  };

  return (
    <div>
      <Tabs onTabSelect={handleTabSelect} />
      <Suspense fallback={<Glimmer />}>
        {tab === 'photos' ? <Photos /> : <Comments />}
      </Suspense>
    </div>
  );
}
```

Jeśli w powyższym przykładzie zakładka zmieni się z `'photos'` na `'comments'`, ale komponent `Comments` zawiesi działanie, na ekranie na chwilę wyświetli się `Glimmer`. Ma to sens, ponieważ użytkownik nie chce już widzieć komponentu `Photos`, zaś komponent `Comments` nie jest jeszcze gotowy na wyrenderowanie treści, a React musi zapewnić spójność zawartości, więc nie ma innego wyjścia, jak tylko wyświetlić komponent `Glimmer`.

Czasem jednak taki ciąg zdarzeń nie jest satysfakcjonujący. Czasem lepiej wyświetlić "stary" widok, podczas gdy nowy jest w trakcie przygotowywania. Można to osiągnąć za pomocą nowej funkcji `startTransition`:

```
function handleTabSelect(tab) {
  startTransition(() => {
    setTab(tab);
  });
}
```

Tym poleciением mówisz Reactowi, że zmiana zakładki na `'comments'` nie jest bardzo pilna, jest jedynie [stanem przejściowym](#), który może chwilę potrwać. React w tym wypadku będzie nadal wyświetlać stary interfejs, który na dodatek będzie w pełni interaktywny. Gdy jednak `<Comments />` skończy się renderować, React podmieni zawartość na ekranie. Aby dowiedzieć się więcej, przejdź do sekcji [stanu przejściowe](#).

Granice błędów

Jeśli inny moduł nie wczyta się poprawnie (na przykład z powodu awarii sieci), spowoduje to błąd. Możesz go obsłużyć aby zapewnić użytkownikowi lepsze doświadczenie, a także aby określić sposób obsługi błędu za pomocą [granicy błędu](#). Taką granicę błędu możesz umieścić w dowolnym miejscu ponad "leniwymi" komponentami i, na przykład, aby wyświetlić stan błędu, gdy połączenie z siecią zostanie zerwane.

```
import React, { Suspense } from 'react';
import MyErrorBoundary from './MyErrorBoundary';

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

const MyComponent = () => (
  <div>
    <MyErrorBoundary>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </MyErrorBoundary>
  </div>
);

;
```

Podział kodu na podstawie ścieżki URL

Decyzja o tym, w których miejscach podzielić kod aplikacji, może okazać się kłopotliwa. Zależy ci na miejscach, że wybierasz miejsca, które równomiernie podzielą twoje pakiety, ale nie chcesz zepsuć doświadczeń użytkownika.

Dobrym punktem startowym są ścieżki (ang. *routes*) w aplikacji. Większość ludzi korzystających z Internetu przyzwyczajona jest, że przejście pomiędzy stronami zajmuje trochę czasu. Dodatkowo, zazwyczaj podczas takiego przejścia spora część ekranu jest renderowana ponownie. Można więc założyć, że użytkownik nie będzie wykonywał żadnych akcji na interfejsie podczas ładowania.

Oto przykład skonfigurowania dzielenia kodu aplikacji opartego na ścieżkach, przy użyciu biblioteki [React Router](#) wraz z funkcją `React.lazy`.

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Wczytywanie...</div>}>
      <Routes>
        <Route path="/" element={<Home />}/>
        <Route path="/about" element={<About />}/>
      </Routes>
    </Suspense>
  </Router>
);

;
```

```
</Suspense>
</Router>
);
```

Eksporty nazwane

`React.lazy` obecnie obsługuje tylko domyślne eksporty. Jeśli moduł, który chcesz zaimportować, używa nazwanych eksportów, możesz utworzyć moduł pośredni, który ponownie eksportuje je jako domyślny eksport. Gwarantuje to działanie mechanizmu "tree-shaking" (pol. *potrząsanie drzewem*), a także zapobiega pobieraniu nieużywanych komponentów.

```
// ManyComponents.js
export const MyComponent = /* ... */;
export const MyUnusedComponent = /* ... */;
```

```
// MyComponent.js
export { MyComponent as default } from "./ManyComponents.js";
```

```
// MyApp.js
import React, { lazy } from 'react';
const MyComponent = lazy(() => import("./MyComponent.js"));
```

Codebase Overview

This section will give you an overview of the React codebase organization, its conventions, and the implementation.

If you want to [contribute to React](#) we hope that this guide will help you feel more comfortable making changes.

We don't necessarily recommend any of these conventions in React apps. Many of them exist for historical reasons and might change with time.

Top-Level Folders

After cloning the [React repository](#), you will see a few top-level folders in it:

- `packages` contains metadata (such as `package.json`) and the source code (`src` subdirectory) for all packages in the React repository. **If your change is related to the code, the `src` subdirectory of each package is where you'll spend most of your time.**
- `fixtures` contains a few small React test applications for contributors.
- `build` is the build output of React. It is not in the repository but it will appear in your React clone after you [build it](#) for the first time.

The documentation is hosted [in a separate repository from React](#).

There are a few other top-level folders but they are mostly used for the tooling and you likely won't ever encounter them when contributing.

Colocated Tests

We don't have a top-level directory for unit tests. Instead, we put them into a directory called `__tests__` relative to the files that they test.

For example, a test for `setInnerHTML.js` is located in `__tests__/setInnerHTML-test.js` right next to it.

Warnings and Invariants

The React codebase uses `console.error` to display warnings:

```
if (__DEV__) {
  console.error('Something is wrong.');
}
```

Warnings are only enabled in development. In production, they are completely stripped out. If you need to forbid some code path from executing, use `invariant` module instead:

```
var invariant = require('invariant');

invariant(
```

```
2 + 2 === 4,
'You shall not pass!'
);
```

The invariant is thrown when the `invariant` condition is `false`.

"Invariant" is just a way of saying "this condition always holds true". You can think about it as making an assertion.

It is important to keep development and production behavior similar, so `invariant` throws both in development and in production. The error messages are automatically replaced with error codes in production to avoid negatively affecting the byte size.

Development and Production

You can use `__DEV__` pseudo-global variable in the codebase to guard development-only blocks of code.

It is inlined during the compile step, and turns into `process.env.NODE_ENV !== 'production'` checks in the CommonJS builds.

For standalone builds, it becomes `true` in the unminified build, and gets completely stripped out with the `if` blocks it guards in the minified build.

```
if (__DEV__) {
  // This code will only run in development.
}
```

Flow

We recently started introducing [Flow](#) checks to the codebase. Files marked with the `@flow` annotation in the license header comment are being typechecked.

We accept pull requests [adding Flow annotations to existing code](#). Flow annotations look like this:

```
ReactRef.detachRefs = function(
  instance: ReactInstance,
  element: ReactElement | string | number | null | false,
): void {
  // ...
}
```

When possible, new code should use Flow annotations. You can run `yarn flow` locally to check your code with Flow.

Multiple Packages

React is a [monorepo](#). Its repository contains multiple separate packages so that their changes can be co-ordinated together, and issues live in one place.

React Core

The "core" of React includes all the top-level `React APIs`, for example:

- `React.createElement()`
- `React.Component`
- `React.Children`

React core only includes the APIs necessary to define components. It does not include the [reconciliation](#) algorithm or any platform-specific code. It is used both by React DOM and React Native components.

The code for React core is located in `packages/react` in the source tree. It is available on npm as the `react` package. The corresponding standalone browser build is called `react.js`, and it exports a global called `React`.

Renderers

React was originally created for the DOM but it was later adapted to also support native platforms with [React Native](#). This introduced the concept of "renderers" to React internals.

Renderers manage how a React tree turns into the underlying platform calls.

Renderers are also located in `packages/`:

- [React DOM Renderer](#) renders React components to the DOM. It implements [top-level ReactDOM APIs](#) and is available as `react-dom` npm package. It can also be used as standalone browser bundle called `react-dom.js` that exports a `ReactDOM` global.
- [React Native Renderer](#) renders React components to native views. It is used internally by React Native.
- [React Test Renderer](#) renders React components to JSON trees. It is used by the [Snapshot Testing](#) feature of [Jest](#) and is available as `react-test-renderer` npm package.

The only other officially supported renderer is `react-art`. It used to be in a separate [GitHub repository](#) but we moved it into the main source tree for now.

Note:

Technically the `react-native-renderer` is a very thin layer that teaches React to interact with React Native implementation. The real platform-specific code managing the native views lives in the [React Native repository](#) together with its components.

Reconcilers

Even vastly different renderers like React DOM and React Native need to share a lot of logic. In particular, the [reconciliation](#) algorithm should be as similar as possible so that declarative rendering, custom components, state, lifecycle methods, and refs work consistently across platforms.

To solve this, different renderers share some code between them. We call this part of React a "reconciler". When an update such as `setState()` is scheduled, the reconciler calls `render()` on components in the tree and mounts, updates, or unmounts them.

Reconcilers are not packaged separately because they currently have no public API. Instead, they are exclusively used by renderers such as React DOM and React Native.

Stack Reconciler

The "stack" reconciler is the implementation powering React 15 and earlier. We have since stopped using it, but it is documented in detail in the [next section](#).

Fiber Reconciler

The "fiber" reconciler is a new effort aiming to resolve the problems inherent in the stack reconciler and fix a few long-standing issues. It has been the default reconciler since React 16.

Its main goals are:

- Ability to split interruptible work in chunks.
- Ability to prioritize, rebase and reuse work in progress.
- Ability to yield back and forth between parents and children to support layout in React.
- Ability to return multiple elements from `render()`.
- Better support for error boundaries.

You can read more about React Fiber Architecture [here](#) and [here](#). While it has shipped with React 16, the async features are not enabled by default yet.

Its source code is located in `packages/react-reconciler`.

Event System

React implements a layer over native events to smooth out cross-browser differences. Its source code is located in `packages/react-dom/src/events`.

What Next?

Read the [next section](#) to learn about the pre-React 16 implementation of reconciler in more detail. We haven't documented the internals of the new reconciler yet.

Komponenty i właściwości

Komponenty pozwalają podzielić interfejs użytkownika na niezależne, pozwalające na ponowne użycie części i myśleć o każdej z nich osobno. Ta strona zawiera wprowadzenie do idei komponentów. W osobnym rozdziale opisaliśmy [szczegółową dokumentację API komponentów](#).

Koncepcyjnie, komponenty są jak javascriptowe funkcje. Przyjmują one arbitralne wartości na wejściu (nazywane "właściwościami" (ang. *props*)) i zwracają reactowe elementy opisujące, co powinno się pojawić na ekranie.

Komponenty funkcyjne i klasowe

Najprostszym sposobem na zdefiniowanie komponentu jest napisanie javascriptowej funkcji:

```
function Welcome(props) {
  return <h1>Cześć, {props.name}</h1>;
}
```

Ta funkcja jest poprawnym reactowym komponentem, ponieważ przyjmuje pojedynczy argument "props" (który oznacza "właściwości", z ang. *properties*), będący obiektem z danymi, i zwraca reactowy element. Takie komponenty nazywamy "komponentami funkcyjnymi", gdyż są one dosłownie javascriptowymi funkcjami.

Do zdefiniowania komponentu możesz również użyć [klasy ze standardu ES6](#):

```
class Welcome extends React.Component {
  render() {
    return <h1>Cześć, {this.props.name}</h1>;
  }
}
```

Obydwa powyższe komponenty są równoważne z punktu widzenia Reacta.

Komponenty funkcyjne i klasowe mają kilka dodatkowych cech, które omówimy w [kolejnych rozdziałach](#).

Renderowanie komponentu

Poprzednio napotykaliśmy reactowe elementy, które reprezentowały znaczniki DOM:

```
const element = <div />;
```

Elementy mogą również reprezentować komponenty zdefiniowane przez użytkownika:

```
const element = <Welcome name="Sara" />;
```

Kiedy React widzi element reprezentujący komponent zdefiniowany przez użytkownika, przekazuje do niego JSX-owe atrybuty i potomków jako jeden obiekt. Obiekt ten nazywamy "właściwościami" komponentu.

Dla przykładu, poniższy kod renderuje na stronie napis "Cześć, Sara":

```
function Welcome(props) {
  return <h1>Cześć, {props.name}</h1>;
}

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

[Przetestuj kod na CodePen](#)

Podsumujmy, co dzieje się w tym przykładzie:

1. Wywołujemy `root.render()` z elementem `<Welcome name="Sara" />`.
2. React wywołuje komponent `Welcome` z właściwościami `{name: 'Sara'}`.
3. Nasz komponent `Welcome` jako wynik zwraca element `<h1>Cześć, Sara</h1>`.
4. React DOM w optymalny sposób aktualizuje drzewo DOM, aby odpowiadało elementowi `<h1>Cześć, Sara</h1>`.

Wskazówka: Zawsze zaczynaj nazwy komponentów od dużej litery.

React traktuje komponenty zaczynające się od małej litery jako tagi drzewa DOM. Na przykład, `<div />` reprezentuje HTML-owy znacznik 'div', ale już `<Welcome />` reprezentuje komponent i wymaga, aby `Welcome` było w zasięgu (ang. *scope*).

Aby dowiedzieć się więcej o uzasadnieniu tej konwencji, przeczytaj [dogłębną analizę składni JSX](#).

Kompozycja komponentów

Komponenty przy zwracaniu wyniku mogą odwoływać się do innych komponentów. Pozwala to używać tej samej abstrakcji komponentu na dowolnym poziomie szczegółowości. Przycisk, formularz, okno dialogowe, ekran - w aplikacjach reactowych tego typu składniki są zwykle reprezentowane przez dedykowane komponenty.

Możemy dla przykładu stworzyć komponent `App`, który wielokrotnie renderuje komponent `Welcome`:

```
function Welcome(props) {
  return <h1>Cześć, {props.name}</h1>;
}

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

[Przetestuj kod na CodePen](#)

Nowe aplikacje reactowe na samej górze drzewa zazwyczaj renderują pojedynczy komponent `App`. Jeśli jednak musisz zintegrować Reacta z istniejącą aplikacją, możesz zacząć od samego dołu, dodając niewielkie komponenty (np. `Button`) i stopniowo przepisywać całą strukturę aż do samej góry.

Wyodrębnianie komponentów

Nie bój się dzielenia komponentów na mniejsze części.

Rozważ poniższy komponent `Comment`:

```
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>
);
```

[Przetestuj kod na CodePen](#)

Przyjmuje on obiekt `author`, napis `text` i datę `date` jako właściwości i zwraca strukturę opisującą komentarz na portalu mediów społecznościowych.

Zmiana tego komponentu czy ponowne użycie jego poszczególnych części może okazać się skomplikowane z powodu całego tego zagnieżdżenia. Rozbijmy go zatem na kilka mniejszych komponentów.

Najpierw wydzielmy komponent `Avatar`:

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}>
  />
);
```

`Avatar` nie musi wiedzieć, że jest renderowany wewnątrz komponentu `Comment`. Dlatego też daliśmy jego właściwości bardziej ogólną nazwę `user` zamiast `author`.

Zalecamy nadawanie nazw właściwościom z punktu widzenia komponentu, a nie kontekstu, w którym jest używany.

Możemy teraz uprościć nieco komponent `Comment`:

```
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>
  );
}
```

Następnie wydzielimy komponent `UserInfo`, który wypisze `Avatar` obok nazwy użytkownika:

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

To pozwala nam uprościć `Comment` jeszcze bardziej:

```
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>
  );
}
```

[Przetestuj kod na CodePen](#)

Wyodrębnianie komponentów może z początku wydawać się żmudnym zajęciem, ale posiadanie palety pozwalających na ponowne użycie komponentów jest opłacalne w większych aplikacjach. Dobrą praktycznąasadą jest to, że jeśli część twojego interfejsu użytkownika jest używana wielokrotnie (np. `Button`, `Panel`, `Avatar`) lub jest ona dostatecznie skomplikowana sama w sobie (np. `App`, `FeedStory`, `Comment`), jest ona dobrym kandydatem do stania się oddzielnym komponentem.

Właściwości są tylko do odczytu

Bez względu na to, czy zadeklarujesz komponent [jako funkcję czy klasę](#), nie może on nigdy modyfikować swoich właściwości. Rozważ następującą funkcję `sum`:

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

Funkcje tego typu nazywane są "[czystymi](#)" (ang. *pure function*), dlatego że nie próbują one zmieniać swoich argumentów i zawsze zwracają ten sam wynik dla tych samych argumentów.

W przeciwieństwie do poprzedniej funkcji, ta poniżej nie jest "czysta", ponieważ zmienia swój argument.

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

React jest bardzo elastyczny, ale ma jedną ścisłą zasadę:

Wszytkie komponenty muszą zachowywać się jak czyste funkcje w odniesieniu do ich właściwości.

Rzec jasna, interfejsy użytkownika w aplikacjach są zwykle dynamiczne, zmieniają się w czasie. W [kolejnym rozdziale](#) wprowadzimy nowe pojęcie "stanu". Stan pozwala komponentom reaktywnym na zmianę swojego wyniku w czasie, w odpowiedzi na akcje użytkownika, żądania sieciowe itp. bez naruszania powyższej zasady.

Kompozycja a dziedziczenie

React posiada potężny model kompozycyjny, z którego zalecamy korzystać zamiast dziedziczenia, aby komponentów można było używać wielokrotnie.

W tej sekcji rozważymy kilka problemów, przy okazji których początkujący użytkownicy Reacta sięgają po dziedziczenie, a następnie pokażemy, jak rozwiązać je za pomocą kompozycji.

Zawieranie

Niektóre komponenty nie wiedzą z góry, co będzie ich "dziećmi". Najczęściej dotyczy to komponentów takich jak `Sidebar` czy `Dialog`, które reprezentują "pojemniki" ogólnego użytku.

Zalecamy, aby komponenty tego typu korzystały ze specjalnego atrybutu `children` i przekazywały go bezpośrednio do renderowanej struktury:

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

Pozwala to innym komponentom przekazywać dowolnych potomków poprzez zagnieżdzanie elementów JSX-owych:

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Witaj
      </h1>
      <p className="Dialog-message">
        Dziękujemy za wizytę na naszym statku kosmicznym!
      </p>
    </FancyBorder>
  );
}
```

[Przetestuj kod na CodePen](#)

Wszystko, co znajdzie się w JSX-owym znaczniku `<FancyBorder>` zostanie przekazane do komponentu `FancyBorder` poprzez atrybut `children`. Jako że `FancyBorder` renderuje `{props.children}` wewnętrz elementu `<div>`, właśnie w takim elemencie pojawią się ostatecznie przekazane komponenty.

Mimo że zdarza się to rzadziej, czasami trzeba wstawić do komponentu wiele takich "dziur". W takich przypadkach można wymyślić własną konwencję i używać jej zamiast `children`:

```

function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
  );
}

```

[Przetestuj kod na CodePen](#)

Elementy reactowe, takie jak `<Contacts />` czy `<Chat />`, są zwykłymi obiektami, dlatego możesz przekazywać je poprzez atrybuty jak każdą inną wartość. To podejście może przypominać koncepcję "slotów" z innych bibliotek, lecz w Reakcie nie ma żadnych ograniczeń co do typu wartości przekazywanych w atrybutach.

Specjalizacja

Czasami wyobrażamy sobie, że niektóre komponenty są "specjalnymi przypadkami użycia" innych komponentów. Na przykład, można by powiedzieć, że `WelcomeDialog` jest specjalnym przypadkiem komponentu `Dialog`.

W Reakcie taką relację również można osiągnąć poprzez kompozycję, gdzie "wyspecjalizowany" komponent renderuje inny, bardziej ogólny komponent i konfiguruje go za pomocą odpowiednich atrybutów:

```

function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
    </FancyBorder>
  );
}

function WelcomeDialog() {
  return (
    <Dialog

```

```

        title="Witaj"
        message="Dziękujemy za wizytę na naszym statku kosmicznym!" />
    );
}

```

[Przetestuj kod na CodePen](#)

Kompozycja działa równie dobrze z komponentami klasowymi:

```

function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
      {props.children}
    </FancyBorder>
  );
}

class SignUpDialog extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSignUp = this.handleSignUp.bind(this);
    this.state = {login: ''};
  }

  render() {
    return (
      <Dialog title="Program Eksploracji Marsa"
        message="Jak powinniśmy się do Ciebie zwracać?">
        <input value={this.state.login}
          onChange={this.handleChange} />
        <button onClick={this.handleSignUp}>
          Zapisz mnie!
        </button>
      </Dialog>
    );
  }

  handleChange(e) {
    this.setState({login: e.target.value});
  }

  handleSignUp() {
    alert(`Witaj na pokładzie, ${this.state.login}!`);
  }
}

```

[Przetestuj kod na CodePen](#)

A co z dziedziczeniem?

W Facebooku korzystamy z Reacta w tysiącach komponentów i nie znaleźliśmy jak dotąd żadnego przypadku użycia, w którym lepszym rozwiązaniem byłoby stworzenie hierarchii dziedziczenia.

Atrybuty i kompozycja dają wystarczającą dowolność w dostosowaniu zarówno wyglądu, jak i zachowania komponentu, w sposób jawni i bezpieczny. Pamiętaj, że komponenty mogą przyjmować atrybuty dowolnego rodzaju: typy podstawowe, elementy reactowe czy funkcje.

Jeśli planujesz wielokrotnie używać w różnych komponentach funkcjonalności niezwiązanego z renderowaniem, sugerujemy wydzielić ją do osobnego modułu javascriptowego. Wtedy komponenty będą mogły ją zimportować bez rozszerzania, bez względu na to, czy to funkcja, obiekt czy klasa.

Renderowanie warunkowe

React umożliwia tworzenie odrębnych komponentów, które hermetyzują (ang. *encapsulate*) pożądane przez ciebie metody. Wybrane komponenty mogą być renderowane bądź nie, w zależności od stanu twojej aplikacji.

Renderowanie warunkowe działa w Reakcie tak samo jak instrukcje warunkowe w javascriptcie. Aby stworzyć elementy odzwierciedlające aktualny stan aplikacji, należy użyć instrukcji `if` lub `operatora warunkowego` oraz pozwolić Reactowi je dopasować poprzez aktualizację interfejsu użytkownika.

Rozważmy następujące dwa komponenty:

```
function UserGreeting(props) {
  return <h1>Witamy ponownie!</h1>;
}

function GuestGreeting(props) {
  return <h1>Proszę się zarejestrować.</h1>;
}
```

Stworzymy komponent `Greeting` (pol. *Powitanie*), który wyświetlał będzie pierwszy lub drugi z powyższych komponentów w zależności od tego czy użytkownik jest zalogowany.

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
// Spróbuj zmienić na isLoggedIn={true}:
root.render(<Greeting isLoggedIn={false} />);
```

[Przetestuj kod na CodePen](#)

Powitanie renderowane przez kod w powyższym przykładzie zależy od wartości właściwości `isLoggedIn`.

Zmienne elementowe

Elementy mogą być przechowywane w zmiennych. Pozwala to na warunkowe renderowanie określonej części komponentu, podczas gdy pozostałe dane wyjściowe nie ulegają zmianie.

Przyjrzyjmy się dwóm nowym komponentom tworzącym przyciski logowania "Zaloguj się" (ang. *Login*) oraz "Wyloguj się" (ang. *Logout*):

```
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      Zaloguj się
    </button>
  );
}
```

```

    );
}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      Wyloguj się
    </button>
  );
}

```

W przykładzie poniżej zbudujemy komponent ze stanem o nazwie `LoginControl` (pol. *kontrola logowania*)

W zależności od aktualnego stanu, będzie on renderował przycisk logowania (`<LoginButton />`) lub wylogowania `<LogoutButton />`. Będzie on również renderował komponent `<Greeting />` z poprzedniego przykładu:

```

class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {isLoggedIn: false};
  }

  handleLoginClick() {
    this.setState({isLoggedIn: true});
  }

  handleLogoutClick() {
    this.setState({isLoggedIn: false});
  }

  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<LoginControl />);

```

[Przetestuj kod na CodePen](#)

Deklarowanie zmiennej oraz stosowanie instrukcji `if` to dobry sposób na warunkowe renderowanie komponentu. Czasem jednak przydaje się nieco krótsza składnia. JSX umożliwia kilka różnych opcji warunków wewnętrzliniowych. Przedstawiamy je poniżej.

Wewnętrzliniowy warunek `if` z użyciem logicznego operatora `&&` [{#inline-if-with-logical--operator}](#)

JSX pozwala [umieszczać w nawiasach klamrowych wyrażenia](#), łącznie z javascriptowym operatorem logicznym `&&`. Jest to przydatne, gdy chcemy jakiś element dołączyć warunkowo.

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Cześć!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          Masz {unreadMessages.length} nieprzeczytanych wiadomości.
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Mailbox unreadMessages={messages} />);
```

[Przetestuj kod na CodePen](#)

Powyższy kod działa, ponieważ w JavaScriptie `true && *wyrażenie*` zawsze jest ewaluowane jako `wyrażenie`, natomiast `false && wyrażenie` jako `false`.

Zatem jeśli warunek zwraca `true`, element następujący bezpośrednio po operatorze `&&` zostanie uwzględniony w danych wyjściowych. Natomiast jeśli warunek zwróci `false`, React zignoruje go i pominie przy renderowaniu.

Wstawienie wyrażenia fałszywego (ang. *falsy expression*) również spowoduje pominięcie elementu umieszczonego za operatorem `&&`, jednak zwróci to wyrażenie. W poniższym przykładzie metoda renderująca zwróci `<div>0</div>`.

```
render() {
  const count = 0;
  return (
    <div>
      {count && <h1>Wiadomości: {count}</h1>}
    </div>
  );
}
```

Skrócona forma `if-else` z operatorem warunkowym

Kolejną metodą renderowania warunkowego wewnątrz wyrażenia jest stosowanie javascriptowego operatora warunkowego `warunek ? true : false`.

W przykładzie poniżej używamy go, aby warunkowo wyrenderować niewielki blok tekstu.

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      Użytkownik jest teraz <b>{isLoggedIn ? 'zalogowany' : 'niezalogowany'}</b>.
    </div>
  );
}
```

Rozwiążanie to może być stosowane również w przypadku dłuższych wyrażeń:

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
      }
    </div>
  );
}
```

Czytelność takich wyrażeń jest oczywiście nieco mniejsza. Podobnie jak w JavaScriptie, wybór odpowiedniego stylu zależy od preferencji twoich i twojego zespołu. Jednocześnie należy pamiętać, że kiedy warunki stają się nazbyt złożone, dobrze jest rozważyć możliwość [wydzielenia osobnego komponentu](#)

Zapobieganie renderowaniu komponentu

W sporadycznych przypadkach może zachodzić potrzeba ukrycia się komponentu, mimo iż został on już wyrenderowany przez inny komponent. Aby to umożliwić, należy zwrócić zamiast niego wartość `null`.

W przykładzie poniżej, renderowanie baneru ostrzegawczego `<WarningBanner />` jest uzależnione od wartości właściwości o nazwie `warn` (pol. ostrzeż). Jeśli jest ona równa `false`, wówczas komponent ten nie jest renderowany.

```
function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }

  return (
    <div className="warning">
      Ostrzeżenie!
    </div>
  );
}
```

```
class Page extends React.Component {
  constructor(props) {
    super(props);
    this.state = {showWarning: true};
    this.handleToggleClick = this.handleToggleClick.bind(this);
  }

  handleToggleClick() {
    this.setState(state => ({
      showWarning: !state.showWarning
    }));
  }

  render() {
    return (
      <div>
        <WarningBanner warn={this.state.showWarning} />
        <button onClick={this.handleToggleClick}>
          {this.state.showWarning ? 'Ukryj' : 'Pokaż'}
        </button>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Page />);
```

Przetestuj kod na CodePen

Zwrócenie wartości `null` w metodzie `render` danego komponentu nie ma wpływu na wywoływanie metod cyklu życia tego komponentu. To znaczy, że np. metoda `componentDidUpdate` w dalszym ciągu zostanie wywołana.

Kontekst

Kontekst umożliwia przekazywanie danych wewnątrz drzewa komponentów bez konieczności przekazywania ich przez właściwości każdego komponentu pośredniego.

W typowej aplikacji reactowej dane przekazywane są z góry w dół (od rodzica do dziecka) poprzez właściwości. Może się to jednak okazać zbyt uciążliwe w przypadku niektórych danych (np. preferencje językowe, schemat UI czy paleta kolorów), z których korzystają komponenty na wielu poziomach struktury. Konteksty umożliwiają współdzielenie pewnych wartości przez takie komponenty bez konieczności przekazywania ich jako właściwości na każdym poziomie drzewa.

- [Kiedy należy użyć kontekstu?](#)
- [Zanim użyjesz kontekstu](#)
- [Interfejs API](#)
 - [React.createContext](#)
 - [Context.Provider](#)
 - [Class.contextType](#)
 - [Context.Consumer](#)
 - [Context.displayName](#)
- [Przykłady](#)
 - [Kontekst dynamiczny](#)
 - [Aktualizacja kontekstu z komponentu zagnieżdżonego](#)
 - [Odczyt z kilku kontekstów jednocześnie](#)
- [Zastrzeżenia](#)
- [Przestarzały interfejs API](#)

Kiedy należy użyć kontekstu?

Konteksty zaprojektowano do współdzielenia danych, które można uznać za "globalne" dla drzewa komponentów, takich jak informacje o zalogowanym użytkowniku, schemat kolorów czy preferowany język. W poniższym przykładzie wartość `theme`, potrzebną do ostylowania przycisku, przekazujemy ręcznie:

```
embed:context/motivation-problem.js
```

Z pomocą kontekstu moglibyśmy uniknąć przekazywania jej na każdym poziomie struktury:

```
embed:context/motivation-solution.js
```

Zanim użyjesz kontekstu

Zwykle kontekstu używa się w sytuacjach, w których pewne dane muszą być dostępne dla *wielu* komponentów na różnych poziomach zagnieżdżenia. Korzystaj jednak z tego mechanizmu z rozwagą, ponieważ utrudnia on wielokrotne używanie komponentów zależnych.

Jeśli twoim celem jest wyłącznie uniknięcie przekazywania wartości przez kilka poziomów drzewa, zwykle łatwiejszym i lepszym rozwiązaniem okazuje się kompozycja komponentów.

Rozważmy komponent `Page`, który musi przekazać właściwości `user` oraz `avatarSize` kilka poziomów w dół, tak aby głęboko zagnieżdżone komponenty `Link` i `Avatar` mogły je odczytać:

```
<Page user={user} avatarSize={avatarSize} />
// ... który renderuje...
<PageLayout user={user} avatarSize={avatarSize} />
// ... który renderuje...
<NavigationBar user={user} avatarSize={avatarSize} />
// ... który renderuje...
<Link href={user.permalink}>
  <Avatar user={user} size={avatarSize} />
</Link>
```

Przekazywanie tych wartości przez tyle poziomów, tylko po to by mógł je odczytać `Avatar`, wydaje się lekka przesadą. Dodatkowo, gdyby `Avatar` w pewnym momencie zaczął wymagać jeszcze jednej wartości z góry, należałoby ją również przekazać przez te wszystkie poziomy.

Jednym z rozwiązań tego problemu **bez używania kontekstów** jest **przekazanie w dół samego komponentu `Avatar`**, dzięki czemu komponenty pośrednie nie muszą niczego wiedzieć o właściwościach `user` czy `avatarSize`:

```
function Page(props) {
  const user = props.user;
  const userLink = (
    <Link href={user.permalink}>
      <Avatar user={user} size={props.avatarSize} />
    </Link>
  );
  return <PageLayout userLink={userLink} />;
}

// Teraz mamy:
<Page user={user} avatarSize={avatarSize} />
// ... który renderuje...
<PageLayout userLink={...} />
// ... który renderuje...
<NavigationBar userLink={...} />
// ... który renderuje...
{props.userLink}
```

Dzięki takiej zmianie tylko komponent `Page`, znajdujący się najwyżej w hierarchii, musi wiedzieć o tym, że `Link` i `Avatar` zależą od wartości `user` i `avatarSize`.

Takie "odwrócenie sterowania" (ang. *inversion of control*) sprawia, że kod staje się czytelniejszy poprzez zmniejszenie liczby wartości przeplecionych przez strukturę aplikacji, jak również daje większą kontrolę komponentom znajdującym się wysoko w hierarchii. Należy jednak pamiętać, że to podejście nie zawsze jest najlepszym rozwiązaniem. Przeniesienie złożoności w górę hierarchii sprawia, że komponenty na górze stają się bardziej złożone, a także wymusza często nadmierną elastyczność na komponentach poniżej.

Warto zwrócić uwagę, że dany komponent nie jest ograniczony do posiadania tylko jednego potomka. Po-przez właściwości do komponentu możemy przekazać wielu potomków naraz lub stworzyć kilka oddzielnych "slotów" dla poszczególnych potomków, jak opisano to w tym rozdziale:

```
function Page(props) {
  const user = props.user;
  const content = <Feed user={user} />;
  const topBar = (
    <NavigationBar>
      <Link href={user.permalink}>
        <Avatar user={user} size={props.avatarSize} />
      </Link>
    </NavigationBar>
  );
  return (
    <PageLayout
      topBar={topBar}
      content={content}
    />
  );
}
```

Takie podejście jest wystarczające w większości przypadków, gdy pojawia się konieczność oddzielenia potomka od jego bezpośrednich przodków. Można pójść jeszcze o krok dalej i skorzystać z wzorca właściwości renderujących (ang. *render props*), jeśli chcemy, by potomek przed wyrenderowaniem mógł skomunikować się z rodzicem.

Czasami jednak te same dane muszą być dostępne dla wielu komponentów w drzewie, na wielu różnych poziomach struktury. Konteksty pozwalają na "rozgłoszenie" (ang. *broadcast*) zarówno samych danych, jak i wszelkich ich modyfikacji, do komponentów znajdujących się poniżej w hierarchii. Używanie kontekstów, zamiast pozostałych wzorców, zwykle sprawdza się przy zarządzaniu aktualnym językiem lub motywem, a także przy przechowywaniu danych we wspólnej pamięci podręcznej.

Interfejs API

`React.createContext` [\[#reactcreatecontext\]](#)

```
const MyContext = React.createContext(defaultValue);
```

Tworzy obiekt kontekstu. Gdy React renderuje komponent, który zasubskrybował się do tego kontekstu, będzie przekazywać mu aktualną wartość z najbliższego "dostawcy" (`Provider`) powyżej w drzewie.

Argument `defaultValue` jest używany **tylko** gdy komponent odczytujący z kontekstu nie ma nad sobą żadnego dostawcy. Przydaje się on podczas testowania komponentów w izolacji, ponieważ nie ma konieczności opakowywania ich w sztucznych dostawców. Uwaga: przekazanie dostawcy wartości `undefined` nie spowoduje, że zasubskrybowane komponenty otrzymają wartość z argumentu `defaultValue`.

`Context.Provider` [\[#contextprovider\]](#)

```
<MyContext.Provider value={/* jakąś wartość */}>
```

Każdy obiekt kontekstu posiada własny komponent dostawcy (ang. *provider*), który pozwala komponentom odczytującym na zasubskrybowanie się na zmiany w tym kontekście.

Wartość przekazana przez dostawcę we właściwości `value` będzie trafiała do "konsumentów" (ang. *consumer*) tego kontekstu znajdujących się poniżej w drzewie. Jeden dostawca może być połączony z wieloma konsumentami. Zagnieżdżanie dostawców jeden pod drugim powoduje nadpisanie wartości kontekstu w danym poddrzewie.

Wszyscy konsumenti znajdujący się poniżej dostawcy będą ponownie renderowani przy każdej zmianie właściwości `value`. Propagacja od dostawcy do jego podległych konsumentów (wliczając w to `.contextType` i `useContext`) nie jest brana pod uwagę przez metodę `shouldComponentUpdate`, a co za tym idzie, konsumenti będą renderowani ponownie nawet jeśli ich przodkowie nie zostali przerenderowani.

Zmiany są wykrywane poprzez porównanie starej i nowej wartości przy użyciu algorytmu podobnego do `Object.is`.

Uwaga

Sposób, w jaki wykrywane są zmiany, może powodować problemy przy przekazywaniu do `value` obiektów (zob. ["Zastrzeżenia"](#)).

`Class.contextType` [\[#classcontexttype\]](#)

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* wykonaj akcję podczas montowania z użyciem aktualnej wartości z MyContext */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
    let value = this.context;
    /* wyrenderuj coś przy użyciu aktualnej wartości z MyContext */
  }
}
MyClass.contextType = MyContext;
```

Do właściwości `contextType` w komponencie klasowym można przypisać obiekt kontekstu utworzony przy pomocy funkcji `React.createContext()`. Dzięki niej wartość najbliższego kontekstu tego typu będzie dostępna pod zmienną `this.context`. Możesz odwoływać się do tej wartości w każdej z metod cyklu życia komponentu, łącznie z metodą renderującą.

Uwaga:

Za pomocą tego interfejsu można zasubskrybować się tylko do jednego kontekstu. Jeśli chcesz połączyć się z wieloma kontekstami, zajrzyj do sekcji pt. "[Odczyt z kilku kontekstów jednocześnie](#)".

Jeśli korzystasz z eksperymentalnej [składni publicznych pól klasy](#), do inicjalizacji `contextType` możesz użyć pola statycznego, oznaczonego przez **static**.

```
class MyClass extends React.Component {
  static contextType = MyContext;
  render() {
    let value = this.context;
    /* wyrenderuj coś na podstawie aktualnej wartości */
  }
}
```

Context.Consumer {#contextconsumer}

```
<MyContext.Consumer>
  {value => /* wyrenderuj coś na podstawie wartości z kontekstu */}
</MyContext.Consumer>
```

Komponent reactowy, który subskrybuje się na zmiany w kontekście. Pozwala na nasłuchiwanie zmian z wnętrza [komponentu funkcyjnego](#).

Jego [potomkiem musi być funkcja](#). Funkcja ta otrzymuje aktualną wartość z kontekstu i zwraca węzeł reaktywy. Argument `value` przekazany do tej funkcji będzie równy właściwości `value` najbliższego dostawcy tego kontekstu powyżej w drzewie. Jeśli ponad komponentem nie ma żadnego dostawcy, zostanie użyta wartość `defaultValue` przekazana do `createContext()`.

Uwaga

Aby dowiedzieć się więcej na temat wzorca "funkcji przekazanej jako potomek", zajrzyj do rozdziału o [właściwościach renderujących](#).

Context.displayName {#contextdisplayname}

Obiekt kontekstu przyjmuje atrybut tekstowy `displayName`. Jest on używany przez React DevTools do wyświetlenia nazwy kontekstu.

Na przykład, poniższy komponent wyświetli się w narzędziach deweloperskich jako "MyDisplayName":

```
const MyContext = React.createContext(/* jakąś wartość */);
MyContext.displayName = 'MyDisplayName';

<MyContext.Provider> // "MyDisplayName.Provider" w DevTools
<MyContext.Consumer> // "MyDisplayName.Consumer" w DevTools
```

Przykłady

Kontekst dynamiczny

Bardziej złożony przykład korzystający z dynamicznej wartości dla motywu:

theme-context.js `embed:context/theme-detailed-theme-context.js`

themed-button.js `embed:context/theme-detailed-themed-button.js`

app.js `embed:context/theme-detailed-app.js`

Aktualizacja kontekstu z komponentu zagnieżdzonego

Często pojawia się potrzeba zaktualizowania kontekstu z komponentu znajdującego się gdzieś głęboko w drzewie. W takim wypadku należy przez kontekst przekazać funkcję, za pomocą której konsumenti będą mogli go aktualizować:

theme-context.js `embed:context/updating-nested-context-context.js`

theme-toggler-button.js `embed:context/updating-nested-context-theme-toggler-button.js`

app.js `embed:context/updating-nested-context-app.js`

Odczyt z kilku kontekstów jednocześnie

Aby zapewnić szybkość ponownego renderowania kontekstu, React musi stworzyć w drzewie osobny węzeł dla każdego konsumenta.

`embed:context/multiple-contexts.js`

Jeśli często używasz dwóch lub więcej wartości z różnych kontekstów, sugerujemy stworzyć oddzielny komponent z właściwością renderującą (ang. *render prop*), który dostarcza je wszystkie.

Zastrzeżenia

Kontekst podczas decydowania, co należy ponownie wyrenderować, sprawdza tożsamość referencji. Z tego powodu w niektórych przypadkach ponowne wyrenderowanie rodzica dostawcy kontekstu może skutkować niechcianym powtórnym wyrenderowaniem wszystkich konsumentów danego kontekstu. W poniższym przykładzie dzieje się tak, ponieważ obiekt przekazywany do właściwości `value` dla dostawcy kontekstu jest za każdym razem tworzony na nowo:

`embed:context/reference-caveats-problem.js`

Aby temu zapobiec, wystarczy przenieść tę wartość do stanu rodzica:

`embed:context/reference-caveats-solution.js`

Przestarzały interfejs API

Uwaga

React poprzednio był wyposażony w eksperymentalny interfejs API dla kontekstów. Mimo iż jest on przestarzały, będzie wspierany we wszystkich wersjach 16.x, jednak aplikacje powinny dążyć do migracji na nową wersję. Przestarzały interfejs zostanie usunięty w kolejnej głównej wersji Reacta. Aby dowiedzieć się więcej na ten temat, [przeczytaj dokumentację przestarzałego kontekstu](#).

Stwórz nową aplikację w Reakcie

Najwygodniej będzie ci się pracowało, jeśli zastosujesz zintegrowany zestaw narzędzi (ang. *toolchain*). Użycie go pozwoli ci też zapewnić dobre doświadczenie użytkownikom twojej aplikacji.

Poniżej przedstawiamy kilka popularnych zestawów narzędziowych używanych w Reakcie. Pomagają one w zadaniach typu:

- zwiększanie skali projektu do wielu plików i komponentów;
- stosowanie dodatkowych bibliotek dostępnych w npm;
- wcześnie wykrywanie często popełnianych błędów;
- ciągła, edycja CSS i JS w trakcie tworzenia aplikacji bez konieczności odświeżania strony w przeglądarce;
- optymalizacja zbudowanego kodu przed wdrożeniem go do środowiska produkcyjnego.

Zestawy narzędziowe, które tutaj podajemy, umożliwiają szybkie rozpoczęcie pracy nad aplikacją i **nie wymagają żadnej wstępnej konfiguracji**

Stosowanie zestawu narzędziowego nie jest konieczne

Jeśli powyższe zadania nie przysparzają ci żadnych problemów lub jeśli nie wiesz jeszcze, jak stosować narzędzia JavaScript, możesz [dodać React jako zwykły znacznik `<script>` do strony HTML](#), ewentualnie [z użyciem JSX](#).

Jest to **najprostszy sposób zintegrowania Reacta z istniejącą już stroną internetową**. Zestaw narzędziowy możesz zawsze dodać do swojej aplikacji, gdy pojawi się taka potrzeba.

Zalecane zestawy narzędziowe

Zespół Reacta rekomenduje następujące rozwiązania:

- Jeśli **uczysz się Reacta** lub jeśli **tworysz nową aplikację jednostronicową**, skorzystaj z [Create React App](#).
- Jeśli tworysz **stronę internetową renderowaną na serwerze z użyciem Node.js**, spróbuj zastosować [Next.js](#).
- Jeśli tworysz **stronę statyczną, która ma głównie wyświetlać określone treści**, spróbuj zastosować [Gatsby](#).
- Jeśli tworysz **bibliotekę komponentów** lub **dokonujesz integracji z istniejącą bazą kodu**, spróbuj zastosować [bardziej elastyczne zestawy narzędziowe](#).

Create React App

[Create React App](#) zapewnia bardzo dogodne środowisko pracy sprzyjające **nauce Reacta**. Jest to najlepszy sposób, aby zacząć tworzyć **nową jednostronicową aplikację** w Reakcie.

Środowisko pracy stworzone przez Create React App nie tylko umożliwia ci stosowanie najnowszych funkcjonalności języka JavaScript, lecz także zoptymalizuje twój kod przed oddaniem go do użytku i ogólnie znacznie usprawni twoją pracę. Aby móc skorzystać z tego rozwiązania na swoim komputerze, będziesz potrzebować [Node >= 14.0.0 and npm >= 5.6](#). Stworzenie projektu umożliwi ci następujące komendy:

```
npx create-react-app moja-aplikacja  
cd moja-aplikacja  
npm start
```

Wskazówka

`npx` w pierwszej linijce powyżej to nie literówka, to [narzędzie uruchamiające pakiety zawarte w npm 5.2+](#).

Create React App nie obsługuje ani logiki backendu ani baz danych; tworzy jedynie frontendowy potok budowania (ang. *build pipeline*). Dzięki temu możesz go używać z dowolnie wybranym przez siebie backendem. Create React App zawiera narzędzia takie jak [Babel](#) i [webpack](#), ale nie musisz nic o nich wieǳieć, aby z powodzeniem używać ich w swoich projektach.

Kiedy uznasz, że Twoja aplikacja jest gotowa do wdrożenia do środowiska produkcyjnego, zastosuj komendę `npm run build`. Dzięki temu uzyskasz zoptymalizowaną wersję swojej aplikacji. Znajdziesz ją w folderze `build`. Więcej informacji na temat Create React App znajdziesz w [pliku README](#) oraz w tym [Przewodniku użytkownika](#).

Next.js

Next.js jest popularnym, lekkim frameworkiem służącym do budowy aplikacji reactowych **statycznych** oraz **renderowanych po stronie serwera**. Zestaw ten zawiera **rozwiązania ułatwiające dodawanie stylu i mechanizmy routingu**. Ponadto, Next.js domyślnie zakłada, że stosujesz [Node.js](#) jako środowisko serwerowe.

Zapoznaj się z zestawem narzędziowym Next.js poprzez [oficjalny przewodnik](#).

Gatsby

Gatsby to najlepsze narzędzie do tworzenia **stron statycznych** korzystających z Reacta. Pozwala ono na tworzenie struktury aplikacji przy użyciu komponentów reactowych, lecz jako wynik końcowy generuje wstępnie wyrenderowane pliki HTML i CSS, co zapewnia maksymalną szybkość ładowania strony.

Zapoznaj się z Gatsbym poprzez [oficjalny przewodnik](#) oraz [galerię zestawów startowych](#).

Bardziej elastyczne zestawy narzędziowe

Podane niżej zestawy narzędziowe są stosunkowo bardziej elastyczne i dają większą dowolność konfiguracji. Polecamy je bardziej doświadczonym użytkownikom:

- **Neutrino** łączy w sobie siłę [webpacka](#) z prostotą ustawień domyślnych. Zestaw ten zawiera ustawienia domyślne dla [aplikacji reactowych](#) i [komponentów reactowych](#).
- **Nx** jest zestawem narzędzi do programistów full-stackowych, z wbudowanym wsparciem dla Reacta, Next.js, [Expressa](#) i wielu innych.
- **Parcel** jest szybkim bundlerem aplikacji webowych, [współgrającym z Reactem](#).
- **Razzle** jest frameworkiem umożliwiającym renderowanie po stronie serwera bez konieczności konfiguracji, ale zarazem dającym większy wybór niż Next.js.

Tworzenie zestawu narzędziowego od podstaw

W skład javascriptowych narzędzi do budowania aplikacji wchodzą zazwyczaj następujące elementy:

- **Menadżer pakietów** np. [Yarn](#) lub [npm](#). Umożliwia on korzystanie z ogromnego ekosystemu dodatkowych pakietów. Pozwala łatwo je instalować i aktualizować.
- **Bundler** np. [webpack](#) lub [Parcel](#). Umożliwia on pisanie kodu modularnego i pakowania go w małe pakiety, aby zoptymalizować czas ładowania.
- **Kompilator** np. [Babel](#). Pozwala on na stosowanie nowych wersji JavaScriptu przy zachowaniu kompatybilności ze starszymi przeglądarkami.

Jeśli chcesz stworzyć swój własny zestaw narzędziowy od podstaw, zajrzyj do [tego przewodnika](#). Dowiesz się z niego, jak odtworzyć niektóre z funkcjonalności Create React App.

Pamiętaj, aby upewnić się, że twój własny zestaw narzędziowy [jest przystosowany do wdrożeń produkcyjnych](#).

Cross-origin Errors

Note:

The following section applies only to the development mode of React. Error handling in production mode is done with regular try/catch statements.

In [development mode](#), React uses a global `error` event handler to preserve the "pause on exceptions" behavior of browser DevTools. It also logs errors to the developer console.

If an error is thrown from a [different origin](#) the browser will mask its details and React will not be able to log the original error message. This is a security precaution taken by browsers to avoid leaking sensitive information.

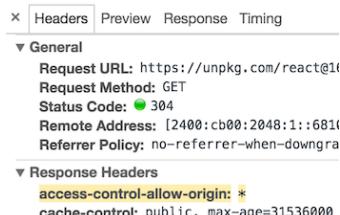
You can simplify the development/debugging process by ensuring that errors are thrown with a same-origin policy. Below are some common causes of cross-origin errors and ways to address them.

CDN

When loading React (or other libraries that might throw errors) from a CDN, add the `crossorigin` attribute to your `<script>` tags:

```
<script crossorigin src="..."></script>
```

Also ensure the CDN responds with the `Access-Control-Allow-Origin: *` HTTP header:



`Access-Control-Allow-Origin: *`

Webpack

Source maps

Some JavaScript bundlers may wrap the application code with `eval` statements in development. (For example Webpack will do this if `devtool` is set to any value containing the word "eval".) This may cause errors to be treated as cross-origin.

If you use Webpack, we recommend using the `cheap-module-source-map` setting in development to avoid this problem.

Code splitting

If your application is split into multiple bundles, these bundles may be loaded using JSONP. This may cause errors thrown in the code of these bundles to be treated as cross-origin.

To resolve this, use the `crossOriginLoading` setting in development to add the `crossorigin` attribute to the `<script>` tags generated for the JSONP requests.

[Go to TOC](#)

Design Principles

We wrote this document so that you have a better idea of how we decide what React does and what React doesn't do, and what our development philosophy is like. While we are excited to see community contributions, we are not likely to choose a path that violates one or more of these principles.

Note:

This document assumes a strong understanding of React. It describes the design principles of *React itself*, not React components or applications.

For an introduction to React, check out [Myślenie reaktywne](#) instead.

Composition

The key feature of React is composition of components. Components written by different people should work well together. It is important to us that you can add functionality to a component without causing rippling changes throughout the codebase.

For example, it should be possible to introduce some local state into a component without changing any of the components using it. Similarly, it should be possible to add some initialization and teardown code to any component when necessary.

There is nothing "bad" about using state or lifecycle methods in components. Like any powerful feature, they should be used in moderation, but we have no intention to remove them. On the contrary, we think they are integral parts of what makes React useful. We might enable [more functional patterns](#) in the future, but both local state and lifecycle methods will be a part of that model.

Components are often described as "just functions" but in our view they need to be more than that to be useful. In React, components describe any composable behavior, and this includes rendering, lifecycle, and state. Some external libraries like [Relay](#) augment components with other responsibilities such as describing data dependencies. It is possible that those ideas might make it back into React too in some form.

Common Abstraction

In general we [resist adding features](#) that can be implemented in userland. We don't want to bloat your apps with useless library code. However, there are exceptions to this.

For example, if React didn't provide support for local state or lifecycle methods, people would create custom abstractions for them. When there are multiple abstractions competing, React can't enforce or take advantage of the properties of either of them. It has to work with the lowest common denominator.

This is why sometimes we add features to React itself. If we notice that many components implement a certain feature in incompatible or inefficient ways, we might prefer to bake it into React. We don't do it lightly. When we do it, it's because we are confident that raising the abstraction level benefits the whole ecosystem. State, lifecycle methods, cross-browser event normalization are good examples of this.

We always discuss such improvement proposals with the community. You can find some of those discussions by the "[big picture](#)" label on the React issue tracker.

Escape Hatches

React is pragmatic. It is driven by the needs of the products written at Facebook. While it is influenced by some paradigms that are not yet fully mainstream such as functional programming, staying accessible to a wide range of developers with different skills and experience levels is an explicit goal of the project.

If we want to deprecate a pattern that we don't like, it is our responsibility to consider all existing use cases for it and [educate the community about the alternatives](#) before we deprecate it. If some pattern that is useful for building apps is hard to express in a declarative way, we will [provide an imperative API](#) for it. If we can't figure out a perfect API for something that we found necessary in many apps, we will [provide a temporary subpar working API](#) as long as it is possible to get rid of it later and it leaves the door open for future improvements.

Stability

We value API stability. At Facebook, we have more than 50 thousand components using React. Many other companies, including [Twitter](#) and [Airbnb](#), are also heavy users of React. This is why we are usually reluctant to change public APIs or behavior.

However we think stability in the sense of "nothing changes" is overrated. It quickly turns into stagnation. Instead, we prefer the stability in the sense of "It is heavily used in production, and when something changes, there is a clear (and preferably automated) migration path."

When we deprecate a pattern, we study its internal usage at Facebook and add deprecation warnings. They let us assess the impact of the change. Sometimes we back out if we see that it is too early, and we need to think more strategically about getting the codebases to the point where they are ready for this change.

If we are confident that the change is not too disruptive and the migration strategy is viable for all use cases, we release the deprecation warning to the open source community. We are closely in touch with many users of React outside of Facebook, and we monitor popular open source projects and guide them in fixing those deprecations.

Given the sheer size of the Facebook React codebase, successful internal migration is often a good indicator that other companies won't have problems either. Nevertheless sometimes people point out additional use cases we haven't thought of, and we add escape hatches for them or rethink our approach.

We don't deprecate anything without a good reason. We recognize that sometimes deprecations warnings cause frustration but we add them because deprecations clean up the road for the improvements and new features that we and many people in the community consider valuable.

For example, we added a [warning about unknown DOM props](#) in React 15.2.0. Many projects were affected by this. However fixing this warning is important so that we can introduce the support for [custom attributes](#) to React. There is a reason like this behind every deprecation that we add.

When we add a deprecation warning, we keep it for the rest of the current major version, and [change the behavior in the next major version](#). If there is a lot of repetitive manual work involved, we release a [codemod](#) script that automates most of the change. Codemods enable us to move forward without stagnation in a massive codebase, and we encourage you to use them as well.

You can find the codemods that we released in the [react-codemod](#) repository.

Interoperability

We place high value in interoperability with existing systems and gradual adoption. Facebook has a massive non-React codebase. Its website uses a mix of a server-side component system called XHP, internal UI libraries that came before React, and React itself. It is important to us that any product team can [start using React for a small feature](#) rather than rewrite their code to bet on it.

This is why React provides escape hatches to work with mutable models, and tries to work well together with other UI libraries. You can wrap an existing imperative UI into a declarative component, and vice versa. This is crucial for gradual adoption.

Scheduling

Even when your components are described as functions, when you use React you don't call them directly. Every component returns a [description of what needs to be rendered](#), and that description may include both user-written components like `<LikeButton>` and platform-specific components like `<div>`. It is up to React to "unroll" `<LikeButton>` at some point in the future and actually apply changes to the UI tree according to the render results of the components recursively.

This is a subtle distinction but a powerful one. Since you don't call that component function but let React call it, it means React has the power to delay calling it if necessary. In its current implementation React walks the tree recursively and calls render functions of the whole updated tree during a single tick. However in the future it might start [delaying some updates to avoid dropping frames](#).

This is a common theme in React design. Some popular libraries implement the "push" approach where computations are performed when the new data is available. React, however, sticks to the "pull" approach where computations can be delayed until necessary.

React is not a generic data processing library. It is a library for building user interfaces. We think that it is uniquely positioned in an app to know which computations are relevant right now and which are not.

If something is offscreen, we can delay any logic related to it. If data is arriving faster than the frame rate, we can coalesce and batch updates. We can prioritize work coming from user interactions (such as an animation caused by a button click) over less important background work (such as rendering new content just loaded from the network) to avoid dropping frames.

To be clear, we are not taking advantage of this right now. However the freedom to do something like this is why we prefer to have control over scheduling, and why `setState()` is asynchronous. Conceptually, we think of it as "scheduling an update".

The control over scheduling would be harder for us to gain if we let the user directly compose views with a "push" based paradigm common in some variations of [Functional Reactive Programming](#). We want to own the "glue" code.

It is a key goal for React that the amount of the user code that executes before yielding back into React is minimal. This ensures that React retains the capability to schedule and split work in chunks according to what it knows about the UI.

There is an internal joke in the team that React should have been called "Schedule" because React does not want to be fully "reactive".

Developer Experience

Providing a good developer experience is important to us.

For example, we maintain [React DevTools](#) which let you inspect the React component tree in Chrome and Firefox. We have heard that it brings a big productivity boost both to the Facebook engineers and to the community.

We also try to go an extra mile to provide helpful developer warnings. For example, React warns you in development if you nest tags in a way that the browser doesn't understand, or if you make a common typo in the API. Developer warnings and the related checks are the main reason why the development version of React is slower than the production version.

The usage patterns that we see internally at Facebook help us understand what the common mistakes are, and how to prevent them early. When we add new features, we try to anticipate the common mistakes and warn about them.

We are always looking out for ways to improve the developer experience. We love to hear your suggestions and accept your contributions to make it even better.

Debugging

When something goes wrong, it is important that you have breadcrumbs to trace the mistake to its source in the codebase. In React, props and state are those breadcrumbs.

If you see something wrong on the screen, you can open React DevTools, find the component responsible for rendering, and then see if the props and state are correct. If they are, you know that the problem is in the component's `render()` function, or some function that is called by `render()`. The problem is isolated.

If the state is wrong, you know that the problem is caused by one of the `setState()` calls in this file. This, too, is relatively simple to locate and fix because usually there are only a few `setState()` calls in a single file.

If the props are wrong, you can traverse the tree up in the inspector, looking for the component that first "poisoned the well" by passing bad props down.

This ability to trace any UI to the data that produced it in the form of current props and state is very important to React. It is an explicit design goal that state is not "trapped" in closures and combinators, and is available to React directly.

While the UI is dynamic, we believe that synchronous `render()` functions of props and state turn debugging from guesswork into a boring but finite procedure. We would like to preserve this constraint in React even though it makes some use cases, like complex animations, harder.

Configuration

We find global runtime configuration options to be problematic.

For example, it is occasionally requested that we implement a function like `React.configure(options)` or `React.register(component)`. However this poses multiple problems, and we are not aware of good solutions to them.

What if somebody calls such a function from a third-party component library? What if one React app embeds another React app, and their desired configurations are incompatible? How can a third-party component specify that it requires a particular configuration? We think that global configuration doesn't work well with composition. Since composition is central to React, we don't provide global configuration in code.

We do, however, provide some global configuration on the build level. For example, we provide separate development and production builds. We may also [add a profiling build](#) in the future, and we are open to considering other build flags.

Beyond the DOM

We see the value of React in the way it allows us to write components that have fewer bugs and compose together well. DOM is the original rendering target for React but [React Native](#) is just as important both to Facebook and the community.

Being renderer-agnostic is an important design constraint of React. It adds some overhead in the internal representations. On the other hand, any improvements to the core translate across platforms.

Having a single programming model lets us form engineering teams around products instead of platforms. So far the tradeoff has been worth it for us.

Implementation

We try to provide elegant APIs where possible. We are much less concerned with the implementation being elegant. The real world is far from perfect, and to a reasonable extent we prefer to put the ugly code into the library if it means the user does not have to write it. When we evaluate new code, we are looking for an implementation that is correct, performant and affords a good developer experience. Elegance is secondary.

We prefer boring code to clever code. Code is disposable and often changes. So it is important that it [doesn't introduce new internal abstractions unless absolutely necessary](#). Verbose code that is easy to move around, change and remove is preferred to elegant code that is prematurely abstracted and hard to change.

Optimized for Tooling

Some commonly used APIs have verbose names. For example, we use `componentDidMount()` instead of `didMount()` or `onMount()`. This is [intentional](#). The goal is to make the points of interaction with the library highly visible.

In a massive codebase like Facebook, being able to search for uses of specific APIs is very important. We value distinct verbose names, and especially for the features that should be used sparingly. For example, `dangerouslySetInnerHTML` is hard to miss in a code review.

Optimizing for search is also important because of our reliance on [codemods](#) to make breaking changes. We want it to be easy and safe to apply vast automated changes across the codebase, and unique verbose names help us achieve this. Similarly, distinctive names make it easy to write custom [lint rules](#) about using React without worrying about potential false positives.

[JSX](#) plays a similar role. While it is not required with React, we use it extensively at Facebook both for aesthetic and pragmatic reasons.

In our codebase, JSX provides an unambiguous hint to the tools that they are dealing with a React element tree. This makes it possible to add build-time optimizations such as [hoisting constant elements](#), safely lint and codemod internal component usage, and [include JSX source location](#) into the warnings.

Dogfooding

We try our best to address the problems raised by the community. However we are likely to prioritize the issues that people are *also* experiencing internally at Facebook. Perhaps counter-intuitively, we think this is the main reason why the community can bet on React.

Heavy internal usage gives us the confidence that React won't disappear tomorrow. React was created at Facebook to solve its problems. It brings tangible business value to the company and is used in many of its products. [Dogfooding](#) it means that our vision stays sharp and we have a focused direction going forward.

This doesn't mean that we ignore the issues raised by the community. For example, we added support for [web components](#) and [SVG](#) to React even though we don't rely on either of them internally. We are actively [listening to your pain points](#) and [address them](#) to the best of our ability. The community is what makes React special to us, and we are honored to contribute back.

After releasing many open source projects at Facebook, we have learned that trying to make everyone happy at the same time produced projects with poor focus that didn't grow well. Instead, we found that picking a small audience and focusing on making them happy brings a positive net effect. That's exactly what we did with React, and so far solving the problems encountered by Facebook product teams has translated well to the open source community.

The downside of this approach is that sometimes we fail to give enough focus to the things that Facebook teams don't have to deal with, such as the "getting started" experience. We are acutely aware of this, and we are thinking of how to improve in a way that would benefit everyone in the community without making the same mistakes we did with open source projects before.

Granice błędów

W przeszłości błędy javascriptowe wewnętrz komponentów uszkadzały wewnętrzny stan Reacta i [wywoływały tajemnicze błędy](#) w kolejnych renderowaniach. Były one następstwem wcześniejszego błędu w kodzie aplikacji, jednakże React nie dostarczał żadnego rozwiązania pozwalającego na właściwe ich obsłużenie wewnątrz komponentów oraz nie potrafił odtworzyć aplikacji po ich wystąpieniu.

Przedstawiamy granice błędów

Błąd w kodzie JavaScript, występujący w jednej z części interfejsu użytkownika (UI), nie powinien psuć całej aplikacji. Aby rozwiązać ten problem, w Reakcie 16 wprowadziliśmy koncepcję granic błędów (ang. *error boundary*).

Granice błędów to komponenty reactowe, które **przechwytują błędy javascriptowe występujące gdziekolwiek wewnątrz drzewa komponentów ich potomków, a następnie logują je i wyświetlają zastępczy interfejs UI**, zamiast pokazywać ten niepoprawnie działający. Granice błędów przechwytują błędy występujące podczas renderowania, w metodach cyklu życia komponentów, a także w konstruktorach całego podzielnego im drzewa komponentów.

Uwaga

Granice błędów **nie obsługują** błędów w:

- Procedurach obsługi zdarzeń (ang. event handlers) ([informacje](#))
- Asynchronicznym kodzie (np. w metodach: `setTimeout` lub w funkcjach zwrotnych `requestAnimationFrame`)
- Komponentach renderowanych po stronie serwera
- Błędach rzuconych w samych granicach błędów (a nie w ich podzielnnych komponentach)

Aby komponent klasowy stał się granicą błędu, musi definiować jedną lub obie metody cyklu życia: `static getDerivedStateFromError()` i/lub `componentDidCatch()`. Należy używać `static getDerivedStateFromError()` do wyrenderowania zastępczego UI po rzuceniu błędu, a `componentDidCatch()`, aby zalogować informacje o błędzie.

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

  static getDerivedStateFromError(error) {
    // Zaktualizuj stan, aby następny render pokazał zastępce UI.
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
```

```
// Możesz także zalogować błąd do zewnętrznego serwisu raportowania błędów
logErrorToMyService(error, errorInfo);
}

render() {
  if (this.state.hasError) {
    // Możesz wyrenderować dowolny interfejs zastępczy.
    return <h1>Something went wrong.</h1>;
  }

  return this.props.children;
}
}
```

Po zdefiniowaniu, granicy błędu można używać jak normalnego komponentu:

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

Granice błędów działają jak bloki `catch {}` w JavaScript, tyle że dla komponentów. Można je zdefiniować tylko w komponentach klasowych. W praktyce, w większości przypadków wystarczy zdefiniować jeden komponent granicy błędu i używać go w całej aplikacji.

Należy pamiętać, że **granice błędów wyłapują błędy w komponentach potomnych**. Nie są one jednak w stanie obsłużyć własnych błędów. W takim przypadku, jeżeli granica błędu nie będzie w stanie wyświetlić zastępczego UI, błąd zostanie przekazany do kolejnej najbliższej granicy błędu powyżej w strukturze komponentów. Jest to zachowanie podobne do tego znanego z javascriptowych bloków `catch {}`.

Demo

[Przykład tworzenia i użycia granicy błędów.](#)

Gdzie umiejszczyć granice błędów

Poziom granularności granic błędów zależy wyłącznie od ciebie. Możesz opakować tylko główny komponent aplikacji i wyświetlać tekst "Coś poszło nie tak", jak to zwykle robi się we frameworkach serwerowych. Możesz też otoczyć każdy z widgetów aplikacji osobną granicą błędów, aby błędy w ich wnętrzu nie zabijały całej aplikacji.

Nowe zachowanie nieobsłużonych błędów

Wprowadzenie granic błędów ma ważne następstwo. **Od Reacta w wersji 16, błędy, które nie zostały obsłużone za pomocą granicy błędów, spowodują odmontowanie całego drzewa komponentów.**

Przedyskutowaliśmy tę zmianę i z naszego doświadczenia wynika, że lepiej jest usunąć całe drzewo komponentów niż wyświetlać zepsute fragmenty UI. Na przykład, w produkcie takim jak Messenger pozostawienie wyświetlonego zepsutego kawałka UI może sprawić, że ktoś nieświadomie wyśle wiadomość do innej osoby. Również w aplikacjach finansowych wyświetlanie złego stanu konta jest gorszą sytuacją niż nie wyświetlenie niczego.

Ta zmiana oznacza, że wraz z migracją do nowej wersji Reacta odkryte zostaną błędy w aplikacjach, które do tej pory nie zostały zauważone. Dodanie granic błędów zapewni lepsze doświadczenie dla użytkownika, gdy coś pójdzie nie tak.

Przykładowo, Facebook Messenger opakowuje w osobne granice błędów następujące fragmenty aplikacji: zawartość paska bocznego, panel z informacjami o konwersacji, listę konwersacji i pole tekstowe na wiadomość. Jeżeli jeden z tych elementów zadziała nieprawidłowo, reszta pozostałe interaktywa i działająca.

Zachęcamy również do używania (lub zbudowania własnego) narzędzia do raportowania błędów, dzięki czemu będzie możliwe poznanie nieobsłużonych błędów występujących w środowisku produkcyjnym.

Ślad stosu komponentów

React 16, w środowisku deweloperskim, wyświetla w konsoli wszystkie błędy złapane podczas renderowania, nawet jeżeli aplikacja przypadkowo je przejmie. Oprócz wiadomości błędu i javascriptowego stosu, dostępny jest również stos komponentów. Dzięki temu wiadomo, gdzie dokładnie w drzewie komponentów wystąpił błąd:

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
  in BuggyCounter (created by App)
  in ErrorBoundary (created by App)
  in div (created by App)
  in App
```

W drzewie komponentów widoczne są również numery linii i nazwy plików. Ten mechanizm domyślnie działa w aplikacjach stworzonych przy użyciu [Create React App](#):

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
  in BuggyCounter (at App.js:26)
  in ErrorBoundary (at App.js:21)
  in div (at App.js:8)
  in App (at index.js:5)
```

Jeżeli nie używasz Create React App, możesz ręcznie dodać [ten plugin](#) do swojej konfiguracji Babela. Został on stworzony do używania tylko podczas fazy deweloperskiej i **powinien zostać wyłączony w środowisku produkcyjnym**

Uwaga

Nazwy komponentów wyświetlane w śladzie stosu zależą od właściwości `Function.name`. Jeżeli obsługujesz starsze przeglądarki, które nie dostarczają jej natywnie (np. IE 11), możesz dodać łatkę taką jak `function.name-polyfill`. Alternatywą jest zadeklarowanie wprost `displayName` we wszystkich komponentach.

A co z try/catch?

`try / catch` jest świetnym rozwiązaniem, ale działa tylko dla imperatywnego kodu:

```
try {
  showButton();
} catch (error) {
  // ...
}
```

Natomiast komponenty reaktywne są deklaratywne i określają, co powinno zostać wyrenderowane:

```
<Button />
```

Granice błędów zachowują deklaratywną naturę Reacta. Na przykład, jeżeli w metodzie `componentDidUpdate` wystąpi błąd podczas aktualizacji stanu, aplikacja poprawnie przekaże błąd do najbliższej granicy błędów.

A co z procedurami obsługi zdarzeń?

Granice błędów nie obsługują błędów z procedur obsługi zdarzeń.

React nie potrzebuje granic błędów do przywrócenia aplikacji po błędzie powstałym w procedurze obsługi zdarzeń. W przeciwieństwie do metod cyklu życia komponentu lub metody renderującej, procedury obsługi zdarzeń nie są wywoływane w trakcie renderowania. Dzięki temu nawet w przypadku błędu React wie, co wyświetlić na ekranie.

Aby obsłużyć błąd w procedurze obsługi zdarzenia, należy użyć javascriptowego `try / catch`:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    try {
      // Kod, który rzuci wyjątek
    } catch (error) {
      this.setState({ error });
    }
  }

  render() {
    if (this.state.error) {
      return <h1>Caught an error.</h1>
    }
    return <button onClick={this.handleClick}>Click Me</button>
  }
}
```

Powyższy przykład prezentuje normalne zachowanie JavaScriptu i nie używa granic błędów.

Zmiany nazewnictwa od Reacta w wersji 15

React 15 zawierał bardzo okrojoną obsługę granic błędów za pomocą metody o nazwie `unstable_handleError`. Ta metoda nie jest już obsługiwana i należy zmienić jej nazwę na `componentDidCatch` począwszy od pierwszych beta wersji Reacta 16.

Ze względu na tę zmianę stworzyliśmy [codemod](#), który automatycznie przekształci twój kod.

Error Decoder

W zminifikowanej wersji produkcyjnej Reacta unikamy wysyłania pełnych komunikatów o błędach w celu zmniejszenia liczby bajtów przesyłanych przez sieć.

Zdecydowanie zalecamy korzystanie z komplikacji deweloperskiej lokalnie podczas debugowania aplikacji, ponieważ śledzi ona dodatkowe informacje debugowania i zapewnia przydatne ostrzeżenia o potencjalnych problemach w aplikacji. Jeśli jednak napotkasz wyjątek podczas korzystania z kodu produkcyjnego, na tej stronie zostanie odtworzony oryginalny tekst błędu.

AJAX i API

Jak mogę wykonać zapytanie AJAX?

Możesz użyć dowolnej biblioteki AJAX. Do najpopularniejszych wyborów należą: [Axios](#), [jQuery AJAX](#) oraz wbudowane w przeglądarki [window.fetch](#).

W którym momencie cyklu życia komponentu powinno się wykonać zapytanie AJAX?

Dane należy uzupełniać z wykorzystaniem zapytań AJAX w metodzie `componentDidMount`. Dzięki temu po pobraniu danych możliwe będzie użycie metody `setState` do zmodyfikowania stanu komponentu.

Przykład: Używanie rezultatu zapytania AJAX do ustawienia lokalnego stanu

Niniejszy przykład pokazuje, jak wykonując zapytania AJAX w metodzie `componentDidMount` można zmodyfikować stan komponentu.

Nasze przykładowe API zwraca następujący obiekt JSON:

```
{
  "items": [
    { "id": 1, "name": "Jabłka", "price": "2 zł" },
    { "id": 2, "name": "Brzoskwinie", "price": "5 zł" }
  ]
}

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      error: null,
      isLoading: false,
      items: []
    };
  }

  componentDidMount() {
    fetch("https://api.example.com/items")
      .then(res => res.json())
      .then(
        (result) => {
          this.setState({
            isLoading: true,
            items: result.items
          });
        },
        // Uwaga: to ważne, żeby obsługiwać błędy tutaj, a
        // nie w bloku catch(), aby nie przetwarzanie błędów
        // mających swoje źródło w komponencie.
        (error) => {
          this.setState({
            isLoading: true,
            error: error
          });
        }
      );
  }
}
```

```

        error
    });
}
}

render() {
  const { error, isLoading, items } = this.state;
  if (error) {
    return <div>Błąd: {error.message}</div>;
  } else if (!isLoading) {
    return <div>Ładowanie...</div>;
  } else {
    return (
      <ul>
        {items.map(item => (
          <li key={item.id}>
            {item.name} {item.price}
          </li>
        ))}
      </ul>
    );
  }
}
}

```

Here is the equivalent with [Hooks](#):

```

function MyComponent() {
  const [error, setError] = useState(null);
  const [isLoading, setIsLoaded] = useState(false);
  const [items, setItems] = useState([]);

  // Note: the empty deps array [] means
  // this useEffect will run once
  // similar to componentDidMount()
  useEffect(() => {
    fetch("https://api.example.com/items")
      .then(res => res.json())
      .then(
        (result) => {
          setIsLoaded(true);
          setItems(result);
        },
        // Note: it's important to handle errors here
        // instead of a catch() block so that we don't swallow
        // exceptions from actual bugs in components.
        (error) => {
          setIsLoaded(true);
          setError(error);
        }
      )
    , []);
  }

  if (error) {
    return <div>Error: {error.message}</div>;
  } else if (!isLoading) {
    return <div>Loading...</div>;
  } else {
    return (
      <ul>

```

```
{items.map(item => (
  <li key={item.id}>
    {item.name} {item.price}
  </li>
))}</ul>
);
}
```

Babel, JSX i kroki budowania

Czy trzeba używać JSX z Reactem?

Nie! Sprawdź "[React bez JSX](#)", aby dowiedzieć się więcej.

Czy trzeba używać ES6 (+) z Reactem?

Nie! sprawdź "[React bez ES6](#)", aby dowiedzieć się więcej.

W jaki sposób napisać komentarz w JSX?

```
<div>
  /* To jest komentarz */
  Witaj, {name}!
</div>
```

```
<div>
  /* A to jest
   wieloliniowy komentarz. */
  Witaj, {name}!
</div>
```

Passing Functions to Components

How do I pass an event handler (like onClick) to a component?

Pass event handlers and other functions as props to child components:

```
<button onClick={this.handleClick}>
```

If you need to have access to the parent component in the handler, you also need to bind the function to the component instance (see below).

How do I bind a function to a component instance?

There are several ways to make sure functions have access to component attributes like `this.props` and `this.state`, depending on which syntax and build steps you are using.

Bind in Constructor (ES2015)

```
class Foo extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

Class Properties (ES2022)

```
class Foo extends Component {
  handleClick = () => {
    console.log('Click happened');
  };
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

Bind in Render

```
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick.bind(this)}>Click Me</button>;
  }
}
```

Note:

Using `Function.prototype.bind` in render creates a new function each time the component renders, which may have performance implications (see below).

Arrow Function in Render

```
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={() => this.handleClick()}>Click Me</button>;
  }
}
```

Note:

Using an arrow function in render creates a new function each time the component renders, which may break optimizations based on strict identity comparison.

Is it OK to use arrow functions in render methods?

Generally speaking, yes, it is OK, and it is often the easiest way to pass parameters to callback functions.

If you do have performance issues, by all means, optimize!

Why is binding necessary at all?

In JavaScript, these two code snippets are **not** equivalent:

```
obj.method();
```

```
var method = obj.method;
method();
```

Binding methods helps ensure that the second snippet works the same way as the first one.

With React, typically you only need to bind the methods you *pass* to other components. For example, `<button onClick={this.handleClick}>` passes `this.handleClick` so you want to bind it. However, it is unnecessary to bind the `render` method or the lifecycle methods: we don't pass them to other components.

[This post by Yehuda Katz](#) explains what binding is, and how functions work in JavaScript, in detail.

Why is my function being called every time the component renders?

Make sure you aren't *calling the function* when you pass it to the component:

```
render() {
  // Wrong: handleClick is called instead of passed as a reference!
  return <button onClick={this.handleClick()}>Click Me</button>
}
```

Instead, *pass the function itself* (without parens):

```
render() {
  // Correct: handleClick is passed as a reference!
  return <button onClick={this.handleClick}>Click Me</button>
}
```

How do I pass a parameter to an event handler or callback?

You can use an arrow function to wrap around an event handler and pass parameters:

```
<button onClick={() => this.handleClick(id)} />
```

This is equivalent to calling `.bind`:

```
<button onClick={this.handleClick.bind(this, id)} />
```

Example: Passing params using arrow functions

```
const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }
  handleClick(letter) {
    this.setState({ justClicked: letter });
  }
  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} onClick={() => this.handleClick(letter)}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    );
  }
}
```

Example: Passing params using data-attributes

Alternately, you can use DOM APIs to store data needed for event handlers. Consider this approach if you need to optimize a large number of elements or have a render tree that relies on `React.PureComponent` equality checks.

```
const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }

  handleClick(e) {
    this.setState({
      justClicked: e.target.dataset.letter
    });
  }

  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} data-letter={letter} onClick={this.handleClick}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    );
  }
}
```

How can I prevent a function from being called too quickly or too many times in a row?

If you have an event handler such as `onClick` or `onScroll` and want to prevent the callback from being fired too quickly, then you can limit the rate at which callback is executed. This can be done by using:

- **throttling**: sample changes based on a time based frequency (eg `_.throttle`)
- **debouncing**: publish changes after a period of inactivity (eg `_.debounce`)
- **requestAnimationFrame throttling**: sample changes based on `requestAnimationFrame` (eg `raf-schd`)

See [this visualization](#) for a comparison of `throttle` and `debounce` functions.

Note:

`_.debounce`, `_.throttle` and `raf-schd` provide a `cancel` method to cancel delayed callbacks. You should either call this method from `componentWillUnmount` or check to ensure that the component is still mounted within the delayed function.

Throttle

Throttling prevents a function from being called more than once in a given window of time. The example below throttles a "click" handler to prevent calling it more than once per second.

```
import throttle from 'lodash.throttle';

class LoadMoreButton extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.handleClickThrottled = throttle(this.handleClick, 1000);
  }

  componentWillMount() {
    this.handleClickThrottled.cancel();
  }

  render() {
    return <button onClick={this.handleClickThrottled}>Load More</button>;
  }

  handleClick() {
    this.props.loadMore();
  }
}
```

Debounce

Debouncing ensures that a function will not be executed until after a certain amount of time has passed since it was last called. This can be useful when you have to perform some expensive calculation in response to an event that might dispatch rapidly (eg scroll or keyboard events). The example below debounces text input with a 250ms delay.

```
import debounce from 'lodash.debounce';

class Searchbox extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.emitChangeDebounced = debounce(this.emitChange, 250);
  }

  componentWillMount() {
    this.emitChangeDebounced.cancel();
  }

  render() {
    return (
      <input
        type="text"
        onChange={this.handleChange}
    )
  }
}
```

```

        placeholder="Search...""
        defaultValue={this.props.value}
      />
    );
}

handleChange(e) {
  this.emitChangeDebounced(e.target.value);
}

emitChange(value) {
  this.props.onChange(value);
}

```

`requestAnimationFrame` throttling

`requestAnimationFrame` is a way of queuing a function to be executed in the browser at the optimal time for rendering performance. A function that is queued with `requestAnimationFrame` will fire in the next frame. The browser will work hard to ensure that there are 60 frames per second (60 fps). However, if the browser is unable to it will naturally *limit* the amount of frames in a second. For example, a device might only be able to handle 30 fps and so you will only get 30 frames in that second. Using `requestAnimationFrame` for throttling is a useful technique in that it prevents you from doing more than 60 updates in a second. If you are doing 100 updates in a second this creates additional work for the browser that the user will not see anyway.

Note:

Using this technique will only capture the last published value in a frame. You can see an example of how this optimization works on [MDN](#)

```

import rafSchedule from 'raf-schd';

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

    this.handleScroll = this.handleScroll.bind(this);

    // Create a new function to schedule updates.
    this.scheduleUpdate = rafSchedule(
      point => this.props.onScroll(point)
    );
  }

  handleScroll(e) {
    // When we receive a scroll event, schedule an update.
    // If we receive many updates within a frame, we'll only publish the latest
    // value.
    this.scheduleUpdate({ x: e.clientX, y: e.clientY });
  }

  componentWillUnmount() {
    // Cancel any pending updates since we're unmounting.
  }
}

```

```
    this.scheduleUpdate.cancel();
}

render() {
  return (
    <div
      style={{ overflow: 'scroll' }}
      onScroll={this.handleScroll}
    >
      
    </div>
  );
}
```

Testing your rate limiting

When testing your rate limiting code works correctly it is helpful to have the ability to fast forward time. If you are using `jest` then you can use `mock timers` to fast forward time. If you are using `requestAnimationFrame` throttling then you may find `raf-stub` to be a useful tool to control the ticking of animation frames.

Wirtualny DOM i organizacja wewnętrzna

Czym jest wirtualny model DOM?

Wirtualny model DOM (VDOM) to koncepcja programistyczna, w której idealna lub inaczej „wirtualna” reprezentacja interfejsu użytkownika jest przechowywana w pamięci i synchronizowana z „prawdziwym” modelem DOM przez bibliotekę taką jak ReactDOM. Proces ten nazywa się [rekoncyliacją \(ang. reconciliation\)](#)

Podejście to pozwala na użycie deklaratywnego interfejsu API Reacta: informujesz Reacta, w jakim stanie ma być interfejs użytkownika, a React upewnia się, że model DOM pasuje do tego stanu. React ukrywa pod warstwą abstrakcji manipulację atrybutami, obsługę zdarzeń i ręczną aktualizację modelu DOM, których w innym przypadku należałyby samodzielnie użyć do zbudowania aplikacji.

Ponieważ „wirtualny model DOM” jest bardziej wzorcem niż konkretną technologią, ludzie, mówiąc o nim, mają na myśli różne rzeczy. W świecie Reacta termin „wirtualny model DOM” jest zwykle kojarzony z [elementami reactowymi](#), ponieważ są to obiekty reprezentujące interfejs użytkownika. React wykorzystuje jednak również wewnętrzne obiekty zwane „włóknami” (ang. *fibers*) do przechowywania dodatkowych informacji o drzewie komponentów. Można je również uznać za część implementacji „wirtualnego modelu DOM” w Reakcie.

Czy model Shadow DOM jest tym samym co wirtualny model DOM?

Nie, oba modele różnią się między sobą. Model Shadow DOM to technologia przeglądarki zaprojektowana przede wszystkim do określania zakresu zmiennych i stylów CSS w komponentach sieciowych (ang. *web components*). Wirtualny DOM to koncepcja implementowana przez javascriptowe biblioteki, które bazują na API przeglądarki.

Czym jest „React Fiber”?

„Fiber” to nazwa nowego silnika rekoncyliującego (ang. *reconciliation engine*) w Reakcie 16. Jego głównym celem jest umożliwienie przyrostowego renderowania wirtualnego modelu DOM. [Dowiedz się więcej](#).

Stan komponentu

Co robi `setState`?

`setState()` tworzy plan aktualizacji obiektu `state` w komponencie. Gdy stan ulega zmianie, komponent reaguje poprzez ponowne renderowanie.

Jaka jest różnica między `state` a `props`?

Właściwości `props` (skrót od *properties*) i stan `state` są zwykłymi obiektami javascriptowymi. Przechowują informacje, które wpływają na wynik renderowania komponentu, jednak jest między nimi istotna różnica: właściwości `props` są przekazywane do komponentu (podobnie jak argumenty do funkcji), podczas gdy stan `state` jest zarządzany *wewnątrz* komponentu (podobnie jak zmienna w ciele funkcji).

Oto kilka sprawdzonych źródeł, z których dowiesz się więcej o tym, kiedy używać właściwości `props`, a kiedy stanu `state`:

- [Właściwości kontra stan](#)
- [ReactJS: Właściwości kontra stan](#)

Dlaczego `setState` błędnie ustawia wartość?

W Reakcie zarówno `this.props`, jak i `this.state` reprezentują *wyrenderowane* wartości, tzn. te, które aktualnie widzimy na ekranie.

Wywołania funkcji `setState` są asynchroniczne - nie spodziewaj się, że `this.state` będzie odzwierciedlać aktualny stan natychmiast po wywołaniu `setState`. Jeśli chcesz obliczyć nowe wartości na podstawie starych, zamiast obiektu przekaż funkcję aktualizującą (więcej o tym poniżej).

Przykład kodu, który *nie* zachowa się tak, jak byśmy się spodziewali:

```
incrementCount() {
  // Uwaga: To *nie* zadziała tak, jak myślisz.
  this.setState({count: this.state.count + 1});
}

handleSomething() {
  // Założmy, że `this.state.count` startuje z wartością 0.
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();
  // Kiedy React ponownie wyrenderuje ten komponent, wartość `this.state.count`
  // będzie 1, a nie 3.

  // Dzieje się tak dlatego, że powyższa funkcja `incrementCount()` odczytuje
  // wartość z `this.state.count`,
  // jednak React nie aktualizuje wartości `this.state.count`, dopóki nie nastąpi
  // ponowne wyrenderowanie.
  // Dlatego `incrementCount()` za każdym razem odczyta `this.state.count` jako 0,
  // a następnie zaplanuje zmianę na 1.
}
```

```
// Jak sobie z tym poradzić? Czytaj dalej!
}
```

Poniżej znajduje się rozwiązanie tego problemu.

Jak aktualizować stan wartościami, które zależą od aktualnego stanu?

Do `setState` zamiast obiektu przekaż funkcję, aby upewnić się, że do obliczeń użyta zostanie najbardziej aktualna wersja stanu (patrz niżej).

Czym różni się przekazanie do `setState` obiektu od przekazania funkcji?

Przekazana funkcja aktualizująca ma dostęp do aktualnej wersji stanu. Jako że wywołania `setState` są grupowane, ten sposób pozwoli ci na stworzenie sekwencji aktualizacji, która zamiast powodować konflikty, będzie operowała na kolejnych aktualnych wersjach stanu:

```
incrementCount() {
  this.setState((state) => {
    // Ważne: zamiast z `this.state` odczytuj wartość z argumentu `state`.
    return {count: state.count + 1}
  });
}

handleSomething() {
  // Założmy, że `this.state.count` startuje z wartością 0.
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();

  // Jeśli sprawdzisz teraz wartość `this.state.count`, nadal będzie równa 0.
  // Jednak gdy React ponownie wypisze komponent, zmieni się ona na 3.
}
```

Dowiedz się więcej na temat funkcji `setState`.

Kiedy `setState` działa asynchronicznie?

Obecnie `setState` działa asynchronicznie wewnętrz procedur obsługi zdarzeń.

Dzięki temu, na przykład, jeśli zarówno komponent-rodzic, jak i komponent-dziecko wywołają `setState` podczas zdarzenia kliknięcia, komponent-dziecko nie zostanie ponownie wyrenderowany dwukrotnie. Zamiast tego React uruchomi wszystkie te aktualizacje stanu na koniec przeglądarkowego zdarzenia. W większych aplikacjach korzystnie wpływa to na wydajność.

Jest to szczegół implementacyjny i staraj się nie polegać na nim bezpośrednio. W przyszłych wersjach React będzie domyślnie grupował aktualizacje w większej liczbie przypadków.

Dlaczego React nie aktualizuje `this.state` synchronicznie?

Jak wyjaśniliśmy w poprzedniej sekcji, React celowo "czeka", aż wszystkie komponenty wywołają `setState()` w swoich procedurach obsługi zdarzeń, zanim zacznie ponownie renderować drzewo komponentów. Dzięki temu unikamy niepotrzebnych ponownych renderowań, co korzystnie wpływa na wydajność aplikacji.

Mimo wszystko może dziwić cię, dlaczego React tak po prostu nie aktualizuje `this.state` natychmiastowo, bez ponownego renderowania.

Są ku temu dwa powody:

- Skutkowałoby to przerwaniem spójności między właściwościami a stanem, powodując bardzo trudne w zlokalizowaniu błędy.
- Uniemożliiłoby to nam zaimplementowanie opracowywanych przez nas nowych funkcjonalności.

Ten [komentarz na GitHubie](#) dużo bardziej zagłębia się w konkretne przykłady.

Czy powinno się używać bibliotek zarządzających stanem, jak Redux czy MobX?

[Być może.](#)

Dobrze jest jednak najpierw dobrze poznać Reacta, zanim zacznie się dodawać kolejne biblioteki do zestawu. W samym tylko Reakcie można napisać dość złożone aplikacje.

[Go to TOC](#)

Struktura plików

Czy istnieje zalecany sposób na strukturyzację projektów w React?

React nie sugeruje, w jaki sposób należy umieścić pliki w folderach. Istnieje jednak kilka popularnych podejść rozpowszechnionych w ekosystemie, które warto rozważyć.

Grupowanie plików według funkcjonalności lub ścieżek

Jednym z powszechnych sposobów strukturyzacji projektów jest umieszczenie plików CSS, JS i testów razem w folderach pogrupowanych według funkcjonalności lub ścieżki.

```
common/
  Avatar.js
  Avatar.css
  APIUtils.js
  APIUtils.test.js
feed/
  index.js
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
  FeedAPI.js
profile/
  index.js
  Profile.js
  ProfileHeader.js
  ProfileHeader.css
  ProfileAPI.js
```

Definicja „funkcjonalności” nie jest jednoznaczna i od ciebie zależy, jak szczegółowo określisz strukturę. Jeśli nie masz pomysłu na foldery najwyższego poziomu, możesz zapytać użytkowników, z jakich głównych części składa się produkt, aby potem użyć ich modelu mentalnego jako schematu.

Grupowanie plików według ich typu

Innym popularnym sposobem strukturyzacji projektów jest grupowanie podobnych plików, na przykład:

```
api/
  APIUtils.js
  APIUtils.test.js
  ProfileAPI.js
  UserAPI.js
components/
  Avatar.js
  Avatar.css
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
  Profile.js
  ProfileHeader.js
  ProfileHeader.css
```

Niektórzy idą o krok dalej i w zależności od roli komponentów w aplikacji, umieszczają je w różnych folderach. Na przykład [Atomic Design](#) to metodologia projektowania oparta na tej zasadzie. Pamiętaj, że często bardziej produktywne jest traktowanie takich metodologii jako pomocnych przykładów, a nie ścisłych zasad, których należy przestrzegać.

Unikaj zbytniego zagnieżdżania

Istnieje wiele problemów związanych z głębokim zagnieżdżaniem katalogów w projektach javascriptowych. Trudniej jest zapisać między nimi import względny lub zaktualizować ten import po przeniesieniu plików. O ile nie masz bardzo ważnego powodu do korzystania z głębokiej struktury folderów, rozważ ograniczenie się do maksymalnie trzech lub czterech zagnieżdżeń folderów w jednym projekcie. Oczywiście jest to tylko zalecenie i może nie być odpowiednie dla twojego projektu.

Nie myśl nad tym za dużo

Jeśli dopiero rozpoczynasz projekt, [nie poświęcaj więcej niż pięć minut](#) na wybór sposobu struktury plików. Wybierz dowolne z powyższych podejść (lub wymyśl własne) i zaczni pisać kod! Prawdopodobnie w trakcie pisania prawdziwego kodu zmienisz podejście jeszcze co najmniej kilka razy.

Jeśli jednak utkniesz, zaczni od przeniesienia wszystkich plików do jednego folderu. W końcu stanie się on na tyle duży, że zechcesz oddzielić niektóre pliki od reszty. Do tego czasu będziesz mieć wystarczającą wiedzę, aby stwierdzić, które pliki najczęściej edytujesz razem. Ogólnie rzecz biorąc, dobrym pomysłem jest trzymanie blisko siebie plików, które często się zmieniają. Zasada ta nazywa się „kolokacją”.

W miarę jak projekty stają się większe, w praktyce często wykorzystują połączenie obu powyższych podejść. Dlatego, zwłaszcza na początku, wybór „właściwego podejścia” nie ma większego znaczenia.

Style i CSS

Jak mogę dodać klasy CSS do komponentów?

Przekaż ciąg znaków używając atrybutu `className`:

```
render() {
  return <span className="menu navigation-menu">Menu</span>
}
```

Klasy CSS mogą być zależne od właściwości i stanu komponentów:

```
render() {
  let className = 'menu';
  if (this.props.isActive) {
    className += ' menu-active';
  }
  return <span className={className}>Menu</span>
}
```

Porada:

Jeśli bardzo często piszesz kod tego typu, pakiet `classnames` może ci pomóc uprościć swój kod.

Czy mogę użyć stylów wewnętrzliniowych?

Tak, zobacz [ten artykuł](#), aby dowiedzieć się więcej.

Czy style wewnętrzliniowe są złe?

Klasy CSS są generalnie bardziej wydajne niż style wewnętrzliniowe.

Czym jest w CSS-w-JS?

Termin "CSS-w-JS" odnosi do wzorca, w którym CSS jest zapisywany razem z kodem JavaScript, a nie w osobnych plikach.

Pamiętaj, że ta funkcjonalność nie stanowi części Reacta, tylko dostarczana jest przez biblioteki stron trzecich. React nie ma opinii na temat sposobu definiowania stylów; jeśli masz wątpliwości, dobrym punktem wyjścia może być definiowanie stylów w osobnym pliku `*.css` i odnoszenie się do nich używając atrybutu `className`.

Czy mogę robić animacje w Reakcie?

React może być wykorzystywany do napędzania animacji. Przykłady znajdziesz na stronach [React Transition Group](#), [React Motion](#) lub [React Spring](#).

react

[Go to TOC](#)

Versioning Policy

React follows [semantic versioning \(semver\)](#) principles.

That means that with a version number **x.y.z**:

- When releasing **critical bug fixes**, we make a **patch release** by changing the **z** number (ex: 15.6.2 to 15.6.3).
- When releasing **new features** or **non-critical fixes**, we make a **minor release** by changing the **y** number (ex: 15.6.2 to 15.7.0).
- When releasing **breaking changes**, we make a **major release** by changing the **x** number (ex: 15.6.2 to 16.0.0).

Major releases can also contain new features, and any release can include bug fixes.

Minor releases are the most common type of release.

This versioning policy does not apply to prerelease builds in the Next or Experimental channels. [Learn more about prereleases](#).

Breaking Changes

Breaking changes are inconvenient for everyone, so we try to minimize the number of major releases – for example, React 15 was released in April 2016 and React 16 was released in September 2017, and React 17 was released in October 2020.

Instead, we release new features in minor versions. That means that minor releases are often more interesting and compelling than majors, despite their unassuming name.

Commitment to Stability

As we change React over time, we try to minimize the effort required to take advantage of new features. When possible, we'll keep an older API working, even if that means putting it in a separate package. For example, [mixins have been discouraged for years](#) but they're supported to this day [via create-react-class](#) and many codebases continue to use them in stable, legacy code.

Over a million developers use React, collectively maintaining millions of components. The Facebook codebase alone has over 50,000 React components. That means we need to make it as easy as possible to upgrade to new versions of React; if we make large changes without a migration path, people will be stuck on old versions. We test these upgrade paths on Facebook itself – if our team of less than 10 people can update 50,000+ components alone, we hope the upgrade will be manageable for anyone using React. In many cases, we write [automated scripts](#) to upgrade component syntax, which we then include in the open-source release for everyone to use.

Gradual Upgrades via Warnings

Development builds of React include many helpful warnings. Whenever possible, we add warnings in preparation for future breaking changes. That way, if your app has no warnings on the latest release, it will be compatible with the next major release. This allows you to upgrade your apps one component at a time.

Development warnings won't affect the runtime behavior of your app. That way, you can feel confident that your app will behave the same way between the development and production builds -- the only differences are that the production build won't log the warnings and that it is more efficient. (If you ever notice otherwise, please file an issue.)

What Counts as a Breaking Change?

In general, we *don't* bump the major version number for changes to:

- **Development warnings.** Since these don't affect production behavior, we may add new warnings or modify existing warnings in between major versions. In fact, this is what allows us to reliably warn about upcoming breaking changes.
- **APIs starting with `unstable_`.** These are provided as experimental features whose APIs we are not yet confident in. By releasing these with an `unstable_` prefix, we can iterate faster and get to a stable API sooner.
- **Alpha and canary versions of React.** We provide alpha versions of React as a way to test new features early, but we need the flexibility to make changes based on what we learn in the alpha period. If you use these versions, note that APIs may change before the stable release.
- **Undocumented APIs and internal data structures.** If you access internal property names like `__SE-CRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED` or `__reactInternalInstance$uk43rzhitjg`, there is no warranty. You are on your own.

This policy is designed to be pragmatic: certainly, we don't want to cause headaches for you. If we bumped the major version for all of these changes, we would end up releasing more major versions and ultimately causing more versioning pain for the community. It would also mean that we can't make progress in improving React as fast as we'd like.

That said, if we expect that a change on this list will cause broad problems in the community, we will still do our best to provide a gradual migration path.

If a Minor Release Includes No New Features, Why Isn't It a Patch?

It's possible that a minor release will not include new features. [This is allowed by semver](#), which states "**[a minor version] MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes.**"

However, it does raise the question of why these releases aren't versioned as patches instead.

The answer is that any change to React (or other software) carries some risk of breaking in unexpected ways. Imagine a scenario where a patch release that fixes one bug accidentally introduces a different bug. This would not only be disruptive to developers, but also harm their confidence in future patch releases. It's especially regrettable if the original fix is for a bug that is rarely encountered in practice.

We have a pretty good track record for keeping React releases free of bugs, but patch releases have an even higher bar for reliability because most developers assume they can be adopted without adverse consequences.

For these reasons, we reserve patch releases only for the most critical bugs and security vulnerabilities.

If a release includes non-essential changes — such as internal refactors, changes to implementation details, performance improvements, or minor bugfixes — we will bump the minor version even when there are no new features.

Formularze

W Reakcie elementy formularza HTML działają trochę inaczej niż pozostałe elementy DOM. Wynika to stąd, że elementy formularza same utrzymują swój wewnętrzny stan. Dla przykładu przyjrzyjmy się zwykłemu formularzowi HTML z jedną wartością - imieniem:

```
<form>
  <label>
    Imię:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Wyślij" />
</form>
```

Powyższy formularz posiada domyślną funkcję automatycznego przekierowania przeglądarki do nowej strony po wysłaniu formularza przez użytkownika. Jeśli zależy ci na tej funkcjonalności, to działa ona również w Reakcie. Jednak w większości przypadków dobrze jest mieć funkcję javascriptową, która obsługuje wysyłanie formularza i ma dostęp do podanych przez użytkownika danych. Standardem stała się obsługa formularzy poprzez tzw. "komponenty kontrolowane".

Komponenty kontrolowane

W HTML-u, elementy formularza takie jak `<input>`, `<textarea>` i `<select>` najczęściej zachowują swój własny stan, który jest aktualizowany na podstawie danych wejściowych podawanych przez użytkownika. Natomiast w Reakcie zmienny stan komponentu jest zazwyczaj przechowywany we właściwości `state` (pol. *stan*) danego komponentu. Jest on aktualizowany jedynie za pomocą funkcji `setState()`.

Możliwe jest łączenie tych dwóch rozwiązań poprzez ustanowienie reactowego stanu jako "wyłącznego źródła prawdy". Wówczas reactowy komponent renderujący dany formularz kontroluje również to, co zachodzi wewnątrz niego podczas wypełniania pól przez użytkownika. Element `input` formularza, kontrolowany w ten sposób przez Reacta, nazywamy "komponentem kontrolowanym"

Gdybyśmy chcieli sprawić, aby podany wcześniej przykładowy formularz wyświetlał przy wysłaniu podane przez użytkownika imię, możemy zrobić z niego komponent kontrolowany w następujący sposób:

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('Podano następujące imię: ' + this.state.value);
  }
}
```

```

    event.preventDefault();
}

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

```

Przetestuj kod na CodePen

Dzięki ustawieniu atrybutu `value` na elemencie formularza, wyświetlane dane zawsze będą odpowiadały `this.state.value`. Tym samym reaktywy stan jest tutaj źródłem prawdy. Ponieważ zaś `handleChange` aktualizuje reaktywy stan przy każdym wciśnięciu klawisza, wyświetlane dane aktualizują się na bieżąco w miarę wpisywania ich przez użytkownika.

W komponentach kontrolowanych wartość elementu formularza zawsze zależy od stanu reaktywnego. Owszem, wymaga to napisania większej ilości kodu, jednak dzięki temu możliwe jest przekazanie wartości do innych elementów interfejsu albo nadpisanie jej wewnątrz procedur obsługi zdarzeń.

Znacznik `textarea` {#the-textarea-tag}

W HTML-u element `<textarea>` definiuje swój tekst poprzez elementy potomne:

```

<textarea>
  Cześć, oto przykład tekstu w polu tekstowym.
</textarea>

```

Natomiast w Reakcie `<textarea>` wykorzystuje w tym celu atrybut `value`. Dzięki temu kod formularza zawierającego `<textarea>` może być podobny do kodu formularza z jednoliniowym elementem `input`:

```

class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Proszę napisać wypracowanie o swoim ulubionym elemencie DOM'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('Wysłano następujące wypracowanie: ' + this.state.value);
  }
}

```

```

        event.preventDefault();
    }

    render() {
        return (
            <form onSubmit={this.handleSubmit}>
                <label>
                    Wypracowanie:
                    <textarea value={this.state.value} onChange={this.handleChange} />
                </label>
                <input type="submit" value="Wyślij" />
            </form>
        );
    }
}

```

Zwróć uwagę, że wartość `this.state.value` jest inicjalizowana w konstruktorze, tak aby pole tekstowe zawierało jakiś domyślny tekst.

Znacznik `select` {#the-select-tag}

W HTML-u element `<select>` tworzy rozwijaną listę. Dla przykładu poniższy kod HTML tworzy rozwijaną listę smaków:

```

<select>
    <option value="grejpfrutowy">Grejpfrutowy</option>
    <option value="limonkowy">Limonkowy</option>
    <option selected value="kokosowy">Kokosowy</option>
    <option value="mango">Mango</option>
</select>

```

Zwróć uwagę na atrybut `selected`, który sprawia, że opcją wybraną domyślnie jest opcja "Kokosowy". W Reakcie zamiast atrybutu `selected` używamy atrybutu `value` na głównym znaczniku `select`. W przypadku komponentów kontrolowanych jest to rozwiązanie bardziej dogodne, ponieważ wartość tego atrybutu aktualizowana jest tylko w jednym miejscu:

```

class FlavorForm extends React.Component {
    constructor(props) {
        super(props);
        this.state = {value: "kokosowy"};

        this.handleChange = this.handleChange.bind(this);
        this.handleSubmit = this.handleSubmit.bind(this);
    }

    handleChange(event) {
        this.setState({value: event.target.value});
    }

    handleSubmit(event) {
        alert('Twój ulubiony smak to: ' + this.state.value);
        event.preventDefault();
    }

    render() {
        return (
            <form onSubmit={this.handleSubmit}>

```

```

<label>
  Wybierz swój ulubiony smak:
  <select value={this.state.value} onChange={this.handleChange}>
    <option value="grejpfrutowy">Grejpfrutowy</option>
    <option value="limonkowy">Limonkowy</option>
    <option value="kokosowy">Kokosowy</option>
    <option value="mango">Mango</option>
  </select>
</label>
<input type="submit" value="Wyślij" />
</form>
);
}
}

```

Przetestuj kod na CodePen

Ogólnie elementy `<input type="text">`, `<textarea>`, i `<select>` działają podobnie. Wszystkie przyjmują atrybut `value`, który można wykorzystać w komponentach kontrolowanych.

Wskazówka

Wartością atrybutu `value` może być także tablica. Daje to możliwość wyboru spośród wielu opcji w znaczniku `select`:

```
<select multiple={true} value={['B', 'C']}>
```

Znacznik `input` dla plików

W HTML-u element `<input type="file">` pozwala użytkownikom wybrać jeden lub więcej plików z pamięci swojego urządzenia, które następnie mogą być wysłane do serwera lub przetworzone z użyciem kodu JavaScript poprzez [interfejs klasy `File`](#).

```
<input type="file" />
```

Ponieważ wartość tego elementu jest wartością przeznaczoną tylko do odczytu, w Reakcie jest to komponent **niekontrolowany**. Przedstawimy go wraz z innymi komponentami tego typu [w dalszej części dokumentacji](#).

Obsługa wielu elementów `input` {#handling-multiple-inputs}

Kiedy zachodzi potrzeba obsługi wielu kontrolowanych elementów `input`, do każdego elementu można dodać atrybut `name` oraz pozwolić funkcji obsługującej (ang. *handler function*) zadecydować o dalszych krokach w zależności od wartości atrybutu `event.target.name`.

Przyjrzymy się następującemu przykładowi:

```

class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };
    this.handleInputChange = this.handleInputChange.bind(this);
  }

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

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

  render() {
    return (
      <form>
        <label>
          Wybiera się:
          <input
            name="isGoing"
            type="checkbox"
            checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
        <br />
        <label>
          Liczba gości:
          <input
            name="numberOfGuests"
            type="number"
            value={this.state.numberOfGuests}
            onChange={this.handleInputChange} />
        </label>
      </form>
    );
  }
}

```

Przetestuj kod na CodePen

Zwróć uwagę na wykorzystaną przez nas składnię **obliczonych nazw właściwości** umówionej przez ES6. Pozwala ona na aktualizację klucza stanu odpowiadającego nazwie danego elementu `input`:

```

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

```

W składni ES5 wyglądałoby to następująco:

```

var partialState = {};
partialState[name] = value;
this.setState(partialState);

```

Ponadto ponieważ `setState()` automatycznie `scala` podany stan częściowy ze stanem aktualnym, funkcja ta wywoływana jest tylko z nowo dostarczonymi danymi.

Wartość `null` w kontrolowanym elemencie `input` "

Określenie właściwości (ang. *prop*) `value` komponentu kontrolowanego zapobiega niepożądanym zmianom danych wejściowych przez użytkownika. Jeśli określisz wartość dla `value`, a dane wejściowe w dalszym ciągu będzie można edytować, sprawdź, czy przez pomyłkę nie przekazujesz wartości `undefined` lub `null`.

Kod poniżej ilustruje ten problem. (Element `input` jest początkowo zablokowany, ale po krótkiej chwili jego zawartość można edytować.)

```
ReactDOM.createRoot(mountNode).render(<input value="Cześć" />);
setTimeout(function() {
  ReactDOM.createRoot(mountNode).render(<input value={null} />);
}, 1000);
```

Inne rozwiązania

Stosowanie kontrolowanych komponentów może być niekiedy uciążliwe, ponieważ wymaga nie tylko tworzenia funkcji obsługujących każdą możliwą zmianę twoich danych, lecz także przekazywania stanu elementu `input` poprzez komponent reactowy. To z kolei może się stać wyjątkowo irytującym doświadczeniem, zwłaszcza gdy konwertujesz istniejący już kod na kod reactowy lub kiedy integrujesz aplikację reactową z biblioteką nie-reactową. W tych sytuacjach warto abyś przyjrzał się `komponentom niekontrolowanym`, które stanowią alternatywną technikę stosowania formularzy.

Rozwiązania całościowe

Jeśli szukasz rozwiązania kompleksowego umożliwiającego walidację, śledzenie odwiedzonych pól oraz obsługi wysyłania danych, często wybieraną opcją jest `biblioteka Formik`. Rozwiązanie to bazuje jednak na tych samych zasadach co komponenty kontrolowane i zarządzanie stanem. Dlatego bardzo ważne jest, abyś przyswoił sobie te zasady.

Przekazywanie referencji

Przekazywanie referencji (ang. *ref forwarding*) to technika, w której [referencje](#) do komponentu "podajemy dalej" do jego dziecka. Dla większości komponentów w aplikacji nie jest to potrzebne, jednak może okazać się przydatne w niektórych przypadkach, zwłaszcza w bibliotekach udostępniających uniwersalne komponenty. Najczęstsze scenariusze opisujemy poniżej.

Przekazywanie referencji do komponentów DOM

Rozważmy komponent `FancyButton`, który renderuje natywny element DOM - przycisk: `embed:forwarding-refs/fancy-button-simple.js`

Komponenty reactowe ukrywają szczegóły swojej implementacji, w tym także wyrenderowany HTML. Inne komponenty używające `FancyButton` **z reguły nie potrzebują mieć dostępu do referencji** do wewnętrznego elementu `button`. Jest to korzystne, gdyż zapobiega sytuacji, w której komponenty są za bardzo uzależnione od struktury drzewa DOM innych komponentów.

Taka enkapsulacja jest pożądana na poziomie aplikacji, w komponentach takich jak `FeedStory` czy `Comment`. Natomiast może się okazać to niewygodne w przypadku komponentów wielokrotnego użytku, będących "liścimi" drzewa. Np. `FancyButton` albo `MyTextInput`. Takie komponenty często używane są w wielu miejscach aplikacji, w podobny sposób jak zwyczajne elementy DOM typu `button` i `input`. W związku z tym, bezpośredni dostęp do ich DOM może okazać się konieczny, aby obsługiwać np. fokus, zaznaczenie czy animacje.

Przekazywanie referencji jest opcjonalną funkcjonalnością, która pozwala komponentom wziąć przekazaną do nich referencję i "podać ją dalej" do swojego dziecka.

W poniższym przykładzie `FancyButton` używa `React.forwardRef`, by przejąć przekazaną do niego referencję i przekazać ją dalej do elementu `button`, który renderuje:

`embed:forwarding-refs/fancy-button-simple-ref.js`

Tym sposobem komponenty używające `FancyButton` mają referencję do elementu `button` znajdującego się wewnętrz. Mogą więc, w razie potrzeby, operować na komponencie tak, jakby operowały bezpośrednio na natywnym elemencie DOM.

Oto wyjaśnienie krok po kroku, opisujące, co wydarzyło się w przykładzie powyżej:

1. Tworzymy [referencję reactową](#) wywołując `React.createRef` i przypisujemy ją do stałej `ref`.
2. Przekazujemy `ref` do `<FancyButton ref={ref}>` przypisując ją do atrybutu JSX.
3. Wewnętrz `forwardRef` React przekazuje `ref` do funkcji `(props, ref) => ...` jako drugi argument.
4. Podajemy argument `ref` dalej do `<button ref={ref}>` przypisując go do atrybutu JSX.
5. Gdy referencja zostanie zamontowana, `ref.current` będzie wskazywać na element DOM `<button>`.

Uwaga

Drugi argument `ref` istnieje tylko, gdy definiujesz komponent przy pomocy wywołania `React.forwardRef`. Zwyczajna funkcja lub klasa nie dostanie argumentu `ref`, nawet jako jednej z właściwości (`props`).

Przekazywanie referencji nie jest ograniczone do elementów drzewa DOM. Możesz także przekazywać referencje do instancji komponentów klasowych.

Uwaga dla autorów bibliotek komponentów

Kiedy zaczniesz używać `forwardRef` w swojej bibliotece komponentów, potraktuj to jako zmianę krytyczną (ang. *breaking change*). W efekcie biblioteka powinna zostać wydana w nowej "wersji głównej" (ang. *major version, major release*). Należy tak postąpić, ponieważ najprawdopodobniej Twoja biblioteka zauważalnie zmieniła zachowanie (np. inaczej przypinając referencje i eksportując inne typy). Może to popsuć działanie aplikacji, które są zależne od dawnego zachowania.

Stosowanie `React.forwardRef` warunkowo, gdy ono istnieje, także nie jest zalecane z tego samego powodu: zmienia to zachowanie biblioteki i może zepsuć działanie aplikacji użytkowników, gdy zmienią wersję Reacta.

Przekazywanie referencji w komponentach wyższego rzędu

Omawiana technika może okazać się wyjątkowo przydatna w **komponentach wyższego rzędu** (KWR; ang. *Higher-Order Components* lub *HOC*). Zaczniemy od przykładu KWR-a, który wypisuje w konsoli wszystkie właściwości komponentu:

```
embed:forwarding-refs/log-props-before.js
```

KWR `logProps` przekazuje wszystkie atrybuty do komponentu, który opakowuje, więc wyrenderowany wynik będzie taki sam. Na przykład, możemy użyć tego KWRa do logowania atrybutów, które zostaną przekazane do naszego komponentu `FancyButton` :

Powyższe rozwiązanie ma jeden minus: referencje nie zostaną przekazane do komponentu. Dzieje się tak, ponieważ `ref` nie jest atrybutem. Tak jak `key`, jest on obsługiwany przez Reacta w inny sposób. Referencja będzie w tym wypadku odnosiła się do najbardziej zewnętrznego kontenera, a nie do opakowanego komponentu.

Oznacza to, że referencje przeznaczone dla naszego komponentu `FancyButton` będą w praktyce wskazywać na komponent `LogProps`.

```
embed:forwarding-refs/fancy-button-ref.js
```

Na szczęście możemy jawnie przekazać referencję do wewnętrznego komponentu `FancyButton` używając API `React.forwardRef`. `React.forwardRef` przyjmuje funkcję renderującą, która otrzymuje parametry `props` oraz `ref`, a zwraca element reactowy. Na przykład: `embed:forwarding-refs/log-props-after.js`

Wyświetlanie własnej nazwy w narzędziach deweloperskich

`React.forwardRef` przyjmuje funkcję renderującą. Narzędzia deweloperskie Reacta (ang. *React DevTools*) używają tej funkcji do określenia, jak wyświetlać komponent, który przekazuje referencję.

Przykładowo, następujący komponent w narzędziach deweloperskich wyświetli się jako "*ForwardRef*":

`embed:forwarding-refs/wrapped-component.js`

Jeśli nazwiesz funkcję renderującą, narzędzia deweloperskie uwzględnią tę nazwę (np. "*ForwardRef(myFunction)*"):

`embed:forwarding-refs/wrapped-component-with-function-name.js`

Możesz nawet ustawić właściwość `displayName` funkcji tak, aby uwzględniała nazwę opakowanego komponentu:

`embed:forwarding-refs/customized-display-name.js`

Fragmenty

Często spotykanym wzorcem w Reakcie jest tworzenie komponentów, które zwracają wiele elementów. Fragmenty pozwalają zgrupować listę potomków bez konieczności dodawania zbędnych węzłów do drzewa DOM.

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  );
}
```

Istnieje również nowy [skrócony zapis](#) do deklarowania fragmentów.

Motywacja

Zdarza się, że potrzebujemy w komponencie zwrócić listę potomków. Rozważmy poniższy przykład:

```
class Table extends React.Component {
  render() {
    return (
      <table>
        <tr>
          <Columns />
        </tr>
      </table>
    );
  }
}
```

Aby zapewnić poprawność wyrenderowanego kodu HTML, komponent `<Columns />` powinien zwrócić kilka elementów `<td>`. Gdyby komórki tabeli, zwarcane przez funkcję `render()` komponentu `<Columns />`, otoczyć np. elementem `<div>`, powstały w ten sposób kod HTML byłby nieprawidłowy.

```
class Columns extends React.Component {
  render() {
    return (
      <div>
        <td>Witaj</td>
        <td>Świecie</td>
      </div>
    );
  }
}
```

daje w rezultacie następującą strukturę dla komponentu `<Table />`:

```
<table>
  <tr>
    <div>
      <td>Witaj</td>
      <td>Świecie</td>
    </div>
  </tr>
</table>
```

Fragmenty rozwiązują ten problem.

Użycie

```
class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Witaj</td>
        <td>Świecie</td>
      </React.Fragment>
    );
  }
}
```

daje w rezultacie następującą strukturę dla komponentu `<Table>`:

```
<table>
  <tr>
    <td>Witaj</td>
    <td>Świecie</td>
  </tr>
</table>
```

Skrócony zapis

Istnieje nowy, krótszy zapis służący do deklarowania fragmentów. Z wyglądu przypomina puste znaczniki:

```
class Columns extends React.Component {
  render() {
    return (
      <>
        <td>Witaj</td>
        <td>Świecie</td>
      </>
    );
  }
}
```

Możesz używać `<></>` tak samo jak innych komponentów. Nie możesz jednak przekazywać do niego klucza (`key`) ani żadnych innych właściwości.

Fragmenty zadeklarowane jawnie przy użyciu składni `<React.Fragment>` mogą posiadać klucze. Ma to zastosowanie, gdy zechcesz przemapować kolekcję na tablicę fragmentów -- na przykład do stworzenia listy opisów:

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Bez `key` React wyrzuci błąd
        <React.Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </React.Fragment>
      ))}
    </dl>
  );
}
```

`key` jest jedyną właściwością, którą można przekazać do fragmentu. Możliwe, że w przyszłości dodamy wsparcie dla innych właściwości, np. procedur obsługi zdarzeń.

Demo

Możesz wypróbować nową składnię JSX dla fragmentów na tym [CodePenie](#).

Podstawowe informacje

- "download.html"
- "downloads.html"

Ta podstrona to przegląd dokumentacji Reacta oraz powiązanych zasobów.

React to javascriptowa biblioteka służąca do budowania interfejsów użytkownika. Dowiedz się, czym tak naprawdę jest React, na naszej [stronie głównej](#) lub [w tutorialu](#).

- [Wypróbuj Reacta](#)
- [Ucz się Reacta](#)
- [Bądź na bieżąco](#)
- [Dokumentacje wersji](#)
- [Czegoś brakuje?](#)

Wypróbuj Reacta

Biblioteka React została zaprojektowana z myślą o stopniowym wdrażaniu. **Dzięki temu zawarte w niej rozwiązania możesz stosować wybiórczo w zależności od potrzeb.** Linki zawarte w tej części przewodnika wprowadzą cię do Reacta. Znajdziesz tu materiały, które pozwolą ci nie tylko dodawać elementy interaktywne do prostych stron HTML, lecz także rozpocząć prace nad bardziej zaawansowanymi aplikacjami bazującymi na Reakcie.

Środowisko doświadczalne

Jeśli chcesz wypróbować Reacta, możesz w tym celu użyć środowisk doświadczalnych online. Wypróbuj szablon "Hello World" na [CodePen](#), [CodeSandbox](#) lub [Stackblitz](#)

Jeśli wolisz używać swojego własnego edytora tekstu, możesz także [ściągnąć ten plik HTML](#), edytować go, a następnie otworzyć lokalnie w swojej przeglądarce. Pamiętaj jednak, że wykonuje on podczas uruchomienia dość mozolną transformację kodu, więc zalecamy go jedynie do prostych przykładów.

Dodaj Reacta do swojej strony

Możesz dodać Reacta do swojej strony HTML w ciągu minuty. Możesz stopniowo zwiększać jego obecność według potrzeb lub użyć go jedynie do kilku dynamicznych widgetów.

Stwórz nową aplikację reactową

Gdy zaczynasz projekt w Reakcie, najlepiej zacząć od [napisania prostej strony w HTML-u przy użyciu tagów <script>](#). Zajmie ci to tylko minutę!

W miarę jak będziesz rozwijać swoją aplikację, możesz zacząć myśleć o bardziej zintegrowanym środowisku. Istnieje kilka JavaScriptowych narzędzi, które polecamy do rozwoju większych aplikacji. Każde z nich, przy drobnej lub znikomej konfiguracji, pozwoli ci korzystać z pełni mocy systemu Reacta. [Kliknij tutaj, aby dowiedzieć się, jak to zrobić.](#)

Ucz się Reacta

Ludzie rozpoczynający przygodę z Reactem wywodzą się z najróżniejszych środowisk, mają też różne sposoby uczenia się. Mamy nadzieję, że ta część przewodnika okaże się pomocna zarówno dla osób o zacięciu teoretycznym jak i dla tych którzy mają podejście bardziej praktyczne.

- Jeśli wolisz **naukę przez praktykę**, zacznij od naszego [przewodnika praktycznego](#).
- Jeśli zaś bardziej zależy ci na systematycznym opanowaniu **podstawowych zagadnień**, zacznij swoją przygodę od [przewodnika po głównych zagadnieniach](#).

Jak każda nowa technologia, z którą spotykasz się po raz pierwszy, React także ma swoją krzywą uczenia się. Jednak przy odrobinie praktyki i cierpliwości *zrozumiesz*, na czym polega.

Podstawowe przykłady

[Strona główna](#) Reacta zawiera kilka małych przykładów z edytorem kodu live. Nawet jeśli nie wiesz jeszcze nic o Reakcie, spróbuj zmienić kod i zobaczyć, jak wpłynie to na wynik działania aplikacji.

React dla początkujących

Jeśli wydaje ci się, że dokumentacja Reacta prowadzi cię w zbyt szybkim tempie, sprawdź [ten przegląd Reacta napisany przez Tanię Rascię](#). Wprowadzi cię on szczegółowo w najistotniejsze zagadnienia Reacta, zachowując styl zrozumiałły dla początkujących. Gdy już skończysz, spróbuj ponownie przeczytać naszą dokumentację!

React dla designerów

Jeśli zaczynając z Reactem przychodzisz ze środowiska grafików i designerów, [te źródła](#) mogą okazać się świetnym punktem startowym.

Źródła JavaScriptowe

Dokumentacja Reacta zakłada znajomość języka programowania JavaScript na pewnym poziomie. Nie musisz być ekspertem, jednakże nauka zarówno Reacta jak i JavaScriptu w tym samym czasie może być nieco trudniejsza.

Proponujemy zapoznanie się z [przeglądem informacji o JavaScriptie](#), by sprawdzić swoją wiedzę. Powinno ci to zająć około 30 minut do godziny, a sprawi, że nauka Reacta będzie bardziej komfortowa.

Podpowiedź

Gdy tylko poczujesz, że pojęcia z języka JavaScript są zbyt zagmatwane, [MDN](#) oraz [javascript.info](#) mogą okazać się świetnymi źródłami do nauki. Istnieją także [fora internetowe](#), na których możesz poprosić o pomoc.

Praktyczne samouczki

Jeśli preferujesz **naukę przez praktykę**, sprawdź nasze [praktyczne samouczki](#). W tym rozdziale, za pomocą Reacta, stworzymy od podstaw grę w kółko i krzyżyk. Jeśli nie zajmujesz się tworzeniem gier i kusi cię, by go pominąć -- najpierw daj mu szansę. Techniki, których nauczysz się w tym poradniku są fundamentami do zbudowania *dowolnej* aplikacji w Reakcie, a zrozumienie ich zapewni ci dogłębne poznanie tematu.

Instrukcja krok po kroku

Jeśli wolisz **naukę pojęć krok po kroku**, zacznij od [przewodnika po głównych zagadnieniach](#). Każdy kolejny rozdział jest tu oparty o wiedzę nabytą w poprzednim, więc radzimy niczego nie pomijać.

Myślenie po reactowemu

Wielu użytkownikom Reacta bardzo pomogła lektura rozdziału pt. [Myślenie po reactowemu](#), bo dzięki niemu coś im w końcu "zaskoczyło" w głowie. To prawdopodobnie najstarszy przewodnik po Reakcie, jednakże jest on wciąż aktualny.

Rekomendowane kursy

Czasem ludzie preferują książki i kursy wideo stworzone przez innych użytkowników, ponieważ są one dla nich bardziej pomocne. Dla takich osób stworzyliśmy [listę rekomendowanych źródeł](#), z których część jest darmowa.

Zaawansowane pojęcia

Kiedy będziesz się już czuć komfortowo z [podstawowymi pojęciami](#), a za sobą będziesz mieć trochę zabawy z kodem w Reakcie, mogą zainteresować cię bardziej zaawansowane zagadnienia. Ta sekcja zapozna cię z przydatnymi, acz rzadziej używanymi funkcjonalnościami, jak [konteksty \(ang. contexts\)](#) czy [referencje \(ang. refs\)](#).

Dokumentacja API

Ta sekcja dokumentacji może okazać się przydatna, jeśli chcesz dowiedzieć się więcej o API Reacta. Na przykład, [dokumentacja klasy React.Component](#) pozwoli ci w szczegółach zrozumieć działanie `setState()` oraz jak działają metody z cyklu życia komponentu.

Słownik pojęć i FAQ

[Słownik pojęć](#) zawiera przegląd najczęściej używanych terminów, które napotkasz w dokumentacji Reacta. Istnieje także sekcja FAQ, poświęcona krótkim pytaniami i odpowiedziom na najpopularniejsze tematy, między innymi "[Jak wykonać zapytanie AJAX-owe?](#)", "[Co to jest stan komponentu?](#)" oraz "[Jak powinna wyglądać struktura plików?](#)".

Bądź na bieżąco

[Blog Reacta](#) jest oficjalnym źródłem aktualizacji i newsów od zespołu Reacta. Wszystkie istotne informacje, w tym notki dotyczące nowych wydań oraz informacje o zdezaktualizowanych elementach, znajdują się tam jako pierwsze.

Możesz obserwować [konto @reactjs](#) na Twitterze, jednak nie ominie cię nic ważnego, jeśli będziesz śledzić jedynie bloga.

Nie każde nowe wydanie Reacta wymaga osobnego posta na blogu, jednak informacje o każdym wydaniu zawsze trafiają do pliku [CHANGELOG.md](#) w repozytorium Reacta, a także na [liście opublikowanych wersji](#).

Dokumentacje wersji

Ta dokumentacja zawsze odzwierciedla najnowszą stabilną wersję Reacta. Począwszy od Reacta w wersji 16, starsze wersje dokumentacji publikowane są na [osobnej podstronie](#). Pamiętaj jednak, że dokumentacje do starszych wersji nie są już aktualizowane.

Czegoś brakuje?

Jeśli brakuje ci czegoś w dokumentacji lub któraś część nie jest wystarczająco jasna, proszę [zgłoś to w repozytorium dokumentacji](#) (w wiadomości zamieszczając swoje sugestie) bądź napisz tweeta do [@reactjs](#). Chętnie dowiemy się, co myślisz!

Obsługa zdarzeń

Obsługa zdarzeń w Reakcie jest bardzo podobna do tej z drzewa DOM. Istnieje jednak kilka różnic w składni:

- Zdarzenia reactowe pisane są camelCazem, a nie małymi literami.
- W JSX procedura obsługi zdarzenia przekazywana jest jako funkcja, a nie łańcuch znaków.

Na przykład, poniższy kod HTML:

```
<button onclick="activateLasers()">
  Aktywuj lasery
</button>
```

w Reakcie wygląda nieco inaczej::

```
<button onClick={activateLasers}>
  Aktywuj lasery
</button>
```

Kolejna różnica polega na tym, że w Reakcie nie można zwrócić `false` w celu zapobiegnięcia wykonania domyślnej akcji. Należy jawnie wywołać `preventDefault`. Na przykład, w czystym HTML-u, aby zapobiec domyślnej akcji formularza (wysłaniu danych), można napisać:

```
<form onsubmit="console.log('Kliknięto na przycisk Wyślij.'); return false">
  <button type="submit">Wyślij</button>
</form>
```

W Reakcie, zamiast tego należy napisać:

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('Kliknięto na przycisk Wyślij.');
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Wyślij</button>
    </form>
  );
}
```

Zmienna `e` to zdarzenie syntetyczne (ang. *synthetic event*). React tworzy zdarzenia tego typu zgodnie ze [specyfikacją W3C](#), dzięki czemu nie trzeba martwić się o kompatybilność z przeglądarkami. Zdarzenia reactowe nie działają dokładnie tak samo jak te natywne. Po więcej informacji sięgnij do specyfikacji obiektu `SyntheticEvent`.

W kodzie reaktywym nie ma potrzeby dodawania obserwatora zdarzenia (ang. *event listener*) do elementu DOM po jego utworzeniu, poprzez wywoływanie funkcji `addEventListener`. Zamiast tego, wystarczy przekazać go podczas pierwszego renderowania komponentu.

Gdy komponent definiowany jest przy użyciu [klasy ze standardu ES6](#), często definiuje się procedurę obsługi zdarzenia jako metodę tej klasy. Na przykład, poniższy komponent `Toggle` wyświetli przycisk, który pozwala użytkownikowi przełączać się między stanami "WŁĄCZONY" i "WYŁĄCZONY":

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

    // Poniższe wiązanie jest niezbędne do prawidłowego przekazania `this` przy
    // wywołaniu funkcji
    this.handleClick = this.handleClick.bind(this);
  }

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

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

[Przetestuj kod na CodePen](#)

Należy zwrócić szczególną uwagę na znaczenie `this` funkcjach zwrotnych (ang. *callbacks*) używanych w JSX. W JavaScripte metody klasy nie są domyślnie [dowiązane](#) do instancji. Jeśli zapomnisz dowiązać metodę `this.handleClick` i przekażesz ją jako atrybut `onClick`, to `this` przy wywołaniu będzie miało wartość `undefined`.

To zachowanie nie jest specyficzne dla Reacta; [tak właśnie działają funkcje w JavaScripte](#). Generalnie, jeśli odwołujesz się do metody bez `()` po nazwie, jak na przykład `onClick={this.handleClick}`, pamiętaj, aby zawsze dowiązywać ją do instancji.

Jeśli denerwuje Cię ciągłe wywoływanie `bind`, istnieją dwa sposoby na obejście tego problemu. Możesz skorzystać ze [składni publicznych pól klasy](#), aby poprawnie dowiązać metody do instancji:

```
class LoggingButton extends React.Component {
  // Poniższy kod wymusza dowiązanie `this` wewnętrz handleClick.
  handleClick = () => {
    console.log('this ma wartość:', this);
  }

  render() {
    return (

```

```

        <button onClick={this.handleClick}>
          Kliknij mnie
        </button>
      );
    }
}

```

Powyższa składnia jest domyślnie włączona w [Create React App](#).

Jeśli nie chcesz używać tej składni, możesz skorzystać z [funkcji strzałkowej](#) (ang. *arrow function*):

```

class LoggingButton extends React.Component {
  handleClick() {
    console.log('this ma wartość:', this);
  }

  render() {
    // Poniższy kod wymusza dowiązanie `this` wewnętrz handleClick.
    return (
      <button onClick={() => this.handleClick()}>
        Kliknij mnie
      </button>
    );
  }
}

```

Problem z taką składnią polega na tym, że za każdym razem, gdy `LoggingButton` jest renderowany, tworzona jest nowa funkcja. W większości przypadków nie ma to większego znaczenia. Jeśli jednak będzie przekazywana do komponentów osadzonych głębiej w strukturze, będzie niepotrzebnie powodowała ich ponowne renderowanie. Zalecamy więc korzystanie ze składni pól klasy lub wiązanie metod w konstruktorze, aby uniknąć tego typu problemów z wydajnością.

Przekazywanie argumentów do procedur obsługi zdarzeń

Dość często, na przykład w pętli, potrzebujemy przekazać do procedury obsługi zdarzenia dodatkowy parametr. Na przykład, jeśli zmienna `id` zawierałaby identyfikator wiersza w tabeli, można by rozwiązać to na dwa sposoby:

```

<button onClick={(e) => this.deleteRow(id, e)}>Usuń wiersz</button>
<button onClick={this.deleteRow.bind(this, id)}>Usuń wiersz</button>

```

Obydwie linie robią to samo, przy użyciu, odpowiednio, [funkcji strzałkowej](#) oraz `Function.prototype.-bind`.

W obydwóch przypadkach argument `e`, reprezentujący zdarzenie reactowe, zostanie przekazany jako drugi w kolejności, zaraz po identyfikatorze wiersza. W przypadku funkcji strzałkowej, musimy przekazać go jawnie, natomiast w `bind` kolejne argumenty są przekazywane do funkcji automatycznie.

[Go to TOC](#)

Witaj, świecie!

Poniżej znajdziesz przykład najprostszej aplikacji reactowej:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<h1>Witaj, świecie!</h1>);
```

Wyświetli ona na stronie nagłówek "Witaj, świecie".

[Przetestuj kod na CodePen](#)

Kliknij w powyższy link, aby otworzyć edytor kodu w przeglądarce. Możesz śmiało dokonywać w nim zmian, zobacysz też jak wpływają one na to, co jest wyświetlane na stronie. Na większości stron w tym kursie znajdziesz przykłady, które można modyfikować.

Jak używać tego przewodnika

W niniejszym kursie zapoznamy się z podstawowymi składnikami aplikacji reactowej: elementami i komponentami. Korzystając z nich, możesz tworzyć rozbudowane aplikacje w oparciu o mniejsze części, dające się wielokrotnie wykorzystywać.

Wskazówka

Kurs ten skierowany jest do osób, które wolą się uczyć nowych koncepcji **krok po kroku**. Jeśli jednak wolisz zdobywać wiedzę poprzez ćwiczenia, możesz skorzystać z [kursu praktycznego](#). Obydwa kursy świetnie się uzupełniają.

Masz przed sobą pierwszy rozdział kursu teoretycznego dotyczącego głównych koncepcji Reacta. Lista dostępnych rozdziałów znajduje się w bocznym pasku nawigacji. Jeśli czytasz ten kurs na urządzeniu mobilnym, możesz przemieszczać się po stronie wciskając przycisk w prawym dolnym rogu ekranu.

Każdy rozdział w tym kursie uzupełnia wiedzę w oparciu o poprzednie rozdziały. **Wielu przypadków użycia Reacta możesz się nauczyć czytając rozdziały w sekcji "Główne idee", zapoznając się z nimi w kolejności, w jakiej pojawiają się w pasku nawigacji.** Przykładowo, rozdział pt. ["Wprowadzenie do JSX"](#) jest następnym w kolejności po tym, który aktualnie czytasz.

Wymagania początkowe

React jest biblioteką javascriptową, stąd też zakładamy, że czytelnik dysponuje pewną podstawową wiedzą o tym języku. **Jeśli czujesz się niezbyt pewnie, polecamy zapoznanie się z kursem "Ponowne wprowadzenie do JavaScriptu", który pozwoli ci ocenić swoją wiedzę** oraz będzie stanowić bazę do sprawnego poruszania się po kursie Reacta. Choć zapoznanie się z tamtym kursem może zajść ci około 30-60 minut, to nie będziesz mieć poczucia, że uczysz się jednocześnie JavaScriptu i Reacta.

Zauważ, że...

W naszym poradniku mogą sporadycznie pojawiać się przykłady oparte o nowszą składnię JavaScriptu. Jeśli od dłuższego czasu nie śledzisz rozwoju języka, [te trzy punkty \(ang.\)](#) mogą okazać się pomocne.

Zaczynajmy!

Kontynuuj przewijanie strony, aby dotrzeć do linku do [następnego rozdziału](#). Znajduje się on tuż przed stopką strony.

[Go to TOC](#)

Komponenty wyższego rzędu

Komponent wyższego rzędu (ang. *Higher-Order Component*), w skrócie KWR (ang. *HOC*), to zaawansowana technika reactowa stosowana w celu wielokrotnego używania logiki komponentu. KWR-y nie są częścią API Reacta *per se*. Są wzorcem, który wyłonił się z kompozycyjnej natury Reacta.

Konkretnie rzecz ujmując, **komponent wyższego rzędu jest funkcją, która przyjmuje jako argument inny komponent i zwraca nowy komponent**.

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

Tak jak zwykły komponent przekształca właściwości (ang. *props*) na fragment UI, tak komponent wyższego rzędu przekształca komponent w inny komponent.

KWR-y pojawiają się często w zewnętrznych bibliotekach reactowych, np. `connect` w Reduksie czy `createFragmentContainer` w Relayu.

W tym artykule wyjaśnimy, dlaczego komponenty wyższego rzędu są użyteczne oraz jak napisać własny.

Używaj KWR-ów do problemów przekrojowych

Uwaga

Dawniej do rozwiązywania problemów przekrojowych sugerowaliśmy korzystanie z mixinów. Zdaliśmy sobie jednak sprawę, iż wprowadzają one więcej zamieszania niż pozytku. [Przeczytaj ten artykuł](#), jeśli chcesz dowiedzieć się, dlaczego odeszliśmy od tego wzorca i w jaki sposób dostosować swoje istniejące komponenty.

Komponenty to podstawowa jednostka wielokrotnie używalnego kodu reactowego. Jednak niektóre wzorce nie pasują idealnie do tradycyjnego zastosowania komponentów.

Dla przykładu, powiedzmy, że mamy komponent `CommentList`, który subskrybuje się do zewnętrznego źródła danych i renderuje listę komentarzy:

```
class CommentList extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      // "DataSource" jest jakimś globalnym źródłem danych
      comments: DataSource.getComments()
    };
  }

  componentDidMount() {
    // Zasubskrybij się na zmiany
    DataSource.addChangeListener(this.handleChange);
  }
}
```

```

}

componentWillUnmount() {
  // Usuń subskrypcję
  DataSource.removeChangeListener(this.handleChange);
}

handleChange() {
  // Zaktualizuj stan komponentu przy każdej zmianie danych źródłowych
  this.setState({
    comments: DataSource.getComments()
  });
}

render() {
  return (
    <div>
      {this.state.comments.map((comment) => (
        <Comment comment={comment.text} key={comment.id} />
      ))}
    </div>
  );
}
}

```

Później możesz chcieć napisać komponent subskrybujący się na pojedynczy wpis na blogu, w którym zastosujesz podobny wzorzec:

```

class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      blogPost: DataSource.getBlogPost(props.id)
    };
  }

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

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

  handleChange() {
    this.setState({
      blogPost: DataSource.getBlogPost(this.props.id)
    });
  }

  render() {
    return <TextBlock text={this.state.blogPost} />;
  }
}

```

Komponenty `CommentList` i `BlogPost` nie są identyczne — wywołują bowiem inne metody `DataSource` i renderują inny fragment interfejsu. Jednak spora część ich implementacji jest taka sama:

- Po zamontowaniu komponentu subskrybują się w `DataSource`.
- Wewnątrz funkcji nasłuchującej wywołują `setState` przy każdej zmianie danych źródłowych.
- Po odmontowaniu komponentu usuwają subskrypcję.

Można sobie wyobrazić, że w większej aplikacji co rusz będziemy pisać podobny kod, który subskrybuje się w `DataSource` i wywołuje `setState`. Chcielibyśmy zbudować warstwę abstrakcji, która pozwoliłaby nam zdefiniować tę logikę w jednym miejscu i współdzielić ją w wielu komponentach. Tu do akcji wkraczają komponenty wyższego rzędu.

Możemy napisać funkcję, tworzącą komponenty takie jak `CommentList` czy `BlogPost`, która subskrybuje się w `DataSource`. Funkcja ta jako jeden z argumentów będzie przyjmować komponent potomny, który otrzyma zasubskrybowane dane poprzez określona właściwość (ang. *prop*). Nazwijmy tę funkcję `withSubscription`:

```
const CommentListWithSubscription = withSubscription(
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id)
);
```

Pierwszy z argumentów to opakowywany komponent. Drugi, po otrzymaniu `DataSource` i aktualnych właściwości komponentu, wyciąga interesujące nas dane ze źródła.

Gdy `CommentListWithSubscription` i `BlogPostWithSubscription` zostaną wyrenderowane, do `CommentList` i `BlogPost` trafi właściwość `data`, zawierająca aktualne dane ze źródła `DataSource`:

```
// Ta funkcja przyjmuje jako argument pewien komponent...
function withSubscription(WrappedComponent, selectData) {
  // ...i zwraca inny komponent...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(DataSource, props)
      };
    }

    componentDidMount() {
      // ... który zajmuje się podpięciem subskrypcji...
      DataSource.addChangeListener(this.handleChange);
    }

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

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

```

    }

    render() {
      // ... i renderuje opakowywany komponent z aktualnymi danymi!
      // Zauważ, że dodatkowo przekazujemy tu też pozostałe właściwości
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };
}

```

Zwróć uwagę, że KWR nie modyfikuje przekazanego mu komponentu ani nie stosuje dziedziczenia w celu skopiowania jego zachowania. Zamiast tego *wkomponowuje* przekazany komponent poprzez jego *opakowanie* w kontener. KWR jest zatem czystą funkcją (ang. *pure function*), nie mającą żadnych efektów ubocznych.

I to by było na tyle! Opakowany komponent otrzyma wszystkie właściwości kontenera, a dodatkowo `data`, używaną do wyrenderowania interfejsu. Dla KWR-a nie ma znaczenia, w jaki sposób wykorzystywane są dane, a z kolei opakowywany komponent nie przejmuje się tym, skąd te dane pochodzą.

Z racji tego, że `withSubscription` jest zwykłą funkcją, możesz przekazać jej tyle argumentów, ile uważasz za stosowne. Możesz, na przykład, zechcieć definiować nazwę dla właściwości `data`, żeby jeszcze bardziej odizolować KWR od opakowanego komponentu. Możesz też przekazać argument, który steruje metodą `shouldComponentUpdate` lub taki, który konfiguruje w jakiś sposób źródło danych. To wszystko jest możliwe dlatego, że KWR ma pełną kontrolę nad opakowywanym komponentem.

Podobnie jak w przypadku zwykłych komponentów, kontrakt pomiędzy `withSubscription` i opakowywanym komponentem w całości opiera się na właściwościach. Pozwala to na łatwą podmianę jednego KWR-a na inny, pod warunkiem że przekazują one renderowanemu komponentowi takie same właściwości. Może się to okazać przydatne np. w razie potrzeby podmiany biblioteki pobierającej dane.

Nie modyfikuj opakowywanego komponentu. Użyj kompozycji.

Powstrzymaj się przed wszelkimi zmianami prototypu komponentu (innymi słowy, przed jego mutowaniem) wewnętrz KWR-a.

```

function logProps(Component) {
  Component.prototype.componentDidUpdate = function(prevProps) {
    console.log('Aktualne właściwości: ', this.props);
    console.log('Poprzednie właściwości: ', prevProps);
  };
  // Fakt, że zwracamy tu oryginalny komponent, może świadczyć o tym,
  // że został on w jakiś sposób zmodyfikowany.
  return Component;
}

// EnhancedComponent wypisze na konsolę informację przy każdej zmianie właściwości
// komponentu
const EnhancedComponent = logProps(Component);

```

Z powyższym kodem jest kilka problemów. Po pierwsze, nie można ponownie użyć opakowywanego komponentu osobno, w innym miejscu aplikacji. Co ważne, jeśli zaaplikujesz kolejny `EnhancedComponent`, który także zmienia metodę `componentDidUpdate`, funkcjonalność pierwszego KWR-a zostanie nadpisana! Ponadto, ten KWR nie zadziała poprawnie z komponentami funkcyjnymi, ponieważ nie mają one metod cyklu życia.

KWR-y mutujące są swego rodzaju "dziurawą abstrakcją" - konsument takiego komponentu musi znać jego implementację, aby uniknąć konfliktów z innymi KWR-ami.

Zamiast modyfikować, KWR-y powinny komponować poprzez opakowywanie otrzymanego komponentu w kontener:

```
function logProps(WrappedComponent) {
  return class extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('Aktualne właściwości: ', this.props);
      console.log('Poprzednie właściwości: ', prevProps);
    }
    render() {
      // Opakowuje otrzymany komponent w kontener, bez jego zmieniania. Dobrze!
      return <WrappedComponent {...this.props} />;
    }
  }
}
```

Powyższy KWR ma podobną funkcjonalność co wersja modyfikująca komponent, lecz nie wprowadza dodatkowych "zgrzytów". Działa zarówno z komponentami klasowymi, jak i funkcyjnymi. A ponieważ jest napisany jako czysta funkcja (ang. *pure function*), można go komponować z innymi KWR-ami, czy nawet z nim samym.

Być może udało ci się zauważyc pewne podobieństwa pomiędzy KWR-ami a wzorcem zwanym **komponenty-kontenery**. Komponenty-kontenery wchodzą w skład strategii oddzielającej odpowiedzialności na niskim i wysokim poziomie abstrakcji. Kontenery zarządzają takimi rzeczami jak subskrypcje czy stan, a także przekazują właściwości do komponentów, które z kolei zajmują się renderowaniem interfejsu. KWR-y używają kontenerów w części swojej implementacji. Można by powiedzieć, że KWR-y to takie definicje sparametryzowanych komponentów-kontenerów.

Konwencja: Przekazuj nieużywane właściwości do opakowywanego komponentu

KWR-y dodają jakąś funkcjonalność do komponentu. Nie powinny jednak zmieniać zbyt drastycznie jego kontraktu. Oczekuje się, że komponent zwracany przez KWR będzie miał podobny interfejs do oryginalnego.

KWR-y powinny przekazywać dalej właściwości, które nie są przez nie używane. Większość KWR-ów zawiera metodę renderującą podobną do tej poniżej:

```
render() {
  // Odfiltruj wszelkie dodatkowe właściwości, które są używane przez KWR
  // i nie powinny być przekazywane dalej
  const { extraProp, ...passThroughProps } = this.props;
```

```
// Wstrzyknij właściwości w opakowywany komponent. Zazwyczaj będą to
// wartości stanu lub metody instancji.
const injectedProp = someStateOrInstanceMethod;

// Przekaż właściwości do renderowanego komponentu
return (
  <WrappedComponent
    injectedProp={injectedProp}
    {...passThroughProps}
  />
);
}
```

Powyższa konwencja pomaga upewnić się, że KWR-y są tak elastyczne i uniwersalne, jak to tylko możliwe.

Konwencja: Maksymalizuj kompozycyjność

Nie wszystkie KWR-y wyglądają tak samo. Czasami przyjmują tylko jeden argument, będący opakowywanym komponentem:

```
const NavbarWithRouter = withRouter(Navbar);
```

Zwykle jednak przyjmują także dodatkowe argumenty. W poniższym przykładzie z biblioteki Relay użyto obiektu konfiguracyjnego, który opisuje dane, od których zależy komponent:

```
const CommentWithRelay = Relay.createContainer(Comment, config);
```

Najczęściej jednak spotyka się KWR-y wyglądające jak ten:

```
// Funkcja `connect` z biblioteki React Redux
const ConnectedComment = connect(commentSelector, commentActions)(CommentList);
```

Że co?! Łatwiej będzie przeanalizować ten kod, jeśli rozbijemy go na części.

```
// `connect` to funkcja, która zwraca inną funkcję
const enhance = connect(commentListSelector, commentListActions);
// Zwracana funkcja jest KWR-em, który zwraca komponent podłączony
// do magazynu Reduksa
const ConnectedComment = enhance(CommentList);
```

Innymi słowy, `connect` jest funkcją wyższego rzędu, która zwraca komponent wyższego rzędu!

Taka forma może wydawać się niejasna czy nawet niepotrzebna, ale ma jedną praktyczną własność. Jednoargumentowe KWR-y, takie jak ten zwrócony przez `connect` w powyższym przykładzie, mają sygnaturę `Component => Component`. Funkcje, których typ wartości zwracanej jest taki sam, jak typ wartości wejściowej, bardzo łatwo dają się komponować.

```
// Zamiast robić tak...
const EnhancedComponent = withRouter(connect(commentSelector)(WrappedComponent))

// ... możesz użyć pomocniczej funkcji komponującej
// compose(f, g, h) daje to samo, co (...args) => f(g(h(...args)))
const enhance = compose(
  // Obydwie poniższe funkcje są jednoargumentowymi KWR-ami
```

```

    withRouter,
    connect(commentSelector)
)
const EnhancedComponent = enhance(WrappedComponent)

```

(Ta sama własność pozwala również funkcji `connect`, jak i innym KWR-om napisanym w stylu "funkcji wzbogacających" (ang. *enhancer*), występować w formie dekoratora - eksperymentalnej funkcjonalności proponowanej dla JavaScriptu.)

Funkcja pomocnicza `compose` jest dostarczana przez wiele bibliotek zewnętrznych, wliczając w to `lodash` (jako `lodash.flowRight`), `Redux` czy `Ramda`.

Konwencja: Opakowuj wyświetlana nazwę dla łatwiejszego debugowania

Komponenty-kontenery stworzone przez KWR-y wyglądają w narzędziu `React Developer Tools` jak zwykłe komponenty. Aby ułatwić sobie debugowanie, możesz zmienić wyświetlana nazwę na inną, informującą o tym, że jest to wynik działania KWR-a.

Najczęściej stosowaną techniką jest opakowywanie wyświetlanej nazwy (ang. *display name*) renderowanego komponentu. Jeśli więc twój komponent wyższego rzędu nazywa się `withSubscription`, a opakowywany komponent to `CommentList`, użyj nazwy `WithSubscription(CommentList)`:

```

function withSubscription(WrappedComponent) {
  class WithSubscription extends React.Component {/* ... */}
  WithSubscription.displayName =
`WithSubscription(${getDisplayName(WrappedComponent)})`;
  return WithSubscription;
}

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';
}

```

Zastrzeżenia

Z komponentami wyższego rzędu wiążą się pewne restrykcje, które mogą nie być oczywiste dla początkujących reactowców.

Nie używaj KWR-ów wewnątrz metody render

Algorytm różnicujący w Reakcie (zwany [rekonsyliacyjnym](#)) korzysta z tożsamości komponentu, aby stwierdzić, czy powinien zaktualizować istniejące poddrzewo, czy też wyrzucić je do kosza i stworzyć nowe. Jeśli komponent zwracany przez funkcję `render` jest identyczny (`==`) jak komponent z poprzedniego renderowania, React aktualizuje drzewo rekurencyjnie, porównując je z tym nowym. Jeśli są różne, poprzednie poddrzewo jest odmontowywane w całości.

W innych przypadkach nie powinno cię to za bardzo obchodzić. Ma to jednak znaczenie dla KWR-ów, ponieważ oznacza to, że nie możesz stworzyć KWR-a wewnątrz metody `render` innego komponentu:

```

render() {
  // Przy każdym renderowaniu tworzona jest nowa wersja komponentu
  EnhancedComponent
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // Powoduje to każdorazowe odmontowanie i ponowne zamontowanie całego poddrzewa!
  return <EnhancedComponent />;
}

```

Problemem nie jest tu sama wydajność aplikacji. Ponowne montowanie komponentu powoduje utratę jego stanu i wszystkich jego potomków.

Zamiast tego używaj KWR-ów na zewnątrz definicji komponentu, tak żeby powstały komponent został stworzony tylko raz. Dzięki temu jego tożsamość zostanie zachowana pomiędzy kolejnymi renderowaniami. A zwykle o to właśnie chodzi.

W szczególnych przypadkach, gdy musisz użyć KWR-a dynamicznie, możesz zrobić to wewnątrz metody cyklu życia komponentu lub w jego konstruktorze.

Pamiętaj o skopiowaniu metod statycznych

Czasami przydatne okazuje się zdefiniowanie metody statycznej dla komponentu reactowego. Przykładowo, kontenery biblioteki Relay udostępniają statyczną metodę `getFragment`, ułatwiającą komponowanie fragmentów GraphQLowych.

Kiedy używasz KWR-a na komponencie, oryginalny komponent jest opakowywany w komponent-kontener. Oznacza to, że nowy komponent nie otrzyma żadnej z metod statycznych oryginalnego komponentu.

```

// Definiujemy metodę statyczną
WrappedComponent.staticMethod = function() {/*...*/}
// Używamy KWR-a
const EnhancedComponent = enhance(WrappedComponent);

// Rozszerzony komponent nie posiada metody statycznej
typeof EnhancedComponent.staticMethod === 'undefined' // true

```

Można sobie z tym poradzić kopiując metody do kontenera przed jego zwroceniem:

```

function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // Musimy wiedzieć, jakie metody trzeba skopiować :
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}

```

Niestety wymaga to przewidzenia, jakie metody będą musiały być skopiowane. Możesz jednak użyć `hoist-non-react-statics`, aby automatycznie skopiować wszystkie nie-reactowe metody statyczne:

```

import hoistNonReactStatic from 'hoist-non-react-statics';
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  hoistNonReactStatic(Enhance, WrappedComponent);
  return Enhance;
}

```

Innym możliwym rozwiązańiem jest wyeksportowanie statycznej metody komponentu niezależnie od niego.

```
// Zamiast...
MyComponent.someFunction = someFunction;
export default MyComponent;

// ...wyeksportuj metodę osobno...
export { someFunction };

// ...a w module korzystającym z nich, zainportuj obydwie rzeczy
import MyComponent, { someFunction } from './MyComponent.js';
```

Referencje nie są przekazywane

Mimo że powszechną konwencją jest przekazywanie w KWR-ach wszystkich właściwości w dół do opakowanego komponentu, nie działa to z referencjami. Dzieje się tak dlatego, że `ref` nie jest zwykłą właściwością — podobnie jak `key`, jest traktowana inaczej przez Reacta. Jeśli dodasz referencję do elementu, którego wynikiem renderowania jest komponent opakowany przez KWR, referencja będzie odnosiła się do instancji najbardziej zewnętrznego komponentu-kontenera, a nie do komponentu opakowywanego.

Rozwiązaniem tego problemu jest użycie interfejsu `React.forwardRef` (wprowadzonego w Reakcie 16.3).
[Więcej o tym mechanizmie dowiesz się z rozdziału o przekazywaniu referencji.](#)

Tworzenie własnych hooków

Hooki są nowym dodatkiem w Reakcie 16.8. Pozwalają one na wykorzystanie stanu i innych funkcjonalności Reacta, bez użycia klas.

Tworzenie własnych hooków pozwala wydzielić logikę z komponentów do funkcji.

Podczas nauki o [używaniu hooka efektów](#) poznaliśmy przedstawiony poniżej komponent aplikacji czatu. Komponent ten wyświetla wiadomość informującą o tym, czy znajomy jest dostępny, czy nie:

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Ładowanie...';
  }
  return isOnline ? 'Dostępny' : 'Niedostępny';
}
```

Założymy, że nasza aplikacja posiada też listę kontaktów i chcemy wyświetlać imiona dostępnych użytkowników w kolorze zielonym. Moglibyśmy skopiować i wkleić powyższą logikę do naszego komponentu `FriendListItem`, ale nie byłoby to idealne rozwiązanie:

```
import React, { useState, useEffect } from 'react';

function FriendListItem(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

```

        </li>
    );
}

```

Zamiast tego chcielibyśmy współdzielić logikę pomiędzy komponentami `FriendStatus` i `FriendListItem`.

W tradycyjnym podejściu mieliśmy do dyspozycji dwa popularne rozwiązania tego problemu: [komponenty wyższego rzędu \(ang. higher-order components\)](#) i [właściwości renderujące \(ang. render props\)](#). Przyjrzyjmy się teraz, jak hooki rozwiążają wiele z tych samych problemów, nie zmuszając przy tym do dodawania kolejnych komponentów do drzewa.

Wyodrębnianie logiki własnego hooka

Kiedy chcemy współdzielić logikę pomiędzy dwoma javascriptowymi funkcjami, wyodrębniamy ją do trzeciej funkcji. Zarówno komponenty, jak i hooki są funkcjami, więc zadziała to także dla nich!

Własny hook to po prostu javascriptowa funkcja, której nazwa zaczyna się od `use` i która może wywoływać inne hooki. Poniższy przykład `useFriendStatus` to nasz pierwszy własny hook:

```

import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}

```

Wewnątrz nie znajdziemy nic nowego -- logika została skopiowana z komponentów powyżej. Pamiętaj żeby, tak jak w komponentach, wywoływać inne hooki tylko z najwyższego poziomu kodu twoich własnych hooków.

W przeciwieństwie do reactowych komponentów, własny hook nie ma narzuconego określonego kształtu. Sami decydujemy, jakie przyjmuje argumenty i jaką, jeśli jakakolwiek, wartość zwróci. Innymi słowy, zachowuje się jak zwykła funkcja. Jego nazwa powinna zawsze zaczynać się od `use`, aby można było już na pierwszy rzut oka stwierdzić, czy mają dla niego zastosowanie [zasady korzystania z hooków](#).

Celem naszego hooka `useFriendStatus` jest zasubskrybowanie się do statusu dostępnosci znajomego. Dlatego przyjmuje on wartość `friendID` jako argument i zwraca informację, czy znajomy jest dostępny:

```

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);
}

```

```
// ...
return isOnline;
}
```

Teraz przyjrzymy się, jak możemy używać własnych hooków.

Używanie własnych hooków

Przypomnijmy, że naszym celem było usunięcie powielonej logiki z komponentów `FriendStatus` i `FriendListItem`. Oba oczekują informacji o tym, czy nasz znajomy jest dostępny.

Teraz, kiedy już wyodrębniliśmy tę logikę do hooka `useFriendStatus`, możemy go *po prostu* użyć:

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Ładowanie...';
  }
  return isOnline ? 'Dostępny' : 'Niedostępny';
}

function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

Czy ten kod jest równoważny oryginalnym przykładom? Tak, działa on dokładnie w ten sam sposób. Jeśli przyjrzyisz się uważniej, zauważysz, że nie dokonaliśmy żadnej zmiany w zachowaniu. Wszystko co zrobiliśmy, to wyodrębnienie wspólnego kodu z dwóch funkcji do jednej, osobnej funkcji. **Własne hooki są konwencją, która wynika naturalnie ze sposobu, w jaki zostały zaprojektowane hooki. Nie są one osobną funkcjonalnością Reacta.**

Czy nazwy moich własnych hooków muszą zaczynać się od „use”? Bardzo prosimy. Ta konwencja jest bardzo ważna. Bez niej nie moglibyśmy automatycznie sprawdzać, czy zostały naruszone [zasady korzystania z hooków](#), ponieważ nie bylibyśmy w stanie stwierdzić, czy w danej funkcji znajdują się wywołania hooków.

Czy dwa komponenty, korzystające z tego samego hooka, współdzielą stan? Nie. Własne hooki to mechanizm pozwalający na współdzielenie *logiki związanej ze stanem* (takiej jak tworzenie subskrypcji i zapamiętywanie bieżącej wartości), ale za każdym razem, kiedy używasz własnego hooka, cały stan i efekty wewnątrz niego są całkowicie odizolowane od siebie nawzajem.

W jaki sposób własny hook otrzymuje odizolowany stan? Każde wywołanie hooka tworzy odizolowany stan. Ponieważ wywołujemy `useFriendStatus` bezpośrednio, z punktu widzenia Reacta nasze komponenty wywołują po prostu funkcje `useState` i `useEffect`. A jak dowiedzieliśmy się już wcześniej, możemy w jednym komponencie wielokrotnie wywoływać funkcje `useState` oraz `useEffect` i będą one całkowicie niezależne.

Porada: Przekazywanie informacji pomiędzy hookami

Jako że hooki to funkcje, możemy pomiędzy nimi przekazywać informacje.

Aby to zilustrować, użyjemy kolejnego komponentu z naszego hipotetycznego przykładu czatu. Jest to rozwijane pole wyboru odbiorcy wiadomości, które wyświetla też, czy aktualnie wybrany znajomy jest dostępny:

```
const friendList = [
];

function ChatRecipientPicker() {
  const [recipientID, setRecipientID] = useState(1);
  const isRecipientOnline = useFriendStatus(recipientID);

  return (
    <>
      <Circle color={isRecipientOnline ? 'green' : 'red'} />
      <select
        value={recipientID}
        onChange={e => setRecipientID(Number(e.target.value))}>
        >
          {friendList.map(friend => (
            <option key={friend.id} value={friend.id}>
              {friend.name}
            </option>
          ))}
        </select>
      </>
    );
}
```

Przechowujemy aktualnie wybrany identyfikator znajomego w zmiennej stanu `recipientID` i aktualizujemy ją, gdy użytkownik wybierze innego znajomego z rozwijanego pola wyboru `<select>`.

Jako że wywołanie hooka `useState` zwraca najnowszą wartość zmiennej stanu `recipientID`, możemy przekazać ją do naszego własnego hooka `useFriendStatus` jako argument:

```
const [recipientID, setRecipientID] = useState(1);
const isRecipientOnline = useFriendStatus(recipientID);
```

Dzięki temu wiemy, czy *aktualnie wybrany* znajomy jest dostępny. Jeżeli wybierzemy innego znajomego, a tym samym zaktualizujemy zmienną stanu `recipientID`, nasz hook `useFriendStatus` anuluje subskrypcję dla poprzednio wybranego znajomego i zasubskrybuje się do statusu nowo wybranego.

Użyj wyobraźni

Własne hooki dają możliwość współdzielenia logiki w sposób, w jaki dotychczas nie było to możliwe w reaktywowych komponentach. Możesz pisać własne hooki, które obejmują szereg różnych przypadków użycia - od obsługi formularzy, animacji, deklaratywnych subskrypcji, liczników, po wiele innych, których jeszcze nie wymyślono. Co więcej, możesz tworzyć hooki, które są równie łatwe w użyciu, jak wbudowane funkcje Reacta.

Spróbuj jednak powstrzymać się od zbyt wcześniego wprowadzania abstrakcji. Teraz, kiedy komponenty funkcyjne mogą znacznie więcej, twój kod źródłowy takich komponentów najprawdopodobniej zacznie „puchnąć”. To normalne, nie *zmuszać się* od razu do dzielenia go na hooki. Ale zachęcamy też do tego, aby zacząć rozglądać się za przypadkami, gdzie własny hook mógłby ukryć skomplikowaną logikę za prostym interfejsem albo pomóc uprątnąć zagmatwany komponent.

Założmy na przykład, że masz w swoim kodzie skomplikowany komponent z dużą ilością zmiennych stanu, zarządzanych w sposób doraźny. Hook `useState` nie jest wcale rozwiązaniem na łatwą centralizację tej logiki. Pewnie lepiej byłoby ci napisać *reduxowy* reduktor (ang. *reducer*):

```
function todosReducer(state, action) {
  switch (action.type) {
    case 'add':
      return [...state, {
        text: action.text,
        completed: false
      }];
    // ... inne akcje ...
    default:
      return state;
  }
}
```

Reduktory są bardzo wygodne do testowania w izolacji i skalowania w celu wyrażenia skomplikowanej logiki aktualizacji. W razie potrzeby możesz je rozbić na mniejsze reduktory. Tym niemniej, być może wolisz korzystać z zalet lokalnego stanu Reacta albo po prostu nie chcesz instalować kolejnej biblioteki.

A co jeśli moglibyśmy napisać hook `useReducer`, który pozwala na zarządzanie *lokalnym* stanem komponentu przy użyciu reduktora? Jego uproszczona wersja mogłaby wyglądać następująco:

```
function useReducer(reducer, initialState) {
  const [state, setState] = useState(initialState);

  function dispatch(action) {
    const nextState = reducer(state, action);
    setState(nextState);
  }

  return [state, dispatch];
}
```

Teraz możemy go użyć w naszym komponencie i pozwolić reduktorowi na zarządzanie jego stanem:

```
function Todos() {  
  const [todos, dispatch] = useReducer(todosReducer, []);  
  
  function handleAddClick(text) {  
    dispatch({ type: 'add', text });  
  }  
  
  // ...  
}
```

Potrzeba zarządzania lokalnym stanem złożonego komponentu za pomocą reduktora jest na tyle powszechna, że wbudowaliśmy hook `useReducer` bezpośrednio w Reacta. Jego opis, wraz z innymi wbudowanymi hookami, znajdziesz w rozdziale pt. „[Hooki - interfejs API](#)”.

Używanie hooka efektów

Hooki są nowym dodatkiem w Reakcie 16.8. Pozwalają one używać stanu i innych funkcjonalności Reacta, bez użycia klas.

Hook efektów pozwala na przeprowadzanie efektów ubocznych w komponentach funkcyjnych:

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Podobnie do metod componentDidMount i componentDidUpdate:
  useEffect(() => {
    // Zaktualizuj tytuł dokumentu korzystając z interfejsu API przeglądarki
    document.title = `Kliknięto ${count} razy`;
  });

  return (
    <div>
      <p>Kliknięto {count} razy</p>
      <button onClick={() => setCount(count + 1)}>
        Kliknij mnie
      </button>
    </div>
  );
}
```

Ten fragment kodu oparty jest na [przykładzie licznika z poprzedniego rozdziału](#), ale dodaliśmy do niego nową funkcjonalność: ustawiamy tytuł dokumentu na własną wiadomość, zawierającą liczbę kliknięć.

Pobieranie danych, tworzenie subskrypcji czy ręczna ingerencja w drzewo DOM z wewnątrz komponentów - to wszystko przykłady efektów ubocznych. Niezależnie od tego, czy znasz je pod nazwą „efekty uboczne” (lub po prostu „efekty”), najprawdopodobniej masz je zaszyte gdzieś w swoim kodzie.

Wskazówka

Jeżeli znasz już metody cyklu życia (ang. *lifecycle methods*) Reacta, możesz myśleć o hooku `useEffect` jako o połączeniu metod `componentDidMount`, `componentDidUpdate` i `componentWillUnmount` w jedną.

W komponentach reactowych występują powszechnie dwa rodzaje efektów ubocznych: te, po których należy „posprzątać” i te, po których nie. Przyjrzyjmy się uważniej temu podziałowi.

Efekty niewymagające sprzątania

Czasami chcemy **uruchomić jakiś dodatkowy kod po tym, jak React zaktualizuje drzewo DOM**. Zapytania sieciowe, ręczna modyfikacja drzewa DOM czy logowanie to powszechnie przykłady efektów, które nie wymagają sprzątania. Mówimy tak, ponieważ możemy je uruchomić i od razu o nich zapomnieć. Porównajmy teraz, jak klasy i hooki pozwalają na przeprowadzanie takich efektów ubocznych.

Przykład wykorzystujący klasy

W klasowych komponentach reactowych metoda `render` nie powinna wywoływać żadnych efektów ubocznych. Działo by się to bowiem zbyt wcześnie -- zwykle chcemy przeprowadzać efekty już *po tym, jak React zaktualizuje drzewo DOM*.

Dlatego też w reactowych klasach umieszczamy efekty uboczne w specjalnych metodach `componentDidMount` i `componentDidUpdate`. Wracając do naszego przykładu, oto klasowy komponent licznika, który aktualizuje tytuł dokumentu po tym, jak React zaktualizuje drzewo DOM:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `Kliknięto ${this.state.count} razy`;
  }

  componentDidUpdate() {
    document.title = `Kliknięto ${this.state.count} razy`;
  }

  render() {
    return (
      <div>
        <p>Kliknięto {this.state.count} razy</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Kliknij mnie
        </button>
      </div>
    );
  }
}
```

Zauważ, że **musieliśmy powtórzyć ten sam kod w dwóch metodach cyklu życia**.

Dzieje się tak dlatego, że w wielu przypadkach chcemy wywołać ten sam efekt uboczny, niezależnie od tego czy komponent właśnie został zamontowany, czy też zaktualizowany. Koncepcja jest taka, żeby działało się to po każdym wyrenderowaniu komponentu. Ale reactowe komponenty klasowe nie mają takiej metody. Moglibyśmy, co prawda, wydzielić osobną metodę, ale wciąż musielibyśmy wywoływać ją w dwóch miejscach.

Teraz sprawdźmy, jak osiągnąć to samo korzystając z hooka `useEffect`.

Przykład wykorzystujący hooki

Widzieliśmy już ten przykład na początku tego rozdziału, ale spójrzmy raz jeszcze:

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `Kliknięto ${count} razy`;
  });

  return (
    <div>
      <p>Kliknięto {count} razy</p>
      <button onClick={() => setCount(count + 1)}>
        Kliknij mnie
      </button>
    </div>
  );
}
```

Co robi wywołanie `useEffect`? Poprzez użycie tego hooka mówisz Reactowi, że twój komponent musi wykonać jakąś czynność po jego wyrenderowaniu. React zapamięta funkcję, którą przekazano do hooka (będziemy odtąd odnosić się do niej jako naszego „efektu”), a potem wywoła ją, gdy już zaktualizuje drzewo DOM. W tym przypadku aktualizujemy tytuł dokumentu, ale moglibyśmy równie dobrze pobrać dane z serwera lub wywołać inne, imperatywne API.

Dlaczego funkcja `useEffect` jest wywoływana wewnątrz komponentu? Umiejscowienie `useEffect` wewnątrz komponentu daje nam dostęp do zmiennej stanu `count` (oraz wszystkich właściwości (ang. *props*)) z wewnątrz efektu. Nie potrzebujemy specjalnego interfejsu API do odczytania tych zmiennych -- znajdują się one w zasięgu funkcji. Hooki wykorzystują javascriptowe domknięcia (ang. *closure*) i unikają wprowadzania nowych, specyficznych dla Reacta interfejsów API -- JavaScript dostarcza przecież gotowe rozwiązanie.

Czy `useEffect` działa przy każdym renderze? Tak! Domyślnie działa on zarówno przy pierwszym wyrenderowaniu komponentu oraz każdej kolejnej jego aktualizacji. (W dalszej części dowiemy się, [jak dostosować to zachowanie](#).) Zamiast myśleć w kategoriach „montowania” i „aktualizacji”, być może łatwiej będzie ci zrozumieć, że efekty wykonywane są „po wyrenderowaniu”. React daje ci gwarancję, że drzewo DOM zostanie zaktualizowane zanim wywoła efekty.

Szczegółowe objaśnienie

Teraz, kiedy wiemy już więcej o efektach, te linijki kodu powinny nabrać sensu:

```
function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `Kliknięto ${count} razy`;
  });
}
```

Deklarujemy zmienną stanu `count`, a następnie mówimy Reactowi, że będziemy chcieli użyć efektu. Zatem do hooka `useEffect` przekazujemy funkcję, która jest naszym efektem. Wewnątrz tego efektu ustawiamy tytuł dokumentu, korzystając z interfejsu API przeglądarki (`document.title`). Możemy odczytać ostatnią wartość zmiennej `count` z wewnątrz efektu, ponieważ znajduje się ona w zasięgu naszej funkcji. Kiedy React będzie renderować nasz komponent, zapamięta użyty przez nas efekt, a następnie wywoła go po aktualizowaniu drzewa DOM. Dzieje się to przy każdym renderowaniu, włączając pierwsze.

Doświadczenci programiści języka JavaScript mogli zauważyc, że funkcja, którą przekazujemy do `useEffect`, będzie inna przy każdym renderze. Jest to celowe działanie. Właściwie to tylko dzięki temu możemy przeczytać wartość zmiennej `count` z wewnątrz efektu, nie martwiąc się, że będzie ona nieaktualna. Za każdym razem, kiedy ponownie renderujemy komponent, planujemy wykonanie innego efektu, który zastąpi poprzedni. W pewnym sensie sprawia to, że efekty zachowują się jak część wyniku renderowania -- każdy efekt „należy” do konkretnego renderowania. W [dalszej części tego rozdziału](#) przyjrzymy się dokładniej, dla czego jest to przydatne.

Wskazówka

W przeciwieństwie do metod `componentDidMount` i `componentDidUpdate`, efekty zaplanowane przy użyciu `useEffect` nie blokują przeglądarki przed odświeżeniem ekranu. Sprawia to wrażenie, że aplikacja działa płynniej. Większość efektów nie musi działać synchronicznie. W sporadycznych przypadkach, gdy muszą to robić (na przykład do pomiaru układu strony (ang. *layout*)), można skorzystać z dedykowanego hooka -- `useLayoutEffect`, z identycznym do `useEffect` interfejsem API.

Efekty wymagające sprzątania

Wcześniej omówiliśmy, jak wyrazić efekty uboczne, które nie potrzebują po sobie „posprzątać”. Jednakże istnieją efekty, które muszą to robić. Na przykład, **możemy chcieć utworzyć subskrypcję** do jakiegoś zewnętrznego źródła danych. W tym przypadku ważne jest, aby po sobie posprzątać i uniknąć tym samym potencjalnego wycieku pamięci! Porównajmy, jak możemy to zrobić z klasami, a jak z hookami.

Przykład wykorzystujący klasy

W klasowych komponentach reactowych zwykle tworzy się subskrypcję w metodzie `componentDidMount`, a następnie sprząta w metodzie `componentWillUnmount`. Założymy, że mamy moduł `ChatAPI`, który pozwala nam zasubskrybować się na zmianę statusu dostępności znajomego. Tak moglibyśmy zasubskrybować się i wyświetlać status znajomego przy użyciu klasy:

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleChange = this.handleChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friendId,
      this.updateStatus
    );
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friendId
    );
  }

  updateStatus(isOnline) {
    this.setState({ isOnline });
  }
}
```

```

        this.props.friend.id,
        this.handleStatusChange
    );
}

componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
        this.props.friend.id,
        this.handleStatusChange
    );
}

handleStatusChange(status) {
    this.setState({
        isOnline: status.isOnline
    });
}

render() {
    if (this.state.isOnline === null) {
        return 'Ładowanie...';
    }
    return this.state.isOnline ? 'Dostępny' : 'Niedostępny';
}
}

```

Zauważ, że metoda `componentDidMount` jest lustrzanym odbiciem metody `componentWillUnmount`. Metody cyklu życia zmuszają nas do podziału tej logiki, mimo że koncepcyjnie stanowią one część tego samego efektu.

Uwaga

Czytelnicy o sokolim wzroku mogli zauważyc, że powyższy przykład potrzebuje też metody `componentDidUpdate`, aby działać w pełni poprawnie. Na razie pominiemy to zagadnienie, ale wróćmy do niego [w dalszej części tego rozdziału](#).

Przykład wykorzystujący hooki

Zobaczmy, jak stworzyć ten sam komponent przy użyciu hooków.

Być może zastanawiasz się, czy będziemy potrzebować jakiegoś osobnego efektu, aby przeprowadzić czyszczenie. Kod do tworzenia i anulowania subskrypcji jest tak ściśle ze sobą powiązany, że `useEffect` został specjalnie zaprojektowany w ten sposób, aby utrzymać go razem. Jeśli efekt zwróci funkcję, React uruchomi ją, gdy nadejdzie pora na sprzątanie:

```

import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
    const [isOnline, setIsOnline] = useState(null);

    useEffect(() => {
        function handleStatusChange(status) {
            setIsOnline(status.isOnline);
        }
    })
}

```

```

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Określ sposób sprzątania po tym efekcie:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Ładowanie...';
  }
  return isOnline ? 'Dostępny' : 'Niedostępny';
}

```

Dlaczego zwróciliśmy funkcję z naszego efektu? Jest to opcjonalny mechanizm sprzątania po efektach. Każdy efekt może zwrócić funkcję, która określa sposób, w jaki sposób należy po nim posprzątać. Pozwala nam to na trzymanie logiki dotyczącej tworzenia i usuwania subskrypcji w jednym miejscu. Są one częścią tego samego efektu!

Kiedy dokładnie React sprząta po naszym efektie? Dzieje się to wtedy, gdy komponent jest odmontowywany. Jednakże, jak dowiedzieliśmy się wcześniej, efekty są wywoływane nie raz, ale przy każdym wyrenderowaniu komponentu. Dlatego React również sprząta po efektach poprzedniego renderowania, zanim wywoła kolejne efekty. Wyjaśnimy [dlaczego pomaga to uniknąć błędów i jak zrezygnować z tego zachowania w przypadku problemów z wydajnością](#) w dalszej części tego rozdziału.

Uwaga

Nie musisz nazywać funkcji zwracanej z efektów. My nazwaliśmy ją `cleanup` (pol. *posprzątaj*) aby lepiej wyjaśnić jej zamysł. Możesz po prostu zwrócić funkcję strzałkową (ang. *arrow function*) albo nazwać funkcję inaczej.

Podsumowanie

Nauczyliśmy się, że hook `useEffect` pozwala nam wyrazić różnego rodzaju efekty uboczne po wyrenderowaniu komponentu. Niektóre efekty mogą wymagać sprzątania, dlatego zwracają odpowiednią funkcję:

```

useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
});

```

Inne efekty mogą nie mieć fazy czyszczenia, nie zwracając więc nic.

```

useEffect(() => {
  document.title = `Kliknięto ${count} razy`;
});

```

Hook efektów łączy oba przypadki użycia w jednym interfejsie API.

Jeżeli czujesz, że masz już przyzwoitą wiedzę na temat hooka efektów albo jeśli rozbolała cię głowa od natłoku wiedzy, możesz od razu przejść do kolejnego rozdziału pt. „[Zasady korzystania z Hooków](#)”.

Porady dotyczące używania efektów

W dalszej części tego rozdziału przyjrzymy się głębiej niektórym aspektom hooka `useEffect`, które najprawdopodobniej zainteresują bardziej doświadczonych użytkowników Reacta. Nie musisz jednak zgłębiać ich wszystkich od razu. Zawsze możesz wrócić do tego rozdziału kiedy indziej i doczytać więcej o hooku efektów.

Porada: Użyj kilku efektów do odseparowania logiki

Jednym z problemów, który przedstawiliśmy we wprowadzeniu do hooków, w podrozdziale pt. „[Motywacja](#)” jest to, że metody cyklu życia w klasach zazwyczaj zawierają niepowiązaną ze sobą logikę. Z kolei wzajemnie powiązana logika jest podzielona na kilka metod. Oto przykład komponentu, który łączy w sobie zarówno logikę licznika, jak i statusu dostępności znajomego z poprzednich przykładów:

```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    document.title = `Kliknięto ${this.state.count} razy`;
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentDidUpdate() {
    document.title = `Kliknięto ${this.state.count} razy`;
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }
  // ...
}
```

Zauważ, jak logika ustawiania właściwości `document.title` jest podzielona pomiędzy metody `componentDidMount` i `componentDidUpdate`. Logika tworzenia subskrypcji jest również rozrzucona pomiędzy `componentDidMount` i `componentWillUnmount`. A metoda `componentDidMount` zawiera kod dla obu tych zadań.

Jak hooki rozwiązują ten problem? Podobnie jak możesz używać hooka *stanu* więcej niż raz, możesz też używać wielu efektów. Pozwala to na wydzielenie niepowiązanej logiki na osobne efekty:

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `Kliknięto ${count} razy`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
  // ...
}
```

Hooki pozwalają na dzielenie kodu na mniejsze fragmenty pod względem ich odpowiedzialności, a nie ze względu na nazwę metody cyklu życia. React wywoła każdy efekt użyty w komponencie w takiej kolejności, w jakiej został dodany.

Wyjaśnienie: Dlaczego efekty działają przy każdej aktualizacji

Jeżeli zwykle używasz klas, pewnie zastanawiasz się, dlaczego faza sprzątania po efektach następuje przy każdym kolejnym renderowaniu, a nie tylko raz, podczas odmontowywania komponentu. Spójrzmy na praktyczny przykład, aby lepiej zrozumieć dlaczego taka konstrukcja pozwala nam tworzyć komponenty z mniejszą liczbą błędów.

Wcześniej w tym rozdziale pokazaliśmy przykład komponentu `FriendStatus`, który wyświetla status dostępności znajomego. Nasza klasa czyta wartość `friend.id` z właściwości `this.props` i tworzy subskrypcję, gdy komponent jest montowany, a następnie usuwa ją, gdy jest odmontowywany.

```
componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
);
```

Ale co wydarzy się, jeżeli właściwość `friend` zmieni się? podczas gdy komponent cały czas widnieje na ekranie? Nasz komponent wciąż będzie wyświetlał status dostępności znajomego, ale nie tego co trzeba. To błąd. Spowodowałbyśmy też wyciek pamięci lub inną katastrofę przy odmontowywaniu, jako że usuwamy subskrypcję z nieprawidłowym ID znajomego.

W komponencie klasowym powinniśmy dodać metodę `componentDidUpdate`, aby obsłużyć ten przypadek:

```
componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentDidUpdate(prevProps) {
  // Usuń subskrypcję dla poprzedniej wartości friend.id
  ChatAPI.unsubscribeFromFriendStatus(
    prevProps.friend.id,
    this.handleStatusChange
  );
  // Utwórz subskrypcję dla nowej wartości friend.id
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}
```

Pominiecie poprawnej obsługi metody `componentDidUpdate` jest częstym źródłem błędów w aplikacjach reactowych.

A teraz rozważ wersję tego komponentu, która korzysta z hooków:

```
function FriendStatus(props) {
  // ...
  useEffect(() => {
    // ...
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
}
```

Nie jest ona podatna na ten błąd. (Ale nie wprowadziliśmy też żadnych zmian.)

Nie ma żadnego specjalnego kodu na obsługę aktualizacji, ponieważ hook `useEffect` obsługuje je domyślnie. Czyści on poprzednie efekty, zanim przeprowadzi kolejne. Aby to lepiej zilustrować pokażemy sekwencję tworzenia i usuwania subskrypcji, jakie wywoła ten komponent w określonym czasie:

```

ChatAPI.subscribeToFriendStatus(100, handleStatusChange); // Wywołaj pierwszy
efekt

ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // Wyczyść poprzedni
efekt
ChatAPI.subscribeToFriendStatus(200, handleStatusChange); // Przeprowadź
 kolejny efekt

ChatAPI.unsubscribeFromFriendStatus(200, handleStatusChange); // Wyczyść poprzedni
efekt
ChatAPI.subscribeToFriendStatus(300, handleStatusChange); // Przeprowadź
 kolejny efekt

// Odmontowanie
ChatAPI.unsubscribeFromFriendStatus(300, handleStatusChange); // Posprzątaj
ostatni efekt

```

Takie zachowanie zapewnia spójność i zapobiega występowaniu błędów typowych dla komponentów klasowych, spowodowanych brakiem implementacji logiki dla aktualizacji.

Porada: Optymalizacja wydajności przez pomijanie efektów

W niektórych przypadkach sprzątanie i przeprowadzanie efektów przy każdym renderze może stworzyć problemy z wydajnością. W komponentach klasowych możemy rozwiązać ten problem, dokładając dodatkowe porównanie wartości `prevProps` i `prevState` wewnątrz metody `componentDidUpdate`:

```

componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `Kliknięto ${this.state.count} razy`;
  }
}

```

To wymaganie jest na tyle powszechnne, że zostało wbudowane w interfejs API hooka `useEffect`. Możesz powiedzieć Reactowi, aby pominął przeprowadzanie efektu, jeśli pewne wartości nie zmieniły się między kolejnymi renderowaniami. Aby to zrobić, przekaż tablicę jako opcjonalny drugi argument do funkcji `useEffect`:

```

useEffect(() => {
  document.title = `Kliknięto ${count} razy`;
}, [count]); // Uruchom ponownie efekt tylko wtedy, gdy zmieni się wartość count

```

W powyższym przykładzie przekazujemy `[count]` jako drugi argument. Co to oznacza? Jeśli `count` ma wartość `5`, a nasz komponent jest ponownie renderowany z `count` wciąż równym `5`, React porówna `[5]` z poprzedniego renderowania i `[5]` z kolejnego renderowania. Ponieważ wszystkie elementy w tablicy są takie same (`5 === 5`), React pominie efekt. Oto nasza optymalizacja.

Kiedy renderujemy z wartością `count` zaktualizowaną do `6`, React porówna elementy tablicy `[5]` z poprzedniego renderowania do elementów w tablicy `[6]` z kolejnego renderowania. Tym razem React ponownie zastosuje efekt, ponieważ `5 !== 6`. Jeśli w tablicy jest wiele elementów, React ponownie uruchomi efekt, nawet jeśli tylko jeden z nich ma inną wartość.

Działa to również w przypadku efektów z fazą czyszczenia:

```
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
}, [props.friend.id]); // Zasubskrybij ponownie tylko wtedy, gdy zmieni się
wartość props.friend.id
```

W przyszłości drugi argument może być automatycznie dodawany przez odpowiednią transformację w kompilatorze.

Uwaga

Jeśli korzystasz z tej techniki optymalizacji, upewnij się, że przekazywana tablica zawiera **wszystkie wartości z zasięgu komponentu (takie jak właściwości (ang. `props`) i stan), które zmieniają się w czasie i które są używane przez efekt**. W przeciwnym razie twój kod odwoła się do starych wartości z poprzednich renderowań. Przeczytaj też, [jak radzić sobie z funkcjami i co robić, gdy tablica zmienia się zbyt często](#).

Jeśli chcesz przeprowadzić efekt i posprzątać po nim tylko raz (podczas montowania i odmontowania), możesz przekazać pustą tablicę ([]) jako drugi argument. Dzięki temu React wie, że twój efekt nie zależy od *jakichkolwiek* wartości właściwości lub stanu, więc nigdy nie musi być ponownie uruchamiany. Nie jest to traktowane jako przypadek specjalny -- wynika to bezpośrednio z tego, jak działa tablica zależności.

Jeśli przekażesz pustą tablicę ([]) właściwości i stan wewnątrz efektu zawsze przyjmą swoje początkowe wartości. Pomimo że przekazywanie [] jest bliższe temu, co znamy z metod `componentDidMount` i `componentWillUnmount`, zwykle istnieją [lepsze rozwiązania](#) pomagające uniknąć zbyt częstego powtarzania efektów. Nie zapominaj też, że React opóźni wywołanie `useEffect` do momentu, aż przeglądarka nie skończy rysować widoku, więc dodatkowa praca tutaj nie jest dużym problemem.

Polecamy użycie reguły `exhaustive-deps`, będącej częścią naszego pakietu `eslint-plugin-react-hooks`. Ostrzega ona, gdy zależności są niepoprawnie zdefiniowane i sugeruje poprawki.

Kolejne kroki

Gratulacje! To był długi rozdział, ale mniejmy nadzieję, że odpowiedzieliśmy na większość pytań dotyczących efektów ubocznych. Poznaliśmy już informacje zarówno o hooku stanu, jak i hooku efektów, a dzięki ich połączeniu możesz zrobić *naprawdę dużo*. Pokrywają one większość przypadków użycia klas -- a tam, gdzie tak nie jest, pomocne mogą okazać się [dodatkowe hooki](#).

Zaczynamy również zauważać, jak hooki rozwiązuje problemy opisane w podrozdziale pt. „[Motywacja](#)”. Zaobserwowałyśmy, jak sprzątanie po efektach pozwala uniknąć duplikacji kodu w metodach `componentDidUpdate` i `componentWillUnmount`, przybliża wzajemnie powiązany kod i pomaga uniknąć błędów. Zobaczyliśmy również, w jaki sposób możemy podzielić efekty w zależności od ich celu, co w klasach nie było wcześniej w ogóle możliwe.

W tym momencie mogłoby paść pytanie, jak działają hooki. Skąd React może wiedzieć, które wywołanie `useState` odpowiada którym zmiennym stanu pomiędzy kolejnymi renderowaniami? W jaki sposób React „dopasowuje” poprzednie i następne efekty przy każdej aktualizacji? **W następnym rozdziale dowiemy się o zasadach korzystania z hooków -- są one niezbędne do ich poprawnego działania.**

Hooki - FAQ

Hooki są nowym dodatkiem do Reacta w wersji 16.8. Pozwalają na użycie stanu i innych funkcji Reacta bez konieczności pisania klas.

Ta strona odpowiada na najczęściej zadawane pytania odnośnie [hooków](#).

- **Strategia wdrażania**

- Które wersje Reacta wspierają hooki?
- Czy muszę przepisać wszystkie komponenty klasowe?
- Co mogę zrobić z hookami, czego nie można było zrobić z klasami?
- Jaka część mojej wiedzy o Reakcie jest nadal aktualna?
- Czy lepiej używać hooków, klas, czy może mieszać obydwa sposoby?
- Czy hooki obejmują wszystkie przypadki użycia, które są dostępne dla klas?
- Czy hooki zastępują "właściwości renderujące" i komponenty wyższego rzędu?
- Co hooki oznaczają dla popularnych API, takich jak `connect()` z Reduxa lub React Router?
- Czy hooki współpracują ze statycznym typowaniem?
- Jak testować komponenty, które używają hooków?
- Co dokładnie narzucają reguły lintera?

- **Od klas do hooków**

- Jak wyglądają metody cyklu życia w odniesieniu do hooków?
- Jak mogę pobrać dane wykorzystując hooki?
- Czy istnieje coś podobnego do zmiennych instancji?
- Lepiej używać jednej czy wielu zmiennych stanu?
- Czy mogę uruchomić efekt tylko podczas aktualizacji komponentu?
- Jak dostać poprzednie właściwości lub stan?
- Dlaczego widzę nieaktualne właściwości lub stan wewnętrz mojej funkcji?
- Jak zaimplementować `getDerivedStateFromProps`?
- Czy istnieje coś takiego jak `forceUpdate`?
- Czy mogę stworzyć referencję do komponentu funkcyjnego?
- Jak mogę zmierzyć węzeł DOM?
- Co oznacza `const [thing, setThing] = useState()`?

- **Optymalizacja wydajności**

- Czy mogę ominąć efekt podczas aktualizacji komponentu?
- Czy bezpieczne jest pomijać funkcje w liście zależności?
- Co zrobić, gdy zależności mojego efektu zmieniają się zbyt często?
- Jak zaimplementować `shouldComponentUpdate`?
- Jak memoizować obliczenia?
- Jak w leniwy sposób tworzyć "ciężkie" obiekty?
- Czy hooki są wolne z powodu tworzenia funkcji podczas renderowania?
- Jak unikać przekazywania funkcji zwrotnych w dół?
- Jak odczytywać często zmieniającą się wartość z `useCallback`?

- [Pod maską](#)
 - [Jak React łączy wywołania hooków z komponentami?](#)
 - [Skąd wziął się pomysł na stworzenie hooków?](#)

Strategia wdrażania

Które wersje Reacta wspierają hooki?

Zaczynając od wersji 16.8.0, React zawiera stabilną implementację hooków dla:

- React DOM
- React Native
- React DOM Server
- React Test Renderer
- React Shallow Renderer

Zauważ, że **aby włączyć hooki, wszystkie paczki Reacta muszą mieć wersję 16.8.0 lub wyższą**. Hooki nie zadziałały, jeżeli zapomnisz zaktualizować, na przykład, React DOM.

[React Native wspiera hooki od wersji 0.59.](#)

Czy muszę przepisać wszystkie komponenty klasowe?

Nie. [Nie ma planów](#) na usunięcie klas z Reacta -- wszyscy musimy stale dostarczać nasze produkty i nie możemy sobie pozwolić na ich przepisywanie. Zachęcamy do wypróbowania hooków w nowym kodzie.

Co mogę zrobić z hookami, czego nie można było zrobić z klasami?

Hooki oferują nowy, potężny i ekspresyjny sposób na wielokrotne używanie funkcjonalności w komponentach. Rozdział pt. "[Tworzenie własnych hooków](#)" zawiera szybki wgląd w to, co można za ich pomocą zrobić. [Ten artykuł](#), napisany przez jednego z głównych członków zespołu Reacta, zawiera bardziej szczegółowe informacje o nowych możliwościach, które pojawiły się wraz z hookami.

Jaka część mojej wiedzy o Reakcie jest nadal aktualna?

Hooki są bardziej bezpośrednim sposobem na użycie dobrze już znanych funkcjonalności Reacta, takich jak na przykład: stan, cykl życia (ang. *lifecycle*), kontekst i referencje (ang. *refs*). Nie zmieniają podstaw działania Reacta, dlatego też twoja wiedza na temat komponentów, właściwości (ang. *props*) i przepływu danych z góry w dół pozostaje ciągle aktualna.

Hooki, same w sobie, posiadają pewną krzywą uczenia się. Jeżeli brakuje czegoś w tej dokumentacji, [zgłoś problem](#), a my postaramy się pomóc.

Czy lepiej używać hooków, klas, czy może mieszać obydwa sposoby?

Zachęcamy do wypróbowania hooków w nowych komponentach. Upewnij się, że wszyscy z twojego zespołu wiedzą, jak ich używać i są zapoznani z tą dokumentacją. Nie zalecamy przepisywania istniejących klas na hooki, chyba że z jakiegoś powodu i tak mieliście to w planach (na przykład w celu naprawy istniejących błędów).

Nie możesz używać hooków *wewnętrz* komponentów klasowych, jednakże bez obaw możesz mieszać komponenty klasowe i funkcyjne z hookami w tym samym drzewie. To, czy komponent jest klasowy, czy funkcyjny i używa hooków, jest detalem implementacyjnym tego komponentu. W dłuższej perspektywie oczekujemy, że hooki będą głównym sposobem pisania komponentów reactowych.

Czy hooki obejmują wszystkie przypadki użycia, które są dostępne dla klas?

Naszym celem dla hooków jest zapewnienie wszystkich przypadków użycia klas, tak szybko jak to tylko możliwe. Brakuje jeszcze odpowiedników dla kilku rzadziej używanych metod cyklu życia komponentu, takich jak `getSnapshotBeforeUpdate`, `getDerivedStateFromError` i `componentDidCatch`, ale zamierzamy je wkrótce dodać.

Czy hooki zastępują "właściwości renderujące" i komponenty wyższego rzędu?

Zazwyczaj właściwości renderujące i komponenty wyższego rzędu renderują tylko pojedynczy komponent potomny. Sądzimy, że hooki są prostszym sposobem na obsługę tego przypadku użycia. Nadal jest miejsce dla obu wzorców (dla przykładu, wirtualny komponent do obsługi suwaka może mieć właściwość `renderItem`, a prezentacyjny komponent kontenera może mieć swoją własną strukturę DOM). Jednak w większości przypadków hooki w zupełności wystarczą, a przy okazji pomogą zmniejszyć liczbę zagnieżdżeń w drzewie.

Co hooki oznaczają dla popularnych API, takich jak Redux connect() i React Router?

Möesz używać tych samych API, co do tej pory - będą nadal działać.

React Redux od wersji v7.1.0 [posiada wsparcie dla API hooków](#) i udostępnia takie funkcje, jak `useDispatch` czy `useSelector`.

React Router [wspiera hooki](#) od wersji 5.1.

W przyszłości być może także inne biblioteki zaczną wspierać hooki.

Czy hooki współpracują ze statycznym typowaniem?

Hooki zostały zaprojektowane z myślą o statycznym typowaniu. Dzięki temu, że są funkcjami, łatwiej jest je poprawnie otypować, w odróżnieniu od wzorców takich jak komponenty wyższego rzędu. Najnowsze definicje Reacta dla Flow i TypeScript wspierają hooki.

Co ważne, przy pomocy bardziej restrykcyjnych typów możesz ograniczyć API Reacta we własnych hookach. React dostarcza podstawowe elementy, ale możesz je łączyć na różne sposoby, odmienne od tych, które zawarliśmy w standardzie.

Jak testować komponenty, które używają hooków?

Z punktu widzenia Reacta komponent wykorzystujący hooki jest zwyczajnym komponentem. Jeżeli twoje narzędzie do testów nie opiera się na wewnętrznej implementacji Reacta, to testowanie komponentów, które używają hooków, nie powinno różnić się od tego, co robisz zazwyczaj.

Uwaga

W rozdziale pt. "[Testy: Przykłady i dobre praktyki](#)" znajdziesz wiele przykładów gotowych do użycia.

Dla przykładu, założmy, że mamy komponent licznika:

```
function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `Kliknięto ${count} razy`;
  });
  return (
    <div>
      <p>Kliknięto {count} razy</p>
      <button onClick={() => setCount(count + 1)}>
        Kliknij mnie
      </button>
    </div>
  );
}
```

Przetestujemy go używając React DOM. Aby upewnić się, że zachowanie komponentu odzwierciedla to w przeglądarce, opakujemy kod renderujący i aktualizujący w funkcję `ReactTestUtils.act()`:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('potrafi wyrenderować i zaktualizować licznik', () => {
  // Testuje pierwsze renderowanie i efekt
  act(() => {
```

```

ReactDOM.createRoot(container).render(<Counter />);
};

const button = container.querySelector('button');
const label = container.querySelector('p');
expect(label.textContent).toBe('Kliknięto 0 razy');
expect(document.title).toBe('Kliknięto 0 razy');

// Testuje drugie renderowanie i efekt
act(() => {
  button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
});
expect(label.textContent).toBe('Kliknięto 1 razy');
expect(document.title).toBe('Kliknięto 1 razy');
});

```

Wywołanie funkcji `act()` opróżni bufor efektów znajdujących się wewnętrz.

Jeżeli musisz przetestować własny hook, możesz stworzyć komponent w teście i wywołać ten hook w ciele jego funkcji. Następnie możesz napisać test do stworzonego w ten sposób komponentu.

Aby zmniejszyć powtarzalność kodu, zalecamy użyć biblioteki `react-testing-library`. Została ona zaprojektowana tak, aby zachęcać do pisania testów używających komponentów w sposób podobny do zachowania docelowych użytkowników aplikacji.

Po więcej informacji zajrzyj do rozdziału pt. ["Testy: Przykłady i dobre praktyki"](#).

Co dokładnie narzucają reguły lintera?

Stworzyliśmy [wtyczkę do ESLinta](#), która zmusza do przestrzegania [zasad hooków](#) w celu uniknięcia potencjalnych błędów. Zakładają one, że każda funkcja zaczynająca się od "`use`" i zaraz po tym wielkiej litery jest hookiem. Zdajemy sobie sprawę, że ta heurystyka nie jest idealna i może wywołać wiele fałszywych alarmów. Ale bez wprowadzenia wspólnej dla całego ekosystemu konwencji, nie ma możliwości, aby hooki działały poprawnie -- dłuższe nazwy zniechęcają ludzi do używania hooków lub do przestrzegania tej konwencji.

W szczególności, reguły te wymuszają, aby:

- Wywołania hooków znajdowały się wewnętrz funkcji pisanej stylem `PascalCase` (zakładają, że jest to komponent) lub innej funkcji `useSomething` (zakładają, że jest to własny hook).
- Hooki przy każdym renderowaniu są wywoływane w tej samej kolejności.

Jest jeszcze kilka innych heurystyk i mogą się one z czasem zmienić, gdy dostroimy reguły tak, aby zbalansować wyszukiwanie błędów i zmniejszyć liczbę fałszywych alarmów.

Od klas do hooków

Jak wyglądają metody cyklu życia w odniesieniu do hooków?

- `constructor`: Komponenty funkcyjne nie potrzebują konstruktora. Stan jest inicjalizowany poprzez wywołanie `useState`. Jeżeli obliczenie stanu początkowego jest kosztowne obliczeniowo, możesz do `useState` przekazać funkcję.

- `getDerivedStateFromProps` : Zamiast tego zaplanuj aktualizację [podczas renderowania](#).
- `shouldComponentUpdate` : Spójrz na [React.memo poniżej](#).
- `render` : Jest to ciało komponentu funkcyjnego.
- `componentDidMount`, `componentDidUpdate`, `componentWillUnmount` : Hook `useEffect` może z powodzeniem zastąpić wszelkie kombinacje tych metod (włączając w to [mniej znane przypadki](#)).
- `getSnapshotBeforeUpdate`, `componentDidCatch` i `getDerivedStateFromError` : W tej chwili nie istnieje hook odzwierciedlający działanie tych metod, ale zostanie wkrótce dodany.

Jak mogę pobrać dane wykorzystując hooki?

Tutaj znajdziesz [małe demo](#), które w tym pomoże. Aby dowiedzieć się więcej, przeczytaj artykuł [o pobieraniu danych z wykorzystaniem hooków](#).

Czy istnieje coś podobnego do zmiennych instancji?

Tak! Hook `useRef()` nie służy tylko do przechowywania referencji DOM. Obiekt "ref" jest generycznym kontenerem, którego właściwość `current` jest zmienna i może przechowywać każdą wartość, tak samo jak właściwości instancji w klasach.

Możesz do niej coś zapisać z wnętrza `useEffect`:

```
function Timer() {
  const intervalRef = useRef();

  useEffect(() => {
    const id = setInterval(() => {
      // ...
    });
    intervalRef.current = id;
    return () => {
      clearInterval(intervalRef.current);
    };
  });
  // ...
}
```

Jeżeli chcielibyśmy po prostu ustawić interwał, nie potrzebowalibyśmy referencji (`id` mogłoby być lokalne dla efektu), jednakże jest to użyteczne w przypadku, gdy chcielibyśmy wyczyścić interwał z wnętrza procedury obsługi zdarzenia:

```
// ...
function handleCancelClick() {
  clearInterval(intervalRef.current);
}
// ...
```

Działanie referencji jest takie samo jak użycie zmiennych instancji w klasie. Jeśli nie korzystasz z [leniwej inicjalizacji](#), unikaj używania referencji podczas renderowania -- może to prowadzić do niepożądanych zachowań. Zamiast tego modyfikuj referencje wewnątrz efektów lub procedur obsługi zdarzeń.

Lepiej używać jednej czy wielu zmiennych stanu?

Jeżeli na co dzień piszesz komponenty klasowe, kuszące może okazać się wywoływanie `useState()` jedno-krotnie i umieszczanie całego stanu wewnątrz pojedynczego obiektu. Jeżeli chcesz, możesz tak robić. Poniżej znajdziesz przykład komponentu, który śledzi ruchy kurSORA. Jego pozycja i stan są trzymane w lokalnym stanie:

```
function Box() {
  const [state, setState] = useState({ left: 0, top: 0, width: 100, height: 100 });
} // ...
}
```

Teraz przyjmijmy, że chcemy napisać logikę, która zmienia `left` i `top`, kiedy użytkownik ruszy myszką. Zauważ, że musimy ręcznie scalać te pola z poprzednim obiektem stanu:

```
// ...
useEffect(() => {
  function handleWindowMouseMove(e) {
    // Rozszczepienie "...state" zapewnia, że nie "stracimy" szerokości i
    // wysokość
    setState(state => ({ ...state, left: e.pageX, top: e.pageY }));
  }
  // Uwaga: ta implementacja jest dość uproszczona
  window.addEventListener('mousemove', handleWindowMouseMove);
  return () => window.removeEventListener('mousemove', handleWindowMouseMove);
}, []);
// ...
```

Gdy aktualizujemy zmienną stanu, *zamieniamy* jej wartość. Różni się to od `this.setState` w klasach, które *scala* zaktualizowane pola do obiektu stanu.

Jeżeli tępknisz za automatycznym scalaniem, możesz napisać własny hook `useLegacyState`, który scala aktualizacje obiektu stanu. Jednak **zalecamy podzielenie stanu na wiele zmiennych stanu, bazując na tym, które wartości mają tendencję do zmianiania się jednocześnie.**

Dla przykładu, możemy podzielić stan naszego komponentu na obiekty `position` oraz `size` i zawsze nadpisywać wartość `position`, bez konieczności scalania stanu z poprzednim:

```
function Box() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  const [size, setSize] = useState({ width: 100, height: 100 });

  useEffect(() => {
    function handleWindowMouseMove(e) {
      setPosition({ left: e.pageX, top: e.pageY });
    }
  }, ...);
```

Oddzielanie niezależnych zmiennych stanu ma także inną zaletę. Pozwala w przyszłości łatwo wyodrębnić powiązaną logikę do własnego hooka, na przykład:

```
function Box() {
  const position = useWindowPosition();
  const [size, setSize] = useState({ width: 100, height: 100 });
  // ...
}

function useWindowPosition() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  useEffect(() => {
    // ...
  }, []);
  return position;
}
```

Zauważ, jak mogliśmy przenieść wywołanie `useState` dla zmiennej stanu `position` i powiązany z nią efekt do własnego hooka, bez konieczności zmiany jego kodu. Jeżeli cały stan znajdowałby się w pojedynczym obiekcie, wyodrębnienie go byłoby trudniejsze.

Zarówno umieszczanie całego stanu wewnętrz pojedynczego wywołania `useState`, jak i wywoływanie `useState` dla każdego pola, będzie działać. Komponenty będą najbardziej czytelne, jeżeli osiągniesz równowagę pomiędzy tymi dwoma skrajnościami i pogrupujesz powiązane ze sobą zmienne stany. Jeżeli logika stanu stanie się zbyt złożona, zalecamy [użycie reduktora](#) lub napisanie własnego hooka.

Czy mogę uruchomić efekt tylko podczas aktualizacji komponentu?

Jest to rzadki przypadek. Jeżeli masz taką potrzebę, możesz [użyć zmiennej referencji](#), aby przechować wartość logiczną, określającą czy jest to pierwsze, czy kolejne renderowanie, a następnie sprawdzać tę flagę w efekcie. (Jeżeli okaże się, że robisz to często, możesz w tym celu stworzyć własnego hooka.)

Jak dostać poprzednie właściwości lub stan?

Poprzednich właściwości i stanu możesz potrzebować w dwóch przypadkach.

Czasami będziesz potrzebować poprzednich właściwości, aby **posprzątać po efekcie**. Przykład: masz efekt, który subskrybuje się do socketu na podstawie właściwości `userId`. Jeżeli wartość `userId` zmieni się, należałoby anulować subskrypcję z *poprzednim* `userId` i stworzyć nową z *następnią* wartością. Aby to zrobić, nie potrzeba niczego nadzwyczajnego:

```
useEffect(() => {
  ChatAPI.subscribeToSocket(props.userId);
  return () => ChatAPI.unsubscribeFromSocket(props.userId);
}, [props.userId]);
```

W powyższym przykładzie, jeśli `userId` zmieni się z `3` na `4`, najpierw wywoła się `ChatAPI.unsubscribeFromSocket(3)`, a następnie `ChatAPI.subscribeToSocket(4)`. Nie ma potrzeby dostępu do "poprzedniego" `userId`, ponieważ funkcja sprzątająca uchwyci go w domknięciu (ang. *closure*).

Innym razem możesz potrzebować **zaktualizować stan przy jakiejś zmianie właściwości lub innego stanu**. Rzadko się tak zdarza i zwykle oznacza, że masz jakiś zduplikowany lub niepotrzebny kawałek stanu. Jeśli jednak naprawdę potrzebujesz skorzystać z tego wzorca, możesz zapamiętać w stanie poprzednią wartość jakiegoś stanu lub właściwości i aktualizować ją podczas renderowania.

Poprzednio do przechowywania poprzedniej wartości zasugerowaliśmy skorzystanie z hooka o nazwie `usePrevious`. Mimo to zauważymy, że w większości przypadku mamy do czynienia z jednym z dwóch powyższych schematów. Jeśli twój przypadek jest inny, możesz zapisać wartość w referencji (ang. `ref`) i ręcznie aktualizować ją w razie potrzeby. Spróbuj jednak unikać odczytywania i aktualizowania referencji podczas renderowania, gdyż zmniejszy to przewidywalność i czytelność twojego komponentu.

Dlaczego widzę nieaktualne właściwości lub stan wewnętrz mojej funkcji?

Każda funkcja wewnętrz komponentu, włączając w to procedury obsługi zdarzeń i efekty, "widzą" właściwości i stan z chwili renderowania, w którym zostały stworzone. Dla przykładu rozważ poniższy kod:

```
function Example() {
  const [count, setCount] = useState(0);

  function handleAlertClick() {
    setTimeout(() => {
      alert('Kliknięto: ' + count);
    }, 3000);
  }

  return (
    <div>
      <p>Kliknięto {count} razy</p>
      <button onClick={() => setCount(count + 1)}>
        Naciśnij mnie
      </button>
      <button onClick={handleAlertClick}>
        Pokaż okno ostrzegawcze
      </button>
    </div>
  );
}
```

Jeżeli najpierw klikniesz "Pokaż okno ostrzegawcze", a następnie zwiększasz licznik, okno ostrzegawcze wyświetli wartość zmiennej `count` z momentu kliknięcia na przycisk "Pokaż okno ostrzegawcze". Zapobiega to błędem powodowanym przez kod zakładający, że właściwości i stan nie zmienią się w czasie.

Jeżeli celowo chcesz odczytać *najświeższy* stan z wnętrza asynchronicznej funkcji zwrotnej, możesz go przechowywać, zmieniać i odczytywać korzystając z [referencji](#).

Ostatecznie, inną możliwą przyczyną tego, że widzisz nieaktualne właściwości lub stan, może być użycie "tablicy zależności" do optymalizacji, ale niepoprawne sprecyzowanie wszystkich zależności. Dla przykładu, jeśli efekt otrzymuje `[]` jako drugi argument, ale wewnętrz odczytuje `someProp`, efekt będzie stale "widział" początkową wartość `someProp`. Rozwiążaniem jest usunięcie tablicy zależności lub naprawienie jej. Tutaj znajdziesz informacje, [jak poradzić sobie z funkcjami](#), a tutaj [inne powszechnie sposoby na uruchamianie efektów rzadziej i bez błędów w zależnościach](#).

Uwaga

Stworzyliśmy regułę `exhaustive-deps` dla ESLinta i dodaliśmy ją do paczki `eslint-plugin-react-hooks`. Wtyczka ostrzega, gdy zależności są sprecyzowane niepoprawnie i zaleca poprawienie kodu.

Jak zaimplementować `getDerivedStateFromProps`?

Prawdopodobnie [w ogóle nie potrzebujesz tej funkcjonalności](#). W rzadkich przypadkach, w których naprawdę będziesz tego potrzebować (na przykład implementacja komponentu `<Transition>`), możesz zaktualizować stan w trakcie renderowania. React wywoła ponownie komponent z zaktualizowanym stanem natychmiast po pierwszym renderowaniu, więc nie wpłynie to znacząco na wydajność.

W poniższym kodzie przechowujemy poprzednią wartość właściwości `row` w zmiennej stanowej, dzięki czemu możemy wykonać porównanie:

```
function ScrollView({row}) {
  const [isScrollingDown, setIsScrollingDown] = useState(false);
  const [prevRow, setPrevRow] = useState(null);

  if (row !== prevRow) {
    // Wiersz zmienił się od ostatniego renderowania. Zaktualizuj isScrollingDown.
    setIsScrollingDown(prevRow !== null && row > prevRow);
    setPrevRow(row);
  }

  return `Przewijanie w dół: ${isScrollingDown}`;
}
```

Na pierwszy rzut oka może to wyglądać dziwnie, ale aktualizacja podczas renderowania jest dokładnie tym samym, czym w założeniu metoda `getDerivedStateFromProps` była od zawsze.

Czy istnieje coś takiego jak `forceUpdate`?

Zarówno `useState`, jak i `useReducer` [wycofują się z aktualizacji](#), jeżeli kolejna wartość jest taka sama jak poprzednia. Zmiana stanu bez użycia `setState`, a następnie wywołanie `setState` nie skutkuje ponownym renderowaniem komponentu.

Zazwyczaj nie powinno się bezpośrednio modyfikować lokalnego stanu w Reakcie. Możesz jednak inkrementować licznik, aby wymusić ponowne renderowanie, nawet jeśli stan się nie zmienił:

```
const [ignored, forceUpdate] = useReducer(x => x + 1, 0);

function handleClick() {
  forceUpdate();
}
```

Jeżeli to możliwe, staraj się unikać tego wzorca.

Czy mogę stworzyć referencję do komponentu funkcyjnego?

Nie powinno się tego robić zbyt często, jednak możesz upublicznić niektóre imperatywne metody dla komponentu rodzica używając hooka `useImperativeHandle`.

Jak mogę zmierzyć węzeł DOM?

Aby zmierzyć pozycję lub rozmiar węzła DOM, możesz użyć [referencji z funkcją zwrotną](#). React wywoła funkcję zwrotną, gdy referencja zostanie przypisana do innego węzła. Tutaj znajdziesz [prosty przykład](#):

```
function MeasureExample() {
  const [height, setHeight] = useState(0);

  const measuredRef = useCallback(node => {
    if (node !== null) {
      setHeight(node.getBoundingClientRect().height);
    }
  }, []);

  return (
    <>
      <h1 ref={measuredRef}>Witaj, świecie</h1>
      <h2>Powyższy nagłówek ma {Math.round(height)} pikseli wysokości</h2>
    </>
  );
}
```

W tym przykładzie nie zdecydowaliśmy się użyć `useRef`, ponieważ obiekt referencji nie powiadamia nas o [zmianach](#) jego aktualnej wartości. Użycie referencji z funkcją zwrotną daje pewność, że [nawet jeśli komponent potomny wyświetli mierzony węzeł później](#) (np. w odpowiedzi na przyciśnięcie przycisku), komponent nadzędny zostanie o tym powiadomiony i może zaktualizować swój pomiar.

Zauważ, że przekazaliśmy `[]` jako tablicę zależności do `useCallback`. Gwarantuje to nam niezmienialność funkcji zwrotnej pomiędzy ponownymi renderowaniami oraz że React nie wywoła jej bez potrzeby.

W tym przykładzie funkcja zwrotna referencji zostanie wywołana tylko w momencie zamontowania i odmontowania komponentu. Dzieje się tak dlatego, że komponent `<h1>` jest obecny w każdym renderowaniu. Jeśli chcesz otrzymywać powiadomienie przy każdej zmianie rozmiaru komponentu, proponujemy skorzystać z `ResizeObserver` lub hooka z jakiejś biblioteki zewnętrznej opartej na tym mechanizmie.

W razie konieczności można [wyodrębnić tę logikę](#) do osobnego hooka i używać wielokrotnie:

```
function MeasureExample() {
  const [rect, ref] = useClientRect();
  return (
    <>
      <h1 ref={ref}>Witaj, świecie</h1>
      {rect !== null &
        <h2>Powyższy nagłówek ma {Math.round(rect.height)} pikseli wysokości</h2>
      }
    </>
  );
}

function useClientRect() {
```

```

const [rect, setRect] = useState(null);
const ref = useCallback(node => {
  if (node !== null) {
    setRect(node.getBoundingClientRect());
  }
}, []);
return [rect, ref];
}

```

Co oznacza `const [thing, setThing] = useState()`?

Jeżeli nie rozpoznajesz tej składni, sprawdź [wyjaśnienie](#) w dokumentacji hooka stanu.

Optymalizacja wydajności

Czy mogę pominąć efekt podczas aktualizacji komponentu?

Tak. Zapoznaj się z [warunkowym uruchamianiem efektów](#). Pamiętaj jednak, że pomijanie aktualizacji często prowadzi do błędów, z tego też powodu nie jest to domyślnie działanie.

Czy bezpiecznie jest pomijać funkcje w liście zależności?

Ogólnie rzecz biorąc, nie.

```

function Example({ someProp }) {
  function doSomething() {
    console.log(someProp);
  }

  useEffect(() => {
    doSomething();
  }, []); // ⚡ Niebezpieczne (wywołuje `doSomething`, które używa `someProp`)
}

```

Trudno jest pamiętać, które właściwości lub stan są używane przez funkcje poza efektem. Dlatego też **zazwyczaj lepiej jest deklarować funkcje wewnętrz efektu**. Dzięki temu łatwo można zauważyc, od których wartości komponentu zależy efekt:

```

function Example({ someProp }) {
  useEffect(() => {
    function doSomething() {
      console.log(someProp);
    }

    doSomething();
  }, [someProp]); // ✅ OK (efekt używa wyłącznie `someProp`)
}

```

Jeżeli po zmianach efekt nadal nie używa wartości z zakresu komponentu, można bezpiecznie użyć `[]`:

```

useEffect(() => {
  function doSomething() {
    console.log('Cześć!');
  }
}

```

```
doSomething();
}, []); // ✅ OK, ponieważ *żadne* wartości z zakresu komponentu nie są używane
wewnętrz efektu
```

W zależności od przypadku użycia, istnieje kilka dodatkowych opcji, które opisaliśmy poniżej.

Uwaga

Stworzyliśmy regułę `exhaustive-deps` (pol. *wyczerpujące zależności*), będącą częścią paczki `eslint-plugin-react-hooks`. Pomaga w znalezieniu komponentów, które nie obsługują aktualizacji w konsekwentny sposób.

Spójrzmy, dlaczego ma to znaczenie.

Kiedy określasz **tablicę zależności**, ostatni argument dla `useEffect`, `useLayoutEffect`, `useMemo`, `useCallback`, lub `useImperativeHandle` powinien zawierać wszystkie wartości biorące udział w przepływie danych, włączając w to właściwości, stan i wszystkie ich pochodne.

Jednym **bezpiecznym** przypadkiem pominięcia argumentu w tablicy zależności jest przekazanie funkcji, która w swoim wnętrzu nie ma odniesień do właściwości, stanu lub wartości z nich dziedziczących. Poniższy przykład zawiera błąd:

```
function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  async function fetchProduct() {
    const response = await fetch('http://myapi/product/' + productId); // Używa
    właściwości productId
    const json = await response.json();
    setProduct(json);
  }

  useEffect(() => {
    fetchProduct();
  }, []); // ❌ Błąd, ponieważ `fetchProduct` używa `productId`
  // ...
}
```

Zalecanym sposobem naprawienia tego, jest przeniesienie funkcji do wnętrza efektu. Dzięki temu łatwiej będzie nam dostrzec stan lub właściwości, których używa efekt, i upewnić się, że wszystkie z nich zostały zadeklarowane:

```
function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  useEffect(() => {
    // Po przeniesieniu funkcji do wnętrza efektu, możemy łatwo dostrzec, których
    wartości używa.
    async function fetchProduct() {
      const response = await fetch('http://myapi/product/' + productId);
      const json = await response.json();
      setProduct(json);
    }
  });
}
```

```

        }

        fetchProduct();
    }, [productId]); // ✅ Poprawnie, ponieważ efekt używa wyłącznie productId
    // ...
}

```

Pozwala to również na obsłużenie asynchronicznych odpowiedzi, stosując zmienną lokalną wewnątrz efektu:

```

useEffect(() => {
    let ignore = false;
    async function fetchProduct() {
        const response = await fetch('http://myapi/product/' + productId);
        const json = await response.json();
        if (!ignore) setProduct(json);
    }

    fetchProduct();
    return () => { ignore = true };
}, [productId]);

```

Przenieśliśmy funkcję do wnętrza efektu, dlatego też nie musi ona znajdować się w tablicy zależności.

Wskazówka

Aby dowiedzieć się więcej o pobieraniu danych za pomocą hooków, sprawdź [ten przykład](#) i [ten artykuł](#).

Jeżeli z jakichś przyczyn *nie możesz przenieść funkcji do wnętrza efektu, istnieje kilka innych opcji:*

- **Możesz spróbować przenieść funkcję poza swój komponent.** W tym przypadku funkcja nie będzie odnosić się do żadnych właściwości czy stanu, dlatego też nie będzie potrzeby dodawania jej do tablicy zależności.
- Jeżeli funkcja, którą wywołujesz, wykonuje jedynie obliczenia i można ją bezpiecznie wywołać podczas renderowania, możesz zechcieć **wywołać ją poza efektem** i uzależnić efekt od zwróconej przez nią wartości.
- W ostateczności, możesz **dodać funkcję do zależności efektu poprzez opakowanie jej definicji**, korzystając z hooka `useCallback`. Zapewnia to niezmienność podczas renderowania, dopóki nie zmieni się również jej własna tablica zależności:

```

function ProductPage({ productId }) {
    // ✅ Opakowanie za pomocą useCallback, aby uniknąć zmian przy każdym
    renderowaniu
    const fetchProduct = useCallback(() => {
        // ... Korzysta z productId ...
    }, [productId]); // ✅ Zdefiniowane zostały wszystkie zależności useCallback

    return <ProductDetails fetchProduct={fetchProduct} />;
}

function ProductDetails({ fetchProduct }) {
    useEffect(() => {

```

```

    fetchProduct();
}, [fetchProduct]); // ✅ Zdefiniowane zostały wszystkie zależności useEffect
// ...
}

```

Zauważ, że w powyższym przykładzie **musieliszmy** przekazać funkcję do tablicy zależności. Dzięki temu zmiana właściwości `productId` w `ProductPage` będzie automatycznie uruchamiała ponowne pobranie danych w komponentie `ProductDetails`.

Co zrobić, gdy zależności mojego efektu zmieniają się zbyt często?

Czasem twój efekt może korzystać ze stanu, który zmienia się zbyt często. Może cię kusić usunięcie go z listy zależności, jednak zwykle prowadzi to do błędów.:

```

function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1); // Ten efekt zależy od wartości `count`
    }, 1000);
    return () => clearInterval(id);
  }, []); // ❌ Błąd: Zmienna `count` nie została wymieniona w zależnościach

  return <h1>{count}</h1>;
}

```

Pusty zbiór zależności, `[]`, oznacza, że efekt ten zostanie uruchomiony tylko jeden raz podczas montowania komponentu, ale już nie przy kolejnych renderowaniach. Problem polega na tym, iż wartość zmiennej `count` wewnętrz funkcji zwrotnej przekazanej do `setInterval` nie będzie się zmieniać. Dzieje się dlatego, że stworzyliśmy dla niej domknięcie (ang. *closure*), w którym `count` ma wartość `0`, ponieważ z taką wartością uruchomiono ten efekt. Co sekundę funkcja zwrotna będzie wywoływała `setCount(0 + 1)`, przez co wartość licznika nigdy nie przekroczy 1.

Podanie `[count]` jako listy zależności mogłoby naprawić ten błąd, jednak spowodowałoby to resetowanie się interwału przy każdej zmianie stanu. W konsekwencji, każdy `setInterval` miałby jedną szansę na wykonanie, zanim zostałaby wyczyszczony (zachowanie podobne do `setTimeout`). Raczej nie o to nam chodzi. Aby temu zapobiec, możemy skorzystać z **funkcyjnego wariantu aktualizacji poprzez `useState`**. Pozwoli to nam określić, jak stan powinien się zmienić, bez odnoszenia się do konkretnego *aktualnego* stanu:

```

function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + 1); // ✅ Nie zależy od zewnętrznej zmiennej `count`
    }, 1000);
    return () => clearInterval(id);
  }, []); // ✅ Nasz efekt nie korzysta z żadnych zmiennych z zakresu komponentu

  return <h1>{count}</h1>;
}

```

(Stałosć referencyjna funkcji `setCount` jest zagwarantowana przez Reacta, więc można ją pominąć na liście zależności.)

Teraz funkcja zwrotna przekazana do `setInterval` wywoływana jest co sekundę, lecz za każdym razem wywołanie `setCount` wewnętrz korzysta z aktualnej wartości licznika `count` (nazwanej lokalnie jako `c`).

W bardziej zawiłych przypadkach (np. gdy jeden stan zależy od drugiego), spróbuj przenieść logikę zmiany stanu poza efekt przy pomocy hooka `useReducer`. W tym artykule pokazano przykład jego zastosowania.

Tożsamość funkcji `dispatch` zwróconej przez `useReducer` jest zawsze stabilna — nawet jeśli reduktor jest deklarowany wewnątrz komponentu i odczytuje jego właściwości.

Ostatecznie, jeśli zechcesz skorzystać z czegoś w rodzaju klasowego `this`, możesz [użyć referencji](#) do przechowania mutowalnej zmiennej. Wtedy możliwe będzie jej nadpisywanie i odczytywanie w dowolnym momencie. Na przykład:

```
function Example(props) {
  // Trzymamy ostatnie właściwości w referencji.
  const latestProps = useRef(props);
  useEffect(() => {
    latestProps.current = props;
  });

  useEffect(() => {
    function tick() {
      // Odczytujemy ostatnie właściwości w dowolnym momencie
      console.log(latestProps.current);
    }
    const id = setInterval(tick, 1000);
    return () => clearInterval(id);
  }, []); // Ten efekt nigdy nie uruchomi się ponownie
}
```

Rób tak tylko, gdy nie znajdziesz lepszej alternatywy, ponieważ poleganie na mutacjach negatywnie wpływa na przewidywalność zachowania się komponentów. Jeśli znasz jakiś wzorzec, którego nie da się w prosty sposób wyrazić za pomocą hooków, [zgłoś to nam](#), załączając przykład działającego kodu, a my postaramy się pomóc.

Jak zaimplementować `shouldComponentUpdate` ?

Możesz opakować komponent funkcyjny za pomocą `React.memo`, aby zastosować płytkie porównanie jego właściwości:

```
const Button = React.memo((props) => {
  // twój komponent
});
```

Nie jest to hook, bo nie komponuje się jak hooki. `React.memo` jest odpowiednikiem klasy `PureComponent`, jednak ogranicza się do porównywania wyłącznie właściwości. (Możesz także jako drugi argument przekazać funkcję porównującą poprzednie i aktualne właściwości. Jeśli zwróci `true`, aktualizacja komponentu zostanie pominięta.)

`React.memo` nie porównuje stanu komponentu, ponieważ komponenty funkcyjne nie mają jednego jedynego obiektu stanu, jak to ma miejsce w komponentach klasowych. Możesz jednak sprawić, by komponenty potomne również były "czystymi" komponentami (ang. *pure components*), a nawet [zoptymalizować poszczególnych potomków za pomocą `useMemo`](#).

Jak memoizować obliczenia?

Za pomocą hooka `useMemo` możesz zapamiętać wynik obliczeń pomiędzy kolejnymi renderowaniami:

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Powyższy kod wywołuje funkcję `computeExpensiveValue(a, b)`, która wykonuje kosztowne obliczenia. Jeśli jednak zależności `[a, b]` nie zmieniły się od ostatniego razu, `useMemo` pominie kolejne wywołanie funkcji i zamiast tego zwróci ostatni wynik.

Pamiętaj, że funkcja przekazana do `useMemo` wywoływana jest podczas renderowania. Nie rób w niej niczego, czego normalnie nie robisz podczas renderowania. Oznacza to, że, na przykład, efekty uboczne należy umieszczać w `useEffect`, a nie w `useMemo`.

Traktuj użycie `useMemo` jako optymalizację szybkości programu, a nie "gwarancję semantyczną" (ang. *semantic guarantee*). W przyszłości React będzie "zapominał" niektóre zapisane wyniki i przeliczał je dopiero przy następnym renderowaniu, np. aby zwolnić pamięć przydzieloną dla komponentów, których nie widać na ekranie. Pisz swój kod tak, aby działał bez użycia `useMemo` — a dopiero później dodawaj ten hook w celach optymalizacyjnych. (W sporadycznych przypadkach, w których wynik *nigdy* nie powinien być przeliczany na nowo, zalecamy skorzystać z [leniwie inicjalizowanej referencji](#).)

Co więcej, `useMemo` pozwala także pominąć kosztowne renderowania komponentów potomnych:

```
function Parent({ a, b }) {
  // Ponownie renderowany tylko wtedy, gdy zmieni się `a`:
  const child1 = useMemo(() => <Child1 a={a} />, [a]);
  // Ponownie renderowany tylko wtedy, gdy zmieni się `b`:
  const child2 = useMemo(() => <Child2 b={b} />, [b]);
  return (
    <>
      {child1}
      {child2}
    </>
  )
}
```

Pamiętaj jednak, że to podejście nie zadziała w pętli, ponieważ [hooków nie można wywoływać w pętlach](#). Możesz jednak wydzielić osobny komponent renderujący element listy, a następnie wywołać w nim `useMemo`.

Jak w leniwy sposób tworzyć "ciężkie" obiekty?

`useMemo` pozwala na [memoizację kosztownych obliczeń](#), pod warunkiem, że ich zależności są takie same. Jest to jednak tylko wskazówka i nie gwarantuje, że obliczenia nie zostaną uruchomione ponownie. Czasem jednak chcesz mieć pewność, że obiekt zostanie stworzony dokładnie raz.

Pierwszy z częstych przypadków dotyczy kosztownego tworzenia stanu początkowego:

```
function Table(props) {
  //⚠ Funkcja createRows() będzie wywoływana przy każdym renderowaniu
  const [rows, setRows] = useState(createRows(props.count));
  // ...
}
```

Aby uniknąć ponownego tworzenia i tak ignorowanego stanu początkowego, możemy do `useState` przekazać **funkcję inicjalizującą**:

```
function Table(props) {
  //✅ Funkcja createRows() będzie wywołana tylko raz
  const [rows, setRows] = useState(() => createRows(props.count));
  // ...
}
```

React wywoła funkcję tylko przy pierwszym renderowaniu. Po więcej informacji zajrzyj do [dokumentacji API hooka useState](#).

Czasem możesz chcieć uniknąć wielokrotnego tworzenia wartości początkowej dla hooka `useRef()`. Na przykład, jeśli chcesz mieć pewność, że zostanie utworzona tylko jedna instancja danej klasy:

```
function Image(props) {
  //⚠ Instancja klasy IntersectionObserver będzie tworzona przy każdym
  //renderowaniu
  const ref = useRef(new IntersectionObserver(onIntersect));
  // ...
}
```

Hook `useRef` **nie przyjmuje** alternatywnego argumentu w postaci funkcji, jak ma to miejsce w `useState`. Zamiast tego możesz napisać własną funkcję, która tworzy i ustawia wartość referencji w sposób leniwy:

```
function Image(props) {
  const ref = useRef(null);

  //✅ Instancja klasy IntersectionObserver zostanie stworzona leniwie tylko raz
  function getObserver() {
    if (ref.current === null) {
      ref.current = new IntersectionObserver(onIntersect);
    }
    return ref.current;
  }

  // W razie potrzeby możesz wywołać getObserver()
  // ...
}
```

Pozwala to uniknąć tworzenia kosztownych obiektów do czasu, aż faktycznie będą potrzebne. Jeśli używasz Flow lub TypeScriptu, możesz dla pewności dodatkowo nadać funkcji `getObserver()` typ nie dopuszczający wartości `null` (ang. *non-nullable type*).

Czy hooki są wolne z powodu tworzenia funkcji podczas renderowania?

Nie. We współczesnych przeglądarkach wydajność domknięć w porównaniu z zastosowaniem klas nie różni się znacząco, za wyjątkiem sytuacji ekstremalnych.

Ponadto, warto zwrócić uwagę, że sposób działania hooków jest bardziej wydajny pod kilkoma względami:

- Hooki unikają sporej części narzutu, jaki wprowadzają klasy - jak choćby koszt tworzenia instancji klasy czy dowiązywanie procedur obsługi zdarzeń w konstruktorze.
- **Kod idiomatyczny używający hooków nie wymaga głębokiego zagnieżdżania drzewa komponentów**, co ma miejsce w kodzie korzystającym z komponentów wyższego rzędu (ang. *higher-order components*), właściwości renderujących (ang. *render props*) i kontekstu. W mniejszych drzewach komponentów React ma mniej do roboty.

Tradycyjnie już, obawy dotyczące wydajności dla *funkcji inline* w Reakcie były związane z sytuacjami, w których przekazywanie każdorazowo nowych funkcji zwrotnych do komponentów potomnych niwelowało optymalizację zapewnioną przez `shouldComponentUpdate` w potomkach. Hooki rozwiązują ten problem na trzy sposoby.

- Hook `useCallback` pozwala na przechowywanie tej samej referencji do funkcji zwrotnej pomiędzy kolejnymi renderowaniami, dzięki czemu metoda `shouldComponentUpdate` może działać poprawnie:

```
// Nie zmieni się, dopóki nie zmienią się `a` lub `b`
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

- Hook `useMemo` ułatwia kontrolowanie tego, kiedy aktualizowane są poszczególne komponenty potomne, zmniejszając potrzebę stosowania czystych komponentów (ang. *pure components*).
- Wreszcie hook `useReducer` zmniejsza potrzebę przekazywania funkcji zwrotnych do dalekich potomków, co wyjaśniono poniżej.

Jak unikać przekazywania funkcji zwrotnych w dół?

Z naszych obserwacji wynika, że programiści nie lubią ręcznego przekazywania funkcji zwrotnych w dół przez kilka poziomów drzewa komponentów. Nawet jeśli kod w ten sposób staje się bardziej bezpośredni, możemy odnieść wrażenie, że zbyt dużo czasu poświęcamy "hydraulice" programu.

W dużych drzewach komponentów sugerujemy przekazywać funkcję `dispatch`, zwróconą przez hooka `useReducer`, poprzez kontekst:

```
const TodosDispatch = React.createContext(null);

function TodosApp() {
  // Uwaga: `dispatch` nie zmieni się pomiędzy renderowaniami
  const [todos, dispatch] = useReducer(todosReducer);

  return (
    <TodosProvider value={{ todos, dispatch }}>
      <Router>
        <Switch>
          <Route path="/" component={Todos} />
        </Switch>
      </Router>
    </TodosProvider>
  );
}
```

```

<TodosDispatch.Provider value={dispatch}>
  <DeepTree todos={todos} />
</TodosDispatch.Provider>
);
}

```

Dowolny komponent poddrzewa wewnętrz `TodosApp` może użyć funkcji `dispatch`, aby uruchomić akcję z `TodosApp`:

```

function DeepChild(props) {
  // Jeśli chcemy wykonać jakąś akcję, możemy wyciągnąć funkcję `dispatch` z kontekstu.
  const dispatch = useContext(TodosDispatch);

  function handleClick() {
    dispatch({ type: 'add', text: 'zrobić pranie' });
  }

  return (
    <button onClick={handleClick}>Dodaj zadanie</button>
  );
}

```

Jest to wygodne zarówno z perspektywy utrzymania kodu (nie trzeba przekazywać funkcji zwrotnych w nieskończoność), jak i ogólnie uniknięcia problemów z tego typu funkcjami. Sugerujemy zatem wykonywać wszelkie "odległe" aktualizacje za pomocą przekazanej w dół funkcji `dispatch`.

Pamiętaj, że nadal możesz wybrać pomiędzy przekazywaniem *stanu* aplikacji w dół za pomocą właściwości (bardziej "wprost") lub za pomocą kontekstu (wygodniejsze w przypadku "odległych" aktualizacji). Jeśli chcesz użyć kontekstu także do przekazania stanu, sugerujemy skorzystać z dwóch niezależnych kontekstów. Kontekst przekazujący `dispatch` nigdy się nie zmienia, dzięki czemu używające go komponenty nie muszą być ponownie renderowane, o ile same w jawnym sposobie nie poproszą o stan.

Jak odczytywać często zmieniającą się wartość wewnętrz `useCallback`?

Uwaga

Zalecamy [przekazywać w dół funkcję `dispatch` za pomocą kontekstu](#), a nie poszczególne funkcje zwrotne za pomocą właściwości. Poniższy sposób został tu umieszczony tylko jako uzupełnienie i "furtka awaryjna".

W rzadkich przypadkach pojawia się potrzeba memoizowania funkcji zwrotnej za pomocą hooka `useCallback`, lecz nie przynosi to żadnej korzyści, ponieważ wewnętrzna funkcja i tak tworzona jest zbyt często. Jeśli memoizowana funkcja jest procedurą obsługi zdarzeń i nie jest wywoływana podczas renderowania, można stworzyć [referencję do zmiennej](#) i ręcznie aktualizować jej wartość:

```

function Form() {
  const [text, updateText] = useState('');
  const textRef = useRef();

```

```

useEffect(() => {
  textRef.current = text; // Nadpisz wartość referencji
});

const handleSubmit = useCallback(() => {
  const currentText = textRef.current; // Odczytaj wartość referencji
  alert(currentText);
}, [textRef]); // Nie twórz ponownie `handleSubmit`, jak byłoby przy `[text]`

return (
  <>
    <input value={text} onChange={e => updateText(e.target.value)} />
    <ExpensiveTree onSubmit={handleSubmit} />
  </>
);
}
}

```

Metoda ta może wydawać się mocno zagmatwana, lecz pokazuje, że można, w razie potrzeby, skorzystać z tego typu optymalizacji. Łatwiej z niej korzystać po wydzieleniu logiki do osobnego hooka:

```

function Form() {
  const [text, updateText] = useState('');
  // Podlega memoizacji, nawet gdy zmienia się `text`:
  const handleSubmit = useEventCallback(() => {
    alert(text);
  }, [text]);

  return (
    <>
      <input value={text} onChange={e => updateText(e.target.value)} />
      <ExpensiveTree onSubmit={handleSubmit} />
    </>
  );
}

function useEventCallback(fn, dependencies) {
  const ref = useRef(() => {
    throw new Error('Niedozwolone wywołanie procedury obsługi zdarzeń podczas renderowania.');
  });

  useEffect(() => {
    ref.current = fn;
  }, [fn, ...dependencies]);

  return useCallback(() => {
    const fn = ref.current;
    return fn();
  }, [ref]);
}

```

Tak czy inaczej, **nie zalecamy korzystania z tego sposobu**, a pokazujemy go tylko dla kompletności dokumentacji. Zamiast tego lepiej jest [unikać przekazywania funkcji zwrotnych głęboko w dół](#).

Pod maską

Jak React łączy wywołania hooków z komponentami?

React sprawuje kontrolę nad aktualnie renderowanym komponentem. Dzięki [zasadom korzystania z hooków](#) wiemy, że hooki mogą być wywoływanie tylko z wnętrza komponentów reaktywnych (lub własnych hooków -- które również można wywoływać tylko w komponentach reaktywnych).

Do każdego komponentu przypisana jest wewnętrzna lista "komórek pamięci". Są to zwykłe obiekty javascriptowe, w których przechowujemy jakieś dane. Kiedy wywołujesz hook, np. `useState()`, odczytuje on aktualną zawartość komórki (lub tworzy nową podczas pierwszego renderowania), a następnie przesywa "wskaźnik" na kolejną komórkę. To dzięki temu każde z kilku wywołań `useState()` może zarządzać niezależną porcją lokalnego stanu.

Skąd wziął się pomysł na stworzenie hooków?

Hooki łączą pomysły z wielu różnych źródeł:

- Nasze stare eksperymenty z funkcyjnymi API w repozytorium [react-future](#).
- Eksperymenty społeczności reaktywnej z interfejsami dla właściwości renderujących, wliczając w to [Reactions Component](#) autorstwa [Ryana Florence'a](#).
- Propozycję [Dominica Gannawaya](#) dotyczącą wprowadzenia [słowa kluczowego](#) `adopt` jako nowej składni dla właściwości renderujących.
- Zmienne stanu i komórki stanu w języku [DisplayScript](#).
- [Komponenty redukujące](#) w ReasonReact.
- [Subskrypcje](#) w Rx.
- [Efekty algebraiczne](#) w Multicore OCaml.

[Sebastian Markbåge](#) wymyślił pierwotny hook, który później został udoskonalony przez [Andrew Clarka](#), [Sophie Alpert](#), [Dominica Gannawaya](#) i innych członków zespołu Reacta.

Wprowadzenie do hooków

Hooki są nowym dodatkiem w Reakcie 16.8. Pozwalają one używać stanu i innych funkcjonalności Reacta, bez użycia klas.

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

function Example() {
  // Zadeklaruj nową zmienną stanu, którą nazwiemy „count”
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Naciśnięto {count} razy</p>
      <button onClick={() => setCount(count + 1)}>
        Naciśnij mnie
      </button>
    </div>
  );
}
```

Funkcja `useState` jest pierwszym „hookiem”, o którym będziemy się uczyć. Przykład ten jest jednak zaledwie zwiastunem. Nie przejmuj się, jeżeli nie ma to jeszcze większego sensu!

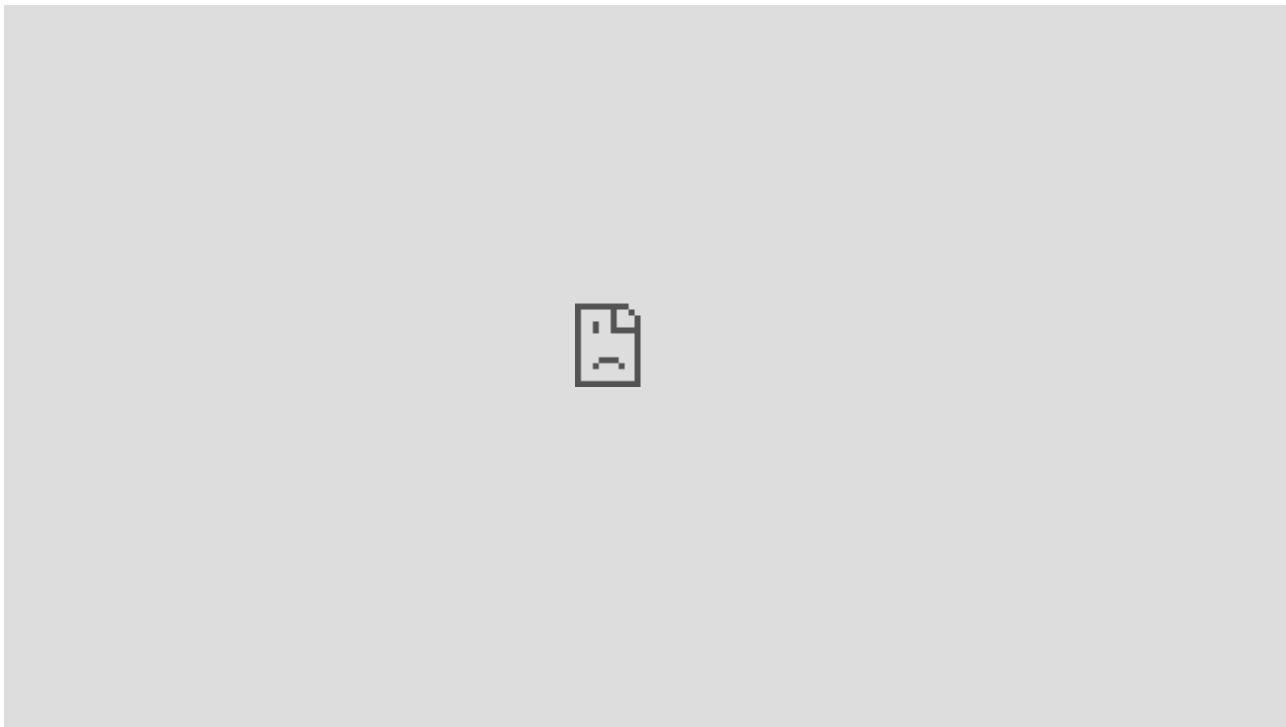
W kolejnym rozdziale możesz rozpoczęć naukę o hookach. Tutaj wyjaśnimy, dlaczego dodaliśmy hooki do Reacta i w jaki sposób pomogą ci one w pisaniu wspaniałych aplikacji.

Uwaga

React 16.8.0 jest pierwszą wersją, która wspiera hooki. Podczas aktualizacji nie zapomnij zaktualizować wszystkich paczek, w tym React DOM. React Native wspiera hooki [od wersji 0.59](#).

Wprowadzenie wideo

Podczas konferencji „React Conf 2018” Sophie Alpert i Dan Abramov zaprezentowali po raz pierwszy hooki. Następnie Ryan Florence zademonstrował, jak przepisać swoją aplikację, by móc ich używać. Wideo z konferencji zamieściliśmy poniżej:



Bez krytycznych zmian

Zanim przejdziemy dalej, zauważ że hooki są:

- **Całkowicie opcjonalne.** Możesz wypróbować hooki w kilku komponentach, bez przepisywania istniejącego kodu. Jeżeli jednak nie masz ochoty, nie musisz ich jeszcze stosować ani uczyć się o nich.
- **100% kompatybilne wstępnie.** Hooki nie zawierają żadnych zmian, które mogłyby zepsuć istniejący kod.
- **Dostępne już teraz.** Hooki są dostępne od wersji 16.8.0.

Nie ma planów na usunięcie klas z Reacta. Możesz przeczytać o strategii stopniowego wdrażania hooków w [kolejnym podrozdziale](#) tej strony.

Hooki nie zastępują twojej wiedzy na temat Reacta. Zamiast tego wprowadzają bardziej bezpośredni interfejs API dla mechanizmów Reacta, które już znasz: właściwości (ang. *props*), stanu, kontekstu, referencji (ang. *refs*) i cyklu życia (ang. *lifecycle*). Jak pokażemy dalej, hooki pozwalają też na łączenie ich w nowy, niezwykle skuteczny sposób.

Jeżeli chcesz rozpoczęć naukę o hookach, przejdź od razu do kolejnego rozdziału! Możesz też kontynuować lekturę tego, aby dowiedzieć się, dlaczego w ogóle dodaliśmy hooki, a także w jaki sposób będziemy je teraz stosować, bez potrzeby przepisywania naszych aplikacji.

Motywacja

Hooki rozwiązują wiele, pozornie niepowiązanych ze sobą, problemów Reacta, na które natknęliśmy się podczas ponad pięciu lat pisania i utrzymywania dziesiątek tysięcy komponentów. Nie ważne, czy dopiero uczysz się Reacta, używasz go na co dzień, czy nawet preferujesz inną bibliotekę (o podobnym, komponentowym modelu działania) - możliwe, że natknąłeś się na część tych problemów.

Współdzielenie logiki związanej ze stanem pomiędzy komponentami jest trudne

React nie oferuje sposobu na „dołączenie” powtarzanego zachowania do komponentu (na przykład, połączenie go z magazynem (ang. *store*)). Jeżeli pracujesz z Reactem już jakiś czas, najprawdopodobniej znasz wzorce, takie jak [właściwości renderujące \(ang. *render props*\)](#) i [komponenty wyższego rzędu \(ang. *higher-order components*\)](#), które próbują rozwiązać ten problem. Wzorce te wymagają jednak modyfikacji komponentów w momencie ich użycia, co może być niewygodne i powodować, że kod jest bardziej skomplikowany. Jeśli spojrzyj na typową aplikację napisaną w Reakcie przy pomocy narzędzia React DevTools, naprawdopodobnie ujrzyś tam „piekło” komponentów opakowujących (ang. *wrapper component*), dostawców (ang. *providers*), konsumentów (ang. *consumers*), komponentów wyższego rzędu, właściwości renderujących i innych abstrakcji. Moglibyśmy, co prawda, [filtrować je w DevToolsach](#), ale to tylko wskazuje na głębszy problem: React potrzebuje lepszego podstawowego mechanizmu do współdzielenia logiki związanej ze stanem.

Korzystając z hooków, możesz wydzielić logikę związaną ze stanem z komponentu. Dzięki czemu, nie wymaga on zależności przy testowaniu i jest łatwy w ponownym wykorzystaniu. **Hooki pozwalają na ponowne użycie logiki związanej ze stanem, bez konieczności zmiany hierarchii komponentów.** Sprawia to, że dzielenie się hookami pomiędzy wieloma komponentami lub ze społecznością jest proste.

Omówimy ten temat szerzej w rozdziale pt. „[Tworzenie własnych hooków](#)”.

Złożone komponenty stają się trudne do zrozumienia

Często musielibyśmy utrzymywać komponenty, które z początku proste, urosły do rangi niemożliwego do utrzymania bałaganu logiki związanej ze stanem i efektów ubocznych (ang. *side effects*). Każda metoda cyklu życia zawiera zwykle mieszankę niepowiązanej ze sobą logiki. Na przykład, komponenty mogą pobierać dane w `componentDidMount` i `componentDidUpdate`. Jednakże metoda `componentDidMount` może też zawierać logikę, która tworzy obserwatory zdarzeń (ang. *event listeners*). Następnie są one czyszczone w `componentWillUnmount`. Wzajemnie powiązany kod zostaje podzielony pomiędzy różne metody, a z kolei zupełnie niezwiązany ze sobą kod trafia do jednej. Sprzyja to niekonsekwencji i popełnianiu błędów.

Wielokrotnie zdarza się, że nie ma możliwości rozbicia tych komponentów na mniejsze części, ponieważ logika związana ze stanem jest już wszędzie. Trudno jest też je testować. Jest to jeden z powodów, dla których wielu woli połączyć Reacta z zewnętrzną biblioteką do zarządzania stanem. To jednak często wprowadza zbyt wiele abstrakcji, zmusza do skakania pomiędzy plikami i utrudnia ponowne wykorzystanie komponentów.

Aby rozwiązać ten problem, **hooki pozwalają podzielić komponent na mniejsze funkcje, bazując na powiązanych ze sobą częściach (takich jak tworzenie subskrypcji czy pobieranie danych)**, zamiast wymuszać sztuczny podział, związany z metodami cyklu życia. Ewentualnie, aby uczynić zachowanie komponentu bardziej przewidywalnym, możesz też dzięki hookom oddelegować zarządzanie lokalnym stanem komponentu do reduktora (ang. *reducer*).

Szerzej omówimy to w rozdziale [Używanie hooka efektów](#).

Klasy dezorientują zarówno ludzi, jak i maszyny

Oprócz tego, że przez klasy trudniej jest ponownie wykorzystać i organizować kod, odkryliśmy, że mogą one również stanowić dużą przeszkodę w nauce Reacta. Trzeba przecież rozumieć, jak działa `this` w JavaScriptie - a działa on tu zupełnie inaczej niż w większości języków programowania. Trzeba pamiętać o wiązaniu (ang. *bind*) funkcji obsługi zdarzeń (ang. *event handler*). Bez [publicznych pól klas w ES2022](#) kod jest bardzo rozwlekły. Ludzie generalnie nie mają problemu ze zrozumieniem właściwości, stanu i kierunku przepływu danych z góry do dołu, a jednak klasy wciąż stanowią pewne wyzwanie. Wybór pomiędzy funkcyjnymi a klasowymi komponentami jest często przyczyną sporów, nawet pomiędzy doświadczonymi programistami Reacta.

Ponadto, React jest dostępny od około pięciu lat i chcielibyśmy mieć pewność, że pozostanie on tak samo istotny przez kolejne pięć. Jak pokazały biblioteki: [Svelte](#), [Angular](#), [Glimmer](#) i inne, [komplikacja komponentów z wyprzedzeniem \(ang. ahead-of-time\)](#) ma ogromny potencjał. Szczególnie jeśli nie jest ograniczona tylko do szablonów. Niedawno eksperymentowaliśmy z [wyliczaniem wartości komponentów](#) z użyciem [Prepacka](#) i wstępnie zaobserwowaliśmy obiecujące rezultaty. Odkryliśmy jednak, że komponenty klasowe mogą zachęcić do nieumyślnego stosowania pewnych wzorców, które spowodują spowolnienie tych optymalizacji. Klasy już teraz stanowią problem dla naszych narzędzi programistycznych. Na przykład, nie za dobrze się minifikują i często zawodzą przez nie „gorące przeładowanie” (ang. hot reloading). Chcemy przedstawić interfejs API, który zwiększy prawdopodobieństwo tego, że kod będzie optymalizowany.

Aby rozwiązać te problemy, **Hooki pozwalają na korzystanie z większej liczby funkcjonalności Reacta, bez użycia klas.** Koncepcyjnie, komponentom reaktywym zawsze bliżej było do funkcji. Hooki zapewniają dostęp do funkcji bez poświęcania praktycznej natury Reacta. Hooki zapewniają dostęp do imperatywnych „furtek” i nie wymagają nauki skomplikowanych technik programowania funkcyjnego lub reaktywnego.

Przykłady

Rozdział pt. „[Hooki w pigułce](#)” jest dobrym miejscem, by zacząć naukę o hookach

Strategia Stopniowego Wdrażania

TLDR: Nie ma planów na usunięcie klas z Reacta.

Zdajemy sobie sprawę, że programiści Reacta są skupieni na dostarczaniu produktów i nie mają czasu przyglądać się każdemu nowemu interfejsowi API, który jest wypuszczany. Hooki są wielką nowością i być może lepiej będzie zaczekać na więcej przykładów i poradników, zanim rozważysz ich naukę lub wdrożenie.

Rozumiemy też, że przy dodawaniu do Reacta nowego, podstawowego mechanizmu poprzeczka została postawiona niezwykle wysoko. Dla zainteresowanych przygotowaliśmy [szczegółowy RFC](#), który dokładnie zgłębia nasze motywy oraz rzuca dodatkowe światło na konkretne decyzje projektowe i obecny stan techniki.

Co najważniejsze, hooki działają równolegle z istniejącym kodem, więc możesz wdrażać je stopniowo. Nie ma pośpiechu, aby migrować kod do hooków. Zalecamy unikanie „wielkiego przepisywania”, szczególnie dla istniejących, skomplikowanych komponentów klasowych. Potrzeba delikatnie przestawić myślenie, aby zacząć „myśleć hookami”. Z naszego doświadczenia wynika, że najlepiej poćwiczyć używanie hooków na nowych, niekrytycznych komponentach i upewnić się, że wszyscy członkowie zespołu czują się z nimi komfortowo. Po wypróbowaniu hooków, prosimy - [prześlij nam opinię](#). Zarówno pozytywną, jak i negatywną.

Chcielibyśmy, żeby hooki objęły wszystkie możliwe przypadki użycia klas, ale **w możliwej do przewidzenia przyszłości, będziemy kontynuować wsparcie komponentów klasowych.** W Facebooku mamy dziesiątki tysięcy komponentów napisanych jako klasy i zdecydowanie nie planujemy ich przepisywania. Zamiast tego zaczynamy używać hooków w nowym kodzie, równolegle do klas.

Najczęściej zadawane pytania

Przygotowaliśmy rozdział pt. „[Hooki - FAQ](#)”, w którym odpowiedzieliśmy na najczęściej zadawane pytania.

Kolejne kroki

Po przeczytaniu tego rozdziału powinien ukształtować ci się obraz tego, jakie problemy rozwiązują hooki, ale wiele szczegółów jest jeszcze prawdopodobnie niejasnych. Nie martw się! [Przejdzmy do następnego rozdziału, gdzie zaczniemy naukę o hookach na przykładach.](#)

[Go to TOC](#)

Hooki w pigułce

Hooki są nowym dodatkiem w Reakcie 16.8. Pozwalają one używać stanu i innych funkcjonalności Reacta, bez użycia klas.

Hooki są [kompatybilne wstępnie](#). Ten rozdział zawiera szybki przegląd wiedzy o hookach i przeznaczony jest dla doświadczonych użytkowników Reacta. Jeżeli w którymś momencie się zgubisz, szukaj żółtych ramek, takich jak ta poniżej:

Szczegółowe wyjaśnienie

Przeczytaj podrozdział pt. „[Motywacja](#)”, aby dowiedzieć się, dlaczego dodaliśmy hooki

↑ ↑ ↑ **Każdy podrozdział zakończony jest taką żółtą ramką.** Zawierają one linki do szczegółowych objaśnień.

Hook stanu

Ten przykład renderuje licznik. Kiedy wciskasz przycisk, zwiększa on wartość w stanie.

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

function Example() {
  // Zadeklaruj nową zmienną stanu, którą nazwiemy „count”
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Naciśnięto {count} razy</p>
      <button onClick={() => setCount(count + 1)}>
        Naciśnij mnie
      </button>
    </div>
  );
}
```

W tym przykładzie `useState` jest *hookiem* (za chwilę wyjaśnimy, co to znaczy). Wywołujemy go z wewnątrz komponentu funkcyjnego, aby wzbogacić go o lokalny stan. React zachowa ten stan pomiędzy kolejnymi renderowaniami. `useState` zwraca parę: *aktualną* wartość stanu i funkcję, która pozwala go aktualizować. Możesz wywołać tę funkcję w procedurze obsługi zdarzenia albo z innego miejsca. Działa to mniej więcej tak samo jak `this.setState` w komponencie klasowym, z tą różnicą, że nie scalą on starych i nowych wartości. (Szerzej omawiamy te różnice w rozdziale pt. „[Używanie hooka stanu](#)“.)

Jedynym argumentem funkcji `useState` jest stan początkowy. W przykładzie powyżej jest to `0`, ponieważ nasz licznik startuje od zera. Zauważ, że w przeciwieństwie do `this.state`, stan nie musi być obiektem. Nic jednak nie stoi na przeszkodzie, by nim był. Wartość argumentu ze stanem początkowym jest wykorzystywana tylko podczas pierwszego renderowania.

Deklaracja kilku zmiennych stanu

Możesz użyć hooka stanu kilka razy w tym samym komponencie:

```
function ExampleWithManyStates() {
  // Zadeklaruj kilka zmiennych stanu!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
  // ...
}
```

Dzięki składni przypisania destrukturyzującego tablic możemy nadać różne nazwy zmiennym stanu, które zadeklarowaliśmy wywołując funkcję `useState`. Nazwy te nie są częścią interfejsu API `useState`. Zamiast tego React zakłada, że jeżeli wywołujesz funkcję `useState` wielokrotnie, za każdym razem (przy każdym renderze) robisz to w tej samej kolejności. W dalszej części tego rozdziału omówimy, jak to działa i dlaczego jest to przydatne.

Ale czym jest hook?

Hooki są to funkcje, które pozwalają „zahaczyć się” w mechanizmy stanu i cyklu życia Reacta, z wewnętrz komponentów funkcyjnych. Hooki nie działają w klasach -- zamiast tego pozwalają korzystać z Reacta bez klas. (Nie zalecamy przepisywania istniejących komponentów z dnia na dzień, ale jeżeli masz ochotę, możesz zacząć korzystać z hooków w nowych komponentach.)

React dostarcza kilka wbudowanych hooków, między innymi `useState`. Ale możesz też stworzyć własne hooki, by współdzielić zachowanie związane ze stanem pomiędzy komponentami. Najpierw rzućmy jednak okiem na wbudowane hooki.

Szczegółowe wyjaśnienie

Hooki stanu poświęciliśmy cały rozdział: „[Używanie hooka stanu](#)”.

⚡ Hook efektów

Najprawdopodobniej zdarzyło ci się już pobierać dane, obsługiwać subskrypcje lub ręcznie modyfikować drzewo DOM wewnętrz komponentów reaktywnych. Tego typu operacje nazywamy „efektami ubocznyimi” (ang. *side effects*), ponieważ mogą one wpływać na inne komponenty i nie mogą zostać przeprowadzone podczas renderowania.

Hook efektów `useEffect` daje możliwość przeprowadzania „efektów ubocznych” z wewnętrz komponentu funkcyjnego. Pełni on taką samą funkcję, jak `componentDidMount`, `componentDidUpdate` i `componentWillUnmount` w komponentach klasowych, ale uproszczoną do jednego interfejsu API. (Różnice między nimi pokażemy na przykładach w rozdziale pt. „[Używanie hooka efektów](#)”.)

Na przykład, ten komponent ustawia tytuł dokumentu HTML po tym, jak React zaktualizuje drzewo DOM:

```

import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Podobnie jak componentDidMount i componentDidUpdate:
  useEffect(() => {
    // Zaktualizuj tytuł dokumentu, korzystając z interfejsu API przeglądarki
    document.title = `Naciśnięto ${count} razy`;
  });

  return (
    <div>
      <p>Naciśnięto {count} razy</p>
      <button onClick={() => setCount(count + 1)}>
        Naciśnij mnie
      </button>
    </div>
  );
}

```

Kiedy wywołujesz funkcję `useEffect`, mówisz Reactowi, żeby uruchomił twój „efekt”, gdy opróżni (ang. *flush*) bufor zmian do drzewa DOM. Efekty są deklarowane wewnątrz komponentu, więc mają dostęp do jego właściwości (ang. *props*) i stanu (ang. *state*). Domyślnie React uruchamia wszystkie efekty po każdym renderowaniu -- *włącznie* z pierwszym. (Różnice między tym a metodami cyklu życia komponentów klasowych omówimy w rozdziale pt. „[Używanie hooka efektów](#)”.)

Poprzez zwrócenie funkcji, efekty mogą też określać, w jaki sposób należy po nich „posprzątać”. Na przykład, ten komponent używa efektu, aby zasubskrybować się do informacji o dostępności znajomego. A następnie sprząta po sobie, anulując tę subskrypcję.

```

import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Ładowanie...';
  }
  return isOnline ? 'Dostępny' : 'Niedostępny';
}

```

W tym przykładzie React anuluje subskrypcję naszego `ChatAPI` w momencie, gdy komponent zostaje odmontowany. Zrobi to też przed każdym kolejnym uruchomieniem efektu (które następuje przy każdym kolejnym renderowaniu komponentu). (Jeśli chcesz, możesz [powiedzieć Reactowi żeby tego nie robił](#), jeśli wartość `props.friend.id`, którą przekazaliśmy do `ChatAPI`, nie uległa zmianie.)

Podobnie jak z `useState`, możesz użyć więcej niż jednego efektu w swoim komponencie:

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `Naciśnięto ${count} razy`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }
  // ...
}
```

Hooki pozwalają organizować kod wewnętrz komponentu, bazując na powiązanych ze sobą fragmentach (takich jak dodawanie i anulowanie subskrypcji). Nie wymuszają sztucznego podziału, jak metody cyklu życia (ang. *lifecycle methods*).

Szczegółowe wyjaśnienie

Możesz dowiedzieć się więcej o `useEffect` w rozdziale pt. „[Używanie hooka efektów](#)”.

👉 Zasady korzystania z hooków

Hooki są funkcjami javascriptowymi, ale narzucają dwie dodatkowe zasady:

- Wywołuj hooki tylko **z najwyższego poziomu kodu**. Nie wywołuj hooków z wewnętrz pętli, warunków czy zagnieżdżonych funkcji.
- Wywołuj hooki tylko **z wewnętrz reactowych komponentów funkcyjnych**. Nie wywołuj hooków z wewnętrz zwykłych javascriptowych funkcji. (Jest jeszcze tylko jedno miejsce, z którego możesz wywoływać hooki -- twoje własne hooki. Za chwilę dowiemy się o tym więcej.)

Stworzyliśmy [wtyczkę do lintera](#), która automatycznie wymusza te zasady. Rozumiemy, że reguły te mogą z początku wydawać się ograniczające i zagmatwane, ale są one niezbędne do prawidłowego funkcjonowania hooków.

Szczegółowe wyjaśnienie

Możesz dowieźć się więcej o tych zasadach w rozdziale pt. „[Zasady korzystania z hooków](#)”.

Tworzenie własnych hooków

Czasami zdarza się, że chcemy ponownie wykorzystać pewną logikę związaną ze stanem pomiędzy komponentami. W tradycyjnym podejściu mieliśmy do dyspozycji dwa popularne rozwiązania tego problemu: [komponenty wyższego rzędu \(ang. higher-order components\)](#) i [właściwości renderujące \(ang. render props\)](#). Własne hooki rozwiązują ten problem, bez konieczności dodawania kolejnych komponentów do drzewa.

W jednym z poprzednich podrozdziałów pokazaliśmy komponent `FriendStatus`, który wykorzystuje hooki `useState` i `useEffect`, aby zasubskrybować się do informacji o dostępności znajomego. Założmy, że chcielibyśmy wykorzystać tę logikę w innym komponencie.

Na początek wydzielmy tę logikę do własnego hooka o nazwie `useFriendStatus`:

```
import React, { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

Jako argument przyjmuje on zmienną `friendID` i zwraca informację o tym, czy nasz znajomy jest dostępny.

Teraz możemy go użyć w obu naszych komponentach:

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Ładowanie...';
  }
  return isOnline ? 'Dostępny' : 'Niedostępny';
}

function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

```

        </li>
    );
}

```

Stan każdego komponentu jest w pełni niezależny. Hooki są metodą na współdzielenie *logiki związanej ze stanem*, nie zaś samego stanu. Tak naprawdę to każde wywołanie hooka tworzy kompletnie wyizolowany stan -- możesz więc użyć tego samego, własnego hooka kilkukrotnie w jednym komponencie.

Własne hooki są bardziej konwencją niż wbudowanym mechanizmem. Jeżeli nazwa funkcji zaczyna się od „`use`” i wywołuje ona inne hooki, mówimy że mamy do czynienia z własnym hookiem. Na tej konwencji nazewnictwa `useSomething` bazuje nasza wtyczka do lintera i pozwala jej to znaleźć błędy w kodzie korzystającym z hooków.

Möesz pisać własne hooki, które obejmują szereg różnych przypadków - od obsługi animacji, deklaratywnych subskrypcji, liczników, po wiele innych, o których nie pomyśleliśmy. Nie możemy się doczekać, żeby zobaczyć, na jakie pomysły wykorzystania własnych hooków wpadnie społeczność Reacta.

Szczegółowe wyjaśnienie

Möesz dowiedzieć się więcej o własnych hookach w rozdziale pt. „[Tworzenie własnych hooków](#)”.

Inne hooki

Istnieje kilka mniej popularnych, wbudowanych hooków, które mogą ci się spodobać. Na przykład `useContext` pozwala zasubskrybować się do zmian kontekstu (ang. *context*) bez wprowadzania zagnieżdżania:

```

function Example() {
  const locale = useContext(LocaleContext);
  const theme = useContext(ThemeContext);
  // ...
}

```

Z kolei `useReducer` pozwala na zarządzanie lokalnym stanem skomplikowanych komponentów przy użyciu reduktora (ang. *reducer*):

```

function Todos() {
  const [todos, dispatch] = useReducer(todosReducer);
  // ...
}

```

Szczegółowe wyjaśnienie

Więcej informacji o wszystkich wbudowanych hookach znajdziesz w rozdziale pt. „[Hooki - interfejs API](#)”.

Kolejne kroki

Uff, to był wyczerpująco szybki przegląd! Jeżeli coś z tego, co napisaliśmy, nie miało dla ciebie sensu lub jeśli chcesz po prostu dowiedzieć się więcej, możesz przeczytać kolejne rozdziały, zaczynając od „[Używanie hooka stanu](#)”.

Możesz też zerkać na rozdziały pt. „[Hooki - interfejs API](#)” i „[Hooki - FAQ](#)”.

I na koniec: nie zapomnij o rozdziale pt. „[Wprowadzenie do hooków](#)”, w którym wyjaśniliśmy, dlaczego daliśmy hooki i jak zacząć z nich korzystać równolegle z istniejącym kodem -- bez przepisywania aplikacji.

Hooki - interfejs API

Hooki są nowym dodatkiem w Reakcie 16.8. Pozwalają one używać stanu i innych funkcjonalności Reacta, bez użycia klas.

Ten rozdział opisuje interfejs API hooków wbudowanych w Reacta.

Jeżeli pierwszy raz stykasz się z hookami, możesz najpierw sprawdzić rozdział pt. „[Hooki w pigułce](#)”. W rozdziale z [najczęściej zadawanymi pytaniami](#) możesz znaleźć inne przydatne informacje.

- [Podstawowe hooki](#)
 - `useState`
 - `useEffect`
 - `useContext`
- [Zaawansowane hooki](#)
 - `useReducer`
 - `useCallback`
 - `useMemo`
 - `useRef`
 - `useImperativeHandle`
 - `useLayoutEffect`
 - `useDebugValue`
 - `useDeferredValue`
 - `useTransition`
 - `useId`
- [Hooki dla bibliotek](#)
 - `useSyncExternalStore`
 - `useInsertionEffect`

Podstawowe hooki

`useState` {#usestate}

```
const [state, setState] = useState(initialState);
```

Zwraca zmienną przechowującą lokalny stan i funkcję do jego aktualizacji.

Podczas pierwszego renderowania zwrócona wartość stanu (`state`) jest taka sama jak wartość przekazana jako pierwszy argument (`initialState`).

Funkcja `setState` jest używana do aktualizacji stanu. Przyjmuje ona nową wartość stanu i kolejkuje ponowne renderowanie komponentu.

```
setState(newState);
```

Podczas kolejnych ponownych renderowań pierwszą wartością zwracaną przez `useState` będzie zawsze najnowszy, zaktualizowany stan.

Uwaga

React daje gwarancje, że funkcja `setState` jest tożsamościowa i że nie zmienia się podczas kolejnych renderowań. Dlatego też można ją bezpiecznie pominąć na liście zależności `useEffect` i `useCallback`.

Aktualizacje funkcyjne

Jeżeli nowy stan wyliczany jest z poprzedniego, możesz przekazać funkcję jako argument do `setState`. Funkcja otrzymuje poprzednią wartość stanu, a zwraca nową, zaktualizowaną wartość. Oto przykład komponentu licznika, który wykorzystuje obie formy `setState`:

```
function Counter({initialCount}) {
  const [count, setCount] = useState(initialCount);
  return (
    <>
      Licznik: {count}
      <button onClick={() => setCount(initialCount)}>Zresetuj</button>
      <button onClick={() => setCount(prevCount => prevCount - 1)}>-</button>
      <button onClick={() => setCount(prevCount => prevCount + 1)}>+</button>
    </>
  );
}
```

Przyciski „+” i „-” wykorzystują formę funkcyjną, ponieważ zaktualizowana wartość bazuje na poprzedniej. Z kolei przycisk „Zresetuj” używa normalnej formy, ponieważ zawsze przywraca początkową wartość.

Jeśli Twoja funkcja aktualizująca zwróci wartość identyczną z aktualnym stanem, nie spowoduje to ponownego wyrenderowania.

Uwaga

W przeciwieństwie do metody `setState` znanej z komponentów klasowych, funkcja `useState` nie scalą automatycznie obiektów reprezentujących aktualizację. Możesz powieścić to zachowanie, łącząc formę aktualizacji funkcyjnej ze składnią operatora rozszczepienia (ang. *spread operator*) obiektu:

```
const [state, setState] = useState({});
setState(prevState => {
  // Object.assign również zadziała
  return {...prevState, ...updatedValues};
});
```

Inną opcją jest hook `useReducer`, który jest bardziej odpowiedni do zarządzania obiektami stanów, zawierającymi wiele pod-wartości.

Leniwa inicjalizacja stanu

Argument `initialState` jest wartością stanu używaną podczas pierwszego rendera. W kolejnych renderowaniach jest on pomijany. Jeśli początkowy stan jest wynikiem kosztownych obliczeń, możesz zamiast tego przekazać funkcję, która zostanie wykonana tylko przy pierwszym renderowaniu:

```
const [state, setState] = useState(() => {
  const initialState = someExpensiveComputation(props);
  return initialState;
});
```

Wycofanie się z aktualizacji stanu

Jeżeli zaktualizujesz hook stanu do takiej samej wartości, jaką jest aktualnie przechowywana w stanie, React „wymiga się”, nie aktualizując potomków i nie uruchamiając efektów. (React używa [algorytmu porównywania `Object.is`](#).)

Pamiętaj, że React może nadal wymagać wyrenderowania tego konkretnego komponentu, zanim wymiga się od dalszych zmian. Nie powinno to być problemem, ponieważ React nie będzie niepotrzebnie wchodził „głębiej” w drzewo. Jeśli wykonujesz kosztowne obliczenia podczas renderowania, możesz je zoptymalizować za pomocą `useMemo`.

Grupowanie zmian stanu

React w celach optymalizacyjnych potrafi zgrupować kilka zmian stanu, powodując tylko jedno ponowne renderowanie. W większości przypadków zwiększa to szybkość aplikacji i nie powinno wpływać na zachowanie twojej aplikacji.

Przed Reactem 18 grupowane były wyłącznie aktualizacje stanu wywołane z procedur obsługi zdarzeń (ang. *event handlers*). Od wersji React 18, [grupowanie jest włączone domyślnie dla wszystkich aktualizacji](#). Pamiętaj jednak, że React sam upewnia się, aby zmiany z kilku różnych zdarzeń zainicjowanych przez użytkownika (np. poprzez kliknięcie na przycisk dwukrotnie) zawsze były przetwarzane osobno i nigdy nie były grupowane. Zapobiega to błędom w logice.

W rzadkich sytuacjach, kiedy zajdzie potrzeba wymuszenia synchronicznej aktualizacji DOM, możesz skorzystać z `flushSync`. Pamiętaj jednak, że taki zabieg wiąże się z spadkiem wydajności, więc korzystaj z niego z rozwagą.

`useEffect` (#useeffect)

```
useEffect(didUpdate);
```

Przyjmuje funkcję zawierającą imperatywny kod, mogący zawierać efekty uboczne.

W ciele głównej funkcji komponentu (określonej jako *faza renderowania* Reacta) nie jest dozwolone modyfikowanie danych, tworzenie subskrypcji, timerów, logowanie i inne efekty uboczne. Robiąc tak należy spodziewać się mylących błędów i niespójności w interfejsie użytkownika.

Zamiast tego użyj `useEffect`. Funkcja przekazana do `useEffect` zostanie uruchomiona po tym, jak zmiany zostaną wyświetcone na ekranie. Traktuj efekty jako furtkę pomiędzy czysto funkcyjnym światem Reacta, a imperatywnym światem.

Domyślnie efekty są uruchamiane po każdym wyrenderowaniu komponentu, ale możesz sprawić, że uruchomią się [tylko jeżeli zmienią się jakieś wartości](#).

Czyszczenie po efekcie

Często efekty tworzą zasoby (np. subskrypcję czy ID timera), które należy uprątnąć zanim komponent opuści ekran. Aby to uczynić funkcja przekazywana do `useEffect` może zwracać funkcję czyszczącą. Na przykład przy tworzeniu subskrypcji:

```
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // Uprątnij subskrypcję
    subscription.unsubscribe();
  };
});
```

Aby zapobiec wyciekom pamięci, funkcja czyszcząca wywoływana jest zanim komponent zostanie usunięty z interfejsu użytkownika. Dodatkowo jeśli komponent jest renderowany kilkukrotnie (co zwykle ma miejsce), **poprzedni efekt jest czyszczony przed wywołaniem kolejnego efektu**. W naszym przykładzie oznacza to, że nowa subskrypcja tworzona jest przy każdej aktualizacji. W kolejnym podrozdziale opisujemy, jak uniknąć wywoływania efektów przy każdej aktualizacji.

Momenty wywoływania efektów

W przeciwieństwie do metod `componentDidMount` i `componentDidUpdate` funkcja przekazana do `useEffect` wywołana zostanie **po tym**, jak skomponowany i namalowany zostanie układ strony. Sprawia to, że nadaje się ona do obsługi wielu typowych efektów ubocznych, takich jak tworzenie subskrypcji czy obsługa zdarzeń. Większość tego typu operacji nie powinna powstrzymywać przeglądarki przed odświeżeniem widoku.

Jednakże nie wszystkie efekty mogą zostać odroczone. Na przykład manipulacja drzewem DOM, widoczna dla użytkownika, musi zostać wywołana synchronicznie przed każdym malowaniem, tak aby użytkownik nie dostrzegł wizualnej niespójności. (Rozróżnienie to w swej koncepcji podobne jest do pasywnych i aktywnych obserwatorów zdarzeń (ang. *event listeners*).) Dla tego typu efektów React przewiduje dodatkowy hook, nazwany `useLayoutEffect`. Ma on tę samą sygnaturę co `useEffect`, różnie się jedynie tym, kiedy jest wywoływany.

Począwszy od Reacta 18, funkcja przekazana do `useEffect` jest wykonywana synchronicznie **przed** skomponowaniem i namalowaniem układu na stronie, jeżeli wywołanie jej wynika z akcji użytkownika, np. kliknięcia, lub jeśli jest wynikiem aktualizacji opakowanej w `flushSync`. Takie zachowanie pozwala na to, by wynik efektu był obserwowalny przez system zdarzeń albo przez kod wywołujący `flushSync`.

Uwaga

Dotyczy to jedynie momentu wywołania funkcji przekazanej do `useEffect` - aktualizacje zaplanowane wewnątrz efektu nadal są odraczane. Różni się to od `useLayoutEffect`, który wywołuje funkcję i przetwarza aktualizacje natychmiast.

Warunkowe uruchamianie efektów

Domyślnym zachowaniem efektów jest ich uruchamianie po każdym pomyślnym renderze. W ten sposób efekt jest zawsze tworzony na nowo, jeśli zmieni się jedna z jego zależności.

Jednakże w pewnych przypadkach może się to okazać zabójcze - choćby w przykładzie subskrypcji z poprzedniego podrozdziału. Nie ma potrzeby tworzyć nowej subskrypcji przy każdej aktualizacji, a jedynie wtedy, gdy zmieni się właściwość `source`.

Aby zaimplementować takie zachowanie należy przekazać do `useEffect` drugi argument, będący tablicą wartości, od których zależy efekt. Nasz zaktualizowany przykład wygląda następująco:

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

Teraz subskrypcja zostanie stworzona ponownie tylko wtedy, gdy zmieni się właściwość `props.source`.

Uwaga

Jeśli korzystasz z tej techniki optymalizacji, upewnij się, że przekazywana tablica zawiera **wszystkie wartości z zasięgu komponentu (takie jak właściwości (ang. `props`) i stan), które zmieniają się w czasie i które są używane przez efekt**. W przeciwnym razie twój kod odwoła się do starych wartości z poprzednich renderowań. Przeczytaj też, [jak radzić sobie z funkcjami i co robić, gdy tablica zmienia się zbyt często](#).

Jeśli chcesz przeprowadzić efekt i posprzątać po nim tylko raz (podczas montowania i odmontowania), możesz przekazać pustą tablicę (`[]`) jako drugi argument. Dzięki temu React wie, że twój efekt nie zależy od *jakichkolwiek* wartości właściwości lub stanu, więc nigdy nie musi być ponownie uruchamiany. Nie jest to traktowane jako przypadek specjalny -- wynika to bezpośrednio z tego, jak zawsze działa tablica wejść.

Jeśli przekażesz pustą tablicę (`[]`) właściwości i stan wewnątrz efektu zawsze przyjmą swoje początkowe wartości. Pomimo że przekazywanie `[]` jest bliższe temu, co znamy z metod `componentDidMount` i `componentWillUnmount`, zwykle istnieją [lepsze rozwiązania](#) pomagające uniknąć zbyt częstego powtarzania efektów. Nie zapominaj też, że React opóźni wywołanie `useEffect` do momentu, aż przeglądarka nie skończy rysować widoku, więc dodatkowa praca tutaj nie jest dużym problemem.

Polecamy użycie reguły `exhaustive-deps`, będącej częścią naszego pakietu `eslint-plugin-react-hooks`. Ostrzega ona, gdy zależności są niepoprawnie zdefiniowane i sugeruje poprawki.

Tablica zależności nie jest przekazywana jako argumenty do funkcji efektu. Koncepcyjnie jednak to właśnie przedstawiają: każda wartość, do której odwołuje się funkcja efektu, powinna również pojawić się w tablicy zależności. W przyszłości dostatecznie zaawansowany kompilator mógłby automatycznie tworzyć tę tablicę.

`useContext` {#usecontext}

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

Przyjmuje obiekt kontekstu (wartość zwroconą przez `React.createContext`) i zwraca jego aktualną wartość. Wartość kontekstu jest określana przez właściwość (ang. *prop*) `value` najbliższego rodzica `<MyContext.Provider>` wywołującego komponentu.

Kiedy najbliższy rodzic `<MyContext.Provider>` zostanie zaktualizowany, ten hook wywoła ponowne renderowanie komponentu z najnowszą wartością kontekstową `value` przekazaną dostawcy (ang. *provider*) `MyContext`. Nawet jeśli któryś z rodziców korzysta z funkcji `React.memo` lub metody `shouldComponentUpdate`, nastąpi ponowne wyrenderowanie poddrzewa, począwszy od komponentu korzystającego z `useContext`.

Pamiętaj, że argument przekazany do `useContext` musi być *samym obiektem kontekstu*:

- **Poprawnie:** `useContext(MyContext)`
- **Niepoprawnie:** `useContext(MyContext.Consumer)`
- **Niepoprawnie:** `useContext(MyContext.Provider)`

Komponent wywołujący `useContext` będzie zawsze ponownie renderowany jeśli zmieni się wartość kontekstu. Jeżeli ponowne renderowanie danego komponentu jest kosztowne, możesz [zoptymalizować to zachowanie](#), używając techniki zapamiętywania (ang. *memoization*).

Podpowiedź

Jeśli znasz już interfejs API kontekstu -- `useContext(MyContext)` jest odpowiednikiem klasowego `static contextType = MyContext` lub też `<MyContext.Consumer>`.

`useContext(MyContext)` pozwala tylko na *czytanie* kontekstu i nasłuchiwanie jego zmian. Wciąż wymagane jest aby w drzewie, ponad komponentem, znalazła się `<MyContext.Provider>` by mógł *dostarczyć* (ang. *provide*) wartość tego kontekstu.

W całości z użyciem `Context.Provider` wygląda to tak:

```
const themes = {
  light: {
    foreground: "#000000",
    background: "#eeeeee"
```

```

    },
    dark: {
      foreground: "#ffffff",
      background: "#222222"
    }
};

const ThemeContext = React.createContext(themes.light);

function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);

  return (
    <button style={{ background: theme.background, color: theme.foreground }}>
      Mam style z motywu kontekstowego!
    </button>
  );
}

```

Ten przykład został przygotowany pod hooki w oparciu o kod z rozdziału pt. [Zaawansowany poradnik dot. kontekstów](#), w którym znajdziesz więcej informacji o tym, kiedy i jak używać kontekstów.

Zaawansowane hooki

Poniższe hooki są albo są wariantami tych podstawowych, z poprzedniego podrozdziału, albo są stosowane tylko w określonych skrajnych wypadkach. Nie stresuj się na myśl o nauce o nich.

`useReducer {#usereducer}`

```
const [state, dispatch] = useReducer(reducer, initialArg, init);
```

Alternatywa dla hooka `useState`. Przyjmuje reduktor (ang. *reducer*), będący funkcją o sygnaturze `(stan, akcja) => nowyStan`, a zwraca aktualny stan w parze z metodą `dispatch`. (Jeżeli znasz bibliotekę Redux, wiesz już, jak to działa.)

`useReducer` sprawdza się lepiej od `useState` tam, gdzie występuje skomplikowana logika związana ze stanem, obejmująca wiele pod-wartości lub gdy następny stan zależy od poprzedniego. `useReducer` pozwala też zoptymalizować wydajność komponentów uruchamiających głębokie aktualizacje, ponieważ zamieścić przekazywać funkcje zwrotne (ang. *callback*), możesz przekazać funkcję `dispatch` w dół drzewa.

Oto przykład licznika z podrozdziału `useState` przepisany z użyciem reduktora:

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Licznik: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

Uwaga

React daje gwarancje, że funkcja `dispatch` jest tożsamościowa i że nie zmienia się podczas kolejnych renderowań. Dlatego też można ją bezpiecznie pominąć na liście zależności `useEffect` i `useCallback`.

Określanie stanu początkowego

Istnieją dwa sposoby na inicjalizację stanu `useReducer`. W zależności od potrzeb, możesz wybrać jeden z nich. Najprostszym sposobem jest przekazanie początkowego stanu, jako drugi argument:

```
const [state, dispatch] = useReducer(
  reducer,
  {count: initialCount}
);
```

Uwaga

React nie używa spopularyzowanej przez Reduxa konwencji argumentu `state = initialState`. Może się zdarzyć, że początkowa wartość zależy od właściwości (ang. *props*), a więc jest ona określana przy wywoływaniu hooka. Nie zalecamy, ale jeśli naprawdę musisz, możesz wywołać `useReducer(reducer, undefined, reducer)`, aby zasymulować zachowanie Reduxa.

Leniwa inicjalizacja

Możesz też leniwe zainicjalizować stan początkowy. Aby to zrobić, musisz przekazać funkcję inicjalizującą `init`, jako trzeci argument. Początkowy stan zostanie ustawiony na wynik wywołania `init(initialArg)`.

Pozwala to na wyodrębnić logikę dotyczącą obliczania stanu początkowego poza reduktorem. Jest to też przydatne do późniejszego resetowania stanu, w odpowiedzi na akcję:

```
function init(initialCount) {
  return {count: initialCount};
}

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    case 'reset':
      return init(action.payload);
    default:
      throw new Error();
  }
}

function Counter({initialCount}) {
  const [state, dispatch] = useReducer(reducer, initialCount, init);
  return (
    <>
      Licznik: {state.count}
      <button
        onClick={() => dispatch({type: 'reset', payload: initialCount})}>
        Zresetuj
      </button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

Wycofanie się z posłania akcji

Jeżeli reduktor zwróci tę samą wartość, jaką aktualnie przyjmuje stan, React „wymiga się” od aktualizacji potomków i uruchomienia efektów. (React używa algorytmu porównywania `Object.is`.)

Pamiętaj, że React może nadal wymagać wyrenderowania tego konkretnego komponentu, zanim wymiga się od dalszych zmian. Nie powinno to być problemem, ponieważ React nie będzie niepotrzebnie wchodził „głębiej” w drzewo. Jeśli wykonujesz kosztowne obliczenia podczas renderowania, możesz je zoptymalizować za pomocą `useMemo`.

`useCallback` {#usecallback}

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

```
},
[a, b],
);
```

Zwraca [zapamiętaną](#) (ang *memoized*) funkcję zwrotną (ang. *callback*).

Przekaż funkcję zwrotną i tablicę zależności. `useCallback` zwróci zapamiętaną wersję funkcji, która zmieni się tylko wtedy, gdy zmieni się któraś z zależności. Jest to przydatne podczas przekazywania funkcji zwrotnych do zoptymalizowanych komponentów podrzędnych, które opierają się na równości referencji, aby zapobiec niepotrzebnym renderowaniom (np. `shouldComponentUpdate`).

`useCallback(fn, deps)` jest równoważne `useMemo(() => fn, deps)`.

Uwaga

Tablica zależności nie jest przekazywana jako argumenty do funkcji zwrotnej. Koncepcyjnie jednak to właśnie przedstawiają: każda wartość, do której odwołuje się funkcja zwrotna, powinna również pojawić się w tablicy zależności. W przyszłości dostatecznie zaawansowany kompilator mógłby automatycznie tworzyć tę tablicę.

Polecamy użycie reguły `exhaustive-deps`, będącej częścią naszego pakietu `eslint-plugin-react-hooks`. Ostrzega ona, gdy zależności są niepoprawnie zdefiniowane i sugeruje poprawki.

useMemo {#usememo}

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Zwraca [zapamiętaną](#) (ang *memoized*) wartość.

Przekaż funkcję tworzącą i tablicę zależności. `useMemo` obliczy ponownie zapamiętaną wartość tylko wtedy, gdy zmieni się któraś z zależności. Ta optymalizacja pozwala uniknąć kosztownych obliczeń przy każdym renderowaniu.

Pamiętaj, że funkcja przekazana do `useMemo` uruchamiana jest podczas renderowania. Nie należy w niej robić niczego, czego normalnie nie robiono by podczas renderowania. Na przykład wszelkie efekty uboczne przynależą do `useEffect`, a nie `useMemo`.

Jeśli nie zostanie przekazana żadna tablica, nowa wartość będzie obliczana przy każdym renderowaniu.

Możesz traktować `useMemo` jako metodę optymalizacji wydajności, nie jako semantyczną gwarancję. W przyszłości React może zdecydować się „zapomnieć” niektóre z wcześniejszych zapamiętywanych wartości i ponownie obliczyć je przy kolejnym renderowaniu, np. aby zwolnić pamięć dla komponentów znajdujących się poza ekranem. Pisz swój kod tak, aby działał bez użycia hooka `useMemo`, a następnie dodaj go aby zoptymalizować wydajność.

Uwaga

Tablica zależności nie jest przekazywana jako argumenty do funkcji zwrotnej. Koncepcyjnie jednak to właśnie przedstawiają: każda wartość, do której odwołuje się funkcja zwrotna, powinna również pojawić się w tablicy zależności. W przyszłości dostatecznie zaawansowany kompilator mógłby automatycznie tworzyć tę tablicę.

Polecamy użycie reguły `exhaustive-deps`, będącej częścią naszego pakietu `eslint-plugin-react-hooks`. Ostrzega ona, gdy zależności są niepoprawnie zdefiniowane i sugeruje poprawki.

`useRef` {#useref}

```
const refContainer = useRef(initialValue);
```

`useRef` zwraca mutowalny (ang. *mutable*) obiekt, którego właściwość `.current` jest inicjalizowana wartością przekazaną jako argument (`initialValue`). Zwrócony obiekt będzie trwał przez cały cykl życia komponentu.

Częstym przypadkiem użycia jest imperatywny dostęp do potomka:

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` wskazuje na zamontowany element kontrolki formularza
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Ustaw skupienie na kontrolce
    formularza</button>
    </>
  );
}
```

Zasadniczo `useRef` jest jak „pudełko”, które może przechowywać mutowalną wartość w swojej właściwości `.current`.

Być może znasz już referencje (ang. `ref`) jako sposób na [dostęp do drzewa DOM](#). Jeśli przekażesz obiekt referencji do Reacta, korzystając ze składni `<div ref={myRef} />`, React ustawi jego właściwość `.current` na odpowiedni węzeł drzewa DOM przy każdej zmianie tego węzła.

Jednakże hook `useRef()` jest przydatny nie tylko jako atrybut `ref`. Jest też [przydatną metodą na przechowywanie mutowalnej wartości](#), podobną do właściwości instancji w przypadku komponentów klasowych.

Sposób ten zadziała ponieważ `useRef` tworzy czysty javascriptowy obiekt. Jedyną różnicą pomiędzy wywołaniem `useRef()`, a samodzielnym tworzeniem obiektu `{current: ...}` jest to, że `useRef`wróci referencję tego samego obiektu przy każdym renderowaniu.

Miej na uwadze fakt, że `useRef` *nie* informuje o tym, że jego wartość się zmieniła. Zmiana (mutowanie) właściwości `.current` nie spowoduje ponownego renderowania. Jeżeli chcesz uruchomić jakiś kod, w momencie gdy React dopina i odpina referencje do węzła DOM, możesz zamiast tego użyć [funkcji zwrotnej](#).

useImperativeHandle {#useimperativehandle}

`useImperativeHandle(ref, createHandle, [deps])`

`useImperativeHandle` dostosowuje wartość instancji, jaka przekazywana jest komponentom przodków, kiedy używają właściwości `ref`. Jak zwykle zalecamy aby w większości przypadków unikać imperatywnego kodu korzystającego z referencji. `useImperativeHandle` powinien być użyty w parze z `forwardRef`:

```
function FancyInput(props, ref) {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} ... />;
}
FancyInput = forwardRef(FancyInput);
```

W tym przykładzie komponent rodzica, który renderuje `<FancyInput ref={inputRef} />` będzie w stanie wywołać `inputRef.current.focus()`.

useLayoutEffect {#uselayouteffect}

Sygnatura funkcji jest taka sama jak `useEffect`, ale jest ona wywoływana synchronicznie po nałożeniu zmian na drzewo DOM. Użyj tego hooka, aby odczytać układ (ang. *layout*) z drzewa DOM i synchronicznie wyrenderować komponent ponownie. Bufor `useLayoutEffect` zostanie opróżniony synchronicznie, zanim przeglądarka będzie miała szansę na malowanie.

Kiedy to tylko możliwe używaj `useEffect` aby uniknąć blokujących aktualizacji wizualnych.

Podpowiedź

Jeżeli przeprowadzasz migrację kodu z komponentu klasowego musisz wiedzieć, że `useLayoutEffect` uruchamiany jest w tych samych fazach, co `componentDidMount` i `componentDidUpdate`. Jednakże **zalecamy zacząć od useEffect i używać useLayoutEffect tylko, jeżeli tamta metoda spowoduje jakieś problemy.**

Jeżeli używasz renderowania po stronie serwera pamiętaj, że ani `useLayoutEffect`, ani `useEffect` nie może działać dopóki kod JavaScript nie zostanie pobrany. Dlatego React ostrzega jeżeli komponent renderowany po stronie serwera zawiera `useLayoutEffect`. Aby to naprawić możesz albo przenieść logikę korzystającą z `useEffect` (jeżeli nie jest niezbędna przy pierwszym renderze), albo opóźnić wyświetlanie tego komponentu do czasu renderowania po stronie klienta (jeżeli kod HTML wygląda na popusty przed uruchomieniem `useLayoutEffect`).

Aby wyłączyć komponent, który korzysta z tego rodzaju efektów z wyrenderowanego po stronie serwera kodu HTML, wyrenderuj go warunkowo, korzystając z zapisu `showChild && <Child />` i opóźnij jego wyświetlanie przy użyciu `useEffect(() => { setShowChild(true); }, []);`. W ten sposób interfejs użytkownika nie będzie wyglądał na zepsuty przed hydracją (ang. *hydration*).

`useDebugValue` {#usedebugvalue}

`useDebugValue(value)`

`useDebugValue` może zostać użyty do wyświetlania etykiet dla własnych hooków w narzędziu React DevTools.

Rozważ na przykład własny hook `useFriendStatus` opisywany w rozdziale "Tworzenie własnych hooków":

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  // ...

  // Wyświetl etykietę w narzędziu DevTools obok tego hooka
  // np. "FriendStatus: Online"
  useDebugValue(isOnline ? 'Online' : 'Offline');

  return isOnline;
}
```

Podpowiedź

Nie zalecamy dodawania „debugowych” wartości każdemu własnemu hookowi. Jest to najbardziej przydatne w przypadku hooków będących częścią współdzielonych bibliotek.

Odroczenie formatowania „debugowych” wartości

W pewnych przypadkach formatowanie wartości może być kosztowną operacją. Jest też zbędne, dopóki hook nie jest rzeczywiście sprawdzany w narzędziach deweloperskich.

Dlatego też `useDebugValue` przyjmuje opcjonalnie jako drugi argument funkcję formatującą. Funkcja ta jest wywoływana tylko wtedy, gdy hooki są sprawdzane w narzędziach deweloperskich. Przyjmuje jako argument „debugową” wartość, a powinna zwrócić sformatowaną wartość.

Na przykład własny hook zwracający obiekt `Date` mógłby uniknąć niepotrzebnego wywołania funkcji `toDateString` poprzez przekazanie następującej funkcji formatującej:

`useDebugValue(date, date => date.toDateString());`

`useDeferredValue` {#usedeferredvalue}

`const deferredValue = useDeferredValue(value);`

`useDeferredValue` przyjmuje wartość i zwraca nową kopię oryginalnej wartości, jednak obliczaną z mniejszym priorytetem. Jeśli dany cykl renderowania został wywołany przez pilną aktualizację, np. akcję użytkownika, React zwróci poprzednią wartość, a nową obliczy, gdy owa pilna aktualizacja się zakończy.

Ten hook przypomina często spotykane hooki korzystające z mechanizmu odbicia (ang. *debounce*) czy dławienia (ang. *throttling*). Zaletą korzystania z `useDeferredValue` jest to, że React zakończy aktualizację, kiedy tylko zakończy się inna, bardziej pilna operacja (zamiast czekać określony czas). Ponadto, podobnie jak w przypadku `startTransition`, odroczone wartości mogą zawieszać (ang. *suspend*) działanie komponentu bez aktywowania niespodziewanych elementów zastępczych (ang. *fallback*).

Memoizacja odroczonych potomków

`useDeferredValue` odraca tylko tę wartość, którą mu przekażesz. Jeśli chcesz zapobiec ponownemu renderowaniu komponentu potomnego podczas pilnej aktualizacji, musisz go zmemoizować za pomocą `React.memo` lub `React.useMemo`:

```
function Typeahead() {
  const query = useSearchQuery('');
  const deferredQuery = useDeferredValue(query);

  // Memoizacja informuje Reacta, żeby wyrenderował ponownie tylko wtedy, gdy
  // zmieni się deferredQuery,
  // a nie samo query.
  const suggestions = useMemo(() =>
    <SearchSuggestions query={deferredQuery} />,
    [deferredQuery]
  );

  return (
    <>
      <SearchInput query={query} />
      <Suspense fallback="Ładowanie wyników...">
        {suggestions}
      </Suspense>
    </>
  );
}
```

Memoizowanie potomków informuje Reacta, że ma je zaktualizować tylko wtedy, gdy zmieni się `deferredQuery`, a nie `query`. Nie jest to jednak zachowanie unikalne dla `useDeferredValue`; podobny wzorzec jest stosowany w hookach korzystających z mechanizmów (ang. *debounce*) czy dławienia (ang. *throttling*).

useTransition {#usetransition}

```
const [isPending, startTransition] = useTransition();
```

Zwraca stan informujący o tym, czy transzycja (ang. *transition*) jest jeszcze w toku, oraz funkcję, która pozwala uruchomić ją uruchomić.

`startTransition` pozwala oznaczyć aktualizacje stanu jako transzycję:

```
startTransition(() => {
  setCount(count + 1);
})
```

Zmienna `isPending` informuje, czy transzycja jest w toku, aby można było na jej podstawie wyświetlić stan ładowania:

```
function App() {
  const [isPending, startTransition] = useTransition();
  const [count, setCount] = useState(0);

  function handleClick() {
    startTransition(() => {
      setCount(c => c + 1);
    })
  }

  return (
    <div>
      {isPending && <Spinner />}
      <button onClick={handleClick}>{count}</button>
    </div>
  );
}
```

Uwaga:

Aktualizacje stanu oznaczone jako transzycje ustępują pierwszeństwa pilniejszym aktualizacjom, np. spowodowanym kliknięciem przez użytkownika.

Aktualizacje zawarte w transzycji nie aktywują elementu zastępczego (ang. *fallback*) dla ponownie zawieszonych (ang. *suspended*) komponentów. Dzięki temu użytkownik może nadal wchodzić w interakcję z aktualną zawartością aplikacji, podczas gdy w tle przygotowywana jest nowa wersja.

useId {#useid}

```
const id = useId();
```

`useId` służy do generowania unikalnych ID, które mają gwarancję stabilności pomiędzy serwerem i klientem, co pozwala uniknąć nieścisłości podczas hydratacji (ang. *hydration*).

Uwaga

Hook `useId` **nie służy** do generowania [kluczy w listach](#). Klucze powinny być generowane na podstawie danych.

Dla przykładu, możesz przekazać wygenerowane w ten sposób `id` bezpośrednio do komponentów, które go potrzebują:

```
function Checkbox() {
  const id = useState();
  return (
    <>
      <label htmlFor={id}>Lubisz Reacta?</label>
      <input id={id} type="checkbox" name="react"/>
    </>
  );
};
```

Jeśli potrzebujesz kilku ID dla tego samego komponentu, dopisz ręcznie przyrostek za wartością `id`:

```
function NameFields() {
  const id = useState();
  return (
    <div>
      <label htmlFor={id + '-firstName'}>Imię</label>
      <div>
        <input id={id + '-firstName'} type="text" />
      </div>
      <label htmlFor={id + '-lastName'}>Nazwisko</label>
      <div>
        <input id={id + '-lastName'} type="text" />
      </div>
    </div>
  );
};
```

Uwaga:

`useState` generuje ciąg znaków zawierający dwukropek `: .`. Zapewnia to unikalność identyfikatora, lecz nie działa w selektorach CSS-owych i API takich jak `querySelectorAll`.

`useState` umożliwia ustawienie opcji `identifierPrefix`, aby uniknąć kolizji w aplikacjach o wielu "koreniach". Aby dowiedzieć się, jak skonfigurować tę opcję, przeczytaj dokumentację dla `hydrateRoot` oraz `ReactDOMServer`.

Hooiki dla bibliotek

Poniższe hooiki przewidziane są dla twórców bibliotek, aby umożliwić im bardziej dogłębna integrację z modelem Reacta, i zwykle nie stosuje się ich w aplikacjach.

`useSyncExternalStore` {#usesyncexternalstore}

```
const state = useSyncExternalStore(subscribe, getSnapshot[], getServerSnapshot);
```

`useSyncExternalStore` jest zalecany do odczytywania i subskrybowania się do zewnętrznych źródeł danych w sposób kompatybilny z funkcjonalnościami renderowania wspólnego, jak np. selektywna hydratacja czy kwantowanie czasu.

Metoda ta zwraca wartość z magazynu i przyjmuje trzy argumenty:

- `subscribe` : służy do zarejestrowania funkcji zwrotnej, która zostanie wywołana przy każdej zmianie wartości w magazynie.
- `getSnapshot` : służy do pobrania aktualnej wartości z magazynu.
- `getServerSnapshot` : służy do pobrania wartości podczas renderowania po stronie serwera.

Najprostszy przykład może subskrybować się na cały magazyn:

```
const state = useSyncExternalStore(store.subscribe, store.getSnapshot);
```

Można jednak zasubskrybować się na konkretną wartość:

```
const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
);
```

Podczas renderowania po stronie serwera musisz zserializować wartość magazynu i przekazać ją do `useSyncExternalStore`. React używa jej podczas hydracji, dzięki czemu nie powstaną niespójności:

```
const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
  () => INITIAL_SERVER_SNAPSHOT.selectedField,
);
```

Uwaga:

`getSnapshot` musi zwracać wartość zbuforowaną wartość. Jeśli `getSnapshot` zostanie wywołana kilka razy z rzędu, powinna zwrócić dokładnie tę samą wartość, chyba że faktycznie uległa ona zmianie.

Dostępny jest "shim" o nazwie `use-sync-external-store/shim` obsługujący wiele wersji Reacta. Preferuje on użycie hooka `useSyncExternalStore`, jeśli jest on dostępny, a w przypadku jego braku korzysta z innych dostępnych narzędzi.

Dla ułatwienia stworzyliśmy wersję tego API z automatycznym wsparciem dla memoizacji wyników z `getSnapshot`, dostępną pod nazwą `use-sync-external-store/with-selector`.

useInsertionEffect {#useinsertioneffect}

```
useInsertionEffect(didUpdate);
```

Sygnatura funkcji jest identyczna jak w przypadku hooka `useEffect`, jednak ten hook wywoływany jest synchronicznie przed wszystkimi mutacjami DOM. Użyj go, aby wstrzyknąć style do DOM, zanim `useLayoutEffect` odczyta układ strony. Jako że ten hook ma ograniczony zakres, nie ma on dostępu do referencji i nie może zlecać aktualizacji stanu.

Uwaga:

`useInsertionEffect` powinien być używany głównie przez autorów bibliotek css-in-js. W innych przypadkach zalecamy korzystanie z `useEffect` lub `useLayoutEffect`.

Zasady korzystania z hooków

Hooki są nowym dodatkiem w Reakcie 16.8. Pozwalają one używać stanu i innych funkcjonalności Reacta, bez użycia klas.

Hooki są javascriptowymi funkcjami, ale podczas korzystania z nich musisz pamiętać o dwóch ważnych zasadach. Stworzyliśmy [wtyczkę do lintera](#), która automatycznie wymusza stosowanie tych zasad:

Wywołuj hooki tylko z najwyższego poziomu kodu

Nie wywołuj hooków wewnętrz pętli, instrukcji warunkowych czy zagnieżdżonych funkcji. Korzystaj z hooków tylko z najwyższego poziomu kodu twoich komponentów funkcyjnych, jeszcze przed zwróceniem wartości. Przestrzegając tej zasady, zyskujesz pewność, że hooki zostaną wywołane w tej samej kolejności, za każdym razem gdy komponent jest renderowany. To właśnie pozwala Reactowi na właściwe przechowywanie stanu pomiędzy kolejnymi wywołaniami `useState` i `useEffect` (Jeśli ciekawi cię ten temat, dogłębnie wyjaśnimy go [w kolejnym podrozdziale](#).)

Wywołuj hooki tylko w komponentach funkcyjnych

Nie wywołuj hooków wewnętrz zwykłych javascriptowych funkcji. Zamiast tego możesz:

- Wywoływać hooki wewnętrz reactowych komponentów funkcyjnych.
- Wywoływać hooki wewnętrz własnych hooków (więcej na ten temat dowiemy się [w następnym rozdziale](#)).

Przestrzegając tej zasady, upewniasz się, że cała logika związana ze stanem komponentu jest wyraźnie widoczna w jego kodzie źródłowym.

Wtyczka dla ESLinta

Wydaliśmy wtyczkę dla ESLinta o nazwie `eslint-plugin-react-hooks`, która wymusza stosowanie tych dwóch zasad. Jeśli chcesz ją wypróbować, możesz dodać ją do swojego projektu w następujący sposób:

Wtyczka ta jest instalowana domyślnie w [Create React App](#).

```
npm install eslint-plugin-react-hooks --save-dev
```

```
// Twoja konfiguracja ESLint
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error", // Sprawdza stosowanie zasad hooków
    "react-hooks/exhaustive-deps": "warn" // Sprawdza zależności efektów
  }
}
```

Możesz teraz przejść do następnego rozdziału, gdzie wyjaśniamy, jak pisać własne hooki. W tym rozdziale postaramy się uzasadnić, dlaczego narzucamy takie zasady.

Wyjaśnienie

Jak [dowiedzieliśmy się wcześniej](#), w ramach pojedynczego komponentu możemy używać wielu hooków stanu i efektów:

```
function Form() {
  // 1. Użyj zmiennej stanu do przechowania imienia
  const [name, setName] = useState('Mary');

  // 2. Użyj efektu, aby zapisać dane formularza
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });

  // 3. Użyj zmiennej stanu do przechowania nazwiska
  const [surname, setSurname] = useState('Poppins');

  // 4. Użyj efektu, aby zaktualizować tytuł strony
  useEffect(function updateTitle() {
    document.title = name + ' ' + surname;
  });

  // ...
}
```

Skąd zatem React wie, jaką wartość stanu zwrócić przy kolejnych wywołaniach funkcji `useState`? Tajemnica tkwi w tym, że **React polega na kolejności, w jakiej hooki są wywoływane**. Nasz przykład zadziała, ponieważ kolejność wywoływanego hooków jest taka sama przy każdym renderowaniu:

```
// -----
// Pierwsze renderowanie
// -----
useState('Mary')           // 1. Zainicjalizuj zmienną stanu imienia wartością
,"Mary"
useEffect(persistForm)    // 2. Dodaj efekt odpowiedzialny za przechowywanie
danych formularza
useState('Poppins')        // 3. Zainicjalizuj zmienną stanu nazwiska wartością
,"Poppins"
useEffect(updateTitle)     // 4. Dodaj efekt odpowiedzialny za aktualizację tytułu

// -----
// Drugie renderowanie
// -----
useState('Mary')           // 1. Odczytaj zmienną stanu przechowującą imię
(argument został zignorowany)
useEffect(persistForm)    // 2. Zastąp efekt odpowiedzialny za przechowywanie
danych
useState('Poppins')        // 3. Odczytaj zmienną stanu przechowującą nazwisko
(argument został zignorowany)
useEffect(updateTitle)     // 4. Zastąp efekt odpowiedzialny za aktualizację
tytułu

// ...
```

Tak długo, jak kolejność wywoływania hooków pozostaje taka sama pomiędzy kolejnymi renderowaniami, React może powiązać lokalny stan z każdym z nich. A co wydarzy się, jeśli umieścimy wywołanie hooka (na przykład efektu `persistForm`) wewnątrz instrukcji warunkowej?

```
// 🔴 Łamiamy pierwszą zasadę, używając hooka wewnątrz instrukcji warunkowej
if (name !== '') {
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });
}
```

Warunek `name !== ''` jest spełniony przy pierwszym renderze, więc uruchamiany jest ten hook. Jednakże przy kolejnym renderze użytkownik może wyczyścić wartości formularza, powodując, że warunek nie będzie spełniony. Teraz, w związku z tym, że pominęliśmy hook podczas renderowania, kolejność wywoływania hooków zostaje zachwiana:

```
useState('Mary')           // 1. Odczytaj zmienną stanu przechowującą imię
// Argument został zignorowany
// useEffect(persistForm) // 🔴 Ten hook został pominięty!
useState('Poppins')        // 🔴 2 (a był 3). Nie uda się odczytać zmiennej stanu
useEffect(updateTitle)     // 🔴 3 (a był 4). Nie uda się zastąpić efektu
```

React nie wiedziałby, co zwrócić dla drugiego wywołania hooka `useState`. React spodziewał się, że drugie wywołanie hooka w tym komponencie będzie odpowiadało wywołaniu efektu `persistForm`, tak jak podczas poprzedniego renderowania. Nie jest to już jednak prawdą. Od tej chwili każde kolejne wywołanie hooka, po tym, jak jeden został pominięty, również przesunęłoby się o jeden, prowadząc do błędów.

Dlatego właśnie hooki muszą być wywoływane z najwyższego poziomu kodu komponentów. Jeśli chcesz, żeby efekt działał pod jakimś warunkiem, możesz umieścić ten warunek *wewnątrz* hooka:

```
useEffect(function persistForm() {
  // 👍 Tym razem nie łamamy pierwszej zasady
  if (name !== '') {
    localStorage.setItem('formData', name);
  }
});
```

Zauważ, że nie musisz o tym pamiętać, jeśli użyjesz dostarczonej przez nas reguły lintera. Teraz jednak wiesz także, *dla czego* hooki działają w ten sposób i jakim problemom zapobiega stosowanie tej reguły.

Kolejne kroki

W końcu jesteśmy gotowi na to, aby nauczyć się [pisać własne hooki!](#) Własne hooki pozwalają łączyć hooki dostarczone przez Reacta we własne abstrakcje i współdzielić logikę związaną ze stanem pomiędzy komponentami.

[Go to TOC](#)

Używanie hooka stanu

Hooki są nowym dodatkiem w Reakcie 16.8. Pozwalają one na wykorzystanie stanu i innych funkcjonalności Reacta, bez użycia klas.

We [wstępie](#) przedstawiliśmy hooki za pomocą następującego przykładu:

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

function Example() {
  // Zadeklaruj nową zmienną stanu, którą nazwiemy „count”
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Kliknięto {count} razy</p>
      <button onClick={() => setCount(count + 1)}>
        Kliknij mnie
      </button>
    </div>
  );
}
```

Naszą naukę o hookach rozpoczęmy, porównując ten kod z jego przykładowym odpowiednikiem w klasie.

Odpowiednik w klasie

Jeżeli używasz już klas w Reakcie, ten kod powinien wyglądać znajomo:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>Kliknięto {this.state.count} razy</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Kliknij mnie
        </button>
      </div>
    );
  }
}
```

Stan inicjalizowany jest wartością `{ count: 0 }`, a następnie, kiedy użytkownik naciska przycisk, inkrementujemy właściwość `state.count`, wywołując metodę `this.setState()`. W dalszej części tego rozdziału będziemy posiłkować się fragmentami kodu z tego przykładu.

Uwaga

Być może zastanawiasz się, dlaczego pokazujemy tutaj licznik zamiast bardziej „życiowego” przykładu. Pozwoli nam to skupić się na samym interfejsie API, póki stawiamy nasze pierwsze kroki z hookami.

Hooki i komponenty funkcyjne

Przypomnijmy, reactowe komponenty funkcyjne wyglądają następująco:

```
const Example = (props) => {
  // Tutaj możesz korzystać z hooków!
  return <div />;
}
```

albo tak:

```
function Example(props) {
  // Tutaj możesz korzystać z hooków!
  return <div />;
}
```

Być może spotkałeś się z nazwą „komponenty bezstanowe”. Jako że właśnie przedstawiamy sposób na korzystanie ze stanu wewnętrz takich komponentów, wolimy nazywać je „komponentami funkcyjnymi”.

Hooki **nie** działają w klasach. Ale możesz używać ich zamiast klas.

Czym jest hook?

Nasz nowy przykład rozpoczęmy importując hook `useState` z Reacta:

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

function Example() {
  // ...
}
```

Czym jest hook? Hook jest specjalną funkcją, która pozwala „zahaczyć się” w wewnętrzne mechanizmy Reacta. Na przykład `useState` jest hookiem, który pozwala korzystać ze stanu w komponencie funkcyjnym. W kolejnych rozdziałach poznamy inne hooki.

Kiedy powinno się korzystać z hooków? Jeśli po stworzeniu komponentu funkcyjnego zorientujesz się, że potrzebujesz przechować kilka wartości w stanie, dotychczas musiałeś zamienić taki komponent na klasę. Teraz możesz skorzystać z hooka z wewnętrz istniejącego komponentu funkcyjnego. Zobaczmy jak to działa!

Uwaga

Istnieje kilka specjalnych zasad, które mówią o tym kiedy możesz, a kiedy nie możesz używać hooków w komponencie. Więcej szczegółów poznamy w rozdziale pt. „[Zasady korzystania z hooków](#)”.

Zadeklarowanie zmiennej stanu

W klasie inicjalizujemy stan `count` z wartością `0`, poprzez ustawienie właściwości `this.state` na `{ count: 0 }` w konstruktorze.

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
}
```

W komponencie funkcyjnym nie mamy dostępu do `this`, więc nie możemy przypisywać, ani odczytać wartości właściwości `this.state`. Zamiast tego wywołamy hook `useState` bezpośrednio z wewnętrz naszego komponentu:

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

function Example() {
  // Zadeklaruj nową zmienną stanu, którą nazwiemy „count”
  const [count, setCount] = useState(0);
```

Co w zasadzie robi wywołanie `useState`? Deklaruje ono „zmienną stanu”. Nasza zmienna nazywa się `count`, ale możemy nazwać ją, jak tylko chcemy, na przykład `banan`. Jest to sposób na „przechowanie” wartości pomiędzy wywołaniami funkcji -- `useState` jest nowym sposobem, na wykorzystanie dokładnie tych samych możliwości, jakie daje `this.state` w klasach. Zwykle zmienne „znikają” kiedy funkcja kończy działanie, ale zmienne stanu są przechowywane przez Reacta.

Co przekazujemy do `useState` jako argumenty? Jedynym argumentem, jaki przyjmuje hook `useState()` jest początkowa wartość stanu. W przeciwieństwie do klas stan nie musi być obiektem. Możemy przechowywać liczbę lub ciąg znaków, jeżeli to wszystko, czego potrzebujemy. W naszym przykładzie chcemy przechować jedynie wartość liczbową, ile razy użytkownik nacisnął przycisk. Dlatego też przekazujemy `0` jako stan początkowy naszej zmiennej. (Jeżeli chcielibyśmy przechować kolejną wartość w stanie, wywołalibyśmy `useState()` po raz drugi.)

Co zwraca wywołanie `useState`? Zwraca ono parę wartości: aktualną wartość stanu i funkcję, która pozwala go aktualizować. Dlatego stosujemy zapis `const [count, setCount] = useState()`. Jest to podobne do właściwości `this.state.count` i metody `this.setState` w klasie, z tą różnicą, że tutaj dostajesz je w parze. Jeżeli nie znasz składni, której użyliśmy, wróćmy do tego [przy końcu tego rozdziału](#).

Teraz, kiedy wiemy już, co robi hook `useState`, nasz przykład powinien nabrać większego sensu:

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

function Example() {
  // Zadeklaruj nową zmienną stanu, którą nazwiemy „count”
  const [count, setCount] = useState(0);
```

Deklarujemy zmienną stanu, którą nazwaliśmy `count` i ustawiamy jej wartość na `0`. React będzie pamiętać jej aktualną wartość pomiędzy kolejnymi renderowaniami i dostarczy najnowszą wartość do naszej funkcji. Jeżeli chcemy zaktualizować obecną wartość `count`, możemy wywołać funkcję `setCount`.

Uwaga

Być może zastanawiasz się -- dlaczego funkcja `useState` (pol. *używaj stanu*) nie nazywa się `createState` (pol. *stwórz stan*)?

Nazwa „Create” (pol. *tworzyć*) nie byłaby zbyt trafna, ponieważ stan tworzony jest tylko wtedy, gdy komponent renderowany jest za pierwszym razem. Podczas kolejnych renderowań `useState` zwraca aktualny stan. Gdyby nie to, nie mogłoby być mowy o żadnym „stanie” (ang. *state*)! Istnieje też powód, dla którego nazwa hooka zawsze rozpoczyna się od `use`. Więcej na ten temat dowiemy się w rozdziale pt. „[Zasady korzystania z hooków](#)”.

Odczytywanie wartości stanu

Jeśli chcemy wyświetlić aktualną wartość stanu licznika w klasie, odczytamy wartość właściwości `this.state.count`:

```
<p>Kliknięto {this.state.count} razy</p>
```

Wewnątrz funkcji, możemy użyć bezpośrednio zmiennej `count`:

```
<p>Kliknięto {count} razy</p>
```

Aktualizowanie wartości stanu

Aby zaktualizować wartość stanu `count` w klasie, musimy wywołać metodę `this.setState()`:

```
<button onClick={() => this.setState({ count: this.state.count + 1 })}>
  Kliknij mnie
</button>
```

Wewnątrz funkcji mamy już zadeklarowane zmienne `setCount` i `count`, więc nie potrzebujemy `this`:

```
<button onClick={() => setCount(count + 1)}>
  Kliknij mnie
</button>
```

Podsumowanie

Teraz **podsumujmy linijkę po linijce to, czego się nauczyliśmy** i sprawdźmy naszą wiedzę.

```

1: import React, { useState } from 'react';
2:
3: function Example() {
4:   const [count, setCount] = useState(0);
5:
6:   return (
7:     <div>
8:       <p>Kliknięto {count} razy</p>
9:       <button onClick={() => setCount(count + 1)}>
10:         Kliknij mnie
11:       </button>
12:     </div>
13:   );
14: }
```

- **Linia 1:** Importujemy hooka `useState` z biblioteki React. Pozwala nam to na używanie lokalnego stanu w komponencie funkcyjnym.
- **Linia 4:** Wewnątrz komponentu `Example`, poprzez wywołanie hooka `useState`, deklarujemy nową zmienną stanu. Wywołanie zwraca parę wartości, którym nadajemy nazwy. Naszą zmienną nazywamy `count` (pol. *licznik*), ponieważ przechowuje ona liczbę naciśnięć przycisku. Inicjalizujemy ją z domyślną wartością, poprzez przekazanie `0` jako jedynego argumentu do funkcji `useState`. Jako druga zwrócona jest funkcja, która pozwala aktualizować wartość `count`, więc nazwiemy ją `setCount` (pol. *ustawić licznik*).
- **Linia 9:** Kiedy użytkownik naciska przycisk wywołujemy funkcję `setCount` z nową wartością. React wyrenderuje ponownie komponent `Example`, przekazując do niego nową wartość zmiennej `count`.

Z początku wszystkie te informacje mogą wydawać się trudne do „przetrawienia”. Nie spiesz się! Jeżeli zgubisz się gdzieś czytając wyjaśnienie, spójrz na kod jeszcze raz i spróbuj go przeczytać ponownie od góry do dołu. Obiecujemy, kiedy spróbujesz „zapomnieć”, jak działa stan w klasach i spojrzyasz na kod świeżym okiem, nabierze to sensu.

Wskazówka: Co oznaczają nawiasy kwadratowe?

Być może zauważysz nawiasy kwadratowe, kiedy deklarowaliśmy zmienną stanu:

```
const [count, setCount] = useState(0);
```

Nazwy po lewej nie są częścią interfejsu API Reacta. Możesz nadać własne nazwy zmiennym stanu:

```
const [fruit, setFruit] = useState('banan');
```

Ta składnia JavaScriptu nazwana jest [przypisaniem destrukturyzującym tablic](#). Oznacza to, że tworzymy dwie zmienne `fruit` i `setFruit` -- gdzie `fruit` jest pierwszą wartością zwróconą przez `useState`, a `setFruit` drugą. Jest to odpowiednik następującego kodu:

```
var fruitStateVariable = useState('banan'); // Zwraca parę
var fruit = fruitStateVariable[0]; // Pierwszy element pary
var setFruit = fruitStateVariable[1]; // Drugi element pary
```

Kiedy deklarujemy zmienną stanu, korzystając z funkcji `useState`, zwraca ona parę -- dwuelementową tablicę. Pierwszym elementem jest aktualna wartość, a drugim funkcja, która pozwala go aktualizować. Używanie zapisu `[0]` i `[1]` do uzyskiwania dostępu do nich jest nieco mylące, ponieważ mają one określone znaczenie. Dlatego używamy składni przypisania destrukturyzującego.

Uwaga

Być może zastanawiasz się, skąd React wie, któremu konkretnie komponentowi odpowiada każde wywołanie `useState`, jeśli nie przekazujemy Reactowi obiektu `this` ani nic podobnego. Na [to pytanie](#) (i wiele innych) odpowiadamy w rozdziale pt. „[Hooki - FAQ](#)”.

Wskazówka: Używanie kilku zmiennych stanu

Deklarowanie zmiennych stanu jako pary `[something, setSomething]` jest przydatne także dlatego, że pozwala ci na używanie *różnych* nazw, dla różnych zmiennych stanu, jeśli chcemy korzystać z więcej niż jednej:

```
function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banan');
  const [todos, setTodos] = useState([{ text: 'Nauczyć się hooków' }]);
}
```

W powyższym komponencie, mamy lokalne zmienne `age`, `fruit` i `todos` i możemy każdą z nich zaktualizować z osobna.

```
function handleOrangeClick() {
  // Similar to this.setState({ fruit: 'orange' })
  setFruit('orange');
}
```

Nie musisz używać kilku zmiennych stanu. Zmienne stanu mogą równie dobrze przechowywać obiekty i tablice, więc wciąż możesz grupować powiązane dane. Jednakże, w przeciwieństwie do metody `this.setState` w klasie, aktualizowanie zmiennej stanu zawsze nadpisuje jej wartość, zamiast scalać stare i nowe wartości.

W rozdziale pt. „[Hooki - FAQ](#)” wypisaliśmy zalecenia dotyczące podziału niezależnych zmiennych stanu.

Kolejne kroki

Ten rozdział poświęciliśmy jednemu z hooków dostarczanych przez Reacta -- `useState`. Czasami będziemy też nazywać go „hookiem stanu”. Pozwala on nam dodać lokalny stan do reaktywnych komponentów funkcyjnych, co zrobiliśmy po raz pierwszy w historii!

Dowiedzieliśmy się też troszeczkę o tym, czym w zasadzie są hooki. Hooki to funkcje, które pozwalają „zahaczyć się” wewnętrzne mechanizmy Reacta z wewnątrz komponentów funkcyjnych. Ich nazwy muszą zaczynać się od `use` i istnieje szereg innych hooków, których jeszcze nie poznaliśmy.

Zapoznajmy się więc z kolejnym hookiem: `useEffect`. Pozwala on na przeprowadzanie „efektów ubocznych” (ang. *side effects*) w komponentach i jest podobny do metod cyklu życia w klasach.

How to Contribute

React is one of Facebook's first open source projects that is both under very active development and is also being used to ship code to everybody on [facebook.com](#). We're still working out the kinks to make contributing to this project as easy and transparent as possible, but we're not quite there yet. Hopefully this document makes the process for contributing clear and answers some questions that you may have.

Code of Conduct {#code-of-conduct}

Facebook has adopted the [Contributor Covenant](#) as its Code of Conduct, and we expect project participants to adhere to it. Please read [the full text](#) so that you can understand what actions will and will not be tolerated.

Open Development

All work on React happens directly on [GitHub](#). Both core team members and external contributors send pull requests which go through the same review process.

Semantic Versioning

React follows [semantic versioning](#). We release patch versions for critical bugfixes, minor versions for new features or non-essential changes, and major versions for any breaking changes. When we make breaking changes, we also introduce deprecation warnings in a minor version so that our users learn about the upcoming changes and migrate their code in advance. Learn more about our commitment to stability and incremental migration in [our versioning policy](#).

Every significant change is documented in the [changelog file](#).

Branch Organization

Submit all changes directly to the `main branch`. We don't use separate branches for development or for upcoming releases. We do our best to keep `main` in good shape, with all tests passing.

Code that lands in `main` must be compatible with the latest stable release. It may contain additional features, but no breaking changes. We should be able to release a new minor version from the tip of `main` at any time.

Feature Flags

To keep the `main` branch in a releasable state, breaking changes and experimental features must be gated behind a feature flag.

Feature flags are defined in `packages/shared/ReactFeatureFlags.js`. Some builds of React may enable different sets of feature flags; for example, the React Native build may be configured differently than ReactDOM. These flags are found in `packages/shared/forks`. Feature flags are statically typed by Flow, so you can run `yarn flow` to confirm that you've updated all the necessary files.

React's build system will strip out disabled feature branches before publishing. A continuous integration job runs on every commit to check for changes in bundle size. You can use the change in size as a signal that a feature was gated correctly.

Bugs

Where to Find Known Issues

We are using [GitHub Issues](#) for our public bugs. We keep a close eye on this and try to make it clear when we have an internal fix in progress. Before filing a new task, try to make sure your problem doesn't already exist.

Reporting New Issues

The best way to get your bug fixed is to provide a reduced test case. This [JSFiddle template](#) is a great starting point.

Security Bugs

Facebook has a [bounty program](#) for the safe disclosure of security bugs. With that in mind, please do not file public issues; go through the process outlined on that page.

How to Get in Touch

- IRC: [#reactjs on freenode](#)
- [Discussion forums](#)

There is also [an active community of React users on the Discord chat platform](#) in case you need help with React.

Proposing a Change

If you intend to change the public API, or make any non-trivial changes to the implementation, we recommend [filing an issue](#). This lets us reach an agreement on your proposal before you put significant effort into it.

If you're only fixing a bug, it's fine to submit a pull request right away but we still recommend to file an issue detailing what you're fixing. This is helpful in case we don't accept that specific fix but want to keep track of the issue.

Your First Pull Request

Working on your first Pull Request? You can learn how from this free video series:

[How to Contribute to an Open Source Project on GitHub](#)

To help you get your feet wet and get you familiar with our contribution process, we have a list of [good first issues](#) that contain bugs that have a relatively limited scope. This is a great place to get started.

If you decide to fix an issue, please be sure to check the comment thread in case somebody is already working on a fix. If nobody is working on it at the moment, please leave a comment stating that you intend to work on it so other people don't accidentally duplicate your effort.

If somebody claims an issue but doesn't follow up for more than two weeks, it's fine to take it over but you should still leave a comment.

Sending a Pull Request

The core team is monitoring for pull requests. We will review your pull request and either merge it, request changes to it, or close it with an explanation. For API changes we may need to fix our internal uses at Facebook.com, which could cause some delay. We'll do our best to provide updates and feedback throughout the process.

Before submitting a pull request, please make sure the following is done:

1. Fork [the repository](#) and create your branch from `main`.
2. Run `yarn` in the repository root.
3. If you've fixed a bug or added code that should be tested, add tests!
4. Ensure the test suite passes (`yarn test`). Tip: `yarn test --watch TestName` is helpful in development.
5. Run `yarn test --prod` to test in the production environment.
6. If you need a debugger, run `yarn debug-test --watch TestName`, open `chrome://inspect`, and press "Inspect".
7. Format your code with `prettier` (`yarn prettier`).
8. Make sure your code lints (`yarn lint`). Tip: `yarn linc` to only check changed files.
9. Run the `Flow` typechecks (`yarn flow`).
10. If you haven't already, complete the CLA.

Contributor License Agreement (CLA)

In order to accept your pull request, we need you to submit a CLA. You only need to do this once, so if you've done this for another Facebook open source project, you're good to go. If you are submitting a pull request for the first time, just let us know that you have completed the CLA and we can cross-check with your GitHub username.

[Complete your CLA here.](#)

Contribution Prerequisites

- You have [Node](#) installed at LTS and [Yarn](#) at v1.2.0+.
- You have [JDK](#) installed.
- You have `gcc` installed or are comfortable installing a compiler if needed. Some of our dependencies may require a compilation step. On OS X, the Xcode Command Line Tools will cover this. On Ubuntu, `apt-get install build-essential` will install the required packages. Similar commands should work on other Linux distros. Windows will require some additional steps, see the [node-gyp installation instructions](#) for details.

- You are familiar with Git.

Development Workflow

After cloning React, run `yarn` to fetch its dependencies. Then, you can run several commands:

- `yarn lint` checks the code style.
- `yarn linc` is like `yarn lint` but faster because it only checks files that differ in your branch.
- `yarn test` runs the complete test suite.
- `yarn test --watch` runs an interactive test watcher.
- `yarn test --prod` runs tests in the production environment.
- `yarn test <pattern>` runs tests with matching filenames.
- `yarn debug-test` is just like `yarn test` but with a debugger. Open `chrome://inspect` and press "Inspect".
- `yarn flow` runs the Flow typechecks.
- `yarn build` creates a `build` folder with all the packages.
- `yarn build react/index,react-dom/index --type=UMD` creates UMD builds of just React and ReactDOM.

We recommend running `yarn test` (or its variations above) to make sure you don't introduce any regressions as you work on your change. However, it can be handy to try your build of React in a real project.

First, run `yarn build`. This will produce pre-built bundles in `build` folder, as well as prepare npm packages inside `build/packages`.

The easiest way to try your changes is to run `yarn build react/index,react-dom/index --type=UMD` and then open `fixtures/packaging/babel-standalone/dev.html`. This file already uses `react.development.js` from the `build` folder so it will pick up your changes.

If you want to try your changes in your existing React project, you may copy `build/node_modules/react/umd/react.development.js`, `build/node_modules/react-dom/umd/react-dom.development.js`, or any other build products into your app and use them instead of the stable version.

If your project uses React from npm, you may delete `react` and `react-dom` in its dependencies and use `yarn link` to point them to your local `build` folder. Note that **instead of `--type=UMD` you'll want to pass `--type=NODE` when building**. You'll also need to build the `scheduler` package:

```
cd ~/path_to_your_react_clone/
yarn build react/index,react/jsx,react-dom/index,scheduler --type=NODE

cd build/node_modules/react
yarn link
cd build/node_modules/react-dom
yarn link

cd ~/path/to/your/project
yarn link react react-dom
```

Every time you run `yarn build` in the React folder, the updated versions will appear in your project's `node_modules`. You can then rebuild your project to try your changes.

If some package is still missing (e.g. maybe you use `react-dom/server` in your project), you can always do a full build with `yarn build`. Note that running `yarn build` without options takes a long time.

We still require that your pull request contains unit tests for any new functionality. This way we can ensure that we don't break your code in the future.

Style Guide

We use an automatic code formatter called [Prettier](#). Run `yarn prettier` after making any changes to the code.

Then, our linter will catch most issues that may exist in your code. You can check the status of your code styling by simply running `yarn linc`.

However, there are still some styles that the linter cannot pick up. If you are unsure about something, looking at [Airbnb's Style Guide](#) will guide you in the right direction.

Request for Comments (RFC)

Many changes, including bug fixes and documentation improvements can be implemented and reviewed via the normal GitHub pull request workflow.

Some changes though are "substantial", and we ask that these be put through a bit of a design process and produce a consensus among the React core team.

The "RFC" (request for comments) process is intended to provide a consistent and controlled path for new features to enter the project. You can contribute by visiting the [rfcs repository](#).

License

By contributing to React, you agree that your contributions will be licensed under its MIT license.

What Next?

Read the [next section](#) to learn how the codebase is organized.

Implementation Notes

This section is a collection of implementation notes for the [stack reconciler](#).

It is very technical and assumes a strong understanding of React public API as well as how it's divided into core, renderers, and the reconciler. If you're not very familiar with the React codebase, read [the codebase overview](#) first.

It also assumes an understanding of the [differences between React components, their instances, and elements](#).

The stack reconciler was used in React 15 and earlier. It is located at [src/renderers/shared/stack/reconciler](#).

Video: Building React from Scratch

[Paul O'Shannessy](#) gave a talk about [building React from scratch](#) that largely inspired this document.

Both this document and his talk are simplifications of the real codebase so you might get a better understanding by getting familiar with both of them.

Overview

The reconciler itself doesn't have a public API. [Renderers](#) like React DOM and React Native use it to efficiently update the user interface according to the React components written by the user.

Mounting as a Recursive Process

Let's consider the first time you mount a component:

```
const root = ReactDOM.createRoot(rootEl);
root.render(<App />);
```

`root.render` will pass `<App />` along to the reconciler. Remember that `<App />` is a React element, that is, a description of *what* to render. You can think about it as a plain object:

```
console.log(<App />);
// { type: App, props: {} }
```

The reconciler will check if `App` is a class or a function.

If `App` is a function, the reconciler will call `App(props)` to get the rendered element.

If `App` is a class, the reconciler will instantiate an `App` with `new App(props)`, call the `componentWillMount()` lifecycle method, and then will call the `render()` method to get the rendered element.

Either way, the reconciler will learn the element `App` "rendered to".

This process is recursive. `App` may render to a `<Greeting />`, `Greeting` may render to a `<Button />`, and so on. The reconciler will "drill down" through user-defined components recursively as it learns what each component renders to.

You can imagine this process as a pseudocode:

```

function isClass(type) {
  // React.Component subclasses have this flag
  return (
    Boolean(type.prototype) &&
    Boolean(type.prototype.isReactComponent)
  );
}

// This function takes a React element (e.g. <App />)
// and returns a DOM or Native node representing the mounted tree.
function mount(element) {
  var type = element.type;
  var props = element.props;

  // We will determine the rendered element
  // by either running the type as function
  // or creating an instance and calling render().
  var renderedElement;
  if (isClass(type)) {
    // Component class
    var publicInstance = new type(props);
    // Set the props
    publicInstance.props = props;
    // Call the lifecycle if necessary
    if (publicInstance.componentWillMount) {
      publicInstance.componentWillMount();
    }
    // Get the rendered element by calling render()
    renderedElement = publicInstance.render();
  } else {
    // Component function
    renderedElement = type(props);
  }

  // This process is recursive because a component may
  // return an element with a type of another component.
  return mount(renderedElement);

  // Note: this implementation is incomplete and recurses infinitely!
  // It only handles elements like <App /> or <Button />.
  // It doesn't handle elements like <div /> or <p /> yet.
}

var rootEl = document.getElementById('root');
var node = mount(<App />);
rootEl.appendChild(node);

```

Note:

This really *is* a pseudo-code. It isn't similar to the real implementation. It will also cause a stack overflow because we haven't discussed when to stop the recursion.

Let's recap a few key ideas in the example above:

- React elements are plain objects representing the component type (e.g. `App`) and the props.
- User-defined components (e.g. `App`) can be classes or functions but they all "render to" elements.
- "Mounting" is a recursive process that creates a DOM or Native tree given the top-level React element (e.g. `<App />`).

Mounting Host Elements

This process would be useless if we didn't render something to the screen as a result.

In addition to user-defined ("composite") components, React elements may also represent platform-specific ("host") components. For example, `Button` might return a `<div />` from its render method.

If element's `type` property is a string, we are dealing with a host element:

```
console.log(<div />);
// { type: 'div', props: {} }
```

There is no user-defined code associated with host elements.

When the reconciler encounters a host element, it lets the renderer take care of mounting it. For example, React DOM would create a DOM node.

If the host element has children, the reconciler recursively mounts them following the same algorithm as above. It doesn't matter whether children are host (like `<div><hr /></div>`), composite (like `<div><Button /></div>`), or both.

The DOM nodes produced by the child components will be appended to the parent DOM node, and recursively, the complete DOM structure will be assembled.

Note:

The reconciler itself is not tied to the DOM. The exact result of mounting (sometimes called "mount image" in the source code) depends on the renderer, and can be a DOM node (React DOM), a string (React DOM Server), or a number representing a native view (React Native).

If we were to extend the code to handle host elements, it would look like this:

```
function isClass(type) {
  // React.Component subclasses have this flag
  return (
    Boolean(type.prototype) &&
    Boolean(type.prototype.isReactComponent)
  );
}

// This function only handles elements with a composite type.
// For example, it handles <App /> and <Button />, but not a <div />.
function mountComposite(element) {
```

```

var type = element.type;
var props = element.props;

var renderedElement;
if (isClass(type)) {
  // Component class
  var publicInstance = new type(props);
  // Set the props
  publicInstance.props = props;
  // Call the lifecycle if necessary
  if (publicInstance.componentWillMount) {
    publicInstance.componentWillMount();
  }
  renderedElement = publicInstance.render();
} else if (typeof type === 'function') {
  // Component function
  renderedElement = type(props);
}

// This is recursive but we'll eventually reach the bottom of recursion when
// the element is host (e.g. <div />) rather than composite (e.g. <App />):
return mount(renderedElement);
}

// This function only handles elements with a host type.
// For example, it handles <div /> and <p /> but not an <App />.
function mountHost(element) {
  var type = element.type;
  var props = element.props;
  var children = props.children || [];
  if (!Array.isArray(children)) {
    children = [children];
  }
  children = children.filter(Boolean);

  // This block of code shouldn't be in the reconciler.
  // Different renderers might initialize nodes differently.
  // For example, React Native would create iOS or Android views.
  var node = document.createElement(type);
  Object.keys(props).forEach(propName => {
    if (propName !== 'children') {
      node.setAttribute(propName, props[propName]);
    }
  });

  // Mount the children
  children.forEach(childElement => {
    // Children may be host (e.g. <div />) or composite (e.g. <Button />).
    // We will also mount them recursively:
    var childNode = mount(childElement);

    // This line of code is also renderer-specific.
    // It would be different depending on the renderer:
    node.appendChild(childNode);
  });

  // Return the DOM node as mount result.
  // This is where the recursion ends.
  return node;
}

function mount(element) {

```

```

var type = element.type;
if (typeof type === 'function') {
  // User-defined components
  return mountComposite(element);
} else if (typeof type === 'string') {
  // Platform-specific components
  return mountHost(element);
}

var rootEl = document.getElementById('root');
var node = mount(<App />);
rootEl.appendChild(node);

```

This is working but still far from how the reconciler is really implemented. The key missing ingredient is support for updates.

Introducing Internal Instances

The key feature of React is that you can re-render everything, and it won't recreate the DOM or reset the state:

```

root.render(<App />);
// Should reuse the existing DOM:
root.render(<App />);

```

However, our implementation above only knows how to mount the initial tree. It can't perform updates on it because it doesn't store all the necessary information, such as all the `publicInstances`, or which DOM `node`s correspond to which components.

The stack reconciler codebase solves this by making the `mount()` function a method and putting it on a class. There are drawbacks to this approach, and we are going in the opposite direction in the [ongoing rewrite of the reconciler](#). Nevertheless this is how it works now.

Instead of separate `mountHost` and `mountComposite` functions, we will create two classes: `DOMComponent` and `CompositeComponent`.

Both classes have a constructor accepting the `element`, as well as a `mount()` method returning the mounted node. We will replace a top-level `mount()` function with a factory that instantiates the correct class:

```

function instantiateComponent(element) {
  var type = element.type;
  if (typeof type === 'function') {
    // User-defined components
    return new CompositeComponent(element);
  } else if (typeof type === 'string') {
    // Platform-specific components
    return new DOMComponent(element);
  }
}

```

First, let's consider the implementation of `CompositeComponent`:

```

class CompositeComponent {
  constructor(element) {
    this.currentElement = element;
    this.renderedComponent = null;
    this.publicInstance = null;
  }

  getPublicInstance() {
    // For composite components, expose the class instance.
    return this.publicInstance;
  }

  mount() {
    var element = this.currentElement;
    var type = element.type;
    var props = element.props;

    var publicInstance;
    var renderedElement;
    if (isClass(type)) {
      // Component class
      publicInstance = new type(props);
      // Set the props
      publicInstance.props = props;
      // Call the lifecycle if necessary
      if (publicInstance.componentWillMount) {
        publicInstance.componentWillMount();
      }
      renderedElement = publicInstance.render();
    } else if (typeof type === 'function') {
      // Component function
      publicInstance = null;
      renderedElement = type(props);
    }

    // Save the public instance
    this.publicInstance = publicInstance;

    // Instantiate the child internal instance according to the element.
    // It would be a DOMComponent for <div /> or <p />,
    // and a CompositeComponent for <App /> or <Button />:
    var renderedComponent = instantiateComponent(renderedElement);
    this.renderedComponent = renderedComponent;

    // Mount the rendered output
    return renderedComponent.mount();
  }
}

```

This is not much different from our previous `mountComposite()` implementation, but now we can save some information, such as `this.currentElement`, `this.renderedComponent`, and `this.publicInstance`, for use during updates.

Note that an instance of `CompositeComponent` is not the same thing as an instance of the user-supplied `element.type`. `CompositeComponent` is an implementation detail of our reconciler, and is never exposed to the user. The user-defined class is the one we read from `element.type`, and `CompositeComponent` creates an instance of it.

To avoid the confusion, we will call instances of `CompositeComponent` and `DOMComponent` "internal instances". They exist so we can associate some long-lived data with them. Only the renderer and the reconciler are aware that they exist.

In contrast, we call an instance of the user-defined class a "public instance". The public instance is what you see as `this` in the `render()` and other methods of your custom components.

The `mountHost()` function, refactored to be a `mount()` method on `DOMComponent` class, also looks familiar:

```
class DOMComponent {
  constructor(element) {
    this.currentElement = element;
    this.renderedChildren = [];
    this.node = null;
  }

  getPublicInstance() {
    // For DOM components, only expose the DOM node.
    return this.node;
  }

  mount() {
    var element = this.currentElement;
    var type = element.type;
    var props = element.props;
    var children = props.children || [];
    if (!Array.isArray(children)) {
      children = [children];
    }

    // Create and save the node
    var node = document.createElement(type);
    this.node = node;

    // Set the attributes
    Object.keys(props).forEach(propName => {
      if (propName !== 'children') {
        node.setAttribute(propName, props[propName]);
      }
    });

    // Create and save the contained children.
    // Each of them can be a DOMComponent or a CompositeComponent,
    // depending on whether the element type is a string or a function.
    var renderedChildren = children.map(instantiateComponent);
    this.renderedChildren = renderedChildren;

    // Collect DOM nodes they return on mount
    var childNodes = renderedChildren.map(child => child.mount());
    childNodes.forEach(childNode => node.appendChild(childNode));

    // Return the DOM node as mount result
    return node;
  }
}
```

The main difference after refactoring from `mountHost()` is that we now keep `this.node` and `this.renderedChildren` associated with the internal DOM component instance. We will also use them for applying non-destructive updates in the future.

As a result, each internal instance, composite or host, now points to its child internal instances. To help visualize this, if a function `<App>` component renders a `<Button>` class component, and `Button` class renders a `<div>`, the internal instance tree would look like this:

```
[object CompositeComponent] {
  currentElement: <App />,
  publicInstance: null,
  renderedComponent: [object CompositeComponent] {
    currentElement: <Button />,
    publicInstance: [object Button],
    renderedComponent: [object DOMComponent] {
      currentElement: <div />,
      node: [object HTMLDivElement],
      renderedChildren: []
    }
  }
}
```

In the DOM you would only see the `<div>`. However the internal instance tree contains both composite and host internal instances.

The composite internal instances need to store:

- The current element.
- The public instance if element type is a class.
- The single rendered internal instance. It can be either a `DOMComponent` or a `CompositeComponent`.

The host internal instances need to store:

- The current element.
- The DOM node.
- All the child internal instances. Each of them can be either a `DOMComponent` or a `CompositeComponent`.

If you're struggling to imagine how an internal instance tree is structured in more complex applications, [React DevTools](#) can give you a close approximation, as it highlights host instances with grey, and composite instances with purple:

```

<TodoApp>
  ▼<div>
    <h3>TODO</h3>
    ▼<TodoList items=[{...}]>
      ▼<ul>
        <li key="1475260226151">Buy milk</li>
      </ul>
    </TodoList>
    ▼<form onSubmit=bound handleSubmit()>
      <input onChange=bound onChange() value="" />
      <button>Add #2</button>
    </form>
  </div>
</TodoApp>

```

To complete this refactoring, we will introduce a function that mounts a complete tree into a container node and a public instance:

```

function mountTree(element, containerNode) {
  // Create the top-level internal instance
  var rootComponent = instantiateComponent(element);

  // Mount the top-level component into the container
  var node = rootComponent.mount();
  containerNode.appendChild(node);

  // Return the public instance it provides
  var publicInstance = rootComponent.getPublicInstance();
  return publicInstance;
}

var rootEl = document.getElementById('root');
mountTree(<App />, rootEl);

```

Unmounting

Now that we have internal instances that hold onto their children and the DOM nodes, we can implement unmounting. For a composite component, unmounting calls a lifecycle method and recurses.

```

class CompositeComponent {

  // ...

  unmount() {
    // Call the lifecycle method if necessary
    var publicInstance = this.publicInstance;
    if (publicInstance) {
      if (publicInstance.componentWillUnmount) {
        publicInstance.componentWillUnmount();
      }
    }
  }
}

```

```
// Unmount the single rendered component
var renderedComponent = this.renderedComponent;
renderedComponent.unmount();
}

}
```

For `DOMComponent`, unmounting tells each child to unmount:

```
class DOMComponent {

// ...

unmount() {
  // Unmount all the children
  var renderedChildren = this.renderedChildren;
  renderedChildren.forEach(child => child.unmount());
}

}
```

In practice, unmounting DOM components also removes the event listeners and clears some caches, but we will skip those details.

We can now add a new top-level function called `unmountTree(containerNode)` that is similar to `ReactDOM.unmountComponentAtNode()`:

```
function unmountTree(containerNode) {
  // Read the internal instance from a DOM node:
  // (This doesn't work yet, we will need to change mountTree() to store it.)
  var node = containerNode.firstChild;
  var rootComponent = node._internalInstance;

  // Unmount the tree and clear the container
  rootComponent.unmount();
  containerNode.innerHTML = '';
}
```

In order for this to work, we need to read an internal root instance from a DOM node. We will modify `mountTree()` to add the `_internalInstance` property to the root DOM node. We will also teach `mountTree()` to destroy any existing tree so it can be called multiple times:

```
function mountTree(element, containerNode) {
  // Destroy any existing tree
  if (containerNode.firstChild) {
    unmountTree(containerNode);
  }

  // Create the top-level internal instance
  var rootComponent = instantiateComponent(element);

  // Mount the top-level component into the container
  var node = rootComponent.mount();
  containerNode.appendChild(node);

  // Save a reference to the internal instance
  node._internalInstance = rootComponent;

  // Return the public instance it provides
}
```

```

var publicInstance = rootComponent.getPublicInstance();
return publicInstance;
}

```

Now, running `unmountTree()`, or running `mountTree()` repeatedly, removes the old tree and runs the `componentWillUnmount()` lifecycle method on components.

Updating

In the previous section, we implemented unmounting. However React wouldn't be very useful if each prop change unmounted and mounted the whole tree. The goal of the reconciler is to reuse existing instances where possible to preserve the DOM and the state:

```

var rootEl = document.getElementById('root');

mountTree(<App />, rootEl);
// Should reuse the existing DOM:
mountTree(<App />, rootEl);

```

We will extend our internal instance contract with one more method. In addition to `mount()` and `unmount()`, both `DOMComponent` and `CompositeComponent` will implement a new method called `receive(nextElement)`:

```

class CompositeComponent {
  // ...

  receive(nextElement) {
    // ...
  }
}

class DOMComponent {
  // ...

  receive(nextElement) {
    // ...
  }
}

```

Its job is to do whatever is necessary to bring the component (and any of its children) up to date with the description provided by the `nextElement`.

This is the part that is often described as "virtual DOM diffing" although what really happens is that we walk the internal tree recursively and let each internal instance receive an update.

Updating Composite Components

When a composite component receives a new element, we run the `componentWillUpdate()` lifecycle method.

Then we re-render the component with the new props, and get the next rendered element:

```

class CompositeComponent {
  // ...

  receive(nextElement) {
    var prevProps = this.currentElement.props;
    var publicInstance = this.publicInstance;
    var prevRenderedComponent = this.renderedComponent;
    var prevRenderedElement = prevRenderedComponent.currentElement;

    // Update *own* element
    this.currentElement = nextElement;
    var type = nextElement.type;
    var nextProps = nextElement.props;

    // Figure out what the next render() output is
    var nextRenderedElement;
    if (isClass(type)) {
      // Component class
      // Call the lifecycle if necessary
      if (publicInstance.componentWillUpdate) {
        publicInstance.componentWillUpdate(nextProps);
      }
      // Update the props
      publicInstance.props = nextProps;
      // Re-render
      nextRenderedElement = publicInstance.render();
    } else if (typeof type === 'function') {
      // Component function
      nextRenderedElement = type(nextProps);
    }
  }
}

```

Next, we can look at the rendered element's `type`. If the `type` has not changed since the last render, the component below can also be updated in place.

For example, if it returned `<Button color="red" />` the first time, and `<Button color="blue" />` the second time, we can just tell the corresponding internal instance to `receive()` the next element:

```

// ...

// If the rendered element type has not changed,
// reuse the existing component instance and exit.
if (prevRenderedElement.type === nextRenderedElement.type) {
  prevRenderedComponent.receive(nextRenderedElement);
  return;
}

// ...

```

However, if the next rendered element has a different `type` than the previously rendered element, we can't update the internal instance. A `<button>` can't "become" an `<input>`.

Instead, we have to unmount the existing internal instance and mount the new one corresponding to the rendered element type. For example, this is what happens when a component that previously rendered a `<button />` renders an `<input />`:

```
// ...

// If we reached this point, we need to unmount the previously
// mounted component, mount the new one, and swap their nodes.

// Find the old node because it will need to be replaced
var prevNode = prevRenderedComponent.getHostNode();

// Unmount the old child and mount a new child
prevRenderedComponent.unmount();
var nextRenderedComponent = instantiateComponent(nextRenderedElement);
var nextNode = nextRenderedComponent.mount();

// Replace the reference to the child
this.renderedComponent = nextRenderedComponent;

// Replace the old node with the new one
// Note: this is renderer-specific code and
// ideally should live outside of CompositeComponent:
prevNode.parentNode.replaceChild(nextNode, prevNode);

}

}
```

To sum this up, when a composite component receives a new element, it may either delegate the update to its rendered internal instance, or unmount it and mount a new one in its place.

There is another condition under which a component will re-mount rather than receive an element, and that is when the element's `key` has changed. We don't discuss `key` handling in this document because it adds more complexity to an already complex tutorial.

Note that we needed to add a method called `getHostNode()` to the internal instance contract so that it's possible to locate the platform-specific node and replace it during the update. Its implementation is straightforward for both classes:

```
class CompositeComponent {
// ...

getHostNode() {
  // Ask the rendered component to provide it.
  // This will recursively drill down any composites.
  return this.renderedComponent.getHostNode();
}

class DOMComponent {
// ...

getHostNode() {
  return this.node;
}
}
```

Updating Host Components

Host component implementations, such as `DOMComponent`, update differently. When they receive an element, they need to update the underlying platform-specific view. In case of React DOM, this means updating the DOM attributes:

```

class DOMComponent {
  // ...

  receive(nextElement) {
    var node = this.node;
    var prevElement = this.currentElement;
    var prevProps = prevElement.props;
    var nextProps = nextElement.props;
    this.currentElement = nextElement;

    // Remove old attributes.
    Object.keys(prevProps).forEach(propName => {
      if (propName !== 'children' && !nextProps.hasOwnProperty(propName)) {
        node.removeAttribute(propName);
      }
    });
    // Set next attributes.
    Object.keys(nextProps).forEach(propName => {
      if (propName !== 'children') {
        node.setAttribute(propName, nextProps[propName]);
      }
    });
  }
}

```

Then, host components need to update their children. Unlike composite components, they might contain more than a single child.

In this simplified example, we use an array of internal instances and iterate over it, either updating or replacing the internal instances depending on whether the received `type` matches their previous `type`. The real reconciler also takes element's `key` in the account and track moves in addition to insertions and deletions, but we will omit this logic.

We collect DOM operations on children in a list so we can execute them in batch:

```

// ...

// These are arrays of React elements:
var prevChildren = prevProps.children || [];
if (!Array.isArray(prevChildren)) {
  prevChildren = [prevChildren];
}
var nextChildren = nextProps.children || [];
if (!Array.isArray(nextChildren)) {
  nextChildren = [nextChildren];
}
// These are arrays of internal instances:
var prevRenderedChildren = this.renderedChildren;
var nextRenderedChildren = [];

// As we iterate over children, we will add operations to the array.
var operationQueue = [];

// Note: the section below is extremely simplified!
// It doesn't handle reorders, children with holes, or keys.
// It only exists to illustrate the overall flow, not the specifics.

for (var i = 0; i < nextChildren.length; i++) {
  // Try to get an existing internal instance for this child
}

```

```

var prevChild = prevRenderedChildren[i];

// If there is no internal instance under this index,
// a child has been appended to the end. Create a new
// internal instance, mount it, and use its node.
if (!prevChild) {
  var nextChild = instantiateComponent(nextChildren[i]);
  var node = nextChild.mount();

  // Record that we need to append a node
  operationQueue.push({type: 'ADD', node});
  nextRenderedChildren.push(nextChild);
  continue;
}

// We can only update the instance if its element's type matches.
// For example, <Button size="small" /> can be updated to
// <Button size="large" /> but not to an <App />.
var canUpdate = prevChildren[i].type === nextChildren[i].type;

// If we can't update an existing instance, we have to unmount it
// and mount a new one instead of it.
if (!canUpdate) {
  var prevNode = prevChild.getHostNode();
  prevChild.unmount();

  var nextChild = instantiateComponent(nextChildren[i]);
  var nextNode = nextChild.mount();

  // Record that we need to swap the nodes
  operationQueue.push({type: 'REPLACE', prevNode, nextNode});
  nextRenderedChildren.push(nextChild);
  continue;
}

// If we can update an existing internal instance,
// just let it receive the next element and handle its own update.
prevChild.receive(nextChildren[i]);
nextRenderedChildren.push(prevChild);
}

// Finally, unmount any children that don't exist:
for (var j = nextChildren.length; j < prevChildren.length; j++) {
  var prevChild = prevRenderedChildren[j];
  var node = prevChild.getHostNode();
  prevChild.unmount();

  // Record that we need to remove the node
  operationQueue.push({type: 'REMOVE', node});
}

// Point the list of rendered children to the updated version.
this.renderedChildren = nextRenderedChildren;

// ...

```

As the last step, we execute the DOM operations. Again, the real reconciler code is more complex because it also handles moves:

```
// ...

// Process the operation queue.
while (operationQueue.length > 0) {
  var operation = operationQueue.shift();
  switch (operation.type) {
    case 'ADD':
      this.node.appendChild(operation.node);
      break;
    case 'REPLACE':
      this.node.replaceChild(operation.nextNode, operation.prevNode);
      break;
    case 'REMOVE':
      this.node.removeChild(operation.node);
      break;
  }
}
}
```

And that is it for updating host components.

Top-Level Updates

Now that both `CompositeComponent` and `DOMComponent` implement the `receive(nextElement)` method, we can change the top-level `mountTree()` function to use it when the element `type` is the same as it was the last time:

```
function mountTree(element, containerNode) {
  // Check for an existing tree
  if (containerNode.firstChild) {
    var prevNode = containerNode.firstChild;
    var prevRootComponent = prevNode._internalInstance;
    var prevElement = prevRootComponent.currentElement;

    // If we can, reuse the existing root component
    if (prevElement.type === element.type) {
      prevRootComponent.receive(element);
      return;
    }

    // Otherwise, unmount the existing tree
    unmountTree(containerNode);
  }

  // ...
}
```

Now calling `mountTree()` two times with the same type isn't destructive:

```
var rootEl = document.getElementById('root');

mountTree(<App />, rootEl);
// Reuses the existing DOM:
mountTree(<App />, rootEl);
```

These are the basics of how React works internally.

What We Left Out

This document is simplified compared to the real codebase. There are a few important aspects we didn't address:

- Components can render `null`, and the reconciler can handle "empty slots" in arrays and rendered output.
- The reconciler also reads `key` from the elements, and uses it to establish which internal instance corresponds to which element in an array. A bulk of complexity in the actual React implementation is related to that.
- In addition to composite and host internal instance classes, there are also classes for "text" and "empty" components. They represent text nodes and the "empty slots" you get by rendering `null`.
- Renderers use [injection](#) to pass the host internal class to the reconciler. For example, React DOM tells the reconciler to use `ReactDOMComponent` as the host internal instance implementation.
- The logic for updating the list of children is extracted into a mixin called `ReactMultiChild` which is used by the host internal instance class implementations both in React DOM and React Native.
- The reconciler also implements support for `setState()` in composite components. Multiple updates inside event handlers get batched into a single update.
- The reconciler also takes care of attaching and detaching refs to composite components and host nodes.
- Lifecycle methods that are called after the DOM is ready, such as `componentDidMount()` and `componentDidUpdate()`, get collected into "callback queues" and are executed in a single batch.
- React puts information about the current update into an internal object called "transaction". Transactions are useful for keeping track of the queue of pending lifecycle methods, the current DOM nesting for the warnings, and anything else that is "global" to a specific update. Transactions also ensure React "cleans everything up" after updates. For example, the transaction class provided by React DOM restores the input selection after any update.

Jumping into the Code

- `ReactMount` is where the code like `mountTree()` and `unmountTree()` from this tutorial lives. It takes care of mounting and unmounting top-level components. `ReactNativeMount` is its React Native analog.
- `ReactDOMComponent` is the equivalent of `DOMComponent` in this tutorial. It implements the host component class for React DOM renderer. `ReactNativeBaseComponent` is its React Native analog.
- `ReactCompositeComponent` is the equivalent of `CompositeComponent` in this tutorial. It handles calling user-defined components and maintaining their state.
- `instantiateReactComponent` contains the switch that picks the right internal instance class to construct for an element. It is equivalent to `instantiateComponent()` in this tutorial.

- `ReactReconciler` is a wrapper with `mountComponent()`, `receiveComponent()`, and `unmountComponent()` methods. It calls the underlying implementations on the internal instances, but also includes some code around them that is shared by all internal instance implementations.
- `ReactChildReconciler` implements the logic for mounting, updating, and unmounting children according to the `key` of their elements.
- `ReactMultiChild` implements processing the operation queue for child insertions, deletions, and moves independently of the renderer.
- `mount()`, `receive()`, and `unmount()` are really called `mountComponent()`, `receiveComponent()`, and `unmountComponent()` in React codebase for legacy reasons, but they receive elements.
- Properties on the internal instances start with an underscore, e.g. `_currentElement`. They are considered to be read-only public fields throughout the codebase.

Future Directions

Stack reconciler has inherent limitations such as being synchronous and unable to interrupt the work or split it in chunks. There is a work in progress on the [new Fiber reconciler](#) with a [completely different architecture](#). In the future, we intend to replace stack reconciler with it, but at the moment it is far from feature parity.

Next Steps

Read the [next section](#) to learn about the guiding principles we use for React development.

Integracja z innymi bibliotekami

Reacta można używać w dowolnej aplikacji webowej. Można go osadzić w innej aplikacji, a także, przy odrobinie wysiłku, inną aplikację można osadzić w kodzie reactowym. W tym poradniku przeanalizujemy kilka powszechnych sytuacji dotyczących integracji z [jQuery](#) i [Backbonem](#). Mimo wszystko te same metody mogą zadziałać przy integracji komponentów z dowolnym kodem.

Integracja z wtyczkami manipulującymi DOM-em

React nie wie nic o zmianach w modelu DOM, które wprowadzono poza Reactem. Decyduje, co należy aktualizować, bazując na własnej, wewnętrznej reprezentacji. Dlatego jeśli węzły DOM zostaną zmienione przez inną bibliotekę, React wpada w zakłopotanie i nie wie, co robić.

Nie oznacza to jednak, że łączenie Reacta z innymi sposobami manipulacji modelu DOM jest niemożliwe czy jakoś szczególnie trudne. Trzeba tylko mieć pewność, że się rozumie, co które z nich robi.

Najprostszym sposobem na uniknięcie konfliktów jest powstrzymanie Reacta przed aktualizowaniem komponentu. Można to zrobić renderując elementy, których React nie ma potrzeby aktualizować, jak np. pusty `<div />`.

Jak podejść do problemu?

Aby lepiej to zobrazować, stwórzmy szkic kodu opakowującego generyczny plugin do jQuery.

Dodamy [referencje](#) (`ref`) do korzenia drzewa DOM. Dostęp do niej otrzymamy wewnątrz metody `componentDidMount`, gdzie będziemy mogli przekazać ją dalej do wtyczki.

Aby powstrzymać Reacta przed ingerowaniem w model DOM po zamontowaniu komponentu, w metodzie `render()` zwrócimy pusty znacznik `<div />`. Taki element `<div />` nie ma żadnych właściwości ani potomków, dlatego React nie ma powodu, żeby go aktualizować, pozwalając tym samym wtyczce na zarządzanie tą częścią drzewa DOM:

```
class SomePlugin extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.somePlugin();
  }

  componentWillUnmount() {
    this.$el.somePlugin('destroy');
  }

  render() {
    return <div ref={el => this.el = el} />;
  }
}
```

Zwróć uwagę, że zdefiniowaliśmy dwie metody cyklu życia: `componentDidMount` i `componentWillUnmount`. Wiele wtyczek do jQuery podpina detektory zdarzeń (ang. *event listeners*) do modelu DOM, dlatego trzeba pamiętać o ich odpięciu w `componentWillUnmount`. Jeśli wtyczka nie udostępnia metody czyszczącej, prawdopodobnie trzeba stworzyć ją samodzielnie, pamiętając, aby odpiąć wszystkie detektory zdarzeń dodane przez wtyczkę i zapobiec tym samym wyciekom pamięci.

Integracja z wtyczką Chosen do jQuery

Aby lepiej zobrazować powyższe koncepcje, napiszmy kawałek kodu opakowującego wtyczkę [Chosen](#), która rozszerza możliwości pól `<select>`.

Uwaga:

Nawet jeśli tak się da, nie znaczy, że jest to najlepsze podejście w przypadku aplikacji reactowych. Zachęcamy do korzystania bezpośrednio z komponentów reactowych, jeśli jest taka możliwość. Są one łatwiejsze w użyciu, a także dają większą kontrolę nad zachowaniem i wyglądem interfejsu użytkownika.

Najpierw przyjrzymy się, co wtyczka Chosen robi z modelem DOM.

Jeśli wywołasz ją na węźle DOM z elementem `<select>`, odczyta ona oryginalne atrybuty tego elementu, ukryje go za pomocą odpowiedniego stylu CSS, a następnie wstawi za nim dodatkowy węzeł DOM z własną reprezentacją wizualną pola wyboru. Na koniec powiadomi nas o wykonaniu pracy za pomocą odpowiednich zdarzeń biblioteki jQuery.

Załóżmy, że interesuje nas komponent opakowujący `<Chosen>` o takim API:

```
function Example() {
  return (
    <Chosen onChange={value => console.log(value)}>
      <option>waniliowe</option>
      <option>czekoladowe</option>
      <option>truskawkowe</option>
    </Chosen>
  );
}
```

Dla uproszczenia zaimplementujemy go jako [komponent niekontrolowany](#).

Najpierw stwórzmy pusty komponent z metodą `render()`, która zwraca `<select>` opakowany w `<div>`:

```
class Chosen extends React.Component {
  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

```

    );
}
}
```

Zwróć uwagę, że wstawiliśmy dodatkowy element `<div>`, który opakowuje `<select>`. Musimy tak zrobić, ponieważ Chosen doda własny element DOM zaraz za `<select>`. Mimo to, z perspektywy Reacta, ten `<div>` ma zawsze tylko jednego potomka. Dzięki temu zmiany wprowadzane przez Reacta nie będą konfliktoły z dodatkowym elementem wstawionym przez wtyczkę Chosen. Pamiętaj, że jeśli w jakikolwiek sposób modyfikujesz DOM poza Reactem, musisz upewnić się, że React nie ma powodu do zmian tych węzłów DOM.

Teraz zaimplementujemy metody cyklu życia. Zainicjalizujemy wtyczkę Chosen, przekazując jej referencję do węzła `<select>` wewnętrz metody `componentDidMount`, a po wszystkim posprzątamy w `componentWillUnmount`.

```

componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();
}

componentWillUnmount() {
  this.$el.chosen('destroy');
}
```

Wypróbuj kod na CodePen

Pole `this.el` nie ma żadnego specjalnego znaczenia dla Reacta. Mimo to kod działa, ponieważ wcześniej, w metodzie `render()`, przypisaliśmy do pola referencję wewnętrz `ref`.

```
<select className="Chosen-select" ref={el => this.el = el}>
```

Powyższy kod wystarczy, aby wyrenderować nasz komponent, jednak dodatkowo chcielibyśmy również być informowani o zmianie wartości pola. Możemy to osiągnąć nasłuchując na zdarzenia `change` wywoływane na elemencie `<select>` zarządzanym przez Chosen.

Nie przekazujemy `this.props.onChange` bezpośrednio do Chosen, ponieważ właściwości komponentu, włącznie z procedurami obsługi zdarzeń, mogą się zmieniać w czasie. Zamiast tego zadeklarujemy metodę `handleChange()`, która wywołuje `this.props.onChange`, i za pomocą jQuery zasubskrybujemy ją na zdarzenie `change`:

```

componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();

  this.handleChange = this.handleChange.bind(this);
  this.$el.on('change', this.handleChange);
}

componentWillUnmount() {
  this.$el.off('change', this.handleChange);
  this.$el.chosen('destroy');
}
```

```
handleChange(e) {
  this.props.onChange(e.target.value);
}
```

Wypróbuj kod na CodePen

Pozostała jeszcze jedna kwestia. W świecie Reacta właściwości są zmienne w czasie. Na przykład, komponent `<Chosen>` może otrzymać innych potomków po zmianie stanu komponentu nadrzędnego. Oznacza to, że w miejscach integracji koniecznie trzeba ręcznie aktualizować DOM w odpowiedzi na zmiany właściwości, ponieważ w tych miejscach React nie zrobi tego za nas.

Dokumentacja wtyczki Chosen sugeruje korzystać z interfejsu `trigger()` biblioteki jQuery do powiadamiania o zmianach na oryginalnym elemencie DOM. Każdy więc Reactowi zająć się aktualizowaniem `this.props.children` wewnętrz `<select>`, a ponadto dodajmy metodę cyklu życia `componentDidUpdate()`, która powiadomi Chosen o zmianach w liście potomków:

```
componentDidUpdate(prevProps) {
  if (prevProps.children !== this.props.children) {
    this.$el.trigger("chosen:updated");
  }
}
```

W ten sposób powiadomimy wtyczkę Chosen, że musi zaktualizować swoje elementy DOM, gdy zmienią się potomkowie elementu `<select>` zarządzanego przez Reacta.

Kompletna implementacja komponentu `Chosen` wygląda następująco:

```
class Chosen extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.chosen();

    this.handleChange = this.handleChange.bind(this);
    this.$el.on('change', this.handleChange);
  }

  componentDidUpdate(prevProps) {
    if (prevProps.children !== this.props.children) {
      this.$el.trigger("chosen:updated");
    }
  }

  componentWillUnmount() {
    this.$el.off('change', this.handleChange);
    this.$el.chosen('destroy');
  }

  handleChange(e) {
    this.props.onChange(e.target.value);
  }

  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
    );
  }
}
```

```

        </div>
    );
}
}

```

[Wypróbuj kod na CodePen](#)

Integracja z innymi bibliotekami do obsługi widoków

Osadzenie Reacta wewnątrz innych aplikacji jest możliwe dzięki elastyczności funkcji `createRoot()`.

Mimo że Reacta zwykle używa się na starcie aplikacji do załadowania jednego głównego komponentu do DOM, funkcję `createRoot()` można wywołać wielokrotnie w niezależnych fragmentach interfejsu, niezależnie od tego, czy są małe jak przycisk, czy dużych jak cała aplikacja.

Prawdę mówiąc, właśnie w taki sposób używamy Reacta na Facebooku. To podejście pozwala nam pisać aplikacje kawałek po kawałku i łączyć je z istniejącymi szablonami wygenerowanymi po stronie serwera czy z innym kodem klienckim.

Zastąpienie renderowania opartego na ciągu znaków

Popularnym wzorcem w starszych aplikacjach webowych było opisywanie fragmentów drzewa DOM za pomocą ciągu znaków, a następnie wstawianie ich do modelu na przykład tak: `$el.html(htmlString)`. Tego typu miejsca w kodzie są idealnymi kandydatami na wprowadzenie Reacta. Wystarczy przepisać owe fragmenty na komponenty reactowe.

Wobec tego taki kod napisany w jQuery...

```

$('#container').html('<button id="btn">Przywitaj się</button>');
$('#btn').click(function() {
  alert('Cześć!');
});

```

...mógłby zostać zastąpiony komponentem reactowym:

```

function Button() {
  return <button id="btn">Przywitaj się</button>;
}

$('#btn').click(function() {
  alert('Cześć!');
});

```

Od tego momentu możesz zacząć przenosić coraz więcej logiki do samego komponentu i stosować coraz więcej praktyk reactowych. Przykładowo, w komponentach zwykle nie stosuje się identyfikatorów, ponieważ komponenty mogą być renderowane wielokrotnie. Zamiast tego możemy skorzystać z [reactowego systemu zdarzeń](#) i podpiąć detektor zdarzeń bezpośrednio na reactowym elemencie `<button>`:

```

function Button(props) {
  return <button onClick={props.onClick}>Przywitaj się</button>;
}

```

```
function HelloButton() {
  function handleClick() {
    alert('Cześć!');
  }
  return <Button onClick={handleClick} />;
}
```

[Wypróbij kod na CodePen](#)

Możesz napisać tyle odizolowanych od siebie komponentów, ile tylko zechcesz, a także renderować je do różnych kontenerów w drzewie DOM za pomocą funkcji `ReactDOM.createRoot()`. Stopniowo gdy będziesz przekształcać coraz więcej kodu aplikacji, niektóre z tych komponentów uda się połączyć w większe komponenty, a wywołania funkcji `ReactDOM.createRoot()` będzie można przenieść w górę hierarchii.

Osadzanie kodu reactowego w widokach backbone'owych

Widoki biblioteki [Backbone](#) (ang. *Backbone Views*), aby wygenerować zawartość dla swoich elementów DOM, zwykle korzystają z HTML-owych ciągów znaków lub funkcji generujących ciągi znaków. Również i ten proces można zastąpić renderowaniem komponentów reactowych.

W poniższym kodzie tworzymy widok backbone'owy o nazwie `ParagraphView`. Nadpisujemy w nim backbone'ową funkcję `render()`, w której do elementu DOM dostarczonego przez Backbone'a (`this.el`) renderujemy komponent reactowy `<Paragraph>`. Tutaj również korzystamy z funkcji `ReactDOM.createRoot()`:

```
function Paragraph(props) {
  return <p>{props.text}</p>;
}

const ParagraphView = Backbone.View.extend({
  initialize(options) {
    this.reactRoot = ReactDOM.createRoot(this.el);
  },
  render() {
    const text = this.model.get('text');
    this.reactRoot.render(<Paragraph text={text} />);
    return this;
  },
  remove() {
    this.reactRoot.unmount();
    Backbone.View.prototype.remove.call(this);
  }
});
```

[Wypróbij kod na CodePen](#)

Musimy również pamiętać o wywołaniu funkcji `root.unmount()` w metodzie `remove`, dzięki której React podczas odmontowywania usunie detektory zdarzeń i inne zasoby powiązane z drzewem komponentów.

Kiedy komponent jest usuwany z *wewnątrz* drzewa Reacta, takie czyszczenie jest wykonywane automatycznie. Jednak ponieważ w tym przypadku ręcznie usuwamy całe drzewo, musimy jawnie wywołać tę metodę.

Integracja z warstwami modeli

Mimo iż ogólnie radzimy korzystać z jednokierunkowego przepływu danych, jak w przypadku [stanu reacto-wego](#), [Fluksa](#) czy [Reduksa](#), komponenty reactowe mogą zależeć od warstwy modelowej dostarczonej przez inne frameworki czy biblioteki.

Korzystanie z modeli backbone'owych w komponentach reactowych

Najprostszym sposobem na skorzystanie z [backbone'owych](#) modeli i kolekcji (ang. *Backbone Models and Collections*) wewnątrz komponentów reactowych jest nasłuchiwanie na różne zdarzenia i ręczne wymuszanie aktualizacji.

Komponenty odpowiedzialne za renderowanie modeli mogą nasłuchiwać na zdarzenia `'change'`, podczas gdy komponenty odpowiedzialne za renderowanie kolekcji mogą nasłuchiwać na zdarzenia `'add'` i `'remove'`. W obydwóch przypadkach należy wywołać metodę `this.forceUpdate()` w celu ponownego wyrenderowania komponentu z nowymi danymi.

W poniższym przykładzie komponent `List` renderuje kolekcję backbone'ową, a poszczególne elementy tej listy renderowane są przy użyciu komponentu `Item`.

```
class Item extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.model.on('change', this.handleChange);
  }

  componentWillUnmount() {
    this.props.model.off('change', this.handleChange);
  }

  render() {
    return <li>{this.props.model.get('text')}</li>;
  }
}

class List extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.collection.on('add', 'remove', this.handleChange);
  }
}
```

```

componentWillUnmount() {
  this.props.collection.off('add', 'remove', this.handleChange);
}

render() {
  return (
    <ul>
      {this.props.collection.map(model => (
        <Item key={model.cid} model={model} />
      ))}
    </ul>
  );
}
}

```

[Wypróbowuj kod na CodePen](#)

Pozyskiwanie danych z modeli backbone'owych

Powyższe podejście wymaga, aby twoje komponenty reactowe były świadome użycia w aplikacji modeli i kolekcji backbone'owych. Jeśli jednak masz w planach późniejszą migrację na inne rozwiązanie do zarządzania danymi, warto skupić wiedzę o Backbone w jak najmniejszej liczbie miejsc w kodzie.

Można to zrobić obsługując wszelkie zmiany w jednym miejscu aplikacji i zapisując je w prostej formie. Można też używać [komponentu wyższego rzędu](#), który pobiera wszystkie atrybuty modelu backbone'owego do stanu i przekazuje je do opakowywanego komponentu.

Dzięki temu tylko komponenty wyższego rzędu muszą umieć korzystać z modeli backbone'owych, natomiast pozostała część aplikacji może działać bez tej wiedzy.

W poniższym przykładzie stworzymy kopię atrybutów modelu i dodamy ją do wewnętrznego stanu komponentu. Zasubskrybijemy się także na zdarzenie `change` (i usuniemy detektor przy odmontowywaniu) i podczas zmiany danych będziemy uzupełniać stan na podstawie aktualnych danych z modelu. Wreszcie upewnimy się, że każda zmiana właściwości `model` spowoduje usunięcie subskrypcji ze starego modelu i podpięcie się na zmiany nowego.

Zwróć uwagę, że ten przykład nie uwzględnia wszystkich sposobów interakcji z Backbone'em, jednak powinien wystarczyć do zobrazowania sposobu, w jaki należy stworzyć docelowy mechanizm:

```

function connectToBackboneModel(WrappedComponent) {
  return class BackboneComponent extends React.Component {
    constructor(props) {
      super(props);
      this.state = Object.assign({}, props.model.attributes);
      this.handleChange = this.handleChange.bind(this);
    }

    componentDidMount() {
      this.props.model.on('change', this.handleChange);
    }

    componentWillReceiveProps(nextProps) {
      this.setState(Object.assign({}, nextProps.model.attributes));
      if (nextProps.model !== this.props.model) {
        this.props.model.off('change', this.handleChange);
      }
    }
  };
}

```

```
    nextProps.model.on('change', this.handleChange);
  }
}

componentWillUnmount() {
  this.props.model.off('change', this.handleChange);
}

handleChange(model) {
  this.setState(model.changedAttributes());
}

render() {
  const propsExceptModel = Object.assign({}, this.props);
  delete propsExceptModel.model;
  return <WrappedComponent {...propsExceptModel} {...this.state} />;
}
}
```

Aby pokazać sposób użycia powyższego kodu, połączymy komponent reaktywny `NameInput` z modelem backbone'owym i będziemy aktualizować jego atrybut `firstName` przy każdej zmianie wartości pola formularza:

```
function NameInput(props) {
  return (
    <p>
      <input value={props.firstName} onChange={props.handleChange} />
      <br />
      Nazywam się {props.firstName}.
    </p>
  );
}

const BackboneNameInput = connectToBackboneModel(NameInput);

function Example(props) {
  function handleChange(e) {
    props.model.set('firstName', e.target.value);
  }

  return (
    <BackboneNameInput
      model={props.model}
      handleChange={handleChange}
    />
  );
}

const model = new Backbone.Model({ firstName: 'Staszek' });
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Example model={model} />);
```

[Wypróbuj kod na CodePen](#)

Ta technika nie jest ograniczona tylko do Backbone'a. Z Reactem możesz używać dowolnej biblioteki modelowej. Wystarczy zasubskrybować się na zmiany w metodach cyklu życia komponentu i, ewentualnie, kopować dane do lokalnego stanu reactowego.

react

[Go to TOC](#)

Wprowadzenie do JSX

Weźmy na warsztat poniższą instrukcję przypisania:

```
const element = <h1> Witaj, świecie! </h1>;
```

Nieco zaskakującą może się tutaj wydawać przypisywana wartość, która nie jest ani typowym łańcuchem znaków, ani HTML-em.

Taką składnię nazywamy JSX i jest to rozszerzenie składni JavaScriptu o możliwość wstawiania znaczników. Zalecamy używanie jej wraz z Reactem do opisywania, jak powinien wyglądać interfejs graficzny strony. JSX może przypominać język oparty o szablony, jednakże daje on do dyspozycji pełnię możliwości JavaScriptu.

JSX jest tłumaczony jeden do jednego na reactowe "elementy". Sposoby ich wyświetlania na stronie poznamy w [następnym rozdziale](#). Poniżej znajdziesz podstawy składni JSX, które pozwolą ci szybko zacząć tworzyć kod.

Dlaczego JSX?

React przyjmuje ideę, że logika związana z prezentacją danych jest z natury rzeczy powiązana z innymi elementami logiki biznesowej UI: sposobami przetwarzania zdarzeń w aplikacji, tym, jak stan aplikacji zmienia się w czasie, jak również tym, jak dane są przygotowywane do wyświetlenia.

Zamiast sztucznie rozdzielać *technologie*, umiejscawiając znaczniki oraz logikę aplikacji w osobnych plikach, React wprowadza [podział odpowiedzialności](#) poprzez wprowadzenie luźno powiązanych jednostek, nazywanych "komponentami", które zawierają zarówno znaczniki HTML, jak i związaną z nimi logikę.

React [nie wymaga](#) używania JSX, jednakże większość programistów uważa go za przydatne narzędzie, unaczyniające to, co dzieje się w kodzie javascriptowym operującym na interfejsach graficznych. Pomaga on również Reactowi wyświetlać bardziej użyteczne informacje o błędach i ostrzeżenia.

Pora sprawdzić, jak to działa - do dzieła!

Osadzanie wyrażeń w JSX

W poniższym przykładzie deklarujemy zmienną `name`, a następnie używamy jej wewnątrz JSX, po prostu otaczając ją nawiasami klamrowymi:

```
const name = 'Grzegorz';
const element = <h1> Witaj, {name} </h1>;
```

Zauważ, że dowolne [wyrażenie JavaScriptowe](#) możesz osadzić w JSX poprzez otoczenie go klamrami. Przykładowo, możesz użyć `2 + 2`, `user.firstName` czy `formatName(user)`, jako że są to prawidłowe wyrażenia w języku JavaScript.

W poniższym przykładzie umiejscawiamy wynik wywołania funkcji javascriptowej `formatName(user)` wewnątrz znacznika `<h1>`:

```

function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Grzegorz',
  lastName: 'Brzęczyszczkiewicz',
};

const element = (
  <h1>
    Witaj, {formatName(user)}!
  </h1>
);

```

[Przetestuj kod na CodePen](#)

Dla czytelności podzieliliśmy kod JSX na kilka linii. Nie jest to wprawdzie wymagane, ale polecamy również otaczanie go w nawiasy. Pozwoli to uniknąć pułapek związanych z [automatycznym wstawianiem średników](#) w JavaScriptie.

JSX również jest wyrażeniem

Po komplikacji, wyrażenia JSX-owe stają się zwykłymi wywołaniami funkcji w JavaScriptie i są ewaluowane do obiektów javascriptowych.

Oznacza to, że możesz używać znaczników JSX również wewnętrz instrukcji warunkowych `if`, pętli `for`, przypisywać je do zmiennych, oczekiwając przekazania jako argumenty do funkcji, zwracać z funkcji i w wielu innych miejscach.

```

function getGreeting(user) {
  if (user) {
    return <h1>Witaj, {formatName(user)}!</h1>;
  }
  return <h1>Witaj, nieznajomy.</h1>;
}

```

Podawanie atrybutów w JSX

Możesz przekazać literał znakowy jako atrybut, używając poniższej składni:

```
const element = <a href="https://www.reactjs.org"> link </a>;
```

Możesz także przypisać wartość dowolnego wyrażenia javascriptowego do atrybutu, używając nawiasów klamrowych:

```
const element = <img src={user.avatarUrl}></img>;
```

Nie otaczaj jednak klamer cudzysłowami, gdy chcesz przekazać wartość jakiegoś wyrażenia do atrybutu. Zalecamy użycie albo cudzysłów (dla statycznych ciągów znaków), albo klamer (dla wartości wyrażeń), ale nie obydwa naraz (dla tego samego atrybutu).

Uwaga:

Jako że składni JSX jest bliżej do JavaScriptu niż do HTML-a, React DOM do nazywania argumentów używa notacji `camelCase` zamiast nazw atrybutów HTML-owych.

Przykładowo, w JSX `class` staje się `className`, zaś zamiast `tabindex` używamy `tabIndex`.

Specyfikowanie elementów potomnych w JSX

Jeśli znacznik jest pusty, możesz zakończyć go bezpośrednio przy pomocy `/>`, podobnie jak ma to miejsce w XML-u:

```
const element = <img src={user.avatarUrl} />;
```

Znaczniki JSX mogą jednak również zawierać elementy potomne:

```
const element = (
  <div>
    <h1>Witaj!</h1>
    <h2>Dobrze cię widzieć.</h2>
  </div>
);
```

JSX jest odporny na ataki przez wstrzyknięcie kodu

W JSX można bezpiecznie osadzać dane wprowadzone przez użytkownika:

```
const title = response.potencjalnieZlosliweDane;
// To jest bezpieczne:
const element = <h1>{title}</h1>;
```

Domyślnie, React DOM stosuje [znaki ucieczki](#) w wartościach używanych w JSX, zanim je wyświetli. W ten sposób upewniamy się, że nic, co nie zostało bezpośrednio napisane w kodzie twojej aplikacji, nie zostanie wstrzyknięte w kod strony. Wszystko, co wyświetlamy, zamieniane jest na statyczne łańcuchy znaków. Pozwala to uniknąć ataków typu [XSS \(Cross-site scripting\)](#).

JSX reprezentuje obiekty

Zanim twój kod JSX będzie mógł zostać wyświetlony w przeglądarce, musi zostać odpowiednio przetworzony przez narzędzie takie jak *Babel*. Babel transpiluje znaczniki JSX do wywołań funkcji `React.createElement`.

Poniższe dwa fragmenty kodu są sobie równoważne:

```
const element = (
  <h1 className="greeting">
    Witaj, świecie!
  </h1>
);
```

```
const element = React.createElement(  
  'h1',  
  {className: 'greeting'},  
  'Witaj, świecie!'  
)
```

Funkcja `React.createElement` wykonuje kilka sprawdzeń, które ułatwiają pisanie kodu wolnego od błędów. W swej istocie jednak, zwraca ona po prostu obiekt podobny do poniższego:

```
// Note: this structure is simplified  
const element = {  
  type: 'h1',  
  props: {  
    className: 'greeting',  
    children: 'Witaj, świecie'  
  };
```

Takie obiekty nazywamy "*elementami reactowymi*". Możesz o nich myśleć jak o sposobie opisania tego, co znajdzie się na ekranie. `ReactDOM` odczytuje te obiekty i używa ich do skonstruowania drzewa DOM twojej strony, jak również do późniejszego odświeżania go.

Więcej o wyświetlaniu elementów reactowych dowiesz się w [następnym rozdziale](#).

Wskaźówka:

Polecamy używanie "[Babela](#)" jako definicji języka w twoim ulubionym edytorze. Pozwoli to na poprawne podświetlanie składni ES6 i JSX.

[Go to TOC](#)

JSX w szczegółach

Zasadniczo, JSX dostarcza lukier składniowy dla funkcji `React.createElement(component, props, ...children)`. Kod JSX:

```
<MyButton color="blue" shadowSize={2}>
  Kliknij mnie
</MyButton>
```

jest komplikowany do:

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Kliknij mnie'
)
```

Jeżeli element nie posiada zagnieżdżonych tagów, możesz użyć tagu samozamykającego się. Wtedy kod:

```
<div className="sidebar" />
```

jest komplikowany do:

```
React.createElement(
  'div',
  {className: 'sidebar'}
)
```

Aby przetestować, jak JSX jest zamieniany na JavaScript, możesz wypróbować [kompilator Babel online](#).

Określanie typu elementu

Pierwsza część tagu JSX określa typ elementu reactowego.

Wielka litera w tagu JSX oznacza, że jest on komponentem reactowym. Takie tagi są bezpośrednio komplikowane na referencję do zmiennej, więc jeżeli używasz w swoim kodzie JSX wyrażenia `<Foo />`, to `Foo` musi znajdować się w zakresie.

React musi znajdować się w zakresie

Jako że JSX komplikowany jest do wywołania `React.createElement`, biblioteka `React` musi być w zakresie, w którym znajduje się twój kod JSX.

W przykładzie poniżej oba importy są konieczne, nawet jeżeli `React` i `CustomButton` nie mają żadnych bezpośrednich odniesień z kodu JavaScript:

```
import React from 'react';
import CustomButton from './CustomButton';

function WarningButton() {
```

```
// return React.createElement(CustomButton, {color: 'red'}, null);
return <CustomButton color="red" />;
}
```

Jeżeli nie używasz rozwiązania budującego paczkę z kodem JavaScript i ładujesz Reacta za pomocą tagu `<script>`, jest on dostępny jako globalna zmienna `React`.

Używanie notacji kropkowej w JSX

Możesz odnosić się do reactowych komponentów za pomocą notacji kropkowej. Jest to wygodne rozwiązanie w sytuacji, gdy z jednego modułu eksportujesz wiele komponentów. Przykładowo, jeżeli `MyComponents.DatePicker` jest komponentem, możesz użyć go bezpośrednio w JSX w następujący sposób:

```
import React from 'react';

const MyComponents = {
  DatePicker: function DatePicker(props) {
    return <div>Wyobraź sobie, że jest tutaj kalendarz w kolorze {props.color}.
  </div>;
}
}

function BlueDatePicker() {
  return <MyComponents.DatePicker color="blue" />;
}
```

Komponenty zdefiniowane przez użytkownika muszą być pisane wielką literą

Jeżeli nazwa elementu zaczyna się od małej litery, oznacza to odniesienie do wbudowanego komponentu, takiego jak na przykład: `<div>` lub ``, i skutkuje przekazaniem ciągu znaków `'div'` lub `'span'` do funkcji `React.createElement`. Nazwy typów, które zaczynają się od wielkiej litery, na przykład: `<Foo />`, są komplikowane do wywołania `React.createElement(Foo)` i odnoszą się do komponentów zdefiniowanych lub zaimportowanych do dwóch plików zawierających kod JavaScript.

Rekomendujemy używanie wielkich liter w nazwach komponentów. Jeżeli twój komponent ma nazwę rozpoczęającą się od małej litery, przypisz ją do zmiennej, której nazwa zaczyna się od wielkiej litery, zanim użyjesz go w JSX.

Na przykład, poniższy kod nie zachowa się tak, jak można by tego oczekiwano:

```
import React from 'react';

// Źle! To jest komponent i jego nazwa powinna zaczynać się wielką literą:
function hello(props) {
  // Dobrze! To użycie <div> jest w porządku, bo jest on poprawnym, wbudowanym
  tagiem HTML:
  return <div>Witaj, {props.toWhat}</div>;
}

function HelloWorld() {
  // Źle! React traktuje <hello /> jak element wbudowany HTML, bo jego nazwa nie
```

```

zaczyna się od wielkiej litery:
return <hello toWhat="Świecie" />;
}

```

Aby to naprawić, zmienimy nazwę z `hello` na `Hello`, a w miejscu użycia napiszemy `<Hello />`:

```

import React from 'react';

// Dobrze! To jest komponent, więc powinien mieć nazwę piszącą wielką literą:
function Hello(props) {
    // Dobrze! Użycie <div> jest w porządku, bo div jest poprawnym elementem
    // HTML:
    return <div>Witaj, {props.toWhat}</div>;
}

function HelloWorld() {
    // Dobrze! React traktuje <Hello /> jak komponent, bo jego nazwa zaczyna się od
    // wielkiej litery:
    return <Hello toWhat="Świecie" />;
}

```

Określanie typu w trakcie działania kodu

Używanie wyrażeń jako elementów w Reakcie jest zabronione. Aby użyć wyrażenia do wskazania komponentu, należy przypisać je do zmiennej o nazwie pisanej z wielkiej litery. To rozwiązanie jest użyteczne w przypadku wyświetlania komponentu w zależności od przekazanego atrybutu:

```

import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
    photo: PhotoStory,
    video: VideoStory
};

function Story(props) {
    // Źle! Komponent w JSX nie może być wyrażeniem.
    return <components[props.storyType] story={props.story} />;
}

```

Aby to naprawić, przypisujemy najpierw wyrażenie do zmiennej z nazwą zaczynającą się od wielkiej litery:

```

import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
    photo: PhotoStory,
    video: VideoStory
};

function Story(props) {
    // Dobrze! Komponenty w JSX mogą być przypisywane i przechowywane w zmiennych
    // pisanych wielką literą.
    const SpecificStory = components[props.storyType];
    return <SpecificStory story={props.story} />;
}

```

Właściwości w JSX

Istnieje kilka sposobów na przekazanie właściwości (ang. *props*) w JSX.

Wyrażenie javascriptowe jako właściwość

Możesz przekazać dowolne wyrażenie javascriptowe jako właściwość, jeżeli otoczysz je klamrami `{}`. Przykładowo:

```
<MyComponent foo={1 + 2 + 3 + 4} />
```

Dla komponentu `MyComponent`, wartość `props.foo` będzie równa `10`, ponieważ wyrażenie `1 + 2 + 3 + 4` zostanie wykonane.

`if` i `for` nie są wyrażeniami w JavaScriptie. Z tego powodu nie mogą być bezpośrednio przekazywane jako właściwość. Zamiast tego, można użyć ich do warunkowego przypisania wartości wyrażenia do zmiennej i zmienną przekazać jako właściwość. Na przykład tak, jak poniżej:

```
function NumberDescriber(props) {
  let description;
  if (props.number % 2 == 0) {
    description = <strong>parzystą</strong>;
  } else {
    description = <i>nieparzystą</i>;
  }
  return <div>{props.number} jest liczbą {description}</div>;
}
```

Możesz dowiedzieć się więcej na temat [warunkowego renderowania](#) i [pętli](#) na odpowiednich stronach.

Literały tekstowe

Literały tekstowe można przekazywać jako wartość właściwości. Poniższe przykłady są równoważne:

```
<MyComponent message="hello world" />
<MyComponent message={['hello world']} />
```

Gdy przekazujesz literał tekstowy, w jego treści zakodowywany jest HTML. Dlatego poniższe wyrażenia będą miały ten sam wynik:

```
<MyComponent message=&lt;3&gt; />
<MyComponent message={['<3'']} />
```

Zwykle jednak konsekwencje tego zachowania nie są istotne, ale zostały wspomniane dla kompletności dokumentacji.

Przekazane właściwości domyślnie mają wartość "true"

Gdy przekażesz atrybut bez jawnego podawania wartości, domyślnie przyjmie on wartość `true`. Poniższe wyrażenia JSX są równoważne:

```
<MyTextBox autocomplete />
<MyTextBox autocomplete={true} />
```

Nie zalecamy jednak *nie* przekazywania żadnej wartości, ponieważ może być to pomyłone ze [skrótnymapisem definicji obiektu w ES6](#) `{foo}`, który jest równoznaczny z `{foo: foo}`, a nie `{foo: true}`. To zachowanie w składni JSX zostało zaimplementowane, żeby odwzorować sposób działania atrybutów HTML.

Właściwości rozszczepione (ang. spread)

Jeżeli zmienna `props` jest obiektem, a chcesz przekazać ją w JSX, możesz użyć operatora rozszczepienia (`...`). Poniższe implementacje komponentów są równoznaczne:

```
function App1() {
  return <Greeting firstName="Ben" lastName="Hector" />;
}

function App2() {
  const props = {firstName: 'Ben', lastName: 'Hector'};
  return <Greeting {...props} />;
}
```

Możesz również wybrać poszczególne atrybuty, których będzie potrzebował twój komponent, a pozostałe przekazać do jego dzieci używając operatora rozszczepienia.

```
const Button = props => {
  const { kind, ...other } = props;
  const className = kind === "primary" ? "PrimaryButton" : "SecondaryButton";
  return <button className={className} {...other} />;
};

const App = () => {
  return (
    <div>
      <Button kind="primary" onClick={() => console.log("kliknięto!")}>
        Witaj, Świecie!
      </Button>
    </div>
  );
};
```

W powyższym przykładzie właściwość `kind` jest używana w komponencie i nie jest przekazywana do elementu DOM `<button>`. Wszystkie pozostałe właściwości są przekazywane poprzez rozszczepiony obiekt `...other`, co powoduje, że komponent jest elastyczny pod względem przyjmowanych właściwości. W tym przykładzie przekazane zostały atrybuty: `onClick` i `children`.

Operator rozszczepienia może być bardzo pomocny, lecz używając go, łatwo jest stracić kontrolę nad przekazywanymi właściwościami. Może się zdarzyć, że przekażesz niepoprawny atrybut HTML lub nadmiarową właściwość do komponentu, który wcale go nie potrzebuje. Dlatego zalecamy ostrożność w używaniu tego rozwiązania.

Elementy potomne w JSX

W JSX, jeżeli wyrażenie posiada zarówno tag otwierający, jak i zamykający, to jego elementy potomne są przekazywane jako specjalna właściwość: `props.children`. Istnieje kilka sposobów na przekazanie właściwości `children`:

Literały tekstowe

Możesz umieścić łańcuch znaków pomiędzy tagiem otwierającym a zamykającym. Wtedy wartość `props.children` będzie równa przekazanemu łańcuchowi znaków. Jest to szczególnie przydatne w użyciu z niektórymi wbudowanymi elementami HTML. Na przykład:

```
<MyComponent>Witaj, świecie!</MyComponent>
```

Powyższy kod jest poprawnym wyrażeniem JSX, a wartość `props.children` w komponencie `MyComponent` będzie równa przekazanemu łańcuchowi znaków ` "Witaj, świecie!". HTML zostanie odkodowany, więc możesz pisać w JSX tak, jak w zwykłym HTML-u:

```
<div>To jest poprawny kod HTML &mdash; ale też JSX.</div>
```

JSX usuwa białe znaki na początku i końcu linii, a także: puste linie, puste linie przylegające do elementów, puste linie wewnętrz łańcuchów znaków. Białe znaki w łańcuchach znaków są zamieniane na pojedyncze spacje. Z tego powodu wszystkie poniższe przykłady są renderowane w ten sam sposób:

```
<div>Witaj, świecie</div>
<div>
  Hello World
</div>
<div>
  Hello
  World
</div>
<div>
  Hello World
</div>
```

Elementy potomne w JSX

W JSX możesz przekazywać więcej niż jeden element jako potomny. Przydaje się to przy wyświetaniu zagnieżdzonych komponentów:

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

Przekazywane elementy mogą być różnego typu, a więc możesz używać łańcuchów znaków wraz z innymi rodzajami elementów potomnych. Jest to kolejne podobieństwo do HTML-a. Poniższy kod jest zarówno poprawnym JSX, jak i HTML:

```
<div>
  Oto lista:
  <ul>
    <li>Element 1</li>
    <li>Element 2</li>
  </ul>
</div>
```

Komponenty mogą również zwracać tablicę elementów::

```
render() {
  // Nie ma potrzeby otaczać elementów listy dodatkowym elementem JSX!
  return [
    // Pamiętaj o kluczach :)
    <li key="A">Pierwszy</li>,
    <li key="B">Drugi</li>,
    <li key="C">Trzeci</li>,
  ];
}
```

Wyrażenia javascriptowe jako komponenty potomne

Möesz przekazać dowolne wyrażenie javascriptowe, obejmując je klamrami `{ }`. Poniższe wyrażenia są równoważne:

```
<MyComponent>foo</MyComponent>
<MyComponent>{'foo'}</MyComponent>
```

To rozwiązanie jest często przydatne przy renderowaniu list. Na przykład, poniższy kod renderuje listę elementów HTML:

```
function Item(props) {
  return <li>{props.message}</li>;
}

function TodoList() {
  const todos = ['dokończyć dokumentację', 'wystawić PR-a', 'namówić Dana na review'];
  return (
    <ul>
      {todos.map((message) => <Item key={message} message={message} />)}
    </ul>
  );
}
```

Wyrażenia javascriptowe można również używać razem z innymi typami danych:

```
function Hello(props) {
  return <div>Cześć, {props.addressee}!</div>;
}
```

Funkcje jako komponenty potomne

Zwykłe wyrażenia javascriptowe przekazywane w JSX są przekształcane na łańcuchy znaków, elementy reaktywe lub listy tych rzeczy. Jednakże właściwość `props.children` działa jak każda inna i może przekazać dowolny rodzaj danych, nie tylko takie, które React wie, jak wyświetlić. Na przykład, twój komponent może przyjmować poprzez `props.children` funkcję zwrotną:

```
// Wywołuje potomną funkcję zwrotną numTimes, aby powtórzyć wyświetlenie elementów
function Repeat(props) {
  let items = [];
  for (let i = 0; i < props.numTimes; i++) {
    items.push(props.children(i));
  }
  return <div>{items}</div>;
}

function ListOfTenThings() {
  return (
    <Repeat numTimes={10}>
      {(index) => <div key={index}>To jest {index}. element listy</div>}
    </Repeat>
  );
}
```

Elementy potomne przekazane do własnych komponentów mogą mieć dowolny typ, pod warunkiem, że React będzie w stanie zamienić je na zrozumiałe dla siebie typ danych przed wyświetleniem. To podejście nie jest zbyt popularne w składni JSX, ale można w ten sposób rozszerzać jej możliwości.

Wartości logiczne, Null, i Undefined są ignorowane

`false`, `null`, `undefined` i `true` są poprawnymi potomkami, ale nie są renderowane. Wszystkie poniższe wyrażenia JSX będą miały ten sam efekt:

```
<div />
<div></div>
<div>{false}</div>
<div>{null}</div>
<div>{undefined}</div>
<div>{true}</div>
```

Może się to okazać przydatne przy warunkowym renderowaniu elementów. Na przykład, poniższy kod wyrenderuje komponent `<Header />` tylko, gdy wartość zmiennej `showHeader` jest równa `true`:

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

Warto pamiętać o tym, że niektóre fałszopodobne wartości (ang. *falsy values*), takie jak `0`, będą renderowane przez Reacta. Przykładowo, poniższy kod nie zachowa się tak, jak można by na pierwszy rzut oka po myśleć, ponieważ gdy atrybut `props.messages` będzie pustą tablicą, wyświetlona zostanie wartość `0`:

```
<div>
  {props.messages.length &&
   <MessageList messages={props.messages} />
  }
</div>
```

Aby to naprawić, należy upewnić się, że wyrażenie przed `&&` zawsze jest wartością logiczną:

```
<div>
  {props.messages.length > 0 &&
   <MessageList messages={props.messages} />
  }
</div>
```

Jeżeli chcesz, aby wartość taka jak: `false`, `true`, `null` lub `undefined` została wyświetlona, w pierwszej kolejności należy przekonwertować ją na łańcuch znaków:

```
<div>
  Moja javascriptowa zmienna to {String(myVariable)}.
</div>
```

Legacy Context

Note:

The legacy context API will be removed in a future major version. Use the [new context API](#) introduced with version 16.3. The legacy API will continue working for all 16.x releases.

How To Use Context

This section documents a legacy API. See the [new API](#).

Suppose you have a structure like:

```
class Button extends React.Component {
  render() {
    return (
      <button style={{background: this.props.color}}>
        {this.props.children}
      </button>
    );
  }
}

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button color={this.props.color}>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  render() {
    const color = "purple";
    const children = this.props.messages.map((message) =>
      <Message text={message.text} color={color} />
    );
    return <div>{children}</div>;
  }
}
```

In this example, we manually thread through a `color` prop in order to style the `Button` and `Message` components appropriately. Using context, we can pass this through the tree automatically:

```
import PropTypes from 'prop-types';

class Button extends React.Component {
```

```

render() {
  return (
    <button style={{background: this.context.color}}>
      {this.props.children}
    </button>
  );
}

Button.contextTypes = {
  color: PropTypes.string
};

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  getChildContext() {
    return {color: "purple"};
  }

  render() {
    const children = this.props.messages.map((message) =>
      <Message text={message.text} />
    );
    return <div>{children}</div>;
  }
}

MessageList.childContextTypes = {
  color: PropTypes.string
};

```

By adding `childContextTypes` and `getChildContext` to `MessageList` (the context provider), React passes the information down automatically and any component in the subtree (in this case, `Button`) can access it by defining `contextTypes`.

If `contextTypes` is not defined, then `context` will be an empty object.

Note:

`React.PropTypes` has moved into a different package since React v15.5. Please use [the prop-type library instead](#) to define `contextTypes`.

We provide a [codemod script](#) to automate the conversion.

Parent-Child Coupling

This section documents a legacy API. See the [new API](#).

Context can also let you build an API where parents and children communicate. For example, one library that works this way is [React Router V4](#):

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';

const BasicExample = () => (
  <Router>
    <div>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/topics">Topics</Link></li>
      </ul>

      <hr />

      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
      <Route path="/topics" component={Topics} />
    </div>
  </Router>
);
```

By passing down some information from the `Router` component, each `Link` and `Route` can communicate back to the containing `Router`.

Before you build components with an API similar to this, consider if there are cleaner alternatives. For example, you can pass entire React components as props if you'd like to.

Referencing Context in Lifecycle Methods

This section documents a legacy API. See the [new API](#).

If `contextTypes` is defined within a component, the following [lifecycle methods](#) will receive an additional parameter, the `context` object:

- `constructor(props, context)`
- `componentWillReceiveProps(nextProps, nextContext)`
- `shouldComponentUpdate(nextProps, nextState, nextContext)`
- `componentWillUpdate(nextProps, nextState, nextContext)`

Note:

As of React 16, `componentDidUpdate` no longer receives `prevContext`.

Referencing Context in Function Components

This section documents a legacy API. See the [new API](#).

Function components are also able to reference `context` if `contextTypes` is defined as a property of the function. The following code shows a `Button` component written as a function component.

```
import PropTypes from 'prop-types';

const Button = ({children}, context) =>
  <button style={{background: context.color}}>
    {children}
  </button>

Button.contextTypes = {color: PropTypes.string};
```

Updating Context

This section documents a legacy API. See the [new API](#).

Don't do it.

React has an API to update context, but it is fundamentally broken and you should not use it.

The `getChildContext` function will be called when the state or props changes. In order to update data in the context, trigger a local state update with `this.setState`. This will trigger a new context and changes will be received by the children.

```
import PropTypes from 'prop-types';

class MediaQuery extends React.Component {
  constructor(props) {
    super(props);
    this.state = {type: 'desktop'};
  }

  getChildContext() {
    return {type: this.state.type};
  }

  componentDidMount() {
    const checkMediaQuery = () => {
      const type = window.matchMedia("(min-width: 1025px)").matches ? 'desktop' :
      'mobile';
      if (type !== this.state.type) {
        this.setState({type});
      }
    };
    checkMediaQuery();
    window.addEventListener('resize', checkMediaQuery);
  }
}
```

```
};

window.addEventListener('resize', checkMediaQuery);
checkMediaQuery();
}

render() {
  return this.props.children;
}

MediaQuery.childContextTypes = {
  type: PropTypes.string
};
```

The problem is, if a context value provided by component changes, descendants that use that value won't update if an intermediate parent returns `false` from `shouldComponentUpdate`. This is totally out of control of the components using context, so there's basically no way to reliably update the context. [This blog post](#) has a good explanation of why this is a problem and how you might get around it.

[Go to TOC](#)

Event Pooling

Note

This page is only relevant for React 16 and earlier, and for React Native.

React 17 on the web **does not** use event pooling.

[Read more](#) about this change in React 17.

The `SyntheticEvent` objects are pooled. This means that the `SyntheticEvent` object will be reused and all properties will be nullified after the event handler has been called. For example, this won't work:

```
function handleChange(e) {
  // This won't work because the event object gets reused.
  setTimeout(() => {
    console.log(e.target.value); // Too late!
  }, 100);
}
```

If you need to access event object's properties after the event handler has run, you need to call `e.persist()`:

```
function handleChange(e) {
  // Prevents React from resetting its properties:
  e.persist();

  setTimeout(() => {
    console.log(e.target.value); // Works
  }, 100);
}
```

Wynoszenie stanu w górę

Bardzo często kilka komponentów jednocześnie musi odzwierciedlać te same zmiany w danych. W takim przypadku proponujemy przeniesienie wspólnego stanu do najbliższego wspólnego przodka. Zobaczmy, jak wygląda to w praktyce.

W tej części poradnika stworzymy kalkulator, który obliczy nam czy woda będzie się gotować w podanej temperaturze.

Rozpoczniemy od komponentu, który nazwiemy `BoilingVerdict`. Komponent ten przyjmie atrybut z temperaturą o nazwie `celsius`, a następnie zwróci informację, czy temperatura jest wystarczająco wysoka.

```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>Woda będzie się gotować</p>;
  }
  return <p>Woda nie będzie się gotować.</p>;
}
```

Następnie stwórzmy komponent o nazwie `Calculator`. Wyrenderuje on element `<input>`, który pozwoli wpisać temperaturę oraz zachowa jego wartość w `this.state.temperature`.

Dodatkowo, będzie on renderował komponent `BoilingVerdict` dla obecnej wartości inputa.

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    return (
      <fieldset>
        <legend>Podaj temperaturę w Celsjuszach:</legend>
        <input
          value={temperature}
          onChange={this.handleChange} />
        <BoilingVerdict
          celsius={parseFloat(temperature)} />
      </fieldset>
    );
  }
}
```

[Przetestuj kod na CodePen](#)

Dodawanie drugiego inputa

Naszym kolejnym wymogiem jest, aby oprócz inputa do wpisywania temperatury w Celsjuszach, dostarczyć także drugi input, który przyjmie temperaturę w Fahrenheitach. Oba inputy powinny być ze sobą zsynchronizowane.

Zacznijmy od wyizolowania komponentu `TemperatureInput` z komponentu `Calculator`. Dodamy do niego nowy atrybut `scale`, który będzie mógł przyjmować wartość `"c"` lub `"f"`:

```
const scaleNames = {
  c: 'Celsjuszach',
  f: 'Fahrenheitach'
};

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Podaj temperaturę w {scaleNames[scale]}:</legend>
        <input value={temperature}>
          onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

Zmieńmy komponent `Calculator` tak, by renderował dwa osobne inputy z temperaturą:

```
class Calculator extends React.Component {
  render() {
    return (
      <div>
        <TemperatureInput scale="c" />
        <TemperatureInput scale="f" />
      </div>
    );
  }
}
```

[Przetestuj kod na CodePen](#)

Mamy teraz dwa inputy, jednak jeśli podamy temperaturę w jednym z nich, drugi nie zostanie zaktualizowany. Jest to sprzeczne z naszymi wymogami: chcemy, by oba inputy były ze sobą zsynchronizowane.

Nie możemy też wyświetlić `BoilingVerdict` z poziomu komponentu `Calculator`. Spowodowane jest to faktem, iż `Calculator` nie ma dostępu do informacji o obecnej temperaturze, która schowana jest w `TemperatureInput`.

Pisanie funkcji konwertujących

Na początek napiszmy dwie funkcje do konwertowania temperatury ze stopni Celsiusza na Fahrenheita i odwrotnie:

```
function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}
```

Obie te funkcje konwertują liczby. Napiszmy jeszcze jedną funkcję, która jako argumenty przyjmie ciąg znaków `temperature` oraz funkcję konwertującą, a zwróci inny ciąg znaków. Użyjemy jej do wyliczenia wartości jednego inputa w oparciu o drugi.

Funkcja zwróci zaokrąglony do trzeciego miejsca po przecinku wynik lub pusty ciąg znaków, jeśli `temperature` jest nieprawidłowe.

```
function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}
```

Na przykład, `tryConvert('abc', toCelsius)` zwróci pusty ciąg znaków, natomiast `tryConvert('10.22', toFahrenheit)` zwróci `'50.396'`.

Wynoszenie stanu w górę

Obecnie obydwa komponenty `TemperatureInput` trzymają swoje wartości niezależnie, w lokalnym stanie:

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }
}
```

```
render() {
  const temperature = this.state.temperature;
  // ...
```

Nam zależy jednak, aby oba te inputy były ze sobą zsynchronizowane. Podczas aktualizacji inputa z Celcjuszami, input z Fahrenheitami powinien odzwierciedlać przekonwertowaną temperaturę i odwrotnie.

W Reakcie współdzielenie stanu komponentu można osiągnąć poprzez utworzenie stanu w najbliższym wspólnym przodku. Nazywa się to "wynoszeniem stanu w górę" (ang. *lifting state up*).

Jeśli wyniesiemy stan do komponentu `Calculator`, zostanie on "źródłem prawdy" dla obecnej temperatury w obu inputach. Komponent ten może poinformować je o trzymaniu zgodnych wartości. Oba inputy będą ze sobą zawsze zsynchronizowane, ponieważ atrybuty dla komponentów `TemperatureInput` przychodzą ze wspólnego przodka - komponentu `Calculator`.

Przeanalizujmy to krok po kroku.

Najpierw zastąpimy `this.state.temperature` używając `this.props.temperature` w komponencie `TemperatureInput`. Na tę chwilę przyjmijmy, że atrybut `this.props.temperature` istnieje, jednakże w przyszłości będziemy musieli go przekazać z komponentu `Calculator`:

```
render() {
  // Wcześniej było: const temperature = this.state.temperature;
  const temperature = this.props.temperature;
  // ...
```

Wiemy już, że [atrybuty są tylko do odczytu](#). Kiedy `temperature` znajdowało się w lokalnym stanie, komponent `TemperatureInput` mógł po prostu wywołać `this.setState()`, by je zmienić. Jednak, ponieważ teraz atrybut `temperature` jest przekazywany od rodzica, komponent `TemperatureInput` nie ma nad nim żadnej kontroli.

W Reakcie rozwiążujemy to na ogół poprzez przekształcenie komponentu w "kontrolowany". Tak samo jak w drzewie DOM, gdzie element `<input>` akceptuje zarówno atrybut `value`, jak i `onChange` jako atrybuty, tak i komponent `TemperatureInput` może akceptować od rodzica atrybuty `temperature` oraz `onTemperatureChange`.

Teraz za każdym razem, gdy komponent `TemperatureInput` zechce zaktualizować temperaturę, wywoła funkcję `this.props.onTemperatureChange`:

```
handleChange(e) {
  // Wcześniej było: this.setState({temperature: e.target.value});
  this.props.onTemperatureChange(e.target.value);
  // ...
```

Wskazówka:

Nazwy, których używamy w tym poradniku, takie jak `temperature` czy `onTemperatureChange`, są tylko nazwami przykładowymi i można je zmienić w dowolny sposób. Jednak przyjęło się nazywać je np. `value` i `onChange`.

Atrybuty `onTemperatureChange` oraz `temperature` zostaną przekazane przez rodzica - komponent `Calculator`. Obsłuży on zmianę poprzez zmodyfikowanie swojego stanu lokalnego, zatem oba inputy zostaną ponownie wyrenderowane z nowymi wartościami. Już wkrótce przyjrzymy się nowej implementacji komponentu `Calculator`.

Zanim jednak zagłębiemy się w zmiany w komponencie `Calculator`, podsumujmy co zmieniło się w `TemperatureInput`. Pozbyliśmy się stanu lokalnego i zamiast korzystać z `this.state.temperature`, możemy teraz używać `this.props.temperature`. Zamiast wywoływać `this.setState()`, kiedy chcemy dokonać zmiany, możemy teraz wywołać `this.props.onTemperatureChange()`, które udostępniane jest przez komponent `Calculator`.

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onTemperatureChange(e.target.value);
  }

  render() {
    const temperature = this.props.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Podaj temperaturę w {scaleNames[scale]}:</legend>
        <input value={temperature}>
          onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

Przyjrzymy się teraz komponentowi `Calculator`.

Wartości atrybutów obecnego inputa - `temperature` oraz `scale` będziemy przechowywać w stanie lokalnym. Użyjemy stanu, który "wynieśliśmy w górę" z inputów i będzie on nam służył za "źródło prawdy". Zarazem jest to najmniejsza ilość danych, wystarczająca do wyrenderowania obydwóch inputów.

Na przykład, jeśli do inputa z Celsjuszami wpiszemy `37`, stan komponentu `Calculator` będzie wyglądał następująco:

```
{
  temperature: '37',
  scale: 'c'
```

Jeśli zdecydujemy się później edytować pole z Fahrenheitami i wpiszemy liczbę 212, stan komponentu `calculator` będzie się przedstawiał tak:

```
{
  temperature: '212',
  scale: 'f'
}
```

Moglibyśmy przechowywać wartości dla obydwóch inputów, jednak jest to zupełnie zbędne. Trzymanie wartości ostatnio zmienianego inputa i jego skali wystarczy do uzyskania pożdanego wyniku. Możemy teraz wyliczyć wartość drugiego inputa na podstawie obecnej wartości `temperature` oraz `scale`.

Inputy są ze sobą zsynchronizowane, dzięki temu, że ich wartości są obliczane na podstawie tego samego stanu.

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    this.state = {temperature: '', scale: 'c'};
  }

  handleCelsiusChange(temperature) {
    this.setState({scale: 'c', temperature});
  }

  handleFahrenheitChange(temperature) {
    this.setState({scale: 'f', temperature});
  }

  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) :
      temperature;
    const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) :
      temperature;

    return (
      <div>
        <TemperatureInput
          scale="c"
          temperature={celsius}
          onTemperatureChange={this.handleCelsiusChange} />
        <TemperatureInput
          scale="f"
          temperature={fahrenheit}
          onTemperatureChange={this.handleFahrenheitChange} />
        <BoilingVerdict
          celsius={parseFloat(celsius)} />
      </div>
    );
  }
}
```

[Przetestuj kod na CodePen](#)

Nie ważne teraz, który z inputów edytujemy, w komponencie `Calculator` aktualizowane będą zarówno `this.state.temperature`, jak i `this.state.scale`.

Podsumujmy teraz, co dzieje się, gdy edytujemy jeden z inputów:

- React wywołuje funkcję `onChange` na elemencie DOM `<input>`. W naszym przypadku jest to metoda `handleChange` z komponentu `TemperatureInput`.
- Metoda `handleChange` w komponencie `TemperatureInput` wywołuje `this.props.onTemperatureChange()` z nową wartością. Atrybuty tego komponentu, takie jak `onTemperatureChange`, pochodzą od rodzica - komponentu `Calculator`.
- Podczas poprzedniego renderu, komponent `Calculator` określił, że atrybut `onTemperatureChange` przychodzący z komponentu `TemperatureInput` z Celsjuszami, jest wywoływany metodą `handleCelsiusChange`, natomiast `onTemperatureChange` z komponentu z Fahrenheitami, jest teraz wywoływany metodą `handleFahrenheitChange`. Dzięki temu, w zależności od tego, który input dostanie nową wartość, jedna z tych metod zostanie wywołana.
- Wewnątrz tych metod komponent `Calculator` prosi Reacta o ponowne wyrenderowanie komponentu poprzez użycie `this.setState()` z nową wartością inputa oraz obecnie używaną skalą.
- React wywołuje metodę `render` komponentu `Calculator`, by dowiedzieć się, jak powinien wyglądać jego UI. Wartości obu inputów są ponownie obliczane, bazując na aktualnej temperaturze i skali. To w tym miejscu odbywa się konwersja temperatury.
- React wywołuje metodę `render` poszczególnych komponentów `TemperatureInput`, która otrzymuje nowe atrybuty od komponentu `Calculator`. Dzięki temu dowiaduje się, jak powinien wyglądać komponent.
- React wywołuje metodę `render` komponentu `BoilingVerdict`, przekazując do niego temperaturę w Celsjuszach jako atrybut.
- Paczka "React DOM" aktualizuje drzewo DOM otrzymaną strukturą, dopasowując się do wartości inputów. Input, którego właśnie edytowaliśmy, otrzymuje swoją obecną wartość, natomiast drugi input otrzymuje temperaturę po konwersji.

Każda aktualizacja przechodzi ten sam proces, więc inputy są zawsze zsynchronizowane.

Wnioski

Wszelkie dane, które zmieniają się w aplikacji reactowej, powinny mieć swoje pojedyncze "źródło prawdy". Na ogół stan dodaje się najpierw do komponentu, który potrzebuje go podczas renderowania. Następnie, jeśli inny komponent potrzebuje tych samych danych, możemy "wynieść je w górę" do najbliższego wspólnego przodka. Zamiast próbować synchronizować stan między różnymi komponentami, lepiej polegać na [przepływie danych "z góry na dół"](#).

Wynoszenie stanu w górę skutkuje powstaniem większej ilości tzw. kodu szablonowego (ang. *boilerplate code*), niż wiązanie dwukierunkowe (ang. *two-way binding*), jednak pozwala na łatwiejsze znalezienie i wyizolowanie błędów. Ponieważ każdy stan "żyje" w jakimś komponencie i tylko ten komponent może go zmienić, ryzyko powstania błędów jest znacznie mniejsze. Ponadto, można dodać własną logikę, która pozwoli odrzucić lub przekształcić dane wejściowe od użytkownika.

Jeśli jakąś wartość można wydedukować na podstawie zarówno atrybutów, jak i stanu komponentu, to prawdopodobnie nie powinna się ona znaleźć w stanie. Na przykład, zamiast przetrzymywać jednocześnie `celsiusValue` i `fahrenheitValue`, przechowujemy jedynie ostatnio edytowane `temperature` oraz `scale`. Wartość drugiego inputa może być przecież wyliczona w metodzie `render()`. Pozwala nam to na czyszczenie lub zaokrąglanie wartości w drugim polu bez utraty dokładności w danych pochodzących od użytkownika.

Jeśli zauważysz nieprawidłowości w interfejsie swojej aplikacji, możesz użyć [Narzędzi deweloperskich dla Reacta](#), aby zbadać atrybuty i przemieszczać się po drzewie, dopóki nie znajdziesz komponentu odpowiedzialnego za zmianę stanu. Pozwoli Ci to znaleźć przyczynę błędów:

Enter temperature in Celsius:

Enter temperature in Fahrenheit:

The water would not boil.

The screenshot shows the React DevTools interface. At the top, there are tabs: Elements, React (which is selected), Console, Sources, Network, Timeline, Profiles, and a more options menu. Below the tabs are three checkboxes: Trace React Updates, Highlight Search, and Use Regular Expressions. The main area displays the component tree under the <Calculator> component. The tree shows a <div> element containing two <TemperatureInput> components and one <BoilingVerdict> component. The <BoilingVerdict> component has a prop 'celsius' set to null. To the right of the tree, there is a summary section for the <Calculator> component. It shows the props as an empty object and the state with 'scale: "c"' and 'temperature: ""'. A note says '\$r in the console'.

Listy i klucze

Na początku przyjrzymy się, jak przekształca się listy w JavaScriptie.

W kodzie poniżej, użyliśmy funkcji `map()`, aby podwoić wartość liczb w tablicy `numbers`. Następnie przypisaliśmy nową tablicę zwróconą z funkcji `map()` do zmiennej `doubled` i wyświetliśmy ją w konsoli:

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
```

Kod ten wyświetla w konsoli `[2, 4, 6, 8, 10]`.

W Reactie przekształcanie tablic w listy `elementów` przebiega niemalże identyczne.

Wyświetlanie wielu komponentów

Możesz zbudować kolekcje elementów i [dodać je do JSX](#) używając nawiasów klamrowych `{}`.

Poniżej iterujemy tablicę liczb `numbers` używając javascriptowej funkcji `map()`. Zwracamy element `` dla każdego elementu tablicy. Na koniec przypisujemy powstałą w ten sposób tablicę do zmiennej `listItems`:

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

Umieszczamy całą tablicę `listItems` wewnątrz elementu ``:

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

[Przetestuj kod na CodePen](#)

Ten kod wyświetla wypunktowaną listę liczb od 1 do 5.

Podstawowy komponent listy

Zazwyczaj będziesz wyświetlać listy wewnątrz `komponentu`.

Możemy przekształcić poprzedni przykład w komponent, który akceptuje tablicę liczb `numbers` i zwraca listę elementów.

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

```

}

const numbers = [1, 2, 3, 4, 5];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<NumberList numbers={numbers} />);

```

Kiedy uruchomisz powyższy kod, dostaniesz ostrzeżenie o tym, że do elementów listy należy dostarczyć właściwość klucza. „Klucz” (ang. *key*) jest specjalnym atrybutem o typie łańcucha znaków, który musisz dodać podczas tworzenia elementów listy. O tym, dlaczego jest to ważne, opowiem w następnej sekcji.

Przypiszmy właściwość `key` do naszych elementów listy wewnętrz `numbers.map()`, czym naprawimy problem brakuującego klucza.

```

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

```

[Przetestuj kod na CodePen](#)

Klucze

Klucze pomagają Reaktowi zidentyfikować, które elementy uległy zmianie, zostały dodane lub usunięte. Klucze powinny zostać nadane elementom wewnętrz tablicy, aby elementy zyskały stabilną tożsamość:

```

const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
    {number}
  </li>
);

```

Najlepszym sposobem wyboru klucza jest użycie unikatowego ciągu znaków, który jednoznacznie identyfikuje element listy spośród jego rodzeństwa. Jako kluczy często będziesz używać identyfikatorów (ang. *IDs*) ze swoich danych:

```

const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);

```

Jeśli w danych nie masz stabilnych identyfikatorów dla wyświetlanych elementów, w skrajnym przypadku do określenia klucza możesz użyć indeksu elementu tablicy:

```
const todoItems = todos.map((todo, index) =>
  // Rób to tylko w przypadku, gdy elementy nie mają stabilnych identyfikatorów
  <li key={index}>
    {todo.text}
  </li>
);
```

Nie zalecamy używania indeksów jako kluczy, jeżeli kolejność elementów może ulec zmianie. Może to negatywnie wpływać na wydajność i spowodować problemy ze stanem komponentu. Sprawdź artykuł Robina Pokornego ze [szczegółowym wyjaśnieniem negatywnych aspektów używania indeksu jako klucza](#). Jeżeli nie zdecydujesz się jawnie ustawić kluczy dla elementów listy, React domyślnie jako kluczy używa indeksów.

Jeżeli chcesz dowiedzieć się więcej, tutaj znajdziesz [szczegółowe wyjaśnienie, dlaczego klucze są niezbędne](#).

Wyodrębnianie komponentów z kluczami

Klucze mają sens tylko w kontekście otaczającej tablicy.

Dla przykładu, jeżeli [wyodrębnisz](#) komponent `ListItem`, ustaw klucz na elementach `<ListItem />` wewnątrz tablicy zamiast na elemencie `` wewnątrz komponentu `ListItem`.

Przykład: Niepoprawne użycie klucza

```
function ListItem(props) {
  const value = props.value;
  return (
    // Źle! Nie ma potrzeby definiowania klucza tutaj:
    <li key={value.toString()}>
      {value}
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Źle! Klucz powinien zostać określony w tym miejscu:
    <ListItem value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

Przykład: Poprawne użycie klucza

```
function ListItem(props) {
  // Dobrze! Nie ma potrzeby definiowania klucza tutaj:
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Dobrze! Klucz powinien zostać ustalony na elementach wewnątrz tablicy.
  );
}
```

```

        <ListItem key={number.toString()} value={number} />
    );
    return (
        <ul>
            {listItems}
        </ul>
    );
}

```

Przetestuj kod na CodePen

Podstawową regułą jest to, że elementy wewnętrz w wywołaniu `map()` potrzebują kluczy.

Klucze muszą być unikalne tylko wśród rodzeństwa

Klucze używane wewnętrz tablic powinny być unikalne w kontekście swojego rodzeństwa. Jednakże nie muszą one być niepowtarzalne globalnie. Możemy użyć tych samych kluczy, gdy tworzymy dwie różne tablice:

```

function Blog(props) {
    const sidebar = (
        <ul>
            {props.posts.map((post) =>
                <li key={post.id}>
                    {post.title}
                </li>
            )}
        </ul>
    );
    const content = props.posts.map((post) =>
        <div key={post.id}>
            <h3>{post.title}</h3>
            <p>{post.content}</p>
        </div>
    );
    return (
        <div>
            {sidebar}
            <hr />
            {content}
        </div>
    );
}

const posts = [
];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Blog posts={posts} />);

```

Przetestuj kod na CodePen

Klucze służą Reactowi jako wskazówka, ale nie są one przekazywane do twoich komponentów. Jeżeli potrzebujesz tych wartości w swoim komponencie, przekaż je jawnie poprzez właściwość o innej nazwie:

```

const content = posts.map((post) =>
    <Post
        key={post.id}

```

```

    id={post.id}
    title={post.title} />
);

```

W powyższym przykładzie, komponent `Post` może odczytać `props.id`, ale nie `props.key`.

Użycie `map()` wewnętrz JSX

W poprzednich przykładach zadeklarowaliśmy oddzielną zmienną `listItems` wewnętrz JSX:

```

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

```

JSX pozwala na [wstawienie dowolnego wyrażenia](#) wewnętrz nawiasów klamrowych, dzięki temu możemy użyć wyniku funkcji `map()` bez przypisywania go do zmiennej:

```

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

```

[Przetestuj kod na CodePen](#)

Czasami dzięki temu kod staje się bardziej przejrzysty, jednak łatwo doprowadzić do nadużywania tego sposobu. Tak samo jak w JavaScriptie, wybór należy do Ciebie, czy warto wyodrębnić zmienną, aby zwiększyć czytelność. Zapamiętaj, że jeżeli ciało funkcji `map()` jest zbyt zagnieżdżone, może to być odpowiedni moment na [wyodrębnienie komponentu](#).

[Go to TOC](#)

Optymalizacja wydajności

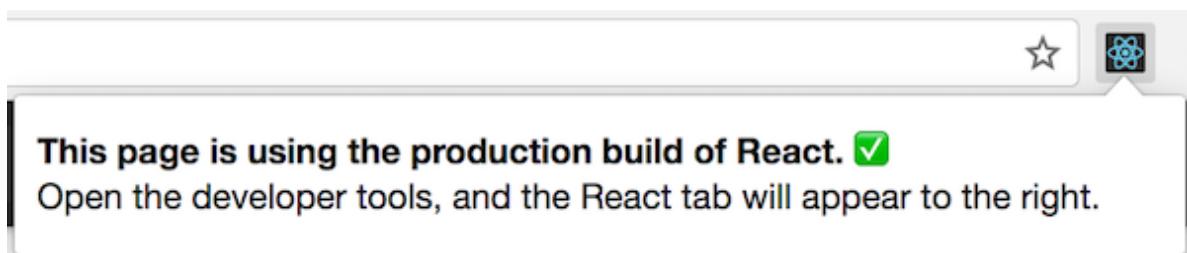
React używa "pod maską" kilku sprytnych sztuczek, dzięki którym może zminimalizować liczbę kosztownych operacji na modelu DOM potrzebnych do aktualizowania interfejsu. W wielu aplikacjach użycie Reacta powinno zapewnić satysfakcjonującą szybkość interfejsu bez stosowania wielu skomplikowanych optymalizacji wydajnościowych. Mimo wszystko, jeśli okaże się inaczej, istnieje kilka sposobów na przyspieszenie twojej aplikacji reactowej.

Użyj kodu produkcyjnego

Jeśli testujesz wydajność swojej aplikacji lub odczuwasz problemy z jej wydajnością, upewnij się, że używasz zminifikowanego kodu produkcyjnego (ang. *production build*).

Domyślnie React dorzuca do kodu wiele ostrzeżeń, które przydają się podczas pisania aplikacji. Niestety z ich powodu React jest cięższy i wolniejszy, dlatego zaleca się wrzucać na produkcję tylko wygenerowany kod produkcyjny.

Jeśli masz wątpliwości co do tego, czy twój proces budowania aplikacji jest ustawiony poprawnie, sprawdź to instalując [React Developer Tools dla Chrome'a](#). Jeśli po instalacji odwiedzisz stronę korzystającą z kodu produkcyjnego, ikona wtyczki będzie miała ciemne tło:



Jeśli jednak strona będzie korzystać z trybu deweloperskiego, ikona będzie czerwona:



Co do zasady, trybu deweloperskiego powinno używać się podczas tworzenia aplikacji, a kod produkcyjny wrzucać tam, gdzie będą z niego korzystać docelowi użytkownicy.

Poniżej znajdziesz więcej instrukcji dotyczących budowania aplikacji produkcyjnej.

Create React App

Jeśli twój projekt powstał przy pomocy [Create React App](#), uruchom polecenie:

```
npm run build
```

Stworzy ono kod produkcyjny twojej aplikacji i umieści go w folderze `build/`.

Pamiętaj, aby używać powyższej komendy tylko przed wrzuceniem kodu na produkcję. Do normalnej pracy nad aplikacją uruchamiaj `npm start`.

Jednoplikowe zbudowane paczki

React oraz React DOM są dostępne jako pojedyncze pliki, wprost gotowe do użycia na produkcji:

```
<script src="https://unpkg.com/react@18/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.production.min.js">
</script>
```

Pamiętaj, że tylko pliki o nazwie kończącej się na `.production.min.js` są przystosowane do działania na produkcji.

Brunch

Aby uzyskać najwydajniejszy build produkcyjny przy użyciu narzędzia Brunch, zainstaluj do niego wtyczkę `terser-brunch`:

```
# Jeśli używasz npma
npm install --save-dev terser-brunch

# Jeśli używasz Yarna
yarn add --dev terser-brunch
```

Następnie stwórz kod produkcyjny dodając flagę `-p` do komendy `build`:

```
brunch build -p
```

Pamiętaj, że uruchamianie powyższej komendy jest koniecznie tylko wtedy, gdy chcesz stworzyć kod produkcyjny. Do codziennej pracy nie korzystaj z flagi `-p` ani nie używaj tej wtyczki, ponieważ spowoduje to ukrycie przydatnych ostrzeżeń reactowych oraz spowolni sam proces budowania aplikacji.

Browserify

Aby uzyskać najwydajniejszy build produkcyjny przy użyciu narzędzia Browserify, zainstaluj kilka wtyczek:

```
# Jeśli używasz npma
npm install --save-dev envify terser uglifyify

# Jeśli używasz Yarna
yarn add --dev envify terser uglifyify
```

Aby stworzyć kod produkcyjny, dodaj poniższe transformacje (**kolejność ma znaczenie**):

- Transformacja `envify` ustawia poprawne środowisko dla procesu budowania. Użyj jej globalnie (`-g`).
- Transformacja `uglifyify` usuwa importy deweloperskie. Również i jej użyj globalnie (`-g`).
- Na koniec powstały kod jest przepuszczany przez `terser`, który dekoruje (ang. *mangle*) kod ([przeczytaj dlaczego](#)).

Na przykład:

```
browserify ./index.js \
-g [ envify --NODE_ENV production ] \
-g uglifyify \
| terser --compress --mangle > ./bundle.js
```

Pamiętaj, że uruchamianie powyższej komendy jest koniecznie tylko wtedy, gdy chcesz stworzyć kod produkcyjny. Do codziennej pracy nie korzystaj z tych wtyczek, ponieważ spowoduje to ukrycie przydatnych ostrzeżeń reactowych oraz spowolni sam proces budowania aplikacji.

Rollup

Aby uzyskać najwydajniejszy build produkcyjny przy użyciu narzędzia Rollup, zainstaluj kilka wtyczek:

```
# Jeśli używasz npma
npm install --save-dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser

# Jeśli używasz Yarna
yarn add --dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser
```

Aby stworzyć kod produkcyjny, dodaj poniższe wtyczki (**kolejność ma znaczenie**):

- Wtyczka `replace` ustawia poprawne środowisko dla procesu budowania.
- Wtyczka `commonjs` dodaje do Rollupa wsparcie dla CommonJS.
- Wtyczka `terser` kompresuje i dekoruje (ang. *mangle*) kod wynikowy.

```
plugins: [
  // ...
  require('rollup-plugin-replace')({
    'process.env.NODE_ENV': JSON.stringify('production')
  }),
  require('rollup-plugin-commonjs')(),
  require('rollup-plugin-terser')(),
  // ...
]
```

Kompletny przykład konfiguracji możesz [zobaczyć w tym giście](#).

Pamiętaj, że uruchamianie powyższej komendy jest koniecznie tylko wtedy, gdy chcesz stworzyć kod produkcyjny. Do codziennej pracy nie korzystaj z wtyczki `terser` ani z `replace` z ustawioną wartością na `'production'`, ponieważ spowoduje to ukrycie przydatnych ostrzeżeń reactowych oraz spowolni sam proces budowania aplikacji.

webpack

Uwaga:

Jeśli korzystasz z Create React App, przeczytaj [tę instrukcję](#).

Poniższa sekcja dotyczy sytuacji, w której webpack jest konfigurowany bezpośrednio.

Webpack w wersji 4+ minifikuje kod domyślnie w trybie produkcyjnym.

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  mode: 'production',
  optimization: {
    minimizer: [new TerserPlugin({ /* dodatkowe opcje */ })],
  },
};
```

Więcej na ten temat możesz przeczytać w [dokumentacji webpacka](#).

Pamiętaj, że uruchamianie powyższej komendy jest koniecznie tylko wtedy, gdy chcesz stworzyć kod produkcyjny. Do codziennej pracy nie korzystaj z wtyczki `TerserPlugin`, ponieważ spowoduje to ukrycie przydatnych ostrzeżeń reactowych oraz spowolni sam proces budowania aplikacji.

Profilowanie komponentów za pomocą Profilera z React DevTools

`react-dom` 16.5+ oraz `react-native` 0.57+ zapewniają zwiększone możliwości profilowania w trybie deweloperskim w połączeniu z Profilerem wtyczki React DevTools. Ogólne informacje na temat Profilera można znaleźć w poście "[Introducing the React Profiler](#)" (pol. "Wprowadzenie do React Profilera"). Filmik opowiadający o profilerze jest także [dostępny na YouTube](#).

Jeśli jeszcze nie masz zainstalowanej wtyczki React DevTools, możesz znaleźć ją tutaj:

- [Wtyczka do Chrome'a](#)
- [Wtyczka do Firefoksa](#)
- [Niezależna paczka Node'owa](#)

Uwaga

Profiler do kodu produkcyjnego jest również dostępny w paczce `react-dom` pod ścieżką `react-dom/profiling`. Więcej informacji na temat jego użycia znajdziesz na [fb.me/react-profiling](#)

Uwaga

Przed wersją 17 Reakta używaliśmy standardowego [User Timing API](#) (pol. *API do pomiarów czasowych*) w celu profilowania komponentów za pomocą zakładki "Performance" (pol. *Wydajność*) w Chromie. Szczegółowy poradnik dotyczący tego tematu znajdziesz w [tym artykule autorstwa Bena Schwarza](#).

Wirtualizacja długich list

Jeśli Twoja aplikacja renderuje długie listy z danymi (setki lub tysiące wierszy), zalecamy użycie techniki zwanej "okienkowaniem" (ang. *windowing*). Technika ta renderuje w danym momencie jedynie niewielką część wszystkich wierszy, co może znacząco wpływać na zredukowanie czasu ponownego renderowania komponentów oraz zmniejszenie liczby tworzonych węzłów DOM.

[react-window](#) oraz [react-virtualized](#) to popularne biblioteki "okienkujące". Dostarczają kilka generycznych komponentów służących do wyświetlania list, siatek i tabel. Jeśli potrzebujesz czegoś bardziej szytego na miarę swojej aplikacji, możesz napisać własny komponent okienkujący, jak [zrobił to Twitter](#).

Unikaj rekoncyliacji

React buduje i zarządza wewnętrzną reprezentacją renderowanego UI. Zawiera ona elementy reactowe zwieracane przez komponenty aplikacji. Dzięki niej React może unikać niepotrzebnych operacji, jak np. tworzenia węzłów DOM i modyfikowania istniejących. Tego typu operacje są wolniejsze niż analogiczne operacje na obiektach javascriptowych. Czasem do tej reprezentacji odnosimy się jako "wirtualny DOM" (ang. *virtual DOM*), lecz podobna występuje w React Native.

Kiedy zmieniają się właściwości lub stan komponentu, React decyduje, czy należy zaktualizować DOM, poprzez porównanie poprzednio zwróconego elementu z tym zwróconym po zmianie. Jeśli nie są takie same, następuje aktualizacja modelu DOM.

Nawet pomimo tego, że React aktualizuje jedynie zmodyfikowane węzły DOM, ponowne wyrenderowanie zajmuje trochę czasu. W wielu przypadkach nie jest to problemem, jednak czasami spowolnienie jest widoczne gołym okiem. Można temu zaradzić nadpisując metodę cyklu życia komponentu o nazwie `shouldComponentUpdate`, która wywoływana jest tuż przed rozpoczęciem ponownego renderowania. Domyslna implementacja tej metody zwraca zawsze `true`, wymuszając na Reakcie każdorazowe ponowne renderowanie:

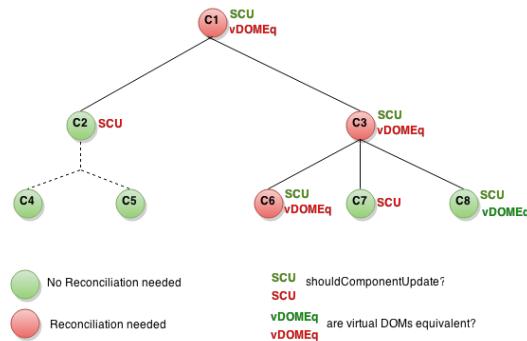
```
shouldComponentUpdate(nextProps, nextState) {
  return true;
}
```

Jeśli wiesz, że w niektórych przypadkach Twój komponent nie musi być aktualizowany, możesz nadpisać `shouldComponentUpdate` w taki sposób, aby zwracało `false`. Dzięki temu React ominie cały proces renderowania, wraz z wywołaniem metody `render()`, na tym komponencie i na wszystkich jego potomkach.

W większości przypadków, zamiast pisania własnej implementacji metody `shouldComponentUpdate()`, wystarczy odziedziczyć po klasie `React.PureComponent`. Daje to ten sam efekt, co zaimplementowanie metody `shouldComponentUpdate()` w taki sposób, aby wykonywała płytke porównanie (ang. *shallow comparison*) na aktualnym oraz poprzednim zestawie właściwości i stanu.

shouldComponentUpdate w akcji

Założymy, że mamy takie poddrzewo komponentów. Dla każdego z nich `SCU` oznacza wartość zwróconą przez metodę `shouldComponentUpdate`, a `vDOMEq` określa, czy wyrenderowane elementy były takie same. Natomiast kolor kółka określa, czy dany komponent musiał zostać sprawdzony przez mechanizm rekoncyliacji, czy nie.



Ponieważ `shouldComponentUpdate` zwróciło `false` dla poddrzewa w węźle C2, React nie musiał renderować C2, a co za tym idzie, nie musiał nawet wywoływać metody `shouldComponentUpdate` na C4 i C5.

Dla C1 i C3 `shouldComponentUpdate` zwróciło `true`, więc React musiał zejść głębiej do liści drzewa i je sprawdzić. Metoda `shouldComponentUpdate` komponentu C6 zwróciła `true`, a ponieważ wyrenderowane elementy różniły się od siebie, React musiał zaktualizować DOM.

Ostatnim interesującym przypadkiem jest C8. React musiał wyrenderować ten komponent, ale wyrenderowany element był taki sam jak poprzednio, w związku z czym nie trzeba było aktualizować modelu DOM.

Zwróć uwagę, że React musiał nanieść poprawki w modelu DOM tylko dla C6, co było nieuniknione. W przypadku C8 nastąpiło wcześnie przerwanie, ponieważ wyrenderowane elementy były takie same, natomiast w poddrzewie C2 oraz węźle C7 React nie musiał nawet niczego porównywać, gdyż `shouldComponentUpdate` zwróciło `false`, przez co `render` nie zostało wywołane w ogóle.

Przykłady

Jeśli twój komponent zmienia się tylko przy zmianie wartości `props.color` lub `state.count`, możesz je sprawdzić wewnątrz metody `shouldComponentUpdate`:

```
class CounterButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

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

```

        return true;
    }
    return false;
}

render() {
    return (
        <button
            color={this.props.color}
            onClick={() => this.setState(state => ({count: state.count + 1}))}>
            Licznik: {this.state.count}
        </button>
    );
}
}

```

W powyższym kodzie `shouldComponentUpdate` sprawdza jedynie, czy zmieniły się wartości `props.color` lub `state.count`. Jeśli nie, wówczas nie ma potrzeby renderować komponentu ponownie. Gdyby komponent nieco się rozrósł, można by zastosować "głębokie porównanie" (ang. *shallow comparison*) wszystkich pól z `props` i `state`, aby stwierdzić, czy należy komponent wyrenderować ponownie. Ten wzorzec jest na tyle popularny, że wyposażyliśmy Reacta w klasę pomocniczą, która to robi. Wystarczy, że twój komponent będzie dziedziczył po klasie `React.PureComponent`, a głębokie porównanie nastąpi automatycznie. Poniższy kod pokazuje prostszy sposób na uzyskanie tego samego efektu:

```

class CounterButton extends React.PureComponent {
    constructor(props) {
        super(props);
        this.state = {count: 1};
    }

    render() {
        return (
            <button
                color={this.props.color}
                onClick={() => this.setState(state => ({count: state.count + 1}))}>
                Licznik: {this.state.count}
            </button>
        );
    }
}

```

W wielu przypadkach, zamiast pisać własną implementację metody `shouldComponentUpdate`, możesz z powodzeniem skorzystać z klasy `React.PureComponent`. Pamiętaj jednak, że wykonuje ona głębokie porównanie, więc na nic się zda, kiedy właściwości lub stan są modyfikowane w sposób, którego ono nie wykrywa.

Problem może pojawić się przy bardziej złożonych strukturach danych. Na przykład, założmy, że mamy komponent `ListOfWords`, który wypisuje słowa po przecinku, oraz nadzędny komponent `WordAdder`, który pozwala kliknąć na guzik i dodać tym samym nowe słowo do listy. Poniższy kod *nie* zadziała poprawnie:

```

class ListOfWords extends React.PureComponent {
    render() {
        return <div>{this.props.words.join(',')}</div>;
    }
}

class WordAdder extends React.Component {

```

```

constructor(props) {
  super(props);
  this.state = {
    words: ['marklar']
  };
  this.handleClick = this.handleClick.bind(this);
}

handleClick() {
  // Ten fragment jest źle napisany i zawiera błąd
  const words = this.state.words;
  words.push('marklar');
  this.setState({words: words});
}

render() {
  return (
    <div>
      <button onClick={this.handleClick} />
      <ListOfWords words={this.state.words} />
    </div>
  );
}
}

```

Problem polega na tym, że `PureComponent` używa prostego porównania starej i nowej wartości `this.props.words`. Jako że ten kod mutuje tablicę `words` wewnętrz metody `handleClick`, stara i nowa wartość dla `this.props.words` będą identyczne, nawet pomimo faktu, iż słowa w tablicy uległy zmianie. Na skutek tego `ListOfWords` nie zostanie zaktualizowany, pomimo że zmieniła się lista słów, które należy wyrenderować.

Potęga niemutowania danych

Najprostszym sposobem na uniknięcie tego problemu jest uniknięcie mutowania wartości, których używasz jako właściwości lub stan. Na przykład, powyższą metodę `handleClick` można napisać z użyciem `concat`:

```

handleClick() {
  this.setState(state => ({
    words: state.words.concat(['marklar'])
  }));
}

```

Standard ES6 dostarcza między innymi [operator rozwinięcia \(ang. spread operator\)](#) dla tablic, który może uprościć ten zapis. Jeśli korzystasz z Create React App, składnia ta jest dostępna domyślnie.

```

handleClick() {
  this.setState(state => ({
    words: [...state.words, 'marklar'],
  }));
}

```

W podobny sposób możesz zmienić kod, który mutuje obiekty, tak, aby uniknąć mutacji. Na przykład, założmy, że mamy obiekt o nazwie `colormap` i chcemy napisać funkcję, która zmienia wartość `colormap.right` na `'blue'`. Możemy napisać:

```
function updateColorMap(colormap) {  
  colormap.right = 'blue';  
}
```

Aby uniknąć mutowania oryginalnego obiektu, możemy użyć metody `Object.assign`:

```
function updateColorMap(colormap) {  
  return Object.assign({}, colormap, {right: 'blue'});  
}
```

`updateColorMap` zwraca teraz nowy obiekt, zamiast mutować stary. Metoda `Object.assign` jest dostępna w standardzie ES6 i wymaga polyfilla.

Składnia rozwinięcia obiektu sprawia, że aktualizowanie obiektów bez ich mutowania jest łatwiejsze:

```
function updateColorMap(colormap) {  
  return {...colormap, right: 'blue'};  
}
```

Ta funkcjonalność została dodana do JavaScriptu w ES2018.

Jeśli korzystasz z Create React App, zarówno `Object.assign`, jak i składnia rozwinięcia obiektu są dostępne domyślnie.

Jeśli działasz na głęboko zagnieżdzonych obiektach, aktualizowanie ich w niemutowalny sposób może okazać się karkołomne. Z pomocą przychodzą `Immer` oraz `immutability-helper`. Biblioteki te pozwalają pisać czytelny kod bez utraty korzyści płynących z niemutowalności (ang. *immutability*).

Portals

Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

```
ReactDOM.createPortal(child, container)
```

The first argument (`child`) is any [renderable React child](#), such as an element, string, or fragment. The second argument (`container`) is a DOM element.

Usage

Normally, when you return an element from a component's render method, it's mounted into the DOM as a child of the nearest parent node:

```
render() {
  // React mounts a new div and renders the children into it
  return (
    <div>
      {this.props.children}
    </div>
  );
}
```

However, sometimes it's useful to insert a child into a different location in the DOM:

```
render() {
  // React does *not* create a new div. It renders the children into `domNode`.
  // `domNode` is any valid DOM node, regardless of its location in the DOM.
  return ReactDOM.createPortal(
    this.props.children,
    domNode
  );
}
```

A typical use case for portals is when a parent component has an `overflow: hidden` or `z-index` style, but you need the child to visually "break out" of its container. For example, dialogs, hovercards, and tooltips.

Note:

When working with portals, remember that [managing keyboard focus](#) becomes very important.

For modal dialogs, ensure that everyone can interact with them by following the [WAI-ARIA Modal Authoring Practices](#).

[Try it on CodePen](#)

Event Bubbling Through Portals

Even though a portal can be anywhere in the DOM tree, it behaves like a normal React child in every other way. Features like context work exactly the same regardless of whether the child is a portal, as the portal still exists in the *React tree* regardless of position in the *DOM tree*.

This includes event bubbling. An event fired from inside a portal will propagate to ancestors in the containing *React tree*, even if those elements are not ancestors in the *DOM tree*. Assuming the following HTML structure:

```
<html>
  <body>
    <div id="app-root"></div>
    <div id="modal-root"></div>
  </body>
</html>
```

A `Parent` component in `#app-root` would be able to catch an uncaught, bubbling event from the sibling node `#modal-root`.

```
// These two containers are siblings in the DOM
const appRoot = document.getElementById('app-root');
const modalRoot = document.getElementById('modal-root');

class Modal extends React.Component {
  constructor(props) {
    super(props);
    this.el = document.createElement('div');
  }

  componentDidMount() {
    // The portal element is inserted in the DOM tree after
    // the Modal's children are mounted, meaning that children
    // will be mounted on a detached DOM node. If a child
    // component requires to be attached to the DOM tree
    // immediately when mounted, for example to measure a
    // DOM node, or uses 'autoFocus' in a descendant, add
    // state to Modal and only render the children when Modal
    // is inserted in the DOM tree.
    modalRoot.appendChild(this.el);
  }

  componentWillUnmount() {
    modalRoot.removeChild(this.el);
  }

  render() {
    return ReactDOM.createPortal(
      this.props.children,
      this.el
    );
  }
}

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

```

this.state = {clicks: 0};
this.handleClick = this.handleClick.bind(this);
}

handleClick() {
  // This will fire when the button in Child is clicked,
  // updating Parent's state, even though button
  // is not direct descendant in the DOM.
  this.setState(state => ({
    clicks: state.clicks + 1
  }));
}

render() {
  return (
    <div onClick={this.handleClick}>
      <p>Number of clicks: {this.state.clicks}</p>
      <p>
        Open up the browser DevTools
        to observe that the button
        is not a child of the div
        with the onClick handler.
      </p>
      <Modal>
        <Child />
      </Modal>
    </div>
  );
}

function Child() {
  // The click event on this button will bubble up to parent,
  // because there is no 'onClick' attribute defined
  return (
    <div className="modal">
      <button>Click</button>
    </div>
  );
}

const root = ReactDOM.createRoot(appRoot);
root.render(<Parent />);

```

Try it on [CodePen](#)

Catching an event bubbling up from a portal in a parent component allows the development of more flexible abstractions that are not inherently reliant on portals. For example, if you render a `<Modal />` component, the parent can capture its events regardless of whether it's implemented using portals.

[Go to TOC](#)

React bez ES6

Komponenty reactowe mogą być javascriptowymi klasami:

```
class Greeting extends React.Component {
  render() {
    return <h1>Witaj, {this.props.name}</h1>;
  }
}
```

Jeśli jednak jeszcze nie używasz składni ES6, możesz skorzystać z modułu `create-react-class`:

```
var createReactClass = require('create-react-class');
var Greeting = createReactClass({
  render: function() {
    return <h1>Witaj, {this.props.name}</h1>;
  }
});
```

Interfejs API klas ES6 jest podobny do tego w `createReactClass()`, jednak istnieje między nimi kilka różnic.

Deklarowanie domyślnych wartości dla właściwości

W przypadku klas ES6 wartości dla `defaultProps` są definiowane na komponencie:

```
class Greeting extends React.Component {
  // ...
}

Greeting.defaultProps = {
  name: 'Maria'
};
```

Jeśli jednak korzystasz z `createReactClass()`, musisz zadeklarować w tym celu funkcję `getDefaultsProps()` jako metodę przekazywanego obiektu:

```
var Greeting = createReactClass({
  getDefaultsProps: function() {
    return {
      name: 'Maria'
    };
  },
  // ...
});
```

Ustawianie stanu początkowego

W klasach ES6 definicja stanu początkowego następuje po przypisaniu w konstruktorze wartości do `this.state`:

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  // ...
}
```

W `createReactClass()` musisz przekazać osobną metodę `getInitialState`, która zwróci stan początkowy komponentu:

```
var Counter = createReactClass({
  getInitialState: function() {
    return {count: this.props.initialCount};
  },
  // ...
});
```

Automatyczne wiązanie

W komponentach reactowych napisanych przy użyciu klas ES6, metody podlegają tym samym zasadom, co metody w zwykłych klasach ES6. Oznacza to, że nie dowiązują one automatycznie `this` do instancji. Musisz jawnie wywołać `.bind(this)` w konstruktorze:

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Witaj!'};
    // Ta linia jest istotna!
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    alert(this.state.message);
  }

  render() {
    // Ponieważ metoda `this.handleClick` jest dowiązana,
    // możemy jej użyć jako procedurę obsługi zdarzeń.
    return (
      <button onClick={this.handleClick}>
        Przywitaj się
      </button>
    );
  }
}
```

W przypadku `createReactClass()` nie jest to wymagane, gdyż funkcja ta automatycznie dowiązuje wszystkie metody:

```
var SayHello = createReactClass({
  getInitialState: function() {
    return {message: 'Witaj!'};
  },
  handleClick: function() {
    alert(this.state.message);
  },
  render: function() {
    return (
      <button onClick={this.handleClick}>
        Przywitaj się
      </button>
    );
  }
});
```

Oznacza to, że korzystanie z klas ES6 wiąże się z pisaniem więcej powtarzanego kodu dla procedur obsługi zdarzeń, jednak na korzyść przemawia znacznie lepsza wydajność w dużych aplikacjach.

Jeśli nie podoba ci się ten nadmiarowy kod, możesz użyć składni [właściwości klas \(ang. class properties\)](#) w [ES2022](#):

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Witaj!'};
  }

  // Użycie funkcji strzałkowej powoduje automatyczne dowiezanie:
  handleClick = () => {
    alert(this.state.message);
  };

  render() {
    return (
      <button onClick={this.handleClick}>
        Przywitaj się
      </button>
    );
  }
}
```

Jeśli wolisz pewniejsze rozwiązania, masz kilka opcji:

- Dowiązuj metody w konstruktorze.
- Używaj funkcji strzałkowych, np. `onClick={(e) => this.handleClick(e)}`.
- Skorzystaj z `createClass`.

Mixiny

Uwaga:

Standard ES6 nie wspiera mixinów, dlatego domyślnie React nie będzie działał z mixinami użytymi w klasach ES6.

Z naszych obserwacji wynika też, że używanie ich często powoduje problemy, dlatego odradzamy korzystania z nich w nowym kodzie.

Ten rozdział istnieje tylko dla zapewnienia kompletności dokumentacji.

Niekiedy różne komponenty mogą współdzielić te same funkcjonalności. Nazywa się je także [problemami przekrojowymi](#). `createReactClass` pozwala na zastosowanie w tym celu przestarzałego systemu `mixinów`.

Jednym z częstych przykładów jest komponent, który chce aktualizować swój stan w różnych odstępach czasu. Łatwo jest skorzystać z funkcji `setInterval()`, lecz należy pamiętać o anulowaniu interwału, gdy już nie jest potrzebny, aby zwolnić pamięć. React dostarcza [metody cyklu życia](#), które informują o tym, kiedy komponent jest tworzony lub niszczony. Stwórzmy prosty mixin, korzystający z tych metod, udostępniający prostą funkcję `setInterval()`, która będzie automatycznie po sobie "sprzątała", gdy komponent ulegnie zniszczeniu.

```
var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function() {
    this.intervals.forEach(clearInterval);
  }
};

var createReactClass = require('create-react-class');

var TickTock = createReactClass({
  mixins: [SetIntervalMixin], // Użyj mixinu
  getInitialState: function() {
    return {seconds: 0};
  },
  componentDidMount: function() {
    this.setInterval(this.tick, 1000); // Wywołaj metodę z mixinu
  },
  tick: function() {
    this.setState({seconds: this.state.seconds + 1});
  },
  render: function() {
    return (
      <p>
        React jest już uruchomiony {this.state.seconds} sekund.
      </p>
    );
  }
});

const root = ReactDOM.createRoot(document.getElementById('example'));
root.render(<TickTock />);
```

Jeśli komponent używa kilku mixinów i niektóre z nich definiują te same metody cyklu życia (tj. kilka z nich chce posprzątać przed zniszczeniem komponentu), React gwarantuje, że wszystkie zostaną wywołane. Metody zdefiniowane w mixinach są uruchamiane zgodnie z kolejnością ich dodania, a na koniec uruchamiana jest metoda samego komponentu.

React bez JSX

JSX nie jest wymagany do korzystania z Reacta. Korzystanie z Reacta bez JSX jest szczególnie wygodne, gdy nie chce się konfigurować kroku komplikacji w środowisku budowania.

Każdy element JSX jest jedynie wygodniejszym odpowiednikiem wywołania metody `React.createElement(component, props, ...children)`. Wszystko więc, co da się zrobić korzystając z JSX, można również uzyskać za pomocą zwykłego JavaScriptu.

Na przykład, ten fragment kodu napisany z użyciem JSX:

```
class Hello extends React.Component {
  render() {
    return <div>Witaj, {this.props.toWhat}</div>;
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Hello toWhat="Świecie" />);
```

może być skompilowany do tego kodu, który nie korzysta z JSX:

```
class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Witaj, ${this.props.toWhat}`);
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(React.createElement(Hello, {toWhat: 'Świecie'}, null));
```

Jeśli chcesz zobaczyć więcej przykładów konwersji składni JSX do kodu javascriptowego, wypróbuj [wersję online kompilatora Babel](#).

Komponent może być ciągiem znaków, podklassą `React.Component` albo zwykłą funkcją.

Aby uniknąć ciągłego pisania `React.createElement`, warto zastosować poniższy wzorzec:

```
const e = React.createElement;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(e('div', null, 'Witaj, Świecie'));
```

Jeśli używa się tej skróconej formy `React.createElement`, korzystanie z Reacta bez JSX może być równie wygodne.

Ewentualnie można zapoznać się z projektami społeczności, takimi jak `react-hyperscript` czy `hyper-script-helpers`, które oferują bardziej zwięzłą składnię.

[Go to TOC](#)

Rekoncyliacja

Deklaratywność API Reacta sprawia, że nie musisz się martwić, co dokładnie zmienia się przy każdej aktualizacji, dzięki czemu pisanie aplikacji staje się dużo prostsze. Dokładna implementacja nie jest jednak tak oczywista. W tym artykule wyjaśniamy decyzje, które podjęliśmy przy projektowaniu algorytmu różnicującego w Reakcie, mając na celu zapewnienie przewidywalności aktualizacji komponentów przy zachowaniu wysokiej wydajności.

Motywacja

Podczas korzystania z Reacta, w danym momencie możesz potraktować funkcję `render()` jako tworzącą drzewo elementów Reacta. Podczas następnej zmiany stanu bądź aktualizacji właściwości funkcja `render()` zwróci inne drzewo elementów. Zadaniem Reacta jest wtedy opracować wydajny sposób na aktualizację UI, tak by dopasować je do najświeższego drzewa elementów.

Istnieją ogólne rozwiązania algorytmicznego problemu generowania najmniejszej liczby operacji wymaganych do przekształcenia drzew elementów. Jednakże nawet najlepsze znane algorytmy mają złożoność rzędu $O(n^3)$, gdzie n jest liczbą elementów w drzewie.

Gdybyśmy wykorzystali ten algorytm w Reakcie, wyświetlenie 1000 elementów wymagałoby miliarda porównań. Byłoby to stanowczo zbyt kosztowne. Zamiast tego, React implementuje heurystyczny algorytm o złożoności $O(n)$, bazujący na dwóch założeniach:

1. Dwa elementy różnych typów produkują różne drzewa.
2. Programista może wskazać, które elementy drzewa mogą być stabilne pomiędzy kolejnymi renderowaniami, używając właściwości `key`.

W praktyce te założenia sprawdzają się w prawie wszystkich zastosowaniach.

Algorytm różnicujący

Podczas różnicowania dwóch drzew React najpierw porównuje elementy nadrzędne. Dalsze kroki zależą od typów elementów nadrzędnych.

Elementy różnych typów

Zawsze, gdy elementy nadrzędne różnią się typem, React pozbywa się starego drzewa i buduje nowe od podstaw. Zamiana `<a>` na ``, czy `<Article>` na `<Comment>` lub `<Button>` na `<div>` - każda z tych zmian spowoduje całkowite przebudowanie drzewa.

Gdy React pozbywa się starego drzewa, wszystkie przypisane do niego węzły DOM są niszczone. W instancjach komponentów wywoływane jest `componentWillUnmount()`. Podczas budowania nowego drzewa do DOM-u dodawane są nowe węzły. W instancjach komponentów wywoływane jest `UNSAFE_componentWillMount()`, a następnie `componentDidMount()`. Jakikolwiek stan przypisany do starego drzewa jest bezpowrotnie tracony.

Jakiekolwiek komponenty poniżej elementu nadzawanego również zostają odmontowane, a ich stan zniszczony. Na przykład, przy zmianie:

```
<div>
  <Counter />
</div>

<span>
  <Counter />
</span>
```

Stary `Counter` zostanie zniszczony i zamontowany zostanie nowy.

Uwaga:

Ta metoda jest przestarzała i powinno się jej [uniknąć](#) w nowym kodzie.

- `UNSAFE_componentWillMount()`

Elementy DOM tego samego typu

Przy porównywaniu dwóch elementów tego samego typu React patrzy na atrybuty obu elementów, zachowuje stary węzeł i aktualizuje jedynie atrybuty, które tego wymagają. Na przykład:

```
<div className="before" title="stuff" />
<div className="after" title="stuff" />
```

Porównując te dwa elementy, React wie, że należy zmienić jedynie `className` na istniejącym węźle DOM-u.

Aktualizując `style`, React wie również, by aktualizować jedynie te właściwości, które uległy zmianie. Na przykład:

```
<div style={{color: 'red', fontWeight: 'bold'}} />
<div style={{color: 'green', fontWeight: 'bold'}} />
```

Przy aktualizacji tych elementów React wie, by zmodyfikować jedynie `color`, a nie również `fontWeight`.

Po obsłużeniu danego węzła DOM-u React rekursywnie wywołuje algorytm na kolejnych potomkach w drzewie.

Komponenty tego samego typu

Gdy komponent jest aktualizowany, jego instancja pozostaje bez zmian, dzięki czemu stan jest zachowany pomiędzy kolejnymi renderowaniami. React aktualizuje właściwości instancji zgodnie z nowym elementem i wywołuje na niej `UNSAFE_componentWillReceiveProps()`, `UNSAFE_componentWillUpdate()` oraz `componentDidUpdate()`.

Następnie wywołana zostaje metoda `render()`, a algorytm różnicujący się dalej w głąb drzewa.

Uwaga:

Te metody są przestarzałe i powinno się ich unikać w nowym kodzie.

- `UNSAFE_componentWillUpdate()`
- `UNSAFE_componentWillReceiveProps()`

Rekurencja po potomkach

Podczas rekurencji przez potomstwo (ang. *children*) węzła DOM-u React iteruje po obu drzewach jednocześnie (starym i nowym), generując zmiany, gdy napotka na różnicę.

Na przykład, gdy dodamy element na koniec potomstwa, przejście pomiędzy tymi drzewami działa poprawnie:

```
<ul>
  <li>pierwszy</li>
  <li>drugi</li>
</ul>

<ul>
  <li>pierwszy</li>
  <li>drugi</li>
  <li>trzeci</li>
</ul>
```

React dopasuje do siebie drzewa `pierwszy`, dopasuje drzewa `drugi`, następnie napotkawszy różnicę doda drzewo `trzeci`.

Przy naiwnej implementacji algorytmu dodanie elementu na początek będzie miało gorszą wydajność. Na przykład, przejście pomiędzy tymi dwoma drzewami jest mało wydajne:

```
<ul>
  <li>Duke</li>
  <li>Villanova</li>
</ul>

<ul>
  <li>Connecticut</li>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
```

React dokona zmian przy każdym potomku, gdyż nie jest świadomym, że `Duke` oraz `Villanova` nie uległy zmianie, a jedynie przesunięciu. Może się to okazać problematyczne.

Klucze

W celu rozwiązania powyższego problemu React wprowadził atrybut `key`. Gdy potomstwo posiada klucz (ang. *key*), React używa go przy porównaniu poprzedniego drzewa z nowym. Na przykład, dodając `key` do poprzedniego przykładu pozbędziemy się problemu z wydajnością:

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>

<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

Teraz React wie, że element z kluczem `'2014'` jest nowy, a elementy o kluczach `'2015'` oraz `'2016'` jedynie zmieniły pozycję.

Dobranie stosownego klucza zazwyczaj nie przysparza trudności. Element, który wyświetlamy, może już posiadać unikalny ID, więc klucz może pochodzić bezpośrednio od twoich danych:

```
<li key={item.id}>{item.name}</li>
```

W przeciwnym wypadku można dodać nowe ID do modelu danych bądź wykorzystać funkcję skrótu (ang. *hash function*) do wygenerowania klucza. Klucz musi być unikalny jedynie względem swego rodzeństwa.

W ostateczności jako klucza można użyć indeksu elementu w tablicy. Rozwiążanie to sprawdzi się, jeśli kolejność elementów w tablicy jest stała; w przypadku zmiennej kolejności różnicowanie będzie mniej wydajne.

W przypadku użycia indeksu jako klucza, zmiany w kolejności elementów tablicy mogą również powodować problemy ze stanem komponentów. Instancje komponentów są aktualizowane bądź nie na podstawie klucza. Jeśli klucz jest indeksem, każda zmiana pozycji elementu w tablicy powoduje zmianę klucza. W rezultacie stan komponentów może zostać zaktualizowany w nieprzewidywalny sposób i powodować trudne do zidentyfikowania błędy.

Na podanym CodePenie można zapoznać się z [przykładowym problemem, jaki stwarza stosowanie indeksów jako kluczy](#), a z kolei tutaj pokazany jest [sposób, w jaki unikanie indeksów w kluczu rozwiązuje problemy z wstawianiem, sortowaniem oraz zmianą pozycji elementów](#).

Kompromisy

Należy pamiętać, że algorytm rekoncyliacji to szczegół implementacyjny. React mógłby ponownie renderować całą aplikację przy każdej akcji; rezultat byłby ten sam. Dla jasności: przez ponowne renderowanie w tym kontekście rozumiemy wywołanie `render` dla wszystkich komponentów; nie oznacza to, że zostaną one odmontowane i zamontowane ponownie. Oznacza jedynie zaaplikowanie zmian według reguł, które dotychczas przedstawiliśmy.

Regularnie usprawniamy algorytm heurystyczny, by zoptymalizować wydajność w najczęstszych przypadkach. W aktualnej implementacji możemy wyrazić fakt, iż poddrzewo zmieniło pozycję względem rodzeństwa, nie jesteśmy jednak w stanie wskazać, że zostało bez zmian przeniesione gdzie indziej. Algorytm wymusi ponowne renderowanie całego poddrzewa.

Ponieważ React opiera się na heurystyce, tracimy na wydajności zawsze wtedy, gdy nie spełniamy jej założeń.

1. Algorytm nie będzie dopasowywał poddrzew komponentów różnych typów. Jeśli naprzemiennie używasz komponentów różnego typu, a zwracających bardzo podobny wynik - sugerujemy ujednolicić typ. W praktyce nie przysporzyło nam to nigdy problemów.
2. Klucze powinny być stabilne, przewidywalne i unikalne. Niestabilne klucze (na przykład generowane przez `Math.random()`) będą skutkować niepotrzebną przebudową wielu instancji komponentów i węzłów DOM-u, powodując spadek wydajności i utratę stanu w komponentach potomnych.

Elementy DOM

Dla poprawy wydajności oraz kompatybilności między przeglądarkami, React wprowadza niezależny od przeglądarki system DOM. Dało to nam możliwość naprawienia pewnych usterek występujących w przeglądarkowych implementacjach DOM.

W Reakcie wszystkie właściwości i atrybuty DOM (włącznie z obsługą zdarzeń) powinny być zapisywane w notacji camelCase. Dla przykładu, atrybut HTML `tabindex` w Reakcie zapisujemy jako `tabIndex`. Wyjątkiem są atrybuty `aria-*` oraz `data-*`, które powinny być zapisywane małą literą, na przykład, `aria-label` pozostaje `aria-label`.

Różnice w atrybutach

W Reakcie kilka atrybutów działa inaczej niż w HTML-u:

`checked`

Atrybut `checked` jest wspierany przez element `<input>` typu `checkbox` lub `radio`. Za jego pomocą można dany element ustawić jako odznaczony. Jest to przydatne podczas budowania kontrolowanych komponentów. W przypadku komponentów niekontrolowanych należy używać `defaultChecked`, który określa, czy komponent ma być odznaczony przy pierwszym zamontowaniu.

`className`

Aby przypisać klasę CSS, należy użyć atrybutu `className`. Dotyczy to wszystkich standardowych elementów DOM oraz SVG, takich jak `<div>`, `<a>` i innych.

Gdy używasz Reacta z Web Components (co jest rzadkie), stosuj atrybut `class`.

`dangerouslySetInnerHTML`

`dangerouslySetInnerHTML` jest reactowym odpowiednikiem dla `innerHTML` z przeglądarkowego DOM. Ogólnie rzecz biorąc, używanie HTML w kodzie jest ryzykowne, ponieważ łatwo jest nieumyślnie narazić użytkowników na atak [cross-site scripting \(XSS\)](#). Z tego powodu jedynym sposobem na wstawienie HTML bezpośrednio w Reakcie jest użycie właściwości `dangerouslySetInnerHTML` i przekazanie obiektu z kluczem `__html` -- wszystko po to, żeby pamiętać, iż jest to niebezpieczne. Przykład:

```
function createMarkup() {
  return {__html: 'Pierwszy &nbsp; Drugi'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

htmlFor

Jako że `for` jest zarezerwowanym słowem w JavaScriptie, React używa zamiast niego `htmlFor`.

onChange

Zdarzenie `onChange` przebiega zgodnie z oczekiwaniemi: jest aktywowane za każdym razem, gdy zmienia się wartość w polu formularza. Specjalnie nie używamy implementacji przeglądarkowej, ponieważ `onChange` ma nazwę nieadekwatną do zachowania. React natomiast używa tego zdarzenia do obsłużenia w czasie rzeczywistym zmian w danych wejściowych kontrolki formularza.

selected

Jeśli chcesz zaznaczyć którąś z opcji `<option>`, przekaż jej wartość jako właściwość `value` elementu `<select>`. Aby dowiedzieć się więcej na ten temat, zajrzyj do sekcji "[Znacznik select](#)".

style

Wskazówka

Niektóre przykłady w dokumentacji używają `style` dla wygody, ale ogólnie rzecz biorąc, **używanie atrybutu `style` do stylowania elementów nie jest zalecane**. W większości przypadków należy korzystać z atrybutu `className`, odnosząc się do klas definiowanych w zewnętrznym arkuszu stylów (ang. `stylesheet`). `style` jest najczęściej używany w aplikacjach reactowych do dodawania stylów dynamicznych, zmieniających się przy kolejnych renderowaniach. Zobacz też [FAQ: Stylowanie i CSS](#).

Atrybut `style` zamiast zwykłego ciągu znaków przyjmuje javascriptowy obiekt z właściwościami CSS zapisanymi w notacji camelCase (nie ciąg znaków jak w CSS). Jest to zgodne z właściwością DOM `style` w JavaScriptie, poprawia wydajność i zapobiega dziurom w zabezpieczeniach przed XSS. Przykład:

```
const divStyle = {
  color: 'blue',
  backgroundImage: `url(${imgUrl})`,
};

function HelloWorldComponent() {
  return <div style={divStyle}>Witaj, świecie</div>;
}
```

Zauważ, że style nie mają automatycznie nadawanego prefiksu. Aby zapewnić kompatybilność ze starszymi przeglądarkami, musisz nadać im odpowiednie właściwości:

```
const divStyle = {
  WebkitTransition: 'all', // zwróć uwagę na wielką literę 'W'
  msTransition: 'all' // 'ms' jest jedynym prefiksem zapisywany małą literą
};
```

```
function ComponentWithTransition() {
  return <div style={divStyle}>To powinno działać na wielu przeglądarkach</div>;
}
```

Nazwy kluczy obiektu stylów zapisujemy w notacji camelCase, aby były zgodne z właściwościami węzłów DOM w JS (np. `node.style.backgroundImage`). Prefiks dostawców (ang. *vendor prefixes*) inne niż ms powinny zaczynać się od wielkiej litery. Dlatego `WebkitTransition` zaczyna się od wielkiej litery "W".

React automatycznie doda przyrostek "px" (piksele) dla pewnych liczbowych właściwości stylów. Jeśli chcesz użyć innych jednostek niż "px", określ wartość jako ciąg znaków z pożądaną jednostką. Przykład:

```
// Rezultat: '10px'
<div style={{ height: 10 }}>
  Witaj, świecie!
</div>

// Rezultat: '10%'
<div style={{ height: '10%' }}>
  Witaj, świecie!
</div>
```

Nie wszystkie właściwości stylów są konwertowane do ciągów znaków z przyrostkiem 'px'. Niektóre z nich pozostają bez jednostek (np. `zoom`, `order`, `flex`). Pełna lista wyjątków jest dostępna [tutaj](#).

suppressContentEditableWarning

Zwykle gdy element posiadający potomka jest oznaczony jako `contentEditable`, pojawia się ostrzeżenie, ponieważ mechanizm ten nie zadziała prawidłowo. Ten atrybut wyłącza to ostrzeżenie. Nie używaj go, chyba że tworzysz bibliotekę ręcznie zarządzającą właściwością `contentEditable`, jak np. [Draft.js](#).

suppressHydrationWarning

Gdy używasz renderowania po stronie serwera (ang. *server-side*), zwykle pojawia się ostrzeżenie, gdy serwer i klient renderują inną zawartość. Jednakże, w niektórych rzadkich przypadkach, dokładne dopasowanie jest bardzo trudne lub niemożliwe do osiągnięcia. Na przykład, znaczniki czasu zwykle różnią się między serwerem i klientem.

Jeżeli ustawisz właściwość `suppressHydrationWarning` na `true`, React nie ostrzeże cię o niedopasowaniu atrybutów i zawartości tego elementu. Działa to tylko na jednym poziomie zagłębiania i jest używane jako "wyjście awaryjne" z problemu. Nie nadużywaj go. Możesz poczytać więcej o hydracji (ang. *hydration*) w [dokumentacji funkcji](#) `ReactDOM.hydrateRoot()`.

value

Atrybut `value` jest wspierany przez elementy `<input>`, `<select>` oraz `<textarea>`. Możesz go użyć do ustawienia wartości komponentu. Jest to przydatne do budowania kontrolowanych komponentów. `defaultValue` jest niekontrolowanym odpowiednikiem tego atrybutu, który określa, jaką wartość powinien mieć komponent przy pierwszym montowaniu.

Wszystkie wspierane atrybuty HTML

Poczytając od wersji 16-stej React wspiera wszystkie standardowe i [niestandardowe](#) atrybuty DOM.

React od zawsze udostępniał interfejs API jak najbardziej podobny do javascriptowego interfejsu DOM. Z racji tego, że komponenty reactowe często przyjmują zarówno właściwości niestandardowe, jak i te związane z modelem DOM, React używa notacji `camelCase`, zupełnie jak interfejs DOM:

```
<div tabIndex={-1} /> // Dokładnie jak node.tabIndex w DOM API
<div className="Button" /> // Dokładnie jak node.className w DOM API
<input readOnly={true} /> // Dokładnie jak node.readOnly w DOM API
```

Powysze właściwości działają podobnie jak odpowiadające im atrybuty HTML, z wyjątkiem specjalnych przypadków wymienionych powyżej.

Niektóre atrybuty DOM wspierane przez Reacta to:

```
accept acceptCharset accessKey action allowFullScreen alt autoComplete
autoFocus autoPlay capture cellPadding cellSpacing challenge charSet checked
cite classID className colSpan cols content contentEditable contextMenu controls
controlsList coords crossOrigin data dateTime default defer dir disabled
download draggable encType form formAction formEncType formMethod formNoValidate
formTarget frameBorder headers height hidden high href hrefLang htmlFor
httpEquiv icon id inputMode integrity is keyParams keyType kind label lang list
loop low manifest marginHeight marginWidth max maxLength media mediaGroup method
min minLength multiple muted name noValidate nonce open optimum pattern
placeholder poster preload profile radioGroup readOnly rel required reversed
role rowSpan rows sandbox scope scoped scrolling seamless selected shape size
sizes span spellCheck src srcDoc srcLang srcSet start step style summary
tabIndex target title type useMap value width wmode wrap
```

Podobnie, wspierane są wszystkie atrybuty SVG:

```
accentHeight accumulate additive alignmentBaseline allowReorder alphabetic
amplitude arabicForm ascent attributeName attributeType autoReverse azimuth
baseFrequency baseProfile baselineShift bbox begin bias by calcMode capHeight
clip clipPath clipPathUnits clipRule colorInterpolation
colorInterpolationFilters colorProfile colorRendering contentScriptType
contentStyleType cursor cx cy d decelerate descent diffuseConstant direction
display divisor dominantBaseline dur dx dy edgeMode elevation enableBackground
end exponent externalResourcesRequired fill fillOpacity fillRule filter
filterRes filterUnits floodColor floodOpacity focusable fontFamily fontSize
fontSizeAdjust fontStretch fontStyle fontVariant fontWeight format from fx fy
g1 g2 glyphName glyphOrientationHorizontal glyphOrientationVertical glyphRef
gradientTransform gradientUnits hanging horizAdvX horizOriginX ideographic
imageRendering in in2 intercept k k1 k2 k3 k4 kernelMatrix kernelUnitLength
kerning keyPoints keySplines keyTimes lengthAdjust letterSpacing lightingColor
limitingConeAngle local markerEnd markerHeight markerMid markerStart
markerUnits markerWidth mask maskContentUnits maskUnits mathematical mode
numOctaves offset opacity operator order orient orientation origin overflow
overlinePosition overlineThickness paintOrder panose1 pathLength
patternContentUnits patternTransform patternUnits pointerEvents points
pointsAtX pointsAtY pointsAtZ preserveAlpha preserveAspectRatio primitiveUnits
r radius refX refY renderingIntent repeatCount repeatDur requiredExtensions
requiredFeatures restart result rotate rx ry scale seed shapeRendering slope
spacing specularConstant specularExponent speed spreadMethod startOffset
stdDeviation stemh stemv stitchTiles stopColor stopOpacity
```

```
strikethroughPosition strikethroughThickness string stroke strokeDasharray  
strokeDashoffset strokeLinecap strokeLinejoin strokeMiterlimit strokeOpacity  
strokeWidth surfaceScale systemLanguage tableValues targetX targetY textAnchor  
textDecoration textLength textRendering to transform u1 u2 underlinePosition  
underlineThickness unicode unicodeBidi unicodeRange unitsPerEm vAlphabetic  
vHanging vIdeographic vMathematical values vectorEffect version vertAdvY  
vertOriginX vertOriginY viewBox viewTarget visibility widths wordSpacing  
writingMode x x1 x2 xChannelSelector xHeight xlinkActuate xlinkArcrole  
xlinkHref xlinkRole xlinkShow xlinkTitle xlinkType xmlns xmlnsXlink xmlBase  
xmlLang xmlSpace y y1 y2 yChannelSelector z zoomAndPan
```

Możesz także używać niestandardowych atrybutów, pod warunkiem, że są one napisane małymi literami.

SyntheticEvent

Poniższa dokumentacja dotyczy `SyntheticEvent` (pol. *zdarzenie syntetyczne*) - klasy opakowującej zdarzenia, będącej częścią systemu obsługi zdarzeń Reacta. Samej [obsłudze zdarzeń w Reakcie](#) poświęciliśmy osobny rozdział.

Informacje ogólne

Napisane przez ciebie procedury obsługi zdarzeń będą otrzymywać jako argument instancję `SyntheticEvent` - klasy opakowującej natywne zdarzenie, niezależnej od przeglądarki. Posiada ona taki sam interfejs jak natywne zdarzenia, wliczając w to metody `stopPropagation()` oraz `preventDefault()`, gwarantuje jednak identyczne działanie na wszystkich przeglądarkach.

Jeśli w którymś momencie zechcesz skorzystać z opakowanego, nativnego zdarzenia, możesz odwołać się do niego poprzez właściwość `nativeEvent`. Syntetyczne zdarzenia różnią się od natywnych zdarzeń przeglądarki i można ich stosować wymiennie. Na przykład, w zdarzeniu `onMouseLeave` wartość `event.nativeEvent` będzie wskazywać na zdarzenie `mouseout`. Specyfikacja mapowania nie jest dostępna publicznie i może ulec zmianie w dowolnym momencie. Każdy obiekt klasy `SyntheticEvent` posiada następujące właściwości:

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
void persist()
DOMEventTarget target
number timeStamp
string type
```

Uwaga:

Od wersji 17, wywołanie metody `e.persist()` nie robi nic, ponieważ `SyntheticEvent` nie jest już współdzielony w [puli](#).

Uwaga:

Od wersji v0.14 wzwyż, zwracanie wartości `false` przez procedurę obsługi nie zatrzymuje propagacji zdarzenia. Zamiast tego należy ręcznie wywoływać odpowiednią metodę: `e.stopPropagation()` lub `e.preventDefault()`.

Obsługiwane zdarzenia

React normalizuje zdarzenia, tak by ich właściwości były jednakowe w różnych przeglądarkach.

Przedstawione na liście poniżej procedury obsługi zdarzeń są wywoływane przez zdarzenie w fazie bąbelkowania (ang. *bubbling phase*). Aby zarejestrować procedurę obsługi w fazie przechwytywania (ang. *capturing phase*), dodaj na końcu nazwy `Capture`, np. zamiast `onClick` użyj `onClickCapture`.

- [Obsługa schowka](#)
- [Obsługa kompozycji](#)
- [Obsługa klawiatury](#)
- [Obsługa skupiania](#)
- [Obsługa formularzy](#)
- [Obsługa zdarzeń generycznych](#)
- [Obsługa myszy](#)
- [Obsługa wskaźnika](#)
- [Obsługa zaznaczania](#)
- [Obsługa dotyku](#)
- [Obsługa interfejsu użytkownika](#)
- [Obsługa pokrętła myszy](#)
- [Obsługa mediów](#)
- [Obsługa obrazów](#)
- [Obsługa animacji](#)
- [Obsługa tranzycji](#)
- [Inne zdarzenia](#)

Dokumentacja

Obsługa schowka

Nazwy zdarzeń:

`onCopy` `onCut` `onPaste`

Właściwości:

`DOMDataTransfer` `clipboardData`

Obsługa kompozycji

Nazwy zdarzeń:

`onCompositionEnd onCompositionStart onCompositionUpdate`

Właściwości:

`string data`

Obsługa klawiatury

Nazwy zdarzeń:

`onKeyDown onKeyPress onKeyUp`

Właściwości:

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

Właściwość `key` przyjmuje dowolną z wartości wymienionych w [specyfikacji Zdarzeń DOM 3. Poziomu](#).

Obsługa skupiania

Nazwy zdarzeń:

`onFocus onBlur`

Powыższe zdarzenia działają na wszystkich elementach w React DOM, nie tylko na kontrolkach formularza.

Właściwości:

`DOMEVENTTARGET relatedTarget`

onFocus

Zdarzenie `onFocus` jest wywoływanego, gdy element (lub któryś z zagnieżdżonych elementów) otrzymuje fokus. Przykładowo, zostanie ono wywołane, gdy użytkownik kliknie na polu tekstowym.

```
function Example() {
  return (
    <input
      onFocus={(e) => {
        console.log('Fokus jest na polu tekstowym');
      }}
      placeholder="onFocus zostanie wywołane po kliknięciu na tym polu."
    />
  )
}
```

onBlur

Zdarzenie `onBlur` jest wywołane, gdy element (lub któryś z zagnieżdżonych elementów) stracił fokus. Przykładowo, zostanie ono wywołane, gdy użytkownik kliknie gdzieś poza polem tekstowym.

```
function Example() {
  return (
    <input
      onBlur={(e) => {
        console.log('Wywołano, bo pole straciło fokus');
      }}
      placeholder="onBlur zostanie wywołane, gdy klikniesz na tym polu, a
      następnie klikniesz poza nim."
    />
  )
}
```

Wykrywanie fokusa i jego utraty

Aby rozróżnić, czy zdarzenia dotyczące fokusa pochodzą spoza rodzica, możesz sprawdzić wartości pól `currentTarget` i `relatedTarget`. Poniżej znajdziesz kod, który pokazuje, jak wykryć fokus na elemencie potomnym, jak na samym elemencie, a jak wykryć taki, który dotyczy całego poddrzewa elementów.

```
function Example() {
  return (
    <div
      tabIndex={1}
      onFocus={(e) => {
        if (e.currentTarget === e.target) {
          console.log('fokus na sobie');
        } else {
          console.log('fokus na elemencie potomnym', e.target);
        }
        if (!e.currentTarget.contains(e.relatedTarget)) {
          // Not triggered when swapping focus between children
          console.log('fokus aktywny poza poddrzewem');
        }
      }}
      onBlur={(e) => {
        if (e.currentTarget === e.target) {
          console.log('fokus utracony na sobie');
        } else {
          console.log('fokus utracony na elemencie potomnym', e.target);
        }
        if (!e.currentTarget.contains(e.relatedTarget)) {
          // Not triggered when swapping focus between children
          console.log('fokus poza poddrzewem');
        }
      }}
    >
```

```

        }
      >
      <input id="1" />
      <input id="2" />
    </div>
  );
}

```

Obsługa formularzy

Nazwy zdarzeń:

`onChange onInput onInvalid onReset onSubmit`

Więcej informacji odnośnie zdarzenia `onChange` opisaliśmy w rozdziale pt. "Formularze".

Obsługa zdarzeń generycznych

Nazwy zdarzeń:

`onError onLoad`

Obsługa myszy

Nazwy zdarzeń:

`onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit
onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave
onMouseMove onMouseOut onMouseOver onMouseUp`

Zdarzenia `onMouseEnter` oraz `onMouseLeave`, zamiast zwykłego bąbelkowania, propagowane są z elementu opuszczanego do elementu wskazywanego, a ponadto nie mają fazy przechwytywania (ang. *capture phase*).

Właściwości:

```

boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey

```

Obsługa wskaźnika

Nazwy zdarzeń:

```
onPointerDown onPointerMove onPointerUp onPointerCancel onGotPointerCapture
onLostPointerCapture onPointerEnter onPointerLeave onPointerOver onPointerOut
```

Zdarzenia `onPointerEnter` oraz `onPointerLeave`, zamiast zwykłego bąbelkowania, propagowane są z elementu opuszczanego do elementu wskazywanego, a ponadto nie mają fazy przechwytywania (ang. *capture phase*).

Właściwości:

Zgodnie ze [specyfikacją W3](#), zdarzenia wskaźnika rozszerzają zdarzenia obsługi myszy o następujące właściwości:

```
number pointerId
number width
number height
number pressure
number tangentialPressure
number tiltX
number tiltY
number twist
string pointerType
boolean isPrimary
```

Informacja o kompatybilności z przeglądarkami:

Zdarzenia wskaźnika nie są jeszcze obsługiwane przez wszystkie przeglądarki (w chwili pisania tego artykułu wspierają je: Chrome, Firefox, Edge i Internet Explorer). React celowo nie dostarcza łatki (ang. *polyfill*) dla pozostałych przeglądarek, ponieważ łatka zgodna ze standardem znaczaco zwiększyłaby rozmiar paczki `react-dom`.

Jeśli twoja aplikacja wymaga obsługi wskaźników, zalecamy dołączenie odpowiedniej paczki zewnętrznej.

Obsługa zaznaczania

Nazwy zdarzeń:

```
onSelect
```

Obsługa dotyku

Nazwy zdarzeń:

```
onTouchCancel onTouchEnd onTouchMove onTouchStart
```

Właściwości:

```
boolean altKey
DOMTouchList changedTouches
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
boolean shiftKey
DOMTouchList targetTouches
DOMTouchList touches
```

Obsługa interfejsu użytkownika

Nazwy zdarzeń:

`onScroll`

Uwaga:

Podczas od Reacta 17, zdarzenie `onScroll` **nie jest propagowane w górę**. Odpowiada to zachowaniu przeglądarki i pozwala uniknąć niejasności, gdy to zagnieżdżony element z suwakiem generował zdarzenia na odległych rodzicach.

Właściwości:

```
number detail
DOMAbstractView view
```

Obsługa pokrętła myszy

Nazwy zdarzeń:

`onWheel`

Właściwości:

```
number deltaMode
number deltaX
number deltaY
number deltaZ
```

Obsługa mediów

Nazwy zdarzeń:

```
onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted
onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay
onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend
onTimeUpdate onVolumeChange onWaiting
```

Obsługa obrazów

Nazwy zdarzeń:

onLoad onError

Obsługa animacji

Nazwy zdarzeń:

onAnimationStart onAnimationEnd onAnimationIteration

Właściwości:

string animationName
string pseudoElement
float elapsedTime

Obsługa tranzycji

Nazwy zdarzeń:

onTransitionEnd

Właściwości:

string propertyName
string pseudoElement
float elapsedTime

Inne zdarzenia

Nazwy zdarzeń:

onToggle

[Go to TOC](#)

Słownik terminów reactowych

Aplikacja jednostronicowa

Aplikacja jednostronicowa to taka, która wczytuje pojedynczą stronę HTML i wszystkie zasoby (takie jak JavaScript czy CSS) potrzebne do uruchomienia aplikacji. Wszelkie interakcje ze stroną i poszczególnymi podstronami nie wymagają ponownego łączenia z serwerem, co oznacza, że strona nie jest przeładowywana.

Pomimo tego, że aplikacje jednostronicowe można tworzyć za pomocą Reacta, nie jest on do tego konieczny. React może posłużyć do wzbogacania niewielkich fragmentów istniejących stron o dodatkową interaktywność. Napisany kod reactowy może z powodzeniem działać razem z kodem wygenerowanym po stronie serwera np. w PHP, lub innymi bibliotekami klienckimi. Prawdę mówiąc, właśnie w ten sposób React jest używany w Facebooku.

ES6, ES2015, ES2016 itp.

Skróty te odnoszą się do najnowszych wersji standardu specyfikacji języka ECMAScript, którego implementacją jest język JavaScript. Wersja ES6 (znana również jako ES2015) zawiera wiele dodatkowych mechanizmów, jak np. funkcje strzałkowe, klasy, literały szablonowe, a także wyrażenia `let` oraz `const`. Po więcej informacji na temat poszczególnych wersji sięgnij do [Wikipedii](#).

Kompilatory

Kompilator javascriptowy wczytuje kod javascriptowy, przekształca go i zwraca go w innym formacie. Najczęściej stosuje się go przekształcania składni ES6 w kod zrozumiały dla starszych przeglądarek. [Babel](#) jest najbardziej powszechnym kompilatorem używanym wraz z Reactem.

Bundlery

Bundlery wczytują kod JavaScript i CSS, napisany w formie odrębnych modułów (często liczonych w setkach) i łączą je w kilka plików zoptymalizowanych pod przeglądarki. W środowisku reactowym do najczęściej używanych bundlerów zaliczane są [Webpack](#) oraz [Browserify](#).

Menedżery pakietów

Menedżery pakietów to narzędzia pozwalające zarządzać zależnościami projektu. W aplikacjach reactowych najczęściej używane są [npm](#) oraz [Yarn](#), przy czym obydwa są klientami tego samego rejestru pakietów npm.

CDN

CDN to skrót od Content Delivery Network (pol. sieć dostarczania zawartości). CDN-y dostarczają statyczną zawartość przechowywaną w pamięci podręcznej poprzez sieć serwerów rozproszonych po całej kuli ziemskiej.

JSX

JSX jest rozszerzeniem składni JavaScript. Przypomina języki szablonów, jednak posiada pełną moc JavaScriptu. Kod napisany w JSX jest komplikowany do wywołań funkcji `React.createElement()`, które zwracają obiekty javascriptowe zwane "elementami reactowymi". Aby dowiedzieć się więcej, przeczytaj [wstęp do składni JSX](#) lub [szczegółowy samouczek](#).

React DOM do nazywania właściwości używa konwencji camelCase zamiast nazw atrybutów HTML. Na przykład, `tabindex` jest zapisywany w składni JSX jako `tabIndex`. Z kolei atrybut `class` piszemy jako `className` ze względu na fakt, iż `class` jest w JavaScriptie zarezerwowanym słowem kluczowym:

```
<h1 className="hello">Mam na imię Klementyna!</h1>
```

Elementy {#elements}

Elementy reactowe są częściami składowymi aplikacji pisanych za pomocą Reacta. Nietrudno pomylić je z szerzej znanym pojęciem "komponentów". Element opisuje to, co zostanie wyświetcone na ekranie. Elementy reactowe są niezmienne (ang. *immutable*).

```
const element = <h1>Witaj, świecie</h1>;
```

Zazwyczaj jednak elementów nie używa się bezpośrednio jako wartości, lecz zwraca w komponentach.

Komponenty {#components}

Komponenty reactowe są niewielkimi kawałkami kodu, często wielokrotnego użytku, które zwracają element reaktywny, który ma zostać wyrenderowany na stronie. Najprostszym wariantem reaktywnego komponentu jest javascriptowa funkcja, która zwraca element reaktywny.:

```
function Welcome(props) {
  return <h1>Witaj, {props.name}</h1>;
}
```

Komponenty mogą także być klasami ze standardu ES6:

```
class Welcome extends React.Component {
  render() {
    return <h1>Witaj, {this.props.name}</h1>;
  }
}
```

Komponenty można podzielić na unikalne fragmenty funkcjonalności, a następnie używać wewnętrz innych komponentów. Komponenty mogą zwracać inne komponenty, ale także tablice, napisy czy liczby. Należy kierować się zasadą, że jeśli jakiś fragment interfejsu użytkownika jest używany wielokrotnie (np. przycisk, panel, awatar) lub sam w sobie jest dość złożony (np. aplikacja, news, komentarz), staje się odpowiednim kandydatem do bycia komponentem wielokrotnego użytku. Nazwy komponentów powinny zaczynać się od wielkiej litery (`<Wrapper/>`, a **nie** `<wrapper/>`). Po więcej informacji sięgnij do [dokumentacji dotyczącej renderowania komponentów](#).

`props {#props}`

`props` (od ang. *properties* = pol. *właściwości*) są danymi wejściowymi dla reactowych komponentów. Przekazuje się je z komponentów nadzorzących do ich potomków.

Pamiętaj, że właściwości `props` są tylko do odczytu. Nie należy ich w jakikolwiek sposób modyfikować:

```
// Źle!
props.number = 42;
```

Jeśli potrzebujesz zmienić którąś wartość w odpowiedzi na akcję użytkownika lub zapytanie do serwera, skorzystaj ze `stanu`.

`props.children {#propschildren}`

Właściwość `props.children` jest dostępna w każdym komponencie. Zawiera wszystko, co znajdzie się między znacznikiem otwierającym i zamkającym danego komponentu, na przykład:

```
<Welcome> Witaj, świecie! </Welcome>
```

Napis `Witaj, świecie!` znajdzie się we właściwości `props.children` komponentu `Welcome`:

```
function Welcome(props) {
  return <p>{props.children}</p>;
}
```

W przypadku komponentów zdefiniowanych jako klasy należy skorzystać z `this.props.children`:

```
class Welcome extends React.Component {
  render() {
    return <p>{this.props.children}</p>;
}
```

`state {#state}`

Komponent potrzebuje własnego stanu (`state`), gdy powiązane z nim dane zmieniają się w czasie. Na przykład, komponent `Checkbox` w zmiennej `isChecked` mógłby śledzić, czy jest zaznaczony, a komponent `NewsFeed` mógłby przechowywać pobrane posty w `fetchedPosts`.

Najistotniejszą różnicą pomiędzy `state` i `props` jest to, że właściwości `props` są dostarczane przez komponent nadzędny, a stanem `state` zarządza sam komponent. Komponent nie może modyfikować swoich właściwości `props`, ale może zmieniać swój stan `state`.

Dla każdego fragmentu danych zmieniających się w czasie powinien istnieć tylko jeden komponent, który taki stan "posiada" na wyłączność. Nie próbuj synchronizować stanów dwóch komponentów. Zamiast tego [wynieś stan w górę](#) do najbliższego przodka i z niego przekaż stan w dół za pomocą właściwości `props`.

Metody cyklu życia {#lifecycle-methods}

Metody cyklu życia to specjalne funkcje uruchamiane w trakcie różnych faz życia komponentu. Istnieją takie, które uruchamiane są podczas tworzenia komponentu i wstawiania go do drzewa DOM (tzw. [montowanie](#)), inne gdy komponent jest aktualizowany, a jeszcze inne gdy jest odmontowywany lub usuwany z drzewa DOM.

Kontrolowane vs. niekontrolowane komponenty

React zapewnia dwa różne podejścia do obsługi pól formularza.

Pole formularza, którego wartością zarządza React, jest nazywane *komponentem kontrolowanym*. Gdy użytkownik wprowadzi do niego dane, wywoływana jest odpowiednia procedura obsługi zdarzenia i to twój kod decyduje, czy wartość jest poprawna (poprzez ponowne wyrenderowanie z nową wartością). Jeśli samodzielnie nie wyrenderujesz ponownie danego pola, nie zmieni się ono na ekranie.

Z kolej *komponentem niekontrolowanym* działa tak, jak wszystkie pola formularza istniejące poza Reactem. Gdy użytkownik wprowadzi do takiego pola (np. pola tekstowego, pola wyboru itp.), zmiana jego wartości następuje automatycznie, bez konieczności obsługiwanego tego w kodzie Reactowym. Oznacza to również, że nie możesz wymusić określonej wartości pola.

W większości przypadków zalecamy korzystanie z komponentów kontrolowanych.

Klucze {#keys}

Klucz `key` to specjalny atrybut tekstowy, wymagany przy tworzeniu tablic elementów. Klucze pozwalają Reactowi zidentyfikować, które elementy listy zostały zmienione, dodane bądź usunięte. Służą również do nadania elementom tablicy stabilnego identyfikatora.

Klucze muszą być unikalne tylko pośród "rodzeństwa" z tej samej tablicy, lecz mogą się powtarzać w ramach całej aplikacji czy nawet wewnętrz tego samego komponentu.

Do określania kluczy nie używaj wartości typu `Math.random()`. Ważne jest, by klucze były "stabilnymi identyfikatorami" w kolejnych renderowaniach. Dzięki temu React może wykryć, które elementy zostały dodane, usunięte lub zmieniły kolejność. Najlepiej nadają się do tego unikalne, stabilne identyfikatory pochodzące z danych, np. `post.id`.

Referencje `ref` `{#refs}`

React wspiera specjalny atrybut, którego można użyć na dowolnym komponencie. Atrybut `ref` może być obiektem utworzonym przy użyciu funkcji `React.createRef()`, dowolną funkcją zwrotną lub ciągiem znaków (w starym API). Gdy `ref` jest funkcją zwrotną, jest ona wywoływaną jednym argumentem: elementem DOM odpowiadającym komponentowi lub instancją klasy (w zależności od typu komponentu). Pozwala to na bezpośredni dostęp do API elementu DOM lub instancji klasy.

Z referencji korzystaj sporadycznie. Jeśli zauważysz, że używasz ich dość często do sprawienia, żeby "coś zaczęło działać", sugerujemy zapoznać się z [przepływem danych z góry na dół](#).

Zdarzenia `{#events}`

Obsługa zdarzeń w elementach reactowych ma kilka różnic składniowych:

- Nazwy procedur obsługi zdarzeń używają konwencji camelCase, a nie są pisane małymi literami.
- W składni JSX procedury obsługi zdarzeń przekazuje się jako funkcje, a nie jako ciągi znaków.

Rekoncyliacja `{#reconciliation}`

Gdy zmieniają się właściwości lub stan komponentu, React decyduje, czy konieczna jest aktualizacja drzewa DOM, poprzez porównanie nowo zwróconego elementu z jego poprzednią wersją. Jeśli elementy różnią się, drzewo DOM jest aktualizowane. Proces ten nazywany jest "rekoncyliacją" (ang. *reconciliation*).

Wymagania środowiska JavaScript

React 18 wspiera wszystkie nowoczesne przeglądarki (Edge, Firefox, Chrome, Safari itd.).

Jeśli Twoja aplikacja wspiera starsze urządzenia i przeglądarki, np. Internet Explorer, które nie dostarczają natywnie funkcjonalności nowoczesnych przeglądarek lub mają niekompatybilne implementacje, rozważ dodanie do swojej paczki globalnych łatek.

Here is a list of the modern features React 18 uses:

- `Promise`
- `Symbol`
- `Object.assign`

Wybór łatek zależy od Twojego środowiska. Dla wielu użytkowników wystarcza skonfigurowanie [Browserlist](#). Inni z kolei muszą importować łatki bezpośrednio np. z `core-js`.

[Go to TOC](#)

Profiler API

The `Profiler` measures how often a React application renders and what the "cost" of rendering is. Its purpose is to help identify parts of an application that are slow and may benefit from [optimizations such as memoization](#).

Note:

Profiling adds some additional overhead, so **it is disabled in the production build**.

To opt into production profiling, React provides a special production build with profiling enabled. Read more about how to use this build at fb.me/react-profiling

Usage

A `Profiler` can be added anywhere in a React tree to measure the cost of rendering that part of the tree. It requires two props: an `id` (string) and an `onRender` callback (function) which React calls any time a component within the tree "commits" an update.

For example, to profile a `Navigation` component and its descendants:

```
render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Main {...props} />
  </App>
);
```

Multiple `Profiler` components can be used to measure different parts of an application:

```
render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Profiler id="Main" onRender={callback}>
      <Main {...props} />
    </Profiler>
  </App>
);
```

`Profiler` components can also be nested to measure different components within the same subtree:

```
render(
  <App>
    <Profiler id="Panel" onRender={callback}>
      <Panel {...props}>
```

```

        <Profiler id="Content" onRender={callback}>
          <Content {...props} />
        </Profiler>
        <Profiler id="PreviewPane" onRender={callback}>
          <PreviewPane {...props} />
        </Profiler>
      </Panel>
    </Profiler>
  </App>
);

```

Note

Although `Profiler` is a light-weight component, it should be used only when necessary; each use adds some CPU and memory overhead to an application.

onRender Callback

The `Profiler` requires an `onRender` function as a prop. React calls this function any time a component within the profiled tree "commits" an update. It receives parameters describing what was rendered and how long it took.

```

function onRenderCallback(
  id, // the "id" prop of the Profiler tree that has just committed
  phase, // either "mount" (if the tree just mounted) or "update" (if it re-
  rendered)
  actualDuration, // time spent rendering the committed update
  baseDuration, // estimated time to render the entire subtree without memoization
  startTime, // when React began rendering this update
  commitTime, // when React committed this update
  interactions // the Set of interactions belonging to this update
) {
  // Aggregate or log render timings...
}

```

Let's take a closer look at each of the props:

The `id` prop of the `Profiler` tree that has just committed. This can be used to identify which part of the tree was committed if you are using multiple profilers.

- `phase: "mount" | "update"` - Identifies whether the tree has just been mounted for the first time or re-rendered due to a change in props, state, or hooks.
- `actualDuration: number` - Time spent rendering the `Profiler` and its descendants for the current update. This indicates how well the subtree makes use of memoization (e.g. `React.memo`, `useMemo`, `shouldComponentUpdate`). Ideally this value should decrease significantly after the initial mount as many of the descendants will only need to re-render if their specific props change.
- `baseDuration: number` - Duration of the most recent `render` time for each individual component within the `Profiler` tree. This value estimates a worst-case cost of rendering (e.g. the initial mount or a tree with no memoization).
- `startTime: number` - Timestamp when React began rendering the current update.

- `commitTime: number` - Timestamp when React committed the current update. This value is shared between all profilers in a commit, enabling them to be grouped if desirable.
- `interactions: Set` - Set of "interactions" that were being traced when the update was scheduled (e.g. when `render` or `setState` were called).

Note

Interactions can be used to identify the cause of an update, although the API for tracing them is still experimental.

Learn more about it at fb.me/react-interaction-tracing

PureRenderMixin

Note

The `PureRenderMixin` mixin predates `React.PureComponent`. This reference doc is provided for legacy purposes, and you should consider using `React.PureComponent` instead.

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
var PureRenderMixin = require('react-addons-pure-render-mixin');
var createReactClass = require('create-react-class');
createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Example using ES6 class syntax:

```
import PureRenderMixin from 'react-addons-pure-render-mixin';
class FooComponent extends React.Component {
  constructor(props) {
    super(props);
    this.shouldComponentUpdate = PureRenderMixin.shouldComponentUpdate.bind(this);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

Under the hood, the mixin implements `shouldComponentUpdate`, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

React.Component

Ta strona zawiera szczegółowe odniesienie do definicji klasy reactowego komponentu. Zakłada ona, że znasz fundamentalne zagadnienia Reacta, takie jak [komponenty i właściwości](#), i [stan i cykl życia](#). Jeśli nie, zapoznaj się najpierw z nimi.

Ogólne informacje

React pozwala na zdefiniowanie komponentów jako klasy lub funkcje. Komponenty zdefiniowane jako klasy obecnie zapewniają więcej funkcjonalności, które szczegółowo opiszemy na tej stronie. Aby komponent mógł być zdefiniowany jako klasa, musi on dziedziczyć po klasie `React.Component`:

```
class Welcome extends React.Component {
  render() {
    return <h1>Cześć, {this.props.name}</h1>;
  }
}
```

Jedyna metoda, która *musi* być zdefiniowana w klasie dziedziczącej po `React.Component` nazywa się `render()`. Wszystkie inne metody opisane na tej stronie są opcjonalne.

Stanowczo odradzamy tworzenie własnych klas bazowych komponentów. W Reactowych komponentach [wielokrotne użycie kodu jest osiągane przede wszystkim przez kompozycję, a nie dziedziczenie](#).

Uwaga:

React nie zmusza cię do stosowania składni klasy ze standardu ES6. Jeśli wolisz jej uniknąć, możesz zamiast niej użyć modułu `create-react-class` lub podobnej niestandardowej abstrakcji. Aby dowiedzieć się więcej, zobacz rozdział [React bez ES6](#).

Cykl życia komponentu

Każdy komponent ma kilka "metod cyklu życia", które możesz nadpisać, aby uruchomić kod w szczególnych momentach programu. **Możesz użyć tego diagramu cyklu życia jako ściągawki.** Na liście poniżej, często używane metody cyklu życia zostały **pogrubione**. Reszta z nich istnieje dla stosunkowo rzadkich przypadków użycia.

Montowanie

Podczas, gdy instancja komponentu zostaje stworzona i włożona do drzewa DOM, w podanej kolejności wywoływane są poniższe metody:

- `constructor()`
- `static getDerivedStateFromProps()`
- **`render()`**

- `componentDidMount()`

Uwaga:

Ta metoda jest uznawana za przestarzałą (ang. *legacy*) i powinno się jej unikać w nowym kodzie:

- `UNSAFE_componentWillMount()`

Aktualizacja

Aktualizacja może być spowodowana zmianami we właściwościach lub stanie komponentu. Kiedy komponent zostaje ponownie wyrenderowany, w podanej kolejności wywołane zostają poniższe metody:

- `static getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `render()`
- `getSnapshotBeforeUpdate()`
- `componentDidUpdate()`

Uwaga:

Te metody są uznawane za przestarzałe i powinno się ich unikać w nowym kodzie:

- `UNSAFE_componentWillUpdate()`
- `UNSAFE_componentWillReceiveProps()`

Odmontowywanie

Kiedy komponent zostaje usunięty z drzewa DOM, wywołana zostaje poniższa metoda:

- `componentWillUnmount()`

Obsługa wyjątków

Poniższe metody zostają wywołane w razie wystąpienia wyjątku podczas renderowania, w metodzie cyklu życia, lub w konstruktorze dowolnych komponentów potomnych.

- `static getDerivedStateFromError()`
- `componentDidCatch()`

Inne API

Każdy komponent zapewnia też kilka innych API:

- `setState()`
- `forceUpdate()`

Właściwości klasy

- `defaultProps`
- `displayName`

Właściwości instancji

- `props`
 - `state`
-

Dokumentacja

Powszechnie używane metody cyklu życia

Metody opisane w tym rozdziale pokrywają zdecydowaną większość przypadków użycia, na które natkniesz się tworząc reactowe komponenty. **Dla wizualnego odniesienia, zobacz ten diagram cyklu życia.**

`render() { #render }`

`render()`

Metoda `render()` jest jedyną metodą wymaganą w komponencie klasowym.

Wywołana, powinna sprawdzić `this.props` i `this.state` oraz zwrócić jeden z poniższych typów:

- **Reactowe elementy.** Zwykle tworzone poprzez [JSX](#). Na przykład, `<div />` i `<MyComponent />` są reaktowymi elementami, które instruują Reacta, aby, odpowiednio, wyrenderował węzeł drzewa DOM, lub inny zdefiniowany przez użytkownika komponent.
- **Tablice i fragmenty.** Pozwalają na zwrócenie wielu elementów z metody render. Po więcej szczegółów odwiedź dokumentację [fragmentów](#).
- **Portale.** Pozwalają na wyrenderowanie elementów potomnych w innym poddrzewie DOM. Po więcej szczegółów odwiedź dokumentację [portali](#).
- **Łańcuchy znaków i liczby.** Zostają wyrenderowane jako węzły tekstowe w drzewie DOM.
- **Typ logiczny lub `null`.** Nie renderuje nic. (Istnieje głównie, aby wspierać wzorzec `return test && <Child />`, gdzie `test` jest wartością logiczną.)

Funkcja `render()` powinna być czysta, to znaczy, że nie modyfikuje stanu komponentu, zwraca ten sam wynik przy każdym wywołaniu, i nie wchodzi w bezpośrednią interakcję z przeglądarką.

Jeśli potrzebujesz wejść w interakcję z przeglądarką, zamiast tego wykonaj swoje instrukcje w `componentDidMount()` lub innych metodach cyklu życia. Utrzymywanie funkcji `render()` w czystości sprawia, że łatwiej jest myśleć o komponentach.

Uwaga

Funkcja `render()` nie zostanie wywołana, jeśli `shouldComponentUpdate()` zwróci `false`.

`constructor()` {#constructor}

`constructor(props)`

Jeśli nie inicjalizujesz stanu i nie wiążesz (ang. *bind*) metod, nie ma potrzeby, abyś implementował konstruktor w swoim reaktywym komponencie.

Konstruktor reaktywego komponentu jest wywoływany przed jego zamontowaniem. Kiedy implementujesz konstruktor w klasie dziedziczącej po klasie `React.Component`, powinieneś wywołać metodę `super(props)` przed jakąkolwiek inną instrukcją. W innym wypadku, `this.props` będzie miało w konstruktorze wartość `undefined`, co może prowadzić do błędów.

Zazwyczaj, konstruktory są używane tylko w dwóch celach:

- Inicjalizacji **stanu lokalnego** przez przypisanie obiektu do `this.state`.
- Związania **metody obsługującej zdarzenia** z instancją komponentu.

Nie powinieneś wywoływać metody `setState()` w funkcji `constructor()`. Zamiast tego, jeśli potrzebujesz użyć w komponencie stanu lokalnego, **przydziel początkowy stan do `this.state` bezpośrednio w konstruktorze:**

```
constructor(props) {
  super(props);
  // Nie wywołuj tutaj this.setState()!
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

Konstruktor jest jedynym miejscem, w którym powinieneś przypisywać `this.state` bezpośrednio. Natomiast we wszystkich innych metodach powinieneś używać `this.setState()`.

Unikaj wprowadzania efektów ubocznych lub subskrypcji w konstruktorze. Używaj zamiast tego `componentDidMount()` dla tych przypadków użycia.

Uwaga

Unikaj kopiowania właściwości do stanu! Jest to częsty błąd:

```
constructor(props) {
  super(props);
  // Nie rób tego!
  this.state = { color: props.color };
}
```

Problem w tym, że jest to jednocześnie niepotrzebne (zamiast tego możesz użyć `this.props.color` bezpośrednio), i jest przyczyną błędów (aktualizacje właściwości `color` nie będą odzwierciedlane w stanie).

Używaj tego wzorca tylko jeśli chcesz celowo ignorować aktualizacje właściwości. W tym wypadku, będzie miała sens zmiana nazwy właściwości na `initialColor` (ang. *początkowy kolor*) lub `defaultColor` (ang. *domyślny kolor*). Możesz wtedy zmusić komponent do "zresetowania" swojego wewnętrznego stanu przez zmianę jego właściwości `key` w razie potrzeby.

Przeczytaj nasz [wpis na blogu na temat unikania stanu pochodnego](#), aby dowiedzieć się co należy zrobić, jeśli wydaje ci się, że potrzebujesz stanu zależnego od właściwości.

componentDidMount() {#componentdidmount}

componentDidMount()

Metoda `componentDidMount()` jest wywoływana bezpośrednio po zamontowaniu komponentu (po jego włożeniu do drzewa). Inicjalizacja, która wymaga węzłów drzewa DOM powinna się tam znaleźć. Jeśli potrzebujesz załadować dane ze zdalnego zasobu, jest to dobre miejsce do wykonania zapytania sieciowego.

Ta metoda jest dobrym miejscem na przygotowanie dowolnych subskrypcji. Jeśli to zrobisz, nie zapomnij ich zakończyć w metodzie `componentWillUnmount()`.

Możesz wywołać metodę `setState()` od razu w `componentDidMount()`. Spowoduje to dodatkowe renderowanie, ale zostanie ono wykonane zanim przeglądarka zaktualizuje ekran. Jest to gwarancją, że pomimo, iż metoda `render()` będzie w tym przypadku wywołana dwa razy, użytkownik nie zobaczy pośredniego stanu. Używaj tego wzorca uważnie, ponieważ często powoduje on problemy z wydajnością. W większości przypadków, powinieneś zamiast tego mieć możliwość przypisania stanu początkowego w konstruktorze. Może to być natomiast konieczne w przypadkach takich jak okna modalne i okienka podpowiedzi, kiedy przed wyrenderowaniem czegoś trzeba zmierzyć węzeł drzewa DOM.

componentDidUpdate() {#componentdidupdate}

componentDidUpdate(prevProps, prevState, snapshot)

Metoda `componentDidUpdate()` jest wywoływana bezpośrednio po wystąpieniu aktualizacji. Nie jest ona wywoływana po początkowym wyrenderowaniu.

Używaj tego jako okazji do operacji na drzewie DOM kiedy komponent został zaktualizowany. Jest to także dobre miejsce na wykonywanie zapytań sieciowych tak długo jak porównujesz obecne właściwości z poprzednimi (na przykład, zapytanie może być niepotrzebne jeśli właściwości się nie zmieniły).

```
componentDidUpdate(prevProps) {
  // Typowy sposób użycia (nie zapomnij porównać właściwości):
  if (this.props.userID !== prevProps.userID) {
```

```

    this.fetchData(this.props.userID);
}
}

```

Możesz wywołać metodę `setState()` od razu w `componentDidUpdate()`, ale weź pod uwagę, że musi ona być owinięta instrukcją warunkową jak w przykładzie powyżej, albo spowodujesz nieskończoną pętlę. Spowodowałoby to również dodatkowe renderowanie które, pomimo że niedostrzegalne dla użytkownika, może negatywnie wpływać na wydajność komponentu. Jeśli próbujesz "odzwierciedlić" pewien stan z właściwością pochodząą z góry, rozważ zamiast tego użycie tej właściwości bezpośrednio. Dowiedz się więcej o tym [dlaczego kopiowanie właściwości do stanu powoduje błędy](#).

Jeśli twój komponent ma zaimplementowaną metodę cyklu życia `getSnapshotBeforeUpdate()` (co jest rzadkie), wartość którą ona zwróci, będzie przekazana jako trzeci parametr ("rzut" (ang.`snapshot`)) do metody `componentDidUpdate()`. W innym wypadku ten parametr będzie miał wartość `undefined`.

Uwaga

Metoda `componentDidUpdate()` nie będzie wywołana jeśli `shouldComponentUpdate()` zwróci false.

componentWillUnmount() {#componentwillunmount}

`componentWillUnmount()`

Metoda `componentWillUnmount()` jest wywoływana zaraz przed odmontowaniem i zniszczeniem komponentu. Przeprowadź potrzebne czyszczenie w tej metodzie, takie jak unieważnienie liczników czasu, anulowanie zapytań sieciowych, lub czyszczenie subskrypcji, które były rozpoczęte w `componentDidMount()`.

Nie powinieneś wywoływać metody `setState()` w `componentWillUnmount()`, ponieważ ten komponent nie zostanie ponownie wrenderowany. Kiedy instancja komponentu zostaje odmonotowana, nigdy nie będzie zamontowana ponownie.

Rzadko używane metody cyklu życia

Metody zawarte w tej sekcji odpowiadają rzadkim przypadkom użycia. Czasem są przydatne, ale większość twoich komponentów najprawdopodobniej nie będzie ich potrzebowała. **Większość poniższych metod możesz zobaczyć na tym diagramie cyklu życia po kliknięciu na checkbox "Pokaż rzadziej używane metody" u góry.**

shouldComponentUpdate() {#shouldcomponentupdate}

`shouldComponentUpdate(nextProps, nextState)`

Używaj metody `shouldComponentUpdate()`, aby dać znać Reactowi, czy obecna zmiana stanu lub właściwości komponentu nie wpłynęła na jego wynik. Domyślnym zachowaniem, na którym powinieneś polegać w większości przypadków, jest ponowne renderowanie przy każdej zmianie stanu.

Metoda `shouldComponentUpdate()` jest wywoływana przed renderowaniem, gdy otrzymywane są nowe właściwości lub stan. Domyślnie wartość zwracana to `true`. Ta metoda nie jest wywoływana przy początkowym renderowaniu lub kiedy została użyta metoda `forceUpdate()`.

Ta metoda istnieje tylko jako **optymalizacja wydajności**. Nie polegaj na niej aby "zapobiegać" renderowaniu, co może prowadzić do błędów. Zamiast pisania `shouldComponentUpdate()` własnoręcznie, **rozważ użycie wbudowanej klasy `PureComponent`**. `PureComponent` przeprowadza płytkie porównanie właściwości i stanu, i obniża szansę na pominięcie niezbędnej aktualizacji.

Jeśli jesteś pewny, że chcesz ją napisać własnoręcznie, możesz porównać `this.props` z `nextProps` i `this.state` z `nextState` oraz zwrócić `false`, aby powiadomić Reacta, że aktualizacja może zostać pominięta. Zauważ, że zwrócenie `false` nie zapobiega ponownemu wyrenderowaniu komponentów potomnych, gdy *ich* stan się zmienia.

Nie zalecamy wykonywania głębokich porównań lub używania `JSON.stringify()` w metodzie `shouldComponentUpdate()`. Jest to bardzo nieefektywne i negatywnie odbije się na wydajności.

Obecnie, jeśli `shouldComponentUpdate()` zwróci `false`, `UNSAFE_componentWillUpdate()`, `render()` i `componentDidUpdate()` nie zostaną wywołane. W przyszłości React może traktować `shouldComponentUpdate()` jako wskazówkę, a nie jako ścisłą dyrektywę, a zwrócenie `false` może mimo wszystko skutkować ponownym wyrenderowaniem komponentu.

`static getDerivedStateFromProps() {#static-getderivedstatefromprops}`

`static getDerivedStateFromProps(props, state)`

Metoda `getDerivedStateFromProps` jest wywoływana zaraz przed wywołaniem metody `render`, zarówno przy początkowym montowaniu, jak i przy dalszych aktualizacjach. Powinna zwrócić obiekt, aby zaktualizować stan, lub zwrócić `null`, aby nie aktualizować nic.

Ta metoda istnieje dla **rzadkich przypadków użycia**, w których stan zależy od zmian właściwości w czasie. Na przykład, może okazać się przydatnym komponentem `<Transition>`, który porównuje swoje obecne komponenty potomne z poprzednimi, aby zdecydować, które z nich mają pojawić się z animacją, a które zniknąć.

Derywowanie stanu sprawia, że kod jest rozwlekły i trudno myśli się o komponentach. [Upewnij się, że znasz prostsze alternatywy:](#)

- Jeśli potrzebujesz **spowodować efekt uboczny** (na przykład pobranie danych, albo animację) w odpowiedzi na zmianę właściwości, zamiast tego użyj metody cyklu życia `componentDidUpdate`.
- Jeśli chcesz **ponownie obliczyć pewne dane tylko, kiedy zmieni się właściwość**, [zamiast tego użyj pomocniczych technik memoizacyjnych](#).

- Jeśli chcesz "zresetować" stan przy zmianie właściwości, rozważ zamiast tego użycie komponentu całkowicie kontrolowanego lub całkowicie niekontrolowanego z właściwością `key`.

Ta metoda nie ma dostępu do instancji komponentu. Jeśli chcesz, możesz używać ponownie kod pomiędzy `getDerivedStateFromProps()` innymi metodami klasy poprzez wyodrębnienie czystych funkcji właściwości i stanu komponentu poza definicję klasy.

Zauważ, że metoda ta wywoływana jest przy *każdym* renderowaniu, bez względu na przyczynę. Jest to kontrastem dla metody `UNSAFE_componentWillReceiveProps`, która zostaje wywołana tylko, kiedy komponent nadrzędny powoduje ponowne wyrenderowanie, a nie jako wynik lokalnego wywołania metody `setState`.

`getSnapshotBeforeUpdate()` { #getsnapshotbeforeupdate }

`getSnapshotBeforeUpdate(prevProps, prevState)`

Metoda `getSnapshotBeforeUpdate()` jest wywoływana zaraz przed tym, gdy ostatnio wyrenderowany wynik zostaje zatwierdzony do np. drzewa DOM. Pozwala to twojemu komponentowi na przejęcie pewnych informacji z drzewa DOM (np. pozycje scrolla) przed ich potencjalną zmianą. Każda wartość zwrocona przez metodę cyklu życia zostanie przekazana jako parametr do metody `componentDidUpdate()`.

Ten przypadek użycia nie jest powszechny, ale może wystąpić w interfejsach użytkownika takich jak wątki czatu, które potrzebują możliwości zarządzania pozycją scrolla w specjalny sposób.

Powinna być zwrocona wartość snapshotu (lub `null`).

Dla przykładu:

`embed:react-component-reference/get-snapshot-before-update.js`

W powyższych przykładach, ważne jest odczytanie właściwości `scrollHeight` w metodzie `getSnapshotBeforeUpdate`, ponieważ mogą wystąpić opóźnienia pomiędzy metodami cyklu życia w fazie "render" (takimi jak `render`), a metodami fazy "commit" (takimi jak `getSnapshotBeforeUpdate` i `componentDidUpdate`).

Granice błędu

Granice błędów to reaktywe komponenty, które wychwytyują javascriptowe wyjątki w dowolnych miejscach swojego drzewa komponentów potomnych, zapisują te wyjątki, i pokazują awaryjny interfejs użytkownika zamiast drzewa komponentów, które rzuciło tym wyjątkiem. Granice błędów wychwytyują wyjątki podczas renderowania, w metodach cyklu życia, i w konstruktorach całego drzewa potomnego.

Komponent klasowy staje się granicą błędu, jeśli ma zdefiniowaną jedną z (lub obie) metod cyklu życia `static getDerivedStateFromError()` lub `componentDidCatch()`. Aktualizacja stanu z tych metod pozwala na wychwycenie nieobsłużonego javascriptowego wyjątku w drzewie komponentów potomnych i pokazać rezerwowy interfejs użytkownika.

Używaj granic błędów tylko do rekonwalescencji po nieoczekiwanych wyjątkach; **nie próbuj używać ich do kontrolowania przebiegu programu.**

Po więcej szczegółów, odwiedź [Obsługa wyjątków w Reakcie 16.](#)

Uwaga

Granice błędów wychwytyują tylko wyjątki w komponentach z drzewa **pod** nimi. Granica błędów nie może wychwycić wyjątku, który wystąpił w niej samej.

static getDerivedStateFromError() {#static-getderivedstatefromerror}

`static getDerivedStateFromError(error)`

Ta metoda cyklu życia jest wywoływana po wyrzuceniu wyjątku przez komponent potomny. Wyrzucony wyjątek zostaje do niej przekazany jako argument, jej wynikiem powinna być wartość, która pozwoli na aktualizowanie stanu.

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

  static getDerivedStateFromError(error) {
    // Aktualizacja stanu, aby kolejne wyrenderowanie pokazało awaryjny interfejs
    // użytkownika.
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      // Możesz wyrenderować dowolny awaryjny interfejs użytkownika
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

Uwaga

Metoda `getDerivedStateFromError()` jest wywoływana podczas fazy "render", więc nie są w niej dozwolone skutki uboczne. Zamiast tego, dla tych przypadków użycia użyj metody `componentDidCatch()`.

componentDidCatch() {#componentdidcatch}

`componentDidCatch(error, info)`

Ta metoda cyklu życia jest wywoływana po wyrzuceniu wyjątku przez komponent potomny. Otrzymuje on dwa argumenty:

1. `error` - Wyjątek, który został wyrzucony.
2. `info` - Obiekt z kluczem `componentStack` zawierający informację o tym, który komponent wyrzucił ten wyjątek.

Metoda `componentDidCatch()` jest wywoływana w fazie "commit", więc dozwolone są w niej skutki uboczne. Powinna być używana do czynności takich jak zapisywanie błędów:

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

  static getDerivedStateFromError(error) {
    // Aktualizacja stanu, aby kolejne wypłaszczenie pokazało awaryjny interfejs
    // użytkownika.
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    // Przykładowy "componentStack":
    //   in ComponentThatThrows (created by App)
    //   in ErrorBoundary (created by App)
    //   in div (created by App)
    //   in App
    logComponentStackToMyService(info.componentStack);
  }

  render() {
    if (this.state.hasError) {
      // Możesz wypłaszczyć dowolny awaryjny interfejs użytkownika
      return <h1>Coś poszło nie tak.</h1>;
    }

    return this.props.children;
  }
}
```

Zbudowana paczka deweloperska będzie nieco różnić się od produkcyjnej pod względem sposobu obsługi błędów przez `componentDidCatch()`.

W środowisku deweloperskim błędy wędrują aż do `window`, co oznacza, że wszelkie procedury zarejestrowane za pomocą `window.onerror` lub `window.addEventListener('error', callback)` również przechwyca te błędy, które złapie `componentDidCatch()`.

Inaczej jest na produkcji, gdzie błędy nie wędrują aż na samą górę. Oznacza to, że nadzędne granice błędów otrzymają błąd tylko wtedy, gdy ich potomkowie wcześniej ich nie przechwycią za pomocą `componentDidCatch()`.

Uwaga

W razie wyjątku, możesz wyrenderować awaryjny interfejs użytkownika za pomocą metody `componentDidCatch()` poprzez wywołanie metody `setState`, ale możliwość ta będzie przestarzała w przyszłych wersjach. Do obsługi renderowania awaryjnego używaj zamiast tego metody `static getDerivedStateFromError()`.

Przestarzałe metody cyklu życia

Poniższe metody cyklu życia są oznaczone jako "przestarzałe". Wciąż działają, zalecamy jednak nie używać ich w nowym kodzie. Możesz dowiedzieć się więcej o migracji od przestarzałych metod cyklu życia [w tym wpisie na blogu](#).

UNSAFE_componentWillMount() {#unsafe_componentwillmount}

UNSAFE_componentWillMount()

Uwaga

Ta metoda cyklu życia była wcześniej nazwana `componentWillMount`. Ta nazwa będzie działać do wersji 17. Użyj [codemoda rename-unsafe-lifecycles](#), aby automatycznie zaktualizować swoje komponenty.

Metoda `UNSAFE_componentWillMount()` jest wywoływana zaraz przed nastaniem montowania. Jest wywoływana przed `render()`, zatem synchroniczne wywoływanie `setState()` w tej metodzie nie spowoduje dodatkowego renderowania. Generalnie, zamiast tego do inicjalizacji stanu zalecamy używania konstruktora.

Unikaj wprowadzania skutków ubocznych lub inicjalizowania subskrypcji w tej metodzie. Dla tych przypadków użycia, używaj zamiast tego metody `componentDidMount()`.

Jest to jedyna metoda cyklu życia wywoływana przy renderowaniu na serwerze.

UNSAFE_componentWillReceiveProps() {#unsafe_componentwillreceiveprops}

UNSAFE_componentWillReceiveProps(nextProps)

Uwaga

Ta metoda cyklu życia była wcześniej nazwana `componentWillReceiveProps`. Ta nazwa będzie działać do wersji 17. Użyj [codemoda rename-unsafe-lifecycles](#), aby automatycznie zaktualizować swoje komponenty.

Uwaga:

Używanie tej metody cyklu życia często prowadzi do błędów i niespójności

- Jeśli potrzebujesz **wykonać efekt uboczny** (na przykład, pobieranie danych lub animację) w odpowiedzi na zmianę właściwości, zamiast tego użyj metody cyklu życia `componentDidUpdate`.
- Jeśli używałeś `componentWillReceiveProps` do **ponownego obliczania pewnych danych tylko w przypadku zmiany właściwości**, zamiast tego użyj pomocniczych technik memoizacyjnych.
- Jeśli używałeś `componentWillReceiveProps` do "**resetowania" stanu w przypadku zmiany właściwości**", zamiast tego rozważ użycie komponentu `całkowicie kontrolowanym` lub `całkowicie niekontrolowanym z właściwością key`.

Dla innych przypadków użycia, [śledź rekomendacje w tym wpisie na blogu na temat stanu pochodnego](#).

Metoda `UNSAFE_componentWillReceiveProps()` jest wywoływana przed tym, jak zamontowany komponent otrzymuje nowe właściwości. Jeśli potrzebujesz zaktualizować stan w odpowiedzi na zmiany właściwości (na przykład, zresetować go), możesz porównać `this.props` i `nextProps` i wykonać przejście stanu w tej metodzie za pomocą `this.setState()`.

Zauważ, że jeśli komponent nadzędny powoduje ponowne wyrenderowanie twojego komponentu, ta metoda będzie wywołana nawet jeśli właściwości nie uległy zmianie. Jeśli chcesz tylko obsłużyć zmiany, upewnij się, że porównujesz poprzednie i obecne wartości.

React nie wywołuje metody `UNSAFE_componentWillReceiveProps()` z początkowymi właściwościami podczas [montowania](#). Wywołuje ją tylko, kiedy właściwości któregoś z komponentów mogą zostać zaktualizowane. Wywołanie metody `this.setState()` przeważnie nie powoduje wywołania `UNSAFE_componentWillReceiveProps()`.

UNSAFE_componentWillUpdate() {#unsafe_componentwillupdate}

`UNSAFE_componentWillUpdate(nextProps, nextState)`

Uwaga

Ta metoda cyklu życia była wcześniej nazwana `componentWillUpdate`. Ta nazwa będzie działać do wersji 17. Użyj [codemoda rename-unsafe-lifecycles](#), aby automatycznie zaktualizować swoje komponenty.

Metoda `UNSAFE_componentWillUpdate()` jest wywoływana zaraz przed renderowaniem, kiedy komponent uzyskuje nowe właściwości lub stan. Używaj tego jako okazji do przygotowania przed nastąpieniem aktualizacji. Ta metoda nie jest wywoływana przy początkowym renderowaniu.

Zauważ, że nie możesz tutaj wywołać `this.setState()`; nie powinieneś też robić niczego innego (np. wysyłania Reduxowych akcji), co spowodowałoby aktualizację reactowego komponentu przed powrotem z metody `UNSAFE_componentWillUpdate()`.

Metoda ta, zazwyczaj może być zastąpiona metodą `componentDidUpdate()`. Jeśli zczytywałeś w tej metodzie informacje z drzewa DOM (np. żeby zapisać pozycje scrolla), możesz przenieść tą logikę do `getSnapshotBeforeUpdate()`.

Uwaga

Metoda `UNSAFE_componentWillUpdate()` nie będzie wywołana jeśli `shouldComponentUpdate()` zwróci `false`.

Inne API

W przeciwieństwie do metod cyklu życia powyżej (które wywołuje dla ciebie React), metody poniżej możesz wywołać *ty* ze swoich komponentów.

Są tylko dwie takie metody: `setState()` i `forceUpdate()`.

`setState()` {#setstate}

 `setState(updater[, callback])`

`setState()` ustawia w kolejce zmiany stanu komponentu i daje znać Reactowi, że komponent i jego komponenty potomne powinny zostać ponownie wyrenderowane ze zaktualizowanym stanem. Jest to podstawaowa metoda używana do aktualizacji interfejsu użytkownika w odpowiedzi na procedury obsługi zdarzeń i odpowiedzi z serwera.

Myśl o metodzie `setState()` bardziej jako o *prośbie* niż o natychmiastowym poleceniu aktualizacji komponentu. Dla lepszej postrzeganej wydajności, React może ją opóźnić, a potem zaktualizować kilka komponentów za jednym zamachem. W rzadkich sytuacjach, kiedy potrzebujesz wymusić synchroniczną aktualizację DOM, możesz opakować ją w `flushSync`. Ale pamiętaj, że może na tym ucierpieć wydajność aplikacji.

Metoda `setState()` nie zawsze od razu aktualizuje komponent. Może ona złączyć lub odłożyć aktualizację na później. Sprawia to, że odczytywanie `this.state` zaraz po wywołaniu `setState()` jest potencjalną pułapką. Zamiast tego, użyj metody `componentDidUpdate` lub funkcji zwrotnej (ang. *callback*) `setState(updater, callback)`, które są wywoływanie po zastosowaniu aktualizacji. Jeśli potrzebujesz zmienić stan w oparciu o poprzedni stan, zapoznaj się z poniższym argumentem `updater`.

`setState()` zawsze powoduje ponowne renderowanie komponentu, chyba że `shouldComponentUpdate()` zwróci `false`. Jeśli przechowujemy mutowalne obiekty, a logiki decydującej o potrzebie ponownego renderowania nie da się zatrzymać w metodzie `shouldComponentUpdate()`, możemy uniknąć zbędnego renderowania wywołując `setState()` tylko wtedy, gdy nowy stan różni się od poprzedniego.

Pierwszym argumentem jest funkcja `updater` posiadająca poniższą sygnaturę:

```
(state, props) => stateChange
```

`state` jest referencją stanu komponentu w momencie, w którym zmiana zostaje zastosowana. Nie powinien on być bezpośrednio zmieniany. Zamiast tego, zmiany powinny być reprezentowane poprzez zbudowanie nowego obiektu na podstawie `state` and `props`. Na przykład założymy, że chcemy powiększyć pewną wartość w stanie o `props.step`:

```
this.setState((state, props) => {
  return {counter: state.counter + props.step};
});
```

Zarówno `state`, jak i `props` otrzymywane przez funkcję aktualizującą są aktualne. Wynik aktualizatora zostaje płytko scalony ze stanem.

Drugi parametrem metody `setState()` jest opcjonalna funkcja zwrotna, która zostanie wywołana kiedy `setState` ukończy swój przebieg i komponent zostanie ponownie wyrenderowany. Ogólnie rzecz biorąc, do tego typu logiki zalecamy zamiast tego używać metody `componentDidUpdate()`.

Opcjonalnie, jako pierwszy argument do metody `setState()` zamiast funkcji możesz przekazać obiekt:

```
setState(stateChange[, callback])
```

Powoduje to przeprowadzenie płytkiego scalenia argumentu `stateChange` do nowego stanu, np., w celu dostosowania ilości przedmiotów w koszyku:

```
this.setState({quantity: 2})
```

Ta forma metody `setState()` także jest asynchroniczna, a wywołanie jej wiele razy podczas jednego cyklu może spowodować ich złączenie. Na przykład, jeśli spróbujesz zwiększyć ilość przedmiotów więcej niż jeden raz w tym samym cyklu, rezultatem tego będzie ekwiwalent:

```
Object.assign(
  previousState,
  {quantity: state.quantity + 1},
  {quantity: state.quantity + 1},
  ...
)
```

Następujące wywołania nadpiszą wartości z poprzednich w tym samym cyklu, więc ilość będzie zwiększona tylko raz. Jeśli kolejny stan zależy od poprzedniego, zamiast tego zalecamy używania formy z funkcją aktualizującą:

```
this.setState((state) => {
  return {quantity: state.quantity + 1};
});
```

Po więcej szczegółów, odwiedź:

- [Przewodnik po stanie i cyklu życia](#)

- Dogłębnie: Kiedy i dlaczego wywołania `setState()` są łączone?
 - Dogłębnie: Dlaczego `this.state` nie jest aktualizowany od razu?
-

`forceUpdate() {#forceupdate}`

`component.forceUpdate(callback)`

Domyślnie, kiedy zmienia się stan lub właściwości twojego komponentu, zrenderuje się on ponownie. Jeśli Twoja metoda `render()` polega na innych danych, możesz powiadomić Reacta, że komponent potrzebuje ponownego wyrenderowania, poprzez wywołanie metody `forceUpdate()`.

Wywołanie `forceUpdate()` spowoduje, że na komponencie zostanie wywołana metoda `render()`, z pominięciem metody `shouldComponentUpdate()`. Spowoduje to wywołanie normalnych metod cyklu życia komponentów potomnych, włączając w to metodę `shouldComponentUpdate()` każdego z nich. React wciąż aktualizuje drzewo DOM tylko w wypadku zmiany znaczników.

Przeważnie powinieneś unikać jakichkolwiek form użycia `forceUpdate()` i odczytywać dane jedynie z `this.props` i `this.state` w metodzie `render()`.

Właściwości Klasy

`defaultProps {#defaultprops}`

`defaultProps` może być zdefiniowana jako własność na samej klasie komponentu, aby ustawić domyślne właściwości dla tej klasy. Jest ona używana dla właściwości równych `undefined`, ale nie `null`. Dla przykładu:

```
class CustomButton extends React.Component {
  // ...
}

CustomButton.defaultProps = {
  color: 'blue'
};
```

Jeśli `props.color` nie jest podany, zostanie domyślnie ustawiony na `'blue'`:

```
render() {
  return <CustomButton />; // props.color zostanie ustawiony na blue
}
```

Jeśli `props.color` zostanie ustawiony jako `null`, pozostanie nim:

```
render() {
  return <CustomButton color={null} />; // props.color pozostanie równy null
}
```

displayName {#displayname}

Napis `displayName` jest używany w komunikatach debugowania. Przeważnie, nie musisz jawnie go definiować, ponieważ jest on wywnioskowany z nazwy funkcji lub klasy, w której zdefiniowany jest ten komponent. Możesz chcieć jawnie go zdefiniować, jeśli chcesz wyświetlić inną nazwę komponentu przy debugowaniu lub kiedy stworzysz komponent wyższego rzędu, po szczegóły odwiedź [Owiń nazwę wyświetlenia dla łatwego debugowania](#).

Właściwości instancji

props {#props}

`this.props` zawiera właściwości, które zostały zdefiniowane przez przywołującego tego komponentu. Po wprowadzenie do właściwości, odwiedź [Komponenty i właściwości](#).

W szczególności, `this.props.children` jest specjalną właściwością, zazwyczaj zdefiniowaną poprzez potomne tagi w wyrażeniu JSX, a nie w samym tagu instancji.

state {#state}

Stan zawiera dane specyficzne dla tego komponentu, które mogą zmieniać się w czasie. Stan jest definiowany przez użytkownika i powinien być zwykłym javascriptowym obiektem.

Jeśli jakaś wartość nie jest używana do renderowania ani przepływu danych (na przykład, ID licznika czasu), nie musisz umieszczać jej w stanie. Wartości tego typu mogą być zdefiniowane jako pola składowe instancji komponentu.

Po informacje na temat stanu, odwiedź [Stan i cykl życia](#).

Nigdy nie zmieniaj `this.state` bezpośrednio, gdyż późniejsze wywołanie `setState()` może zastąpić wykonaną przez ciebie zmianę. Traktuj `this.state` jako niezmienny.

ReactDOMClient

The `react-dom/client` package provides client-specific methods used for initializing an app on the client. Most of your components should not need to use this module.

```
import * as ReactDOM from 'react-dom/client';
```

If you use ES5 with npm, you can write:

```
var ReactDOM = require('react-dom/client');
```

Overview

The following methods can be used in client environments:

- `createRoot()`
- `hydrateRoot()`

Browser Support

React supports all modern browsers, although [some polyfills are required](#) for older versions.

Note

We do not support older browsers that don't support ES5 methods or microtasks such as Internet Explorer. You may find that your apps do work in older browsers if polyfills such as [es5-shim](#) and [es5-sham](#) are included in the page, but you're on your own if you choose to take this path.

Reference

`createRoot()` [\[#createroot\]](#)

```
createRoot(container[, options]);
```

Create a React root for the supplied `container` and return the root. The root can be used to render a React element into the DOM with `render`:

```
const root = createRoot(container);
root.render(element);
```

`createRoot` accepts two options:

- `onRecoverableError` : optional callback called when React automatically recovers from errors.
- `identifierPrefix` : optional prefix React uses for ids generated by `React.useId`. Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix used on the server.

The root can also be unmounted with `unmount`:

```
root.unmount();
```

Note:

`createRoot()` controls the contents of the container node you pass in. Any existing DOM elements inside are replaced when `render` is called. Later calls use React's DOM diffing algorithm for efficient updates.

`createRoot()` does not modify the container node (only modifies the children of the container). It may be possible to insert a component to an existing DOM node without overwriting the existing children.

Using `createRoot()` to hydrate a server-rendered container is not supported. Use `hydrateRoot()` instead.

hydrateRoot() {#hydrateroot}

```
hydrateRoot(container, element[, options])
```

Same as `createRoot()`, but is used to hydrate a container whose HTML contents were rendered by `ReactDOMServer`. React will attempt to attach event listeners to the existing markup.

`hydrateRoot` accepts two options:

- `onRecoverableError`: optional callback called when React automatically recovers from errors.
- `identifierPrefix`: optional prefix React uses for ids generated by `React.useId`. Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix used on the server.

Note

React expects that the rendered content is identical between the server and the client. It can patch up differences in text content, but you should treat mismatches as bugs and fix them. In development mode, React warns about mismatches during hydration. There are no guarantees that attribute differences will be patched up in case of mismatches. This is important for performance reasons because in most apps, mismatches are rare, and so validating all markup would be prohibitively expensive.

[Go to TOC](#)

ReactDOMServer

Obiekt `ReactDOMServer` pozwala na renderowanie komponentów do statycznych znaczników. Zazwyczaj jest używany na serwerze Node'owym:

```
// ES modules
import * as ReactDOMServer from 'react-dom/server';
// CommonJS
var ReactDOMServer = require('react-dom/server');
```

Ogólne informacje

Następujące metody mogą zostać użyte tylko w **środowiskach obsługujących strumienie Node.js (ang. Node.js Streams)**:

- `renderToPipeableStream()`
- `renderToNodeStream()` (przestarzałe)
- `renderToStaticNodeStream()`

Następujące metody mogą być użyte tylko w **środowiskach obsługujących strumienie webowe (ang. Web Streams)** (m.in. przeglądarki, Deno, niektóre nowoczesne wersje Edge):

- `renderToReadableStream()`

Następujące metody mogą być użyte w środowiskach, które nie obsługują strumieni:

- `renderToString()`
- `renderToStaticMarkup()`

Dokumentacja

`renderToPipeableStream() {#rendertopipeablestream}`

`ReactDOMServer.renderToPipeableStream(element, options)`

Renderuje reaktywy element do jego początkowego kodu HTML. Zwraca strumień z metodą `pipe(res)` do przekierowania wyniku oraz `abort()` do przerwania żądania. Posiada pełne wsparcie dla zawieszeń (ang. *Suspense*) i strumieniowania kodu HTML o "opóźnionych" blokach treści "wskakujących" jako znaczniki `<script>`. [Czytaj więcej na ten temat](#)

Jeśli wywołasz metodę `ReactDOM.hydrateRoot()` na węźle, który zawiera już znaczniki wypierane po stronie serwera, React zachowa je i dołączy jedynie procedury obsługi zdarzeń. Poprawi to wydajność i wrzeciona przy pierwszym ładowaniu strony.

```
let didError = false;
const stream = renderToPipeableStream(
  <App />,
  {
```

```

onShellReady() {
  // Treść ponad wszystkimi granicami zawieszeń (ang. *Suspense boundaries*)
  jest gotowa.
  // Jeśli coś się "wykrzeczyło" przed początkiem strumieniowania, ustawiamy
  odpowiedni kod błędu.
  res.statusCode = didError ? 500 : 200;
  res.setHeader('Content-type', 'text/html');
  stream.pipe(res);
},
onShellError(error) {
  // Coś się "wykrzeczyło", zanim zakończyliśmy powłokę, więc wysyłamy
  alternatywną powłokę.
  res.statusCode = 500;
  res.send(
    '<!doctype html><p>Ładowanie...</p><script src="clientrender.js">
</script>'
  );
},
onAllReady() {
  // Jeśli nie potrzebujesz strumieniowania, skorzystaj z tej metody zamiast
  onShellReady.
  // Zostanie ona wywołana po wygenerowaniu treści całej strony.
  // Można jej użyć w celu dostarczenia treści dla wyszukiwarek lub dla
  wygenerowania statycznej treści.

  // res.statusCode = didError ? 500 : 200;
  // res.setHeader('Content-type', 'text/html');
  // stream.pipe(res);
},
onError(err) {
  didError = true;
  console.error(err);
},
}
);

```

Zobacz pełną listę opcji.

Uwaga:

To jest funkcjonalność specyficzna dla Node.js. Środowiska z obsługą [strumieni webowych \(ang. Web Streams\)](#), jak np. Deno czy współczesne wersje Edge, powinny korzystać z `renderToReadableStream`.

`renderToReadableStream()` [\[#rendertoreadablestream\]](#)

`ReactDOMServer.renderToReadableStream(element, options);`

Strumieniuje reaktywy element do jego początkowego kodu HTML. Zwraca obietnicę (ang. *Promise*), która zwraca [strumień do odczytu \(ang. Readable Stream\)](#). Posiada pełne wsparcie dla zawieszania (ang. *Suspense*) oraz strumieniowania HTML. [Czytaj więcej na ten temat](#)

Jeśli wywołasz metodę `ReactDOM.hydrateRoot()` na węźle, który zawiera już znaczniki wyrenderowane po stronie serwera, React zachowa je i dołączy jedynie procedury obsługi zdarzeń. Poprawi to wydajność i wrażenia przy pierwszym ładowaniu strony.

```
let controller = new AbortController();
let didError = false;
try {
  let stream = await renderToReadableStream(
    <html>
      <body>Sukces</body>
    </html>,
  {
    signal: controller.signal,
    onError(error) {
      didError = true;
      console.error(error);
    }
  }
);

// Można w ten sposób poczekać na zakończenie wszystkich zawieszeń (ang.
*Suspense*). Możesz
// odkomentować tę linię, jeśli chcesz zbuforować cały kod HTML, zamiast go
strumieniować.
// Możesz także użyć tego do przystosowania aplikacji dla wyszukiwarek lub do
wygenerowania statycznej treści:

// await stream.allReady;

return new Response(stream, {
  status: didError ? 500 : 200,
  headers: {'Content-Type': 'text/html'},
});
} catch (error) {
  return new Response(
    '<!doctype html><p>Ładowanie...</p><script src="clientrender.js"></script>',
  {
    status: 500,
    headers: {'Content-Type': 'text/html'},
  }
);
}
```

Zobacz pełną listę opcji.

Uwaga:

Ta funkcjonalność jest zależna od [strumieni web \(ang. Web Streams\)](#). W Node.js użyj `renderToPipableStream`.

`renderToNodeStream()` (przestarzałe)

ReactDOMServer `renderToNodeStream(element)`

Renderuje reaktywy element do jego początkowego kodu HTML. Zwraca strumień Node.js do odczytu, który na wyjściu zwróci ciąg znaków HTML. Zwrócony przez strumień kod HTML jest identyczny z tym, który zwróciłaby funkcja `ReactDOMServer.renderToString`. Możesz użyć tej metody, aby wygenerować kod HTML po stronie serwera, a następnie wysłać znaczniki jako odpowiedź na pierwsze żądanie, co pozwoli przyspieszyć ładowanie strony i umożliwić wyszukiwarekom indeksowanie jej w celach SEO.

Jeśli wywołasz metodę `ReactDOM.hydrateRoot()` na węźle, który zawiera już znaczniki wyrenderowane po stronie serwera, React zachowa je i dołączy jedynie procedury obsługi zdarzeń. Poprawi to wydajność i wrażenia przy pierwszym ładowaniu strony.

Uwaga:

Do użycia tylko po stronie serwera. Ten interfejs API nie jest dostępny w przeglądarce.

Strumień zwrócony przez tę metodę zwróci strumień bajtów zakodowany w utf-8. Jeśli potrzebujesz strumienia z innym kodowaniem, skorzystaj na przykład z paczki `iconv-lite`, która dostarcza strumienie transformujące do transkodowania tekstu.

`renderToStaticNodeStream()` `{#rendertostaticnodestream}`

`ReactDOMServer.renderToStaticNodeStream(element)`

Działa analogicznie do `renderToNodeStream`, z tą różnicą, że nie tworzy dodatkowych atrybutów DOM, takich jak `data-reactroot` (używanych wewnętrznie przez Reacta). Przydaje się, jeśli chcesz używać Reacta jako prostego generatora statycznych stron, gdzie usunięcie dodatkowych atrybutów pozwoli zaoszczędzić kilka bajtów.

Kod HTML zwrócony przez strumień jest identyczny z tym, jaki zwróciłaby funkcja `ReactDOMServer.renderToStaticMarkup`.

Jeżeli planujesz używać Reacta po stronie klienta w celu dodania znacznikom interaktywności, nie używaj tej metody. Zamiast niej użyj `renderToNodeStream` na serwerze i `ReactDOM.hydrateRoot()` po stronie klienta.

Uwaga:

Do użycia tylko po stronie serwera. Ten interfejs API nie jest dostępny w przeglądarce.

Strumień zwrócony przez tę metodę zwróci strumień bajtów zakodowany w utf-8. Jeśli potrzebujesz strumienia z innym kodowaniem, skorzystaj na przykład z paczki `iconv-lite`, która dostarcza strumienie transformujące do transkodowania tekstu.

renderToString() {#rendertostring}

ReactDOMServer.renderToString(element)

Renderuje element reaktywy do jego początkowego kodu HTML. Zwraca kod HTML w formie tekstowej. Możesz użyć tej metody, aby wygenerować kod HTML po stronie serwera, a następnie wysłać znaczniki jako odpowiedź na pierwsze żądanie, co pozwoli przyspieszyć ładowanie strony i umożliwić wyszukiwarem indeksowanie jej w celach SEO.

Jeśli wywołasz `ReactDOM.hydrateRoot()` na węźle, który zawiera już taki kod wyrenderowany po stronie serwera, React zachowa go i jedynie doda do niego procedury obsługi zdarzeń, zapewniając szybkie wstępne ładowanie stronie.

Uwaga

Ta funkcjonalność na ograniczone wsparcie dla zawieszeń (ang. *Suspense*) i nie obsługuje strumieniowania.

Sugerujemy po stronie serwera korzystać z `renderToPipeableStream` (dla Node.js) lub `renderToReadableStream` (dla Web Streams) instead.

renderToStaticMarkup() {#rendertostaticmarkup}

ReactDOMServer.renderToStaticMarkup(element)

Działa podobnie do `renderToString`, lecz nie tworzy dodatkowych atrybutów DOM, których React potrzebuje do działania, np. `data-reactroot`. Przydaje się, jeśli chcesz użyć Reacta jako prostego generowania statycznych stron, ponieważ usunięcie ich pozwala zaoszczędzić kilka cennych bajtów.

Jeśli planujesz używać Reacta po stronie klienta, aby dodać aplikacji nieco interakcyjności, nie używaj tej metody. Zamiast niej skorzystaj z `renderToString` po stronie serwera, a następnie `ReactDOM.hydrateRoot()` po stronie klienta.

[Go to TOC](#)

ReactDOM

Pakiet `react-dom` udostępnia metody specyficzne dla DOM, które mogą być używane na najwyższym poziomie aplikacji i, w razie potrzeby, jako "wyjście awaryjne" poza model Reacta.

```
import * as ReactDOM from 'react-dom';
```

If you use ES5 with npm, you can write:

```
var ReactDOM = require('react-dom');
```

Pakiet `react-dom` dostarcza również moduły właściwe dla aplikacji klienckich i serwerowych:

- `react-dom/client`
- `react-dom/server`

Informacje ogólne

Pakiet `react-dom` eksportuje:

- `createPortal()`
- `flushSync()`

These `react-dom` methods are also exported, but are considered legacy:

- `render()`
- `hydrate()`
- `findDOMNode()`
- `unmountComponentAtNode()`

Note:

Both `render` and `hydrate` have been replaced with new [client methods](#) in React 18. These methods will warn that your app will behave as if it's running React 17 (learn more [here](#)).

Wsparcie dla przeglądarek

React wspiera wszystkie popularne przeglądarki, jednak dla starszych wersji [wymagane są niektóre łatki](#) (ang. *polyfills*).

Uwaga

Nie wspieramy starszych przeglądarek, które nie obsługują metod ze standardu ES5 lub mikrozadań, takich jak Internet Explorer. Twój aplikacja może zadziałać na nich, jeśli użyjesz odpowiednich latek, jak na przykład [es5-shim](#) i [es5-sham](#). Pamiętaj jednak, że pójście tą drogą skazuje cię na całkowitą samodzielność.

Dokumentacja

`createPortal()` [\[#createportal\]](#)

```
createPortal(child, container)
```

Tworzy portal. Portale umożliwiają renderowanie elementów do węzła DOM istniejącego poza hierarchią danego komponentu .

`flushSync()` [\[#flushsync\]](#)

```
flushSync(callback)
```

Wymusza synchroniczne wykonanie wszelkich aktualizacji zawartych wewnętrz przekazanej funkcji. Metoda ta jest przydatna, gdy chcemy mieć natychmiastowy dostęp do wyników tych aktualizacji.

```
// Spraw, żeby ta zmiana stanu była synchroniczna.
flushSync(() => {
  setCount(count + 1);
});
// W tym miejscu stan jest już zaktualizowany.
```

Uwaga:

`flushSync` może mieć znaczący wpływ na wydajność aplikacji. Używaj z rozwagą.

`flushSync` może wymusić na granicach zawieszenia (ang. *Suspense boundaries*), aby wyświetliły swój komponent zastępczy (ang. *fallback*).

`flushSync` przed zakończeniem swojego działania może wymusić wywołanie efektów i synchroniczne zaaplikowanie wszelkich zmian z nimi związanych.

`flushSync` może również wymusić aktualizacje spoza przekazanej funkcji. Na przykład, jeśli aktualnie przetwarzane są jakieś aktualizacje z kliknięcia na ekranie, React może również je wymusić zanim przejdzie do aktualizacji z przekazanej funkcji.

Przestarzałe funkcjonalności

`render()` [\[#render\]](#)

```
render(element, container[, callback])
```

Uwaga:

W wersji React 18, `render` zostało zastąpione przez `createRoot`. Aby dowiedzieć się więcej, przeczytaj sekcję o [createRoot](#).

Renderuje element reaktywy do podanego kontenera w drzewie DOM i zwraca [referencję](#) do komponentu (lub zwraca `null` dla [komponentów bezstanowych](#)).

Jeśli element reaktywy był już wcześniej renderowany do kontenera `container`, zostanie on automatycznie zaktualizowany przez Reacta, który odpowiednio zmodyfikuje DOM tak, aby odzwierciedlić najnowszą wersję komponentu.

Jeśli w argumencie `callback` przekażesz funkcję zwrotną, zostanie ona wywołana po wyrenderowaniu lub zaktualizowaniu komponentu.

Uwaga:

`render()` kontroluje zawartość podanego węzła kontenera. Po pierwszym wywołaniu zastępowane są wszystkie elementy DOM wewnątrz niego. Każde kolejne wywołania, z pomocą reaktywego algorytmu różnicującego, efektywnie aktualizują drzewo.

`render()` nie modyfikuje węzła kontenera (jedynie jego elementy potomne). Możliwe jest wstawienie komponentu do istniejącego węzła DOM bez nadpisywania istniejących elementów podległych.

`render()` obecnie zwraca referencję do instancji klasy `ReactComponent`, będącej korzeniem drzewa. Jednak używanie tej referencji jest uznawane za przestarzałą praktykę i należy jej unikać, ponieważ przyszłe wersje Reacta mogą w niektórych przypadkach renderować komponenty asynchronicznie. Jeśli potrzebujesz referencji do instancji korzenia, sugerujemy przekazanie do niego [referencyjnej funkcji zwrotnej](#).

Używanie `render()` do hydratacji (ang. *hydrating*) kontenera renderowanego po stronie serwera jest przestarzałą praktyką. Zamiast tego użyj funkcji `hydrateRoot()`.

`hydrate()` {#hydrate}

`hydrate(element, container[, callback])`

Uwaga:

W Reakcie 18 funkcja `hydrate` została zastąpiona przez `hydrateRoot`. Aby dowiedzieć się więcej, przeczytaj sekcję o [hydrateRoot](#).

Działa podobnie do funkcji `render()`, ale służy do odtworzenia kontenera, którego struktura HTML została wyrenderowana przez `ReactDOMServer`. React podejmie próbę dołączenia detektorów zdarzeń do istniejących elementów.

React oczekuje, że renderowana treść będzie identyczna między serwerem a klientem. Potrafi, co prawda, poprawić różnice w treści tekstu, ale należy traktować niedopasowania jako błędy i zawsze je naprawiać. W trybie deweloperskim React ostrzega przed niedopasowaniem podczas procesu odtwarzania struktury. Nie ma gwarancji, że różnice w atrybutach zostaną odpowiednio poprawione w przypadku niedopasowania. Jest to ważne ze względu na wydajność, ponieważ w większości aplikacji niedopasowania są rzadkie, a zatem sprawdzenie wszystkich znaczników byłoby zbyt kosztowne obliczeniowo.

Jeśli któryś z atrybutów elementu lub treść tekstu intencjonalnie różnią się między serwerem a klientem (jak w przypadku znacznika czasu), możesz wyłączyć ostrzeżenie, dodając do elementu atrybut `suppressHydrationWarning={true}`. Działa to tylko na tym konkretnym elemencie i jest swego rodzaju "wyjściem awaryjnym". Nie nadużywaj go. O ile nie jest to treść tekstowa, React nie będzie próbował na siłę nanosić poprawek, więc wartość może pozostać niespójna do momentu jej kolejnej aktualizacji.

Jeśli potrzebujesz celowo renderować coś innego po stronie serwera i klienta, możesz wykonać renderowanie dwuprzebiegowe. Komponenty, które renderują coś innego po stronie klienta, mogą bazować na zmiennej stanu, np. `this.state.isClient`, którą można ustawić na `true` w metodzie `componentDidMount()`. W ten sposób początkowe renderowania zwróci tę samą zawartość co serwer, unikając niedopasowania. Jednak zaraz po odtworzeniu struktury w sposób synchroniczny nastąpi dodatkowe renderowanie. Zauważ, że to podejście spowolni działanie komponentów, ponieważ będą musiały być renderowane dwukrotnie - dlatego używaj go z rozwagą.

Pamiętaj, aby zwrócić uwagę na tzw. "user experience" użytkowników z wolnym połączeniem internetowym. Kod javascriptowy może załadować się znacznie później niż nastąpi pierwsze renderowanie kodu HTML. Z tego powodu, jeśli wyrenderujesz coś innego podczas przebiegu po stronie klienta, strona może się "przycinać". Możliwe, że w tej sytuacji pomoże wyrenderowanie "powłoki" (ang. *shell*) aplikacji po stronie serwera, a w kliencie wyświetlenie jedynie dodatkowych widgetów. Aby dowiedzieć się, jak to zrobić, nie napotykając problemów związanych z niedopasowaniem znaczników, zapoznaj się z wyjaśnieniem zawartym w poprzednim akapicie.

`unmountComponentAtNode()` {#unmountcomponentatnode}

```
unmountComponentAtNode(container);
```

Uwaga:

W Reakcie 18 funkcja `unmountComponentAtNode` została zastąpiona przez `root.unmount()`. Aby dowiedzieć się więcej, przeczytaj sekcję o [createRoot](#).

Usuwa zamontowany komponent z drzewa DOM, usuwając jego stan i procedury obsługi zdarzeń. Jeśli we wskazanym kontenerze nie zamontowano jeszcze żadnego elementu, wywoływanie tej funkcji nie daje żadnego efektu. Zwraca `true`, jeśli komponent został odmontowany lub `false`, jeśli kontener był pusty.

`findDOMNode()` `{#findDOMNode}`

Uwaga:

`findDOMNode` jest swego rodzaju "wyjściem awaryjnym", za pomocą którego możemy uzyskać dostęp do szukanego węzła DOM. Odradzamy korzystania z tej funkcji, ponieważ zaburza ona abstrakcję struktury komponentów. [Została wycofana w StrictMode](#).

`findDOMNode(component)`

Jeśli wskazany komponent został zamontowany w drzewie DOM, funkcja zwróci odpowiadający mu natywny element DOM. Przydaje się do odczytywania wartości z drzewa DOM, np. danych z pól formularza lub wymiarów interfejsu. **W większości przypadków wystarczy jednak "podpiąć" się do węzła DOM za pomocą właściwości `ref`, całkowicie unikając stosowania funkcji `findDOMNode`.**

Jeśli komponent renderuje `null` lub `false`, `findDOMNode` zwróci `null`. Jeśli renderuje ciąg znaków, `findDOMNode` zwróci tekst danego węzła DOM. Od Reakta w wersji 16 w górę komponenty mogą również zwracać tzw. fragmenty, zawierające kilku potomków. W takim przypadku `findDOMNode` zwróci węzeł DOM odpowiadający pierwszemu niepustemu potomkowi.

Uwaga:

`findDOMNode` działa tylko na zamontowanych komponentach (czyli takich, które zostały umieszczone w drzewie DOM). Jeśli spróbujesz wywołać tę funkcję na niezamontowanym komponencie (np. jeśli wywołasz `findDOMNode()` w metodzie `render` komponentu, który jeszcze nie został utworzony), zostanie zgłoszony wyjątek.

`findDOMNode` nie może być używane w komponentach funkcyjnych.

Główne API Reacta

Obiekt `React` jest punktem wejściowym do biblioteki React. Jeśli załadowujesz ją używając tagu `<script>`, główny interfejs API będzie dostępny w zmiennej globalnej `React`. Jeśli używasz standardu ES6, możesz skorzystać ze składni `import React from 'react'`. Jeśli używasz starszego standardu ES5, użyj składni `var React = require('react')`.

Ogólne informacje

Komponenty

Komponenty reaktywne pozwalają podzielić interfejs użytkownika na niezależne, enkapsulowane elementy, z których możesz korzystać w wielu miejscach. Komponenty w Reakcie mogą być definiowane poprzez dziedziczenie po klasie `React.Component` lub `React.PureComponent`.

- `React.Component`
- `React.PureComponent`

Jeśli nie używasz standardu ES6 do definiowania klas, możesz skorzystać z modułu `create-react-class`. W innym rozdziale opisaliśmy sposób na [korzystanie z Reacta bez ES6](#).

Komponenty reaktywne można również zdefiniować jako funkcje, a następnie otoczyć:

- `React.memo`

Tworzenie elementów reaktywnych

Do opisania wyglądu interfejsu użytkownika polecamy [korzystanie ze składni JSX](#). W JSX elementy są tylko wygodniejszymi odpowiednikami wywołania metody `React.createElement()`. Jeśli korzystasz z JSX, prawdopodobnie nigdy nie przyjdzie ci korzystać z tych metody bezpośrednio.

- `createElement()`
- `createFactory()`

Aby dowiedzieć się więcej na ten temat, zajrzyj do rozdziału pt. ["React bez JSX"](#).

Przekształcanie elementów

`React` udostępnia kilka API do manipulowania elementami:

- `cloneElement()`
- `isValidElement()`
- `React.Children`

Fragmenty

`React` pozwala także renderować wiele komponentów bez konieczności używania komponentu opakowującego.

- `React.Fragment`

Referencje (ang. refs)

- `React.createRef`
- `React.forwardRef`

Suspense

`Suspense` pozwala komponentowi "poczekać" na konkretne zdarzenie przez wyrenderowaniem. Obecnie `Suspense` obsługuje tylko jeden przypadek użycia: [dynamiczne ładowanie komponentów przy użyciu `React.lazy`](#). W przyszłości będzie wspierał także inne przypadki użycia, jak na przykład pobieranie danych.

- `React.lazy`
- `React.Suspense`

Tranzycje

`Tranzycje` są nową funkcjonalnością współbieżną dodaną w Reakcie 18. Pozwalają oznaczyć aktualizacje stanu jako tranzycje, informując tym samym Reacta, że mogą one zostać przerwane i nie wymagają wyświetlenia komponentu zastępczego `Suspense` dla widocznej już treści.

- `React.startTransition`
- `React.useTransition`

Hooki

`Hooki` są nowym dodatkiem w Reakcie 16.8. Pozwalają one używać stanu i innych funkcjonalności Reacta bez użycia klas. Hooki mają [dedykowany rozdział w dokumentacji](#) oraz osobny interfejs API:

- [Podstawowe hooki](#)
 - `useState`
 - `useEffect`
 - `useContext`
- [Dodatkowe hooki](#)
 - `useReducer`
 - `useCallback`
 - `useMemo`
 - `useRef`
 - `useImperativeHandle`
 - `useLayoutEffect`
 - `useDebugValue`
 - `useDeferredValue`

- `useTransition`
 - `useId`
 - [Library Hooks](#)
 - `useSyncExternalStore`
 - `useInsertionEffect`
-

Dokumentacja

`React.Component` [{#reactcomponent}](#)

`React.Component` to klasa bazowa dla komponentów reaktowych, definiowanych przez użycie [klasy ze standardu ES6](#):

```
class Greeting extends React.Component {
  render() {
    return <h1>Witaj, {this.props.name}</h1>;
  }
}
```

Lista metod i właściwości związanych z klasą `React.Component` znajduje się w [dokumentacji API dla React.Component](#).

`React.PureComponent` [{#reactpurecomponent}](#)

Klasa `React.PureComponent` jest podobna do `React.Component`. Różnica między nimi jest taka, że `React.Component` nie posiada implementacji metody `shouldComponentUpdate()`, natomiast `React.PureComponent` implementuje ją z użyciem płytkiego porównania właściwości (*ang. props*) i stanu.

Jeśli metoda `render()` danego komponentu wyświetla ten sam rezultat przy tych samych właściwościach i stanie, możesz przekształcić go na `React.PureComponent`, by poprawić wydajność.

Uwaga

Metoda `shouldComponentUpdate()` klasy `React.PureComponent` wykonuje jedynie płytkie porównanie obiektów. Jeśli zawierają one skomplikowaną strukturę danych, może to wprowadzić nieprawidłowości podczas zmian na głębszych poziomach struktury. Korzystaj z `PureComponent` tylko, jeśli spodziewasz się prostej struktury właściwości i stanu, lub użyj `forceUpdate()`, jeśli wiesz, że dane umieszczone głęboko w strukturze uległy zmianie. Możesz także zastanowić się nad skorzystaniem z [niezmiennych obiektów](#) (*ang. immutable objects*), by ułatwić szybkie porównanie zagnieżdżonych danych.

Dodatkowo, `shouldComponentUpdate()` klasy `React.PureComponent` pomija aktualizację właściwości dla całego poddrzewa komponentu. Upewnij się, że wszystkie komponenty potomne również korzystają z tego rozwiązania.

React.memo {#reactmemo}

```
const MyComponent = React.memo(function MyComponent(props) {
  /* renderuj korzystając z właściwości */
});
```

React.memo jest komponentem wyższego rzędu (ang. *higher-order component*).

Jeśli twój komponent przy takich samych właściwościach zawsze renderuje tę samą strukturę, możesz opakować go w React.memo w celu poprawy wydajności. Oznacza to, że React pominie renderowanie tego komponentu i użyje jego ostatnio wyrenderowanej wersji.

React.memo reaguje tylko na zmiany we właściwościach. Jeśli twój komponent funkcyjny opakowany w React.memo używa hooków useState, useReducer lub useContext, nadal będzie się aktualizował przy zmianie stanu komponentu lub kontekstu.

Domyślnie, komponent wykona jedynie płytkie porównanie obiektów przekazanych we właściwościach. Jeśli chcesz zastosować własny mechanizm porównujący, możesz przekazać odpowiednią funkcję jako drugi argument.

```
function MyComponent(props) {
  /* renderuj korzystając z właściwości */
}
function areEqual(prevProps, nextProps) {
  /*
    zwróć true, jeśli przekazując nextProps, komponent zwróciłby
    taki sam rezultat, jak po przekazaniu prevProps;
    w innym przypadku zwróć false
  */
}
export default React.memo(MyComponent, areEqual);
```

Ta metoda wykorzystywana jest jedynie do **optymalizacji wydajności**. Nie powinno się używać jej do "zapobiegania" renderowaniu, ponieważ może to doprowadzić do większej ilości błędów.

Uwaga

W odróżnieniu od metody shouldComponentUpdate() w komponencie klasowym, funkcja areEqual zwraca true, jeśli właściwości są jednakowe, lub false, jeśli nie są. Jest to odwrócona logika metody shouldComponentUpdate.

createElement() {#createelement}

```
React.createElement(
  type,
  [props],
  [...children]
)
```

Tworzy i zwraca nowy element reaktywy danego typu. Argument `type` może być zarówno nazwą znacznika HTML (np. `'div'` lub `'span'`), komponentem reaktowym (klassą lub funkcją), jak i fragmentem reaktowym.

Kod pisany w JSX jest konwertowany do wywołań funkcji `React.createElement()`. Jeśli korzystasz z JSX, w większości przypadków nie będziesz bezpośrednio wywoływać `React.createElement()`. Przeczytaj rozdział pt. "React bez JSX", aby dowiedzieć się więcej.

`cloneElement()` [#cloneelement](#)

```
React.cloneElement(
  element,
  [config],
  [...children]
)
```

Zacznij od sklonowania i zwrócenia elementu reaktywego przy pomocy argumentu `element`. `config` powinien zawierać wszystkie nowe właściwości, `key` lub `ref`. Nowo utworzony element będzie posiadał pierwotne właściwości scalone płytko z nowymi. Nowe elementy potomne zastąpią obecne. `key` oraz `ref` z pierwotnego elementu zostaną zachowane, jeśli nie przekażesz ich w `config`.

Wywołanie `React.cloneElement()` jest niemal równoznaczne z napisaniem:

```
<element.type {...element.props} {...props}>{children}</element.type>
```

Jednakże zachowuje on także wszystkie referencje `ref`. Oznacza to, że jeśli jeden z komponentów potomnych posiada `ref`, nie zostanie on przypadkowo zabrany z jednego z przodków. Nowy element będzie posiadał ten sam `ref` co przed klonowaniem. Nowy `ref` lub `key` zastąpią poprzednie, jeśli je przekażesz.

Powyższy interfejs API zastąpił przestarzałą funkcję `React.addons.cloneWithProps()`.

`createFactory()` [#createfactory](#)

```
React.createFactory(type)
```

Zwraca funkcję, która tworzy reaktywy element danego typu. Tak jak `React.createElement()`, argument `type` może być zarówno nazwą znacznika HTML (np. `'div'` lub `'span'`), komponentem reaktowym (klassą lub funkcją) lub fragmentem reaktowym.

Ta funkcja pomocnicza jest uznawana za przestarzałą i radzimy korzystać z JSX lub bezpośrednio z `React.createElement()`.

Jeśli korzystasz z JSX, nie ma potrzeby korzystania z `React.createFactory()`. Przeczytaj rozdział pt. "React bez JSX", aby dowiedzieć się więcej.

`isValidElement()` **#isvalidelement}**

`React.isValidElement(object)`

Weryfikuje, czy obiekt jest elementem reactowym. Zwraca `true` lub `false`.

`React.Children` **#reactchildren}**

`React.Children` udostępnia narzędzia do obsługi struktury danych przekazanej w `this.props.children`.

`React.Children.map` **#reactchildrenmap}**

`React.Children.map(children, function[(thisArg)])`

Wywołuje funkcję na każdym bezpośrednim komponencie potomnym zawartym w `children`, z `this` ustalonym na `thisArg`. Jeśli `children` jest tablicą, funkcja zostanie wywołana na każdym elemencie tej tablicy. Jeśli potomek jest wartością `null` lub `undefined`, metoda ta zamiast tablicy zwróci `null` lub `undefined`.

Uwaga

Jeśli `children` jest typu `Fragment`, zostanie potraktowany jako pojedynczy potomek, nie jak tablica.

`React.Children.forEach` **#reactchildrenforeach}**

`React.Children.forEach(children, function[(thisArg)])`

Działa tak samo jak `React.Children.map()`, ale nie zwraca tablicy.

`React.Children.count` **#reactchildrencount}**

`React.Children.count(children)`

Zwraca liczbę komponentów w `children`, równą liczbie potencjalnych wywołań funkcji zwrotnej (ang. *callback*) przekazanej do `map` lub `forEach`.

`React.Children.only` **#reactchildrenonly}**

`React.Children.only(children)`

Weryfikuje, czy `children` ma tylko jednego potomka (element reactowy), i zwraca go. W innym przypadku metoda rzuci wyjątkiem.

Uwaga:

`React.Children.only()` nie akceptuje wartości zwracanej przez `React.Children.map()`, ponieważ jest to tablica, a nie element reactowy.

React.Children.toArray {#reactchildrentoArray}

React.Children.toArray(children)

Zwraca strukturę z `children` jako płaską tablicę, z kluczem przypisanym do każdego z potomków. Przydatne, jeśli chcemy manipulować grupą potomków w metodzie renderującej, zwłaszcza jeśli chcemy zmienić ich kolejność lub podzielić przed przekazaniem dalej.

Uwaga:

`React.Children.toArray()` zmienia klucze, by zachować semantyczną poprawność zagłębionych tablic podczas przygotowywania płaskiej struktury. Oznacza to, że `toArray` doda prefiks do nazwy każdego klucza, tak by każdy element należał do odpowiedniego zakresu.

React.Fragment {#reactfragment}

Komponent `React.Fragment` pozwala zwrócić wiele elementów w metodzie `render()` bez opakowywania ich w dodatkowy element DOM:

```
render() {
  return (
    <React.Fragment>
      Jakiś tekst.
      <h2>Nagłówek</h2>
    </React.Fragment>
  );
}
```

Możesz także skorzystać ze skróconego zapisu `<></>`. Po więcej informacji zajrzyj do wpisu na blogu - [React v16.2.0: Improved Support for Fragments](#).

React.createRef {#reactcreateref}

`React.createRef` tworzy `referencję`, którą możesz przypiąć do dowolnego elementu reactowego poprzez właściwość `ref`. [embed:16-3-release-blog-post/create-ref-example.js](#)

React.forwardRef {#reactforwardref}

`React.forwardRef` tworzy komponent reactowy, który przekazuje właściwość `ref` do kolejnego komponentu w dół drzewa. Ta technika nie jest zbyt popularna, ale przydatna w dwóch konkretnych przypadkach:

- [Przekazywanie referencji do komponentów DOM](#)
- [Przekazywanie referencji w komponentach wyższego rzędu](#)

`React.forwardRef` przyjmuje funkcję renderującą jako argument. React wywoła tę funkcję, przekazując `props` oraz `ref` jako argumenty. Funkcja ta powinna zwrócić węzeł reaktywy (ang. *React node*).

`embed:reference-react-forward-ref.js`

W powyższym przykładzie React przekazuje `ref` poprzez `<FancyButton ref={ref}>` jako drugi argument funkcji renderującej wewnątrz `React.forwardRef`. Funkcja renderująca przekaże `ref` do elementu `<button ref={ref}>`.

W rezultacie, po tym, jak referencja zostanie przypięta przez Reacta, `ref.current` będzie wskazywał bezpośrednio na element `<button>`.

Aby dowiedzieć się więcej, przeczytaj rozdział poświęcony [przekazywaniu referencji](#).

React.lazy {#reactlazy}

`React.lazy()` pozwala definiować komponenty, które są ładowane dynamicznie. Zmniejsza się w ten sposób rozmiar paczki, ponieważ wyodrębnione zostają komponenty, które nie są używane podczas wstępnego renderowania.

W naszej [dokumentacji poświęconej rozdzielaniu kodu](#) zamieściliśmy więcej informacji na ten temat. [Ten artykuł](#) również może okazać się pomocny w wyjaśnieniu zasad działania tej metody.

```
// Ten komponent jest ładowany dynamicznie
const SomeComponent = React.lazy(() => import('./SomeComponent'));
```

Pamiętaj, że renderowanie "leniwych komponentów" (ang. *lazy components*) wymaga użycia komponentu `<React.Suspense>` na wyższym poziomie drzewa. W ten sposób definiuje się wskaźnik ładowania.

React.Suspense {#reactsuspense}

`React.Suspense` pozwala zdefiniować wskaźnik ładowania, w razie gdyby któryś z komponentów poniżej nie był jeszcze gotowy do wyrenderowania. W przyszłości planujemy dodanie do `Suspense` obsługi większej liczby scenariuszy, jak np. pobieranie danych. Możesz poczytać o tym więcej w [naszym planie działania](#).

Obecnie **jedynym** przypadkiem użycia `<React.Suspense>` jest dynamiczne ładowanie komponentów.

```
// Ten komponent jest ładowany dynamicznie
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    // Wyświetla <Spinner> dopóki OtherComponent nie zostanie załadowany
    <React.Suspense fallback={<Spinner />}>
      <div>
        <OtherComponent />
      </div>
    </React.Suspense>
  );
}
```

W naszej [dokumentacji poświęconej rozdzielaniu kodu](#) zamieściliśmy więcej informacji na ten temat. Zwróć uwagę na to, że komponenty `lazy` mogą być zawarte w drzewie wiele poziomów poniżej `Suspense`. Dobrą praktyką jest umieszczanie `<Suspense>` w miejscu, w którym powinien pojawić się wskaźnik ładowania, natomiast `lazy()` w miejscu, w którym chcesz rozdzielić kod.

Uwaga

Jeśli jakaś treść jest już wyświetlona na ekranie, przełączenie na wskaźnik aktywności może być dezorientujące dla użytkownika. Czasami lepiej jest wyświetlić "stary" interfejs, podczas gdy nowy jest jeszcze przygotowywany. Aby to zrobić, możesz użyć nowego API `startTransition` oraz `useTransition`, oznaczając w ten sposób niektóre aktualizacje jako "tranzycje" i unikając niepotrzebnego mrugnięcia ekranu.

`React.Suspense` w renderowaniu po stronie serwera

Podczas renderowania po stronie serwera granice zawieszenia (ang. *Suspense boundaries*) pozwalają wysłać aplikację w mniejszych kawałkach poprzez zawieszanie (ang. *suspending*) komponentów. Kiedy komponent jest zawieszony, React każe najbliższej granicy `Suspense` wyrenderować swój komponent zastępczy. Jeśli komponent się odwiesi przed wysłaniem kodu do klienta, komponent zastępczy jest wyrzucany, a w jego miejsce trafia właściwa zawartość.

`React.Suspense` podczas hydratacji

Granice zawieszenia (ang. *Suspense boundaries*) są zależne od tego, czy ich granice nadzędne ulegną hydratacji przed nimi, lecz nie muszą czekać na swoje "rodzeństwo", czyli granice na tym samym poziomie. W przypadku wystąpienia jakiegoś zdarzenia na którejś z granic przed jej całkowitąhydratacją, otrzyma ona wyższy priorytet niż pozostałe. [Czytaj więcej](#).

`React.startTransition` {#starttransition}

`React.startTransition(callback)`

`React.startTransition` pozwala oznaczyć aktualizacje stanu wewnątrz jakiejś funkcji jako "tranzycję". Tej funkcji należy używać w miejscach, gdzie nie można skorzystać z `React.useTransition`.

Uwaga:

Aktualizacje w tranzycjach ustępują pierwszeństwa bardziej pilnym aktualizacjom, jak np. kliknięciom na ekranie.

Aktualizacje w tranzycji nie powodują wyświetlenia komponentu zastępczego (ang. *fallback*) przy ponownym zawieszeniu, dzięki czemu użytkownik może kontynuować interakcję ze "starym" interfejsem, dopóki nie zakończy się tranzycja.

`React.startTransition` nie zwraca flagi `isPending`. Aby śledzić status transzycji, użyj hooka `React.useTransition`.

Test Renderer

Importowanie

```
import TestRendererer from 'react-test-renderer'; // ES6
const TestRendererer = require('react-test-renderer'); // ES5 z zainstalowanym npm
```

Ogólne informacje

Paczka ta udostępnia narzędzie, które renderuje komponenty reactowe do czysto javascriptowych obiektów, bez użycia drzewa DOM czy natywnego środowiska mobilnego.

Istotą tej paczki jest łatwość wygenerowania "migawki" (ang. *snapshot*) hierarchii widoków (podobnej do drzewa DOM), wyrenderowanej przez komponent z React DOM lub React Native bez pomocy przeglądarki czy `jsdom`.

Przykład:

```
import TestRendererer from 'react-test-renderer';

function Link(props) {
  return <a href={props.page}>{props.children}</a>;
}

const testRendererer = TestRendererer.create(
  <Link page="https://www.facebook.com/">Facebook</Link>
);

console.log(testRendererer.toJSON());
// { type: 'a',
//   props: { href: 'https://www.facebook.com/' },
//   children: [ 'Facebook' ] }
```

Przy pomocy funkcjonalności biblioteki Jest do generowania snapshotów można automatycznie zapisać do pliku kopię drzewa w formacie JSON, a w teście sprawdzać, czy się ono nie zmieniło ([Więcej informacji na ten temat](#)).

Zwrócone drzewo można również przeszukiwać w celu znalezienia konkretnych węzłów i sprawdzenia ich właściwości.

```
import TestRendererer from 'react-test-renderer';

function MyComponent() {
  return (
    <div>
      <SubComponent foo="bar" />
      <p className="my">Cześć</p>
    </div>
  )
}

function SubComponent() {
```

```

    return (
      <p className="sub">Potomek</p>
    );
}

const testRenderer = TestRenderer.create(<MyComponent />);
const testInstance = testRenderer.root;

expect(testInstance.findByType(SubComponent).props.foo).toBe('bar');
expect(testInstance.findByProps({className:
  "sub"}).children).toEqual(['Potomek']);

```

TestRenderer

- `TestRenderer.create()`
- `TestRenderer.act()`

Instancja TestRenderer

- `testRenderer.toJSON()`
- `testRenderer.toTree()`
- `testRenderer.update()`
- `testRenderer.unmount()`
- `testRenderer.getInstance()`
- `testRenderer.root`

TestInstance

- `testInstance.find()`
- `testInstance.findByType()`
- `testInstance.findByProps()`
- `testInstance.findAll()`
- `testInstance.findAllByType()`
- `testInstance.findAllByProps()`
- `testInstance.instance`
- `testInstance.type`
- `testInstance.props`
- `testInstance.parent`
- `testInstance.children`

Dokumentacja

`TestRenderer.create()` **#testrenderercreate**

`TestRenderer.create(element, options);`

Tworzy instancję `TestRenderer` przy użyciu przekazanego elementu reactowego. Nie korzysta z prawdziwego drzewa DOM, lecz renderuje całe drzewo komponentów do pamięci, aby można było wykonać na nim asercje. Zwraca `instancję TestRenderera`.

TestRenderer.act() **{#testrendereract}**

`TestRenderer.act(callback);`

Podobnie jak funkcja pomocnicza `act()` z `react-dom/test-utils`, `TestRenderer.act` przygotowuje komponent do późniejszych asercji. Używaj tej wersji funkcji `act()` do opakowania wywołań `TestRenderer.create` i `testRenderer.update`.

```
import {create, act} from 'react-test-renderer';
import App from './app.js'; // Testowany komponent

// wyrenderuj komponent
let root;
act(() => {
  root = create(<App value={1}>)
});

// wykonaj sprawdzenia na korzeniu drzewa
expect(root.toJSON()).toMatchSnapshot();

// zaktualizuj komponent przy użyciu innych właściwości
act(() => {
  root.update(<App value={2}>);
})

// wykonaj sprawdzenia na korzeniu drzewa
expect(root.toJSON()).toMatchSnapshot();
```

testRenderer.toJSON() **{#testrenderertojson}**

`testRenderer.toJSON()`

Zwraca obiekt reprezentujący wyrenderowane drzewo. W drzewie znajdą się wyłącznie węzły specyficzne dla platformy, np. `<div>` lub `<View>`, wraz ze swoimi właściwościami. Nie zostaną wyrenderowane natomiast żadne niestandardowe komponenty użytkownika. Funkcja przydaje się przy [testowaniu za pomocą snapshotów](#).

testRenderer.toTree() **{#testrenderertotree}**

`testRenderer.toTree()`

Zwraca obiekt reprezentujący wyrenderowane drzewo. W przeciwieństwie do `toJSON()`, ta reprezentacja jest bardziej szczegółowa i zawiera również niestandardowe komponenty użytkownika. Prawdopodobnie ta funkcja nigdy ci się nie przyda, chyba że piszesz w oparciu o tę paczkę własną bibliotekę do testów.

testRenderer.update() **{#testrendererupdate}**

`testRenderer.update(element)`

Przy użyciu nowego elementu głównego ponownie renderuje drzewo przechowywane w pamięci. Symuluje aktualizację przeprowadzaną przez Reacta na korzeniu drzewa. Jeśli nowy element ma ten sam typ i klucz, co poprzedni, drzewo zostanie zaktualizowane; w przeciwnym wypadku drzewo zostanie odmontowane i zamontowane ponownie.

`testRenderer.unmount()` **{#testrendererunmount}**

```
testRenderer.unmount()
```

Odmontowuje drzewo przechowywane w pamięci, wywołując odpowiednie zdarzenia z cyklu życia komponentów.

`testRenderer.getInstance()` **{#testrenderergetinstance}**

```
testRenderer.getInstance()
```

Zwraca instancję korzenia drzewa, jeśli takowy istnieje. Nie zadziała, jeśli komponent główny jest funkcja, ponieważ funkcje nie mają własnych instancji.

`testRenderer.root` **{#testrendererroot}**

```
testRenderer.root
```

Zwraca "instancję testową" korzenia drzewa. Przydatne do wykonywania asercji na poszczególnych węzłach drzewa, a także do wyszukiwania innych "instancji testowych" w poddrzewach.

`testInstance.find()` **{#testinstancefind}**

```
testInstance.find(test)
```

Wyszukuje w poddrzewie dokładnie jedną instancję testową, dla której wywołanie `test(testInstance)` zwróci `true`. Jeśli funkcja `test(testInstance)` nie zwróci `true` dla dokładnie jednej instancji, rzucony zostanie wyjątek.

`testInstance.findByType()` **{#testinstancefindbytype}**

```
testInstance.findByType(type)
```

Wyszukuje w poddrzewie dokładnie jedną instancję testową o podanym typie `type`. Jeśli funkcja znajdzie inną liczbę instancji, rzucony zostanie wyjątek.

`testInstance.findByProps()` **{#testinstancefindbyprops}**

```
testInstance.findByProps(props)
```

Wyszukuje w poddrzewie dokładnie jedną instancję testową o podanych właściwościach `props`. Jeśli funkcja znajdzie inną liczbę instancji, rzucony zostanie wyjątek.

testInstance.findAll() {#testinstancefindall}

```
testInstance.findAll(test)
```

Wyszukuje w poddrzewie wszystkie instancje testowe, dla których funkcja `test(testInstance)` zwraca `true`.

testInstance.findAllByType() {#testinstancefindallbytype}

```
testInstance.findAllByType(type)
```

Wyszukuje w poddrzewie wszystkie instancje testowe o podanym typie `type`.

testInstance.findAllByProps() {#testinstancefindallbyprops}

```
testInstance.findAllByProps(props)
```

Wyszukuje w poddrzewie wszystkie instancje testowe o podanych właściwościach `props`.

testInstance.instance {#testinstanceinstance}

```
testInstance.instance
```

Zwraca instancję komponentu powiązanego z daną instancją testową. Działa tylko dla komponentów klasowych, ponieważ funkcyjne nie mają instancji. Wynik jest referencją odpowiadającą `this` w danym komponencie.

testInstance.type {#testinstancetype}

```
testInstance.type
```

Zwraca typ komponentu powiązanego z daną instancją testową. Przykładowo, typem komponentu `<Button />` jest `Button`.

testInstance.props {#testinstanceprops}

```
testInstance.props
```

Zwraca atrybuty powiązane z daną instancją testową. Przykładowo, właściwościami komponentu `<Button size="small" />` są: `{size: 'small'}`.

testInstance.parent {#testinstanceparent}

```
testInstance.parent
```

Zwraca instancję rodzica dla danej instancji testowej.

testInstance.children {#testinstancechildren}

```
testInstance.children
```

Zwraca instancje potomków dla danej "instancji testowej".

Pomysły

Do metody `TestRenderer.create` jako argument można przekazać funkcję `createNodeMock`, która pozwala na tworzenie własnych atrap referencji (ang. *mock refs*). `createNodeMock` jako argument przyjmuje element, a zwraca obiekt będący atrapą referencji. Przydaje się to w testach komponentów, które korzystają z referencji (ang. *refs*) do innych komponentów.

```
import TestRenderer from 'react-test-renderer';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.input = null;
  }
  componentDidMount() {
    this.input.focus();
  }
  render() {
    return <input type="text" ref={el => this.input = el} />
  }
}

let focused = false;
TestRenderer.create(
  <MyComponent />,
  {
    createNodeMock: (element) => {
      if (element.type === 'input') {
        // atrapa funkcji "focus"
        return {
          focus: () => {
            focused = true;
          }
        };
      }
      return null;
    }
);
expect(focused).toBe(true);
```

Refs and the DOM

Refs provide a way to access DOM nodes or React elements created in the render method.

In the typical React dataflow, [props](#) are the only way that parent components interact with their children. To modify a child, you re-render it with new props. However, there are a few cases where you need to imperatively modify a child outside of the typical dataflow. The child to be modified could be an instance of a React component, or it could be a DOM element. For both of these cases, React provides an escape hatch.

When to Use Refs

There are a few good use cases for refs:

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

Avoid using refs for anything that can be done declaratively.

For example, instead of exposing `open()` and `close()` methods on a `Dialog` component, pass an `isOpen` prop to it.

Don't Overuse Refs

Your first inclination may be to use refs to "make things happen" in your app. If this is the case, take a moment and think more critically about where state should be owned in the component hierarchy. Often, it becomes clear that the proper place to "own" that state is at a higher level in the hierarchy. See the [Lifting State Up](#) guide for examples of this.

Note

The examples below have been updated to use the `React.createRef()` API introduced in React 16.3. If you are using an earlier release of React, we recommend using [callback refs](#) instead.

Creating Refs

Refs are created using `React.createRef()` and attached to React elements via the `ref` attribute. Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
```

```

        return <div ref={this.myRef} />;
    }
}
```

Accessing Refs

When a ref is passed to an element in `render`, a reference to the node becomes accessible at the `current` attribute of the ref.

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

The value of the ref differs depending on the type of the node:

- When the `ref` attribute is used on an HTML element, the `ref` created in the constructor with `React.createRef()` receives the underlying DOM element as its `current` property.
- When the `ref` attribute is used on a custom class component, the `ref` object receives the mounted instance of the component as its `current`.
- **You may not use the `ref` attribute on function components** because they don't have instances.

The examples below demonstrate the differences.

Adding a Ref to a DOM Element

This code uses a `ref` to store a reference to a DOM node:

```

class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // create a ref to store the textInput DOM element
    this.textInput = React.createRef();
    this.focusTextInput = this.focusTextInput.bind(this);
  }

  focusTextInput() {
    // Explicitly focus the text input using the raw DOM API
    // Note: we're accessing "current" to get the DOM node
    this.textInput.current.focus();
  }

  render() {
    // tell React that we want to associate the <input> ref
    // with the `textInput` that we created in the constructor
    return (
      <div>
        <input
          type="text"
          ref={this.textInput} />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

React will assign the `current` property with the DOM element when the component mounts, and assign it back to `null` when it unmounts. `ref` updates happen before `componentDidMount` or `componentDidUpdate` lifecycle methods.

Adding a Ref to a Class Component

If we wanted to wrap the `CustomTextInput` above to simulate it being clicked immediately after mounting, we could use a ref to get access to the custom input and call its `focusTextInput` method manually:

```
class AutoFocusTextInput extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  componentDidMount() {
    this.textInput.current.focusTextInput();
  }

  render() {
    return (
      <CustomTextInput ref={this.textInput} />
    );
  }
}
```

Note that this only works if `CustomTextInput` is declared as a class:

```
class CustomTextInput extends React.Component {
  // ...
}
```

Refs and Function Components

By default, **you may not use the `ref` attribute on function components** because they don't have instances:

```
function MyFunctionComponent() {
  return <input />;
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  render() {
    // This will *not* work!
    return (
      <MyFunctionComponent ref={this.textInput} />
    );
  }
}
```

If you want to allow people to take a `ref` to your function component, you can use `forwardRef` (possibly in conjunction with `useImperativeHandle`), or you can convert the component to a class.

You can, however, **use the `ref` attribute inside a function component** as long as you refer to a DOM element or a class component:

```
function CustomTextInput(props) {
  //textInput must be declared here so the ref can refer to it
  const textInput = useRef(null);

  function handleClick() {
    textInput.current.focus();
  }

  return (
    <div>
      <input
        type="text"
        ref={textInput} />
      <input
        type="button"
        value="Focus the text input"
        onClick={handleClick}
      />
    </div>
  );
}
```

Exposing DOM Refs to Parent Components

In rare cases, you might want to have access to a child's DOM node from a parent component. This is generally not recommended because it breaks component encapsulation, but it can occasionally be useful for triggering focus or measuring the size or position of a child DOM node.

While you could [add a ref to the child component](#), this is not an ideal solution, as you would only get a component instance rather than a DOM node. Additionally, this wouldn't work with function components.

If you use React 16.3 or higher, we recommend to use [ref forwarding](#) for these cases. **Ref forwarding lets components opt into exposing any child component's ref as their own**. You can find a detailed example of how to expose a child's DOM node to a parent component [in the ref forwarding documentation](#).

If you use React 16.2 or lower, or if you need more flexibility than provided by ref forwarding, you can use [this alternative approach](#) and explicitly pass a ref as a differently named prop.

When possible, we advise against exposing DOM nodes, but it can be a useful escape hatch. Note that this approach requires you to add some code to the child component. If you have absolutely no control over the child component implementation, your last option is to use `findDOMNode()`, but it is discouraged and deprecated in `StrictMode`.

Callback Refs

React also supports another way to set refs called "callback refs", which gives more fine-grain control over when refs are set and unset.

Instead of passing a `ref` attribute created by `createRef()`, you pass a function. The function receives the React component instance or HTML DOM element as its argument, which can be stored and accessed elsewhere.

The example below implements a common pattern: using the `ref` callback to store a reference to a DOM node in an instance property.

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

    this.textInput = null;

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

    this.focusTextInput = () => {
      // Focus the text input using the raw DOM API
      if (this.textInput) this.textInput.focus();
    };
  }

  componentDidMount() {
    // autofocus the input on mount
    this.focusTextInput();
  }

  render() {
    // Use the `ref` callback to store a reference to the text input DOM
    // element in an instance field (for example, this.textInput).
    return (
      <div>
        <input
          type="text"
          ref={this.setTextInputRef}
        />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

React will call the `ref` callback with the DOM element when the component mounts, and call it with `null` when it unmounts. Refs are guaranteed to be up-to-date before `componentDidMount` or `componentDidUpdate` fires.

You can pass callback refs between components like you can with object refs that were created with `React.createRef()`.

```
function CustomTextInput(props) {
  return (
    <div>
```

```

        <input ref={props.inputRef} />
      </div>
    );
}

class Parent extends React.Component {
  render() {
    return (
      <CustomTextInput
        inputRef={el => this.inputElement = el}
      />
    );
  }
}

```

In the example above, `Parent` passes its ref callback as an `inputRef` prop to the `CustomTextInput`, and the `CustomTextInput` passes the same function as a special `ref` attribute to the `<input>`. As a result, `this.inputElement` in `Parent` will be set to the DOM node corresponding to the `<input>` element in the `CustomTextInput`.

Legacy API: String Refs

If you worked with React before, you might be familiar with an older API where the `ref` attribute is a string, like `"textInput"`, and the DOM node is accessed as `this.refs.textInput`. We advise against it because string refs have [some issues](#), are considered legacy, and **are likely to be removed in one of the future releases**.

Note

If you're currently using `this.refs.textInput` to access refs, we recommend using either the [callback pattern](#) or the [createRef API](#) instead.

Caveats with callback refs

If the `ref` callback is defined as an inline function, it will get called twice during updates, first with `null` and then again with the DOM element. This is because a new instance of the function is created with each render, so React needs to clear the old ref and set up the new one. You can avoid this by defining the `ref` callback as a bound method on the class, but note that it shouldn't matter in most cases.

Release Channels

React relies on a thriving open source community to file bug reports, open pull requests, and [submit RFCs](#). To encourage feedback we sometimes share special builds of React that include unreleased features.

This document will be most relevant to developers who work on frameworks, libraries, or developer tooling. Developers who use React primarily to build user-facing applications should not need to worry about our prerelease channels.

Each of React's release channels is designed for a distinct use case:

- **Latest** is for stable, semver React releases. It's what you get when you install React from npm. This is the channel you're already using today. **Use this for all user-facing React applications.**
- **Next** tracks the main branch of the React source code repository. Think of these as release candidates for the next minor semver release. Use this for integration testing between React and third party projects.
- **Experimental** includes experimental APIs and features that aren't available in the stable releases. These also track the main branch, but with additional feature flags turned on. Use this to try out upcoming features before they are released.

All releases are published to npm, but only Latest uses [semantic versioning](#). Prereleases (those in the Next and Experimental channels) have versions generated from a hash of their contents and the commit date, e.g. `0.0.0-68053d940-20210623` for Next and `0.0.0-experimental-68053d940-20210623` for Experimental.

The only officially supported release channel for user-facing applications is Latest. Next and Experimental releases are provided for testing purposes only, and we provide no guarantees that behavior won't change between releases. They do not follow the semver protocol that we use for releases from Latest.

By publishing prereleases to the same registry that we use for stable releases, we are able to take advantage of the many tools that support the npm workflow, like [unpkg](#) and [CodeSandbox](#).

Latest Channel

Latest is the channel used for stable React releases. It corresponds to the `latest` tag on npm. It is the recommended channel for all React apps that are shipped to real users.

If you're not sure which channel you should use, it's Latest. If you're a React developer, this is what you're already using.

You can expect updates to Latest to be extremely stable. Versions follow the semantic versioning scheme. Learn more about our commitment to stability and incremental migration in our [versioning policy](#).

Next Channel

The Next channel is a prerelease channel that tracks the main branch of the React repository. We use prereleases in the Next channel as release candidates for the Latest channel. You can think of Next as a superset of Latest that is updated more frequently.

The degree of change between the most recent Next release and the most recent Latest release is approximately the same as you would find between two minor semver releases. However, **the Next channel does not conform to semantic versioning**. You should expect occasional breaking changes between successive releases in the Next channel.

Do not use prereleases in user-facing applications.

Releases in Next are published with the `next` tag on npm. Versions are generated from a hash of the build's contents and the commit date, e.g. `0.0.0-68053d940-20210623`.

Using the Next Channel for Integration Testing

The Next channel is designed to support integration testing between React and other projects.

All changes to React go through extensive internal testing before they are released to the public. However, there are a myriad of environments and configurations used throughout the React ecosystem, and it's not possible for us to test against every single one.

If you're the author of a third party React framework, library, developer tool, or similar infrastructure-type project, you can help us keep React stable for your users and the entire React community by periodically running your test suite against the most recent changes. If you're interested, follow these steps:

- Set up a cron job using your preferred continuous integration platform. Cron jobs are supported by both [CircleCI](#) and [Travis CI](#).
- In the cron job, update your React packages to the most recent React release in the Next channel, using `next` tag on npm. Using the npm cli:

```
npm update react@next react-dom@next
```

Or yarn:

```
yarn upgrade react@next react-dom@next
```

- Run your test suite against the updated packages.
- If everything passes, great! You can expect that your project will work with the next minor React release.
- If something breaks unexpectedly, please let us know by [filing an issue](#).

A project that uses this workflow is Next.js. (No pun intended! Seriously!) You can refer to their [CircleCI configuration](#) as an example.

Experimental Channel

Like Next, the Experimental channel is a prerelease channel that tracks the main branch of the React repository. Unlike Next, Experimental releases include additional features and APIs that are not ready for wider release.

Usually, an update to Next is accompanied by a corresponding update to Experimental. They are based on the same source revision, but are built using a different set of feature flags.

Experimental releases may be significantly different than releases to Next and Latest. **Do not use Experimental releases in user-facing applications.** You should expect frequent breaking changes between releases in the Experimental channel.

Releases in Experimental are published with the `experimental` tag on npm. Versions are generated from a hash of the build's contents and the commit date, e.g. `0.0.0-experimental-68053d940-20210623`.

What Goes Into an Experimental Release?

Experimental features are ones that are not ready to be released to the wider public, and may change drastically before they are finalized. Some experiments may never be finalized -- the reason we have experiments is to test the viability of proposed changes.

For example, if the Experimental channel had existed when we announced Hooks, we would have released Hooks to the Experimental channel weeks before they were available in Latest.

You may find it valuable to run integration tests against Experimental. This is up to you. However, be advised that Experimental is even less stable than Next. **We do not guarantee any stability between Experimental releases.**

How Can I Learn More About Experimental Features?

Experimental features may or may not be documented. Usually, experiments aren't documented until they are close to shipping in Next or Latest.

If a feature is not documented, they may be accompanied by an [RFC](#).

We will post to the [React blog](#) when we're ready to announce new experiments, but that doesn't mean we will publicize every experiment.

You can always refer to our public GitHub repository's [history](#) for a comprehensive list of changes.

Właściwość renderująca

Określenie "właściwość renderująca" odnosi się do techniki współdzielenia kodu pomiędzy komponentami reaktywnymi, przy użyciu właściwości, której wartością jest funkcja.

Komponent z właściwością renderującą przyjmuje funkcję, zwracającą element reactowy, a następnie wywołuje ją, nie implementując jednocześnie logiki renderowania tego komponentu.

```
<DataProvider render={data => (
  <h1>Cześć, {data.target}</h1>
)} />
```

Biblioteki używające właściwości renderujących to m.in. [React Router](#), [Downshift](#) oraz [Formik](#).

W tym dokumencie przedyskutujemy przydatność właściwości renderujących i dowiemy się, jak napisać je samodzielnie.

Użycie właściwości renderujących przy zagadnieniach przekrojowych (ang. *cross-cutting concerns*)

Komponenty są podstawowym sposobem na wielokrotne używanie funkcjonalności w Reakcie. Nie zawsze jednak jest oczywiste, jak należy współdzielić stan czy zachowanie zawarte w jednym komponencie, tak aby miały do nich dostęp inne komponenty.

Za przykład niech posłuży komponent śledzący pozycję kurSORA w aplikacji internetowej:

```
class MouseTracker extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        <h1>Porusz myszką!</h1>
        <p>Aktualna pozycja kurSORA to ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}
```

Podczas gdy kurSOR przemieszcza się po ekranie, komponent wyświetla jego koordynaty (x, y) wewnątrz znacznika `<p>`.

Pytanie brzmi: jak użyć ponownie tego zachowania w innym komponencie? Innymi słowy, jeżeli inny komponent potrzebuje współrzędnych pozycji kurSora, czy możemy zaimplementować to zachowanie tak, abyśmy mogli łatwo współdzielić je z tym komponentem?

Ponieważ komponenty są podstawowym sposobem na wielokrotne używanie funkcjonalności w Reakcie, spróbujmy przekształcić trochę powyższy kod tak, aby wykorzystać komponent `<Mouse>` zawierający w sobie zachowanie wymagane przez inny komponent.

```
// Komponent <Mouse> posiada funkcjonalność, której potrzebujemy...
class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        {/* ...ale jak wyrenderować coś innego niż <p>? */}
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <>
        <h1>Porusz myszką!</h1>
        <Mouse />
      </>
    );
  }
}
```

Teraz komponent `<Mouse>` implementuje wszystkie zachowania związane z nasłuchiwaniem na zdarzenie `mousemove` i przechowuje pozycję kurSora (x, y), jednak nie jest jeszcze w pełni reużywalny.

Dla przykładu, powiedzmy że mamy komponent `<Cat>`, który renderuje obrazek kota goniącego kurSOR na ekranie. Moglibyśmy użyć właściwości `<Cat mouse={{ x, y }}>`, aby przekazać komponentowi koordynaty kurSora, tak aby wiedział, gdzie umieścić obrazek kota.

Początkowo możesz chcieć wyrenderować komponent `<Cat>` wewnątrz metody `render` komponentu `<Mouse>`, na przykład w ten sposób:

```

class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}

class MouseWithCat extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        /*
          Moglibyśmy w tym miejscu po prostu zamielić <p> na <Cat>... ale później
          musielibyśmy stworzyć oddzielnny komponent <MouseWithSomethingElse>
          dla każdego przypadku użycia, dlatego też <MouseWithCat>
          nie jest jeszcze w pełni używalny wielokrotnie.
        */
        <Cat mouse={this.state} />
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Porusz myszką!</h1>
        <MouseWithCat />
      </div>
    );
  }
}

```

To podejście zadziałałoby w naszym specyficznym przypadku, ale nie osiągneliśmy prawdziwego celu, jakim jest hermetyzacja zachowania w taki sposób, aby można było je wykorzystywać wielokrotnie. W takim przypadku, za każdym razem gdy potrzebowalibyśmy inaczej wyświetlić coś na podstawie pozycji kurSORA, musielibyśmy stworzyć nowy komponent (innymi słowy, kolejny `<MouseWithCat>`), który wyrenderuje coś w odpowiedni sposób.

I tutaj do akcji wkracza właściwość renderująca. Zamiast na stałe osadzać `<Cat>` wewnątrz komponentu `<Mouse>`, jednocześnie zmieniając wyrenderowany przez niego wynik, możemy dostarczyć do komponentu `<Mouse>` właściwość będącą funkcją. Funkcja ta w sposób dynamiczny będzie określać, co powinno zostać wyrenderowane.

```

class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}

class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        /*
          Zamiast używać statycznej reprezentacji tego co renderuje <Mouse>,
          użyj właściwości `render`, aby dynamicznie określić, co należy
        wyrenderować.
        */
        {this.props.render(this.state)}
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Porusz myszką!</h1>
        <Mouse render={mouse => (
          <Cat mouse={mouse} />
        )}/>
      </div>
    );
  }
}

```

Teraz, zamiast faktycznie klonować komponent `<Mouse>` i na stałe wpisywać coś innego w jego metodzie `render` dla poszczególnych przypadków, dostarczamy właściwość `render`, której komponent `<Mouse>` może użyć do dynamicznego określenia renderowanego wyniku.

Konkretnie rzecz ujmując, **właściwość renderująca jest funkcją, której komponent używa, aby wiezieć, co ma wyrenderować.**

Ta technika powoduje, że zachowanie, które chcemy współdzielić, staje się niezwykle przenośne. Aby uzyskać zamierzony efekt, wyrenderuj komponent `<Mouse>` wraz z właściwością `render`, która określi, co powinno zostać wyrenderowane biorąc pod uwagę aktualną pozycję (x, y) kurSORA myszy.

Ciekawostką, o której warto wspomnieć podczas opisywania właściwości renderujących, jest to, że większość komponentów wyższego rzędu (ang. *higher-order components, HOC*) można zaimplementować przy użyciu zwykłego komponentu z właściwością renderującą. Dla przykładu, jeżeli chcesz korzystać z komponentu wyższego rzędu `withMouse` zamiast komponentu `<Mouse>`, możesz z łatwością go stworzyć przy użyciu `<Mouse>` z właściwością renderującą:

```
// Jeżeli z jakiegoś powodu naprawdę potrzebujesz HOC-a, możesz go łatwo
// stworzyć używając zwykłego komponentu z właściwością renderującą!
function withMouse(Component) {
  return class extends React.Component {
    render() {
      return (
        <Mouse render={mouse => (
          <Component {...this.props} mouse={mouse} />
        )}/>
      );
    }
  }
}
```

Użycie właściwości renderującej powoduje, że jest możliwe użycie obu wzorców.

Używanie właściwości innych niż `render` {#using-props-other-than-render}

Pamiętaj, że pomimo iż wzorzec nosi nazwę "właściwości renderującej" (ang. *render prop*), nie musisz używać właściwości o nazwie `render`, aby go zastosować. W zasadzie *każda właściwość, która jest funkcją służącą do określenia, co powinno zostać wyrenderowane, jest technicznie rzecz ujmując "właściwością renderującą"*.

Pomimo że powyższe przykłady używają nazwy `render`, moglibyśmy po prostu użyć właściwości `children`!

```
<Mouse children={mouse => (
  <p>Pozycja kurSORA to {mouse.x}, {mouse.y}</p>
)} />
```

Pamiętaj, że w JSX właściwości `children` nie musisz przekazywać na liście "atrybutów" twojego elementu. Zamiast tego możesz ją umieścić bezpośrednio *wewnątrz* znacznika!

```
<Mouse>
  {mouse => (
    <p>Pozycja kurSORA to {mouse.x}, {mouse.y}</p>
  )}
</Mouse>
```

Powyższa technika stosowana jest w bibliotece [react-motion](#).

Jako że ta technika wydaje się trochę niecodzienna, podczas projektowania swojego API prawdopodobnie zechcesz jawnie określić w `propTypes` komponentu, że właściwość `children` powinna być funkcją.

```
Mouse.propTypes = {
  children: PropTypes.func.isRequired
};
```

Ostrzeżenia

Uważaj przy stosowaniu właściwości renderującej dla komponentów dziedziczących po React.PureComponent

Jeżeli stworzysz funkcję wewnętrz metodę `render`, użycie właściwości renderującej może zniwelować korzyści płynące z użycia `React.PureComponent`. Dzieje się tak ponieważ płytka porównanie (ang. *shallow comparison*) dla nowych właściwości zawsze będzie zwracać wartość `false`, a w tym przypadku każde wywołanie `render` będzie generować nową wartość dla właściwości renderującej.

Dla przykładu, kontynuując z komponentem `<Mouse>`, jeżeli `Mouse` rozszerzałaby klasę `React.PureComponent` zamiast `React.Component`, nasz przykład wyglądałby tak:

```
class Mouse extends React.PureComponent {
  // Ta sama implementacja co wyżej...
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Porusz myszką!</h1>

        {/* 
          Źle! Wartość właściwości `render` będzie
          inna przy każdym wywołaniu metody render.
        */}
        <Mouse render={mouse => (
          <Cat mouse={mouse} />
        )}>
      </div>
    );
  }
}
```

W tym przykładzie, za każdym razem gdy `<MouseTracker>` jest renderowany, generowana jest nowa funkcja jako wartość dla właściwości `<Mouse render>`. Zatem w pierwszej kolejności neguje to efekt rozszerzania `React.PureComponent` przez komponent `<Mouse>`!

Aby obejść ten problem, możesz zdefiniować właściwość jako metodę instancji klasy:

```
class MouseTracker extends React.Component {  
  // Zdefiniowana jako metoda instancji, `this.renderTheCat` zawsze  
  // odnosi się do *tej samej* funkcji podczas użycia w metodzie render  
  renderTheCat(mouse) {  
    return <Cat mouse={mouse} />;  
  }  
  
  render() {  
    return (  
      <div>  
        <h1>Porusz myszką!</h1>  
        <Mouse render={this.renderTheCat} />  
      </div>  
    );  
  }  
}
```

W przypadkach gdy nie możesz zdefiniować właściwości statycznej (na przykład, dlatego że musisz domknąć właściwości komponentu i/lub jego stan), `<Mouse>` powinien rozszerzać klasę `React.Component`.

Renderowanie elementów

Elementy to najmniejsze bloki budujące reaktywne aplikacje.

Element opisuje, co chcesz zobaczyć na ekranie:

```
const element = <h1>Witaj, świecie!</h1>;
```

W przeciwieństwie do elementów drzewa DOM w przeglądarce, reaktywne elementy są zwykłymi obiektami i mają niski koszt tworzenia. React DOM zajmuje się aktualizowaniem drzewa DOM tak, aby odpowiadało strukturze elementów reaktywnych.

Wskazówka:

Łatwo pomylić elementy z szerzej znaną koncepcją "komponentów". Komponenty przedstawimy w [kolejnym rozdziale](#). Elementy są tym, z czego komponenty "są zbudowane". Zachęcamy do przeczytania tego rozdziału przed przejściem dalej.

Renderowanie elementu w drzewie DOM

Powiedzmy, że gdzieś w twoim pliku HTML jest `<div>`:

```
<div id="root"></div>
```

Ten węzeł drzewa DOM nazywamy "korzeniem", bo wszystko, co się w nim znajduje będzie zarządzane przez React DOM.

Aplikacje zbudowane przy pomocy samego Reacta zazwyczaj posiadają pojedynczy węzeł drzewa DOM. Jeśli natomiast integrujesz reaktywną aplikację z już istniejącą aplikacją, możesz mieć tyle odizolowanych "korzeni", ile chcesz.

Aby wypchnąć reaktywny element w węźle drzewa DOM, najpierw przekaż element DOM do `ReactDOM.createRoot()`, a następnie przekaż element reaktywny do `root.render()`:

```
embed:rendering-elements/render-an-element.js
```

[Przetestuj kod na CodePen](#)

Na stronie wyświetli się napis "Witaj, świecie!".

Aktualizowanie wyrenderowanego elementu

Reaktywne elementy są [niezmienne](#) (ang. *immutable*). Kiedy już stworzysz element, nie możesz zmienić jego komponentów potomnych ani właściwości. Element jest jak pojedyncza klatka z filmu: reprezentuje interfejs użytkownika w pewnym punkcie czasu.

Przy naszej dotychczasowej wiedzy, jedynym sposobem aktualizacji interfejsu użytkownika jest stworzenie nowego elementu i przekazanie go do `root.render()`.

Rozważ ten przykład tykającego zegara:

```
embed:rendering-elements/update-rendered-element.js
```

[Przetestuj kod na CodePen](#)

Wywołuje on `root.render()` z wewnętrz funkcji zwrotnej `setInterval()` co sekundę.

Wskazówka:

W praktyce większość reactowych aplikacji wywołuje `root.render()` tylko raz. W kolejnych rozdziałach dowiemy się, jak można taki kod wyizolować do [komponentów stanowych](#).

Radzimy jednak nie pomijać żadnych tematów, ponieważ kolejne rozdziały oparte są o wiedzę z poprzednich.

React uaktualnia tylko to, co potrzebne

React DOM porównuje element i jego potomków do poprzedniego oraz nakłada tylko te aktualizacje drzewa DOM, które konieczne są do doprowadzenia go do pożądanego stanu.

Możesz to sprawdzić przez zbadanie (ang. *inspect*) [ostatniego przykładu](#) przy użyciu narzędzi deweloperskich:

Hello, world!

It is 12:26:46 PM.

```
Console Sources Network Timeline
▼<div id="root">
  ▼<div data-reactroot>
    <h1>Hello, world!</h1>
    ▼<h2>
      <!-- react-text: 4 -->
      "It is "
      <!-- /react-text -->
      <!-- react-text: 5 -->
      "12:26:46 PM"
      <!-- /react-text -->
      <!-- react-text: 6 -->
      ","
      <!-- /react-text -->
    </h2>
  </div>
</div>
```

inspektor DOM pokazujący cząstkowe aktualizacje

Mimo że przy każdym tyknięciu zegara tworzymy element opisujący cały interfejs użytkownika, tylko węzeł tekstowy, którego treść uległa zmianie, zostaje zaktualizowany przez React DOM.

Według naszego doświadczenia, myślenie o tym, jak powinien wyglądać interfejs użytkownika w danym momencie, a nie jak zmieniać go w czasie, eliminuje całą klasę błędów.

react

[Go to TOC](#)

Stan i cykl życia

W tym poradniku wprowadzimy pojęcie stanu (ang. *state*) i cyklu życia (ang. *lifecycle*) komponentu reactowego. Więcej informacji na ten temat znajdziesz w [szczegółowej dokumentacji API komponentów](#).

Wróćmy do przykładu tykającego zegara z [jednej z poprzednich lekcji](#). W sekcji "Renderowanie elementów" nauczyliśmy się tylko jednego sposobu aktualizowania interfejsu aplikacji. Aby zmienić wynik renderowania, wywołujemy funkcję `root.render()`:

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function tick() {
  const element = (
    <div>
      <h1>Witaj, świecie!</h1>
      <h2>Aktualny czas: {new Date().toLocaleTimeString()}</h2>
    </div>
  );
  root.render(element);
}

setInterval(tick, 1000);
```

Przetestuj kod na CodePen

W tym rozdziale dowiemy się, jak sprawić, by komponent `Clock` był w pełni hermetyczny i zdatny do wielokrotnego użytku. Wypożymy go we własny timer, który będzie aktualizował się co sekundę.

Zacznijmy od wyizolowania kodu, który odpowiada za wygląd zegara:

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function Clock(props) {
  return (
    <div>
      <h1>Witaj, świecie!</h1>
      <h2>Aktualny czas: {props.date.toLocaleTimeString()}</h2>
    </div>
  );
}

function tick() {
  root.render(<Clock date={new Date()} />);
}

setInterval(tick, 1000);
```

Przetestuj kod na CodePen

Brakuje jeszcze fragmentu, który spełniałby kluczowe założenie: inicializacja timera i aktualizowanie UI co sekundę powinny być zaimplementowane w komponencie `Clock`.

Idealnie byłoby móc napisać tylko tyle i oczekiwąć, że `Clock` zajmie się resztą:

```
root.render(<Clock />);
```

Aby tak się stało, musimy dodać do komponentu "stan".

Stan przypomina trochę atrybuty (ang. *props*), jednak jest prywatny i w pełni kontrolowany przez dany komponent.

Przekształcanie funkcji w klasę

Proces przekształcania komponentu funkcyjnego (takiego jak nasz `Clock`) w klasę można opisać w pięciu krokach:

1. Stwórz [klasę zgodną ze standardem ES6](#) o tej samej nazwie i odziedzicz po klasie `React.Component` przy pomocy słowa kluczowego `extend`.
2. Dodaj pustą metodę o nazwie `render()`.
3. Przenieś ciało funkcji do ciała metody `render()`.
4. W `render()` zamień wszystkie `props` na `this.props`.
5. Usuń starą deklarację funkcji.

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Witaj, świecie!</h1>
        <h2>Aktualny czas: {this.props.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

[Przetestuj kod na CodePen](#)

Komponent `Clock` przestał już być funkcją i od teraz jest klasą.

Metoda `render` zostanie automatycznie wywołana przy każdej zmianie. Dopóki będziemy renderować `<Clock />` do tego samego węzła drzewa DOM, dopóty używana będzie jedna i ta sama instancja klasy `Clock`. Pozwala to na skorzystanie z dodatkowych funkcjonalności, takich jak lokalny stan czy metody cyklu życia komponentu.

Dodawanie lokalnego stanu do klasy

Przenieśmy teraz `date` z atrybutów do stanu w trzech krokach:

1. Zamień wystąpienia `this.props.date` na `this.state.date` w ciele metody `render()`:

```
class Clock extends React.Component {
  render() {
    return (
      <div>
```

```

        <div>
          <h1>Witaj, świecie!</h1>
          <h2>Aktualny czas: {this.state.date.toLocaleTimeString()}.</h2>
        </div>
      );
    }
}

```

2. Dodaj [konstruktor klasy](#) i zainicjalizuj w nim pole `this.state`:

```

class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Witaj, świecie!</h1>
        <h2>Aktualny czas: {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

```

Zwróć uwagę na argument `props` przekazywany do konstruktora bazowego za pomocą specjalnej funkcji `super()`:

```

constructor(props) {
  super(props);
  this.state = {date: new Date()};
}

```

Komponenty klasowe zawsze powinny przekazywać `props` do konstruktora bazowego.

3. Usuń atrybut `date` z elementu `<Clock />`:

```
root.render(<Clock />);
```

Timer dodamy do komponentu nieco później.

W rezultacie powinniśmy otrzymać następujący kod:

```

class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Witaj, świecie!</h1>
        <h2>Aktualny czas: {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

```

```

    }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);

```

[Przetestuj kod na CodePen](#)

Teraz sprawimy, by komponent `Clock` uruchomił własny timer i aktualizował go co sekundę.

Dodawanie metod cyklu życia do klasy

W aplikacjach o wielu komponentach istotne jest zwalnianie zasobów przy niszczeniu każdego z komponentów.

Chcielibyśmy [uruchamiać timer](#) przy każdym pierwszym wyrenderowaniu komponentu `Clock` do drzewa DOM. W Reakcie taki moment w cyklu życia komponentu nazywamy "montowaniem" (ang. *mounting*).

Chcemy również [resetować timer](#) za każdym razem, gdy DOM wygenerowany przez `Clock` jest usuwany z dokumentu. W Reakcie taki moment nazywamy to "odmontowaniem" (ang. *unmounting*) komponentu.

W klasie możemy zadeklarować specjalne metody, które będą uruchamiały kod w momencie montowania i odmontowywania komponentu:

```

class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {}

  componentWillUnmount() {}

  render() {
    return (
      <div>
        <h1>Witaj, świecie!</h1>
        <h2>Aktualny czas: {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}

```

Takie metody nazywamy "metodami cyklu życia".

Metoda `componentDidMount()` uruchamiana jest po wyrenderowaniu komponentu do drzewa DOM. To dobre miejsce na inicjalizację timera:

```
componentDidMount() {
  this.timerID = setInterval(
    () => this.tick(),
    1000
  );
}
```

Zwróć uwagę, że identyfikator timera zapisujemy bezpośrednio do `this (this.timerID)`.

Mimo że `this.props` jest ustawiane przez Reacta, a `this.state` jest specjalnym polem, to nic nie stoi na przeszkodzie, aby stworzyć dodatkowe pola, w których chcielibyśmy przechowywać wartości niezwiązane bezpośrednio z przepływem danych (jak nasz identyfikator timera).

Zatrzymaniem timera zajmie się metoda cyklu życia zwana `componentWillUnmount()`:

```
componentWillUnmount() {
  clearInterval(this.timerID);
}
```

Na koniec zaimplementujemy metodę o nazwie `tick()`, którą komponent `Clock` będzie wywoływał co sekundę.

Użyjemy w niej `this.setState()`, aby zaplanować aktualizację lokalnego stanu komponentu:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Witaj, świecie!</h1>
        <h2>Aktualny czas: {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

[Przetestuj kod na CodePen](#)

Teraz timer powinien już tykać co sekundę.

Podsumujmy, co dzieje się w powyższym kodzie i w jakiej kolejności wywoływane są metody:

1. Kiedy element `<Clock />` przekazywany jest do funkcji `root.render()`, React wywołuje konstruktor komponentu `Clock`. Jako że `Clock` będzie wyświetlać aktualny czas, musi on zainicjalizować `this.state` obiektem zawierającym aktualną datę. Później ten stan będzie aktualizowany.
2. Następnie React wywołuje metodę `render()` komponentu `Clock`. W ten sposób uzyskuje informację, co powinno zostać wyświetlone na stronie. Gdy otrzyma odpowiedź, odpowiednio aktualizuje drzewo DOM.
3. Po wyrenderowaniu komponentu `Clock` do drzewa DOM, React wywołuje metodę cyklu życia o nazwie `componentDidMount()`. W jej ciele komponent `Clock` prosi przeglądarkę o zainicjalizowanie nowego timera, który będzie wywoływać metodę `tick()` co sekundę.
4. Co sekundę przeglądarka wywołuje metodę `tick()`. W jej ciele komponent `Clock` żąda aktualizacji UI poprzez wywołanie metody `setState()`, przekazując jako argument obiekt z aktualnym czasem. Dzięki wywołaniu `setState()` React wie, że zmienił się stan i że może ponownie wywołać metodę `render()`, by dowiedzieć się, co powinno zostać wyświetlone na ekranie. Tym razem wartość zmiennej `this.state.date` w ciele metody `render()` będzie inna, odpowiadająca nowemu czasowi - co React odzwierciedli w drzewie DOM.
5. Jeśli kiedykolwiek komponent `Clock` zostanie usunięty z drzewa DOM, React wywoła na nim metodę cyklu życia o nazwie `componentWillUnmount()`, zatrzymując tym samym timer.

Poprawne używanie stanu

Są trzy rzeczy, które musisz wiedzieć o `setState()`.

Nie modyfikuj stanu bezpośrednio

Na przykład, poniższy kod nie spowoduje ponownego wyrenderowania komponentu:

```
// Źle
this.state.comment = 'Witam';
```

Zamiast tego używaj `setState()`:

```
// Dobrze
this.setState({comment: 'Witam'});
```

Jedynym miejscem, w którym wolno Ci użyć `this.state` jest konstruktor klasy.

Aktualizacje stanu mogą działać się asynchroniczne

React może zgrupować kilka wywołań metody `setState()` w jedną paczkę w celu zwiększenia wydajności aplikacji.

Z racji tego, że zmienne `this.props` i `this.state` mogą być aktualizowane asynchronicznie, nie powinno się polegać na ich wartościach przy obliczaniu nowego stanu.

Na przykład, poniższy kod może nadpisać `counter` błędna wartością:

```
// Źle
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

Aby temu zaradzić, wystarczy użyć alternatywnej wersji metody `setState()`, która jako argument przyjmuje funkcję zamiast obiektu. Funkcja ta otrzyma dwa argumenty: aktualny stan oraz aktualne atrybuty komponentu.

```
// Dobrze
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

W powyższym kodzie użyliśmy [funkcji strzałkowej](#), lecz równie dobrze moglibyśmy użyć zwykłej funkcji:

```
// Dobrze
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

Aktualizowany stan jest scalany

Gdy wywołujesz `setState()`, React scala (ang. *merge*) przekazany obiekt z aktualnym stanem komponentu.

Na przykład, gdyby komponent przechowywał w stanie kilka niezależnych zmiennych:

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

można byłoby je zaktualizować niezależnie za pomocą osobnych wywołań metody `setState()`:

```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
});
```

```

    });
});

fetchComments().then(response => {
  this.setState({
    comments: response.comments
  });
}
}

```

Scalanie jest płytkie (ang. *shallow*), tzn. `this.setState({comments})` nie zmieni `this.state.posts`, lecz całkowicie nadpisze wartość `this.state.comments`.

Dane płyną z góry na dół

Ani komponenty-rodzice, ani ich dzieci nie wiedzą, czy jakiś komponent posiada stan, czy też nie. Nie powinny się również przejmować tym, czy jest on funkcyjny, czy klasowy.

Właśnie z tego powodu stan jest nazywany lokalnym lub enkapsulowanym. Nie mają do niego dostępu żadne komponenty poza tym, który go posiada i modyfikuje.

Komponent może zdecydować się na przekazanie swojego stanu w dół struktury poprzez atrybuty jego komponentów potomnych:

```
<FormattedDate date={this.state.date} />
```

Komponent `FormattedDate` otrzyma `date` jako atrybut i nie będzie w stanie rozróżnić, czy pochodzi on ze stanu lub jednego z atrybutów komponentu `Clock`, czy też został przekazany bezpośrednio przez wartość:

```

function FormattedDate(props) {
  return <h2>Aktualny czas: {props.date.toLocaleTimeString()}</h2>;
}

```

[Przetestuj kod na CodePen](#)

Taki przepływ danych nazywany jest powszechnie jednokierunkowym (ang. *unidirectional*) lub "z góry na dół" (ang. *top-down*). Stan jest zawsze własnością konkretnego komponentu i wszelkie dane lub części UI, powstałe w oparciu o niego, mogą wpływać jedynie na komponenty znajdujące się "poniżej" w drzewie.

Wyobraź sobie, że drzewo komponentów to wodospad atrybutów, a stan każdego z komponentów to dodatkowe źródło wody, które go zasila, jednocześnie spadając w dół wraz z resztą wody.

Aby pokazać, że wszystkie komponenty są odizolowane od reszty, stwórzmy komponent `App`, który renderuje trzy elementy `<Clock>`:

```

function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}

```

```
    </div>
  );
}
```

Przetestuj kod na CodePen

Każdy `Clock` tworzy swój własny timer i aktualizuje go niezależnie od pozostałych.

W aplikacjach reactowych to, czy komponent ma stan, czy nie, jest tylko jego szczegółem implementacyjnym, który z czasem może ulec zmianie. Możesz dowolnie używać bezstanowych komponentów (ang. *stateless components*) wewnętrz tych ze stanem (ang. *stateful components*), i vice versa.

Statyczne sprawdzanie typów

Narzędzia statycznie sprawdzające typy, jak [Flow](#) i [TypeScript](#), identyfikują określone rodzaje problemów jeszcze zanim uruchomisz kod. Usprawniają one również pracę programisty poprzez dodanie przydatnych funkcjonalności, jak np. autouzupełniania. Z tego powodu w większych aplikacjach sugerujemy zastąpienie [PropTypes](#) narzędziami Flow lub TypeScript.

Flow

[Flow](#) jest narzędziem statycznie sprawdzającym typy w kodzie javascriptowym. Jest rozwijany przez Facebooka i często używany w połączeniu z Reactem. Pozwala opisać zmienne, funkcje i komponenty reactowe za pomocą specjalnej składni, a także wyłapuje zawczasu potencjalne problemy. Aby zapoznać się z jego podstawami, przeczytaj [wstęp do Flow](#).

Aby móc używać Flow, musisz:

- Dodać Flow do projektu jako zależność.
- Upewnić się, że składnia Flow jest usuwana podczas komplikacji.
- Dodać adnotacje typów i uruchomić Flow w celu ich sprawdzenia.

Wszystkie te kroki opisaliśmy szczegółowo poniżej.

Dodawanie Flow do projektu

Najpierw, będąc w terminalu, przejdź do folderu z projektem, a następnie uruchom następujące polecenie:

Jeśli używasz [Yarna](#), uruchom:

```
yarn add --dev flow-bin
```

Jeśli używasz [npma](#), uruchom:

```
npm install --save-dev flow-bin
```

Powyższa komenda zainstaluje w twoim projekcie najnowszą wersję narzędzia Flow.

Teraz dodaj `flow` do sekcji `"scripts"` wewnątrz pliku `package.json`, aby móc skorzystać z tego skryptu w terminalu:

```
{
  // ...
  "scripts": {
    "flow": "flow",
    // ...
  },
  // ...
}
```

Na koniec uruchom jedno z następujących poleceń:

Jeśli używasz [Yarna](#), uruchom:

```
yarn run flow init
```

Jeśli używasz [npma](#), uruchom:

```
npm run flow init
```

Powyższe komendy stworzą plik konfiguracyjny dla Flow, który należy umieścić w repozytorium.

Usuwanie składni Flow podczas komplikacji

Flow rozszerza język JavaScript poprzez dodanie specjalnej składni, służącej do deklaracji typów. Niestety przeglądarki nie rozumieją tej składni, dlatego musimy się upewnić, że nie trafi ona do skompilowanego kodu javascriptowego wysyłanego do przeglądarki.

Dokładny zestaw kroków opisujący sposób skonfigurowania tego zależy od narzędzi, którymi komplujesz swój kod javascriptowy.

Create React App

Jeśli twój projekt został stworzony przy pomocy [Create React App](#) - gratulujemy! Adnotacje dla Flow są już domyślnie usuwane podczas komplikacji, więc nie musisz niczego więcej konfigurować.

Babel

Uwaga:

Poniższe instrukcje *nie* dotyczą użytkowników Create React App. Pomimo iż Create React App używa "pod maską" Babela, jest on w tym przypadku już skonfigurowany tak, aby rozumieć Flow. Wykonaj poniższe kroki, jeśli *nie* używasz Create React App.

Jeśli Babel w twoim projekcie został skonfigurowany ręcznie, musisz zainstalować specjalny zestaw narzędzi (ang. *preset*) dla Flow.

Jeśli używasz Yarna, uruchom:

```
yarn add --dev @babel/preset-flow
```

Jeśli używasz npma, uruchom:

```
npm install --save-dev @babel/preset-flow
```

Następnie dodaj zestaw `flow` do swojej [konfiguracji Babela](#). Na przykład, jeśli konfiguracja w twoim projekcie jest trzymana w pliku `.babelrc`, mógłby on wyglądać następująco:

```
{
  "presets": [
    "@babel/preset-flow",
    "react"
  ]
}
```

Pozwoli ci to używać w kodzie składni Flow.

Uwaga:

Flow nie wymaga użycia zestawu `react`, jednak często są one używane w parze. Flow sam w sobie rozumie składnię JSX.

Inne sposoby konfiguracji etapu budowania

Jeśli nie korzystasz ani z Create React App, ani z Babelu, możesz użyć paczki `flow-remove-types`, która usunięcie deklaracje typów.

Korzystanie z Flow

Jeśli udało ci się skonfigurować wszystko, co opisaliśmy powyżej, możesz śmiało uruchomić Flow po raz pierwszy.

```
yarn flow
```

Jeśli używasz npma, uruchom:

```
npm run flow
```

Powinien pojawić się następujący komunikat ("Brak błędów!"):

```
No errors!
⭐ Done in 0.17s.
```

Dodawanie adnotacji w Flow

Domyślnie Flow sprawdza tylko te pliki, które zawierają następującą linię:

```
// @flow
```

Zwykle umieszcza się ją na początku pliku. Spróbuj dodać ją do kilku plików i uruchom `yarn flow` lub `npm run flow`, aby sprawdzić, czy Flow znalazł jakieś błędy.

Istnieje również [pewien sposób](#), by wymusić na Flow sprawdzanie *wszystkich* plików, bez względu na to, czy zawierają powyższą adnotację. W istniejących projektach może to wprowadzić niemałe zamieszanie, jednak sensowne jest ustawienie tej opcji w nowych projektach, w których od początku chcemy mieć zapewnione sprawdzanie typów przez Flow.

Masz już wszystko! Jeśli chcesz dowiedzieć się więcej na temat Flow, przeczytaj:

- [Dokumentacja Flow: Adnotacje typów](#)
- [Dokumentacja Flow: Edytory](#)
- [Dokumentacja Flow: React](#)
- [Linting w Flow](#)

TypeScript

[TypeScript](#) jest językiem programowania rozwijanym przez Microsoft. Jest typowanym nadzbiorem języka JavaScript i posiada własny kompilator. Jako język typowany, TypeScript może wyłapywać błędy na etapie budowania, długo zanim aplikacja "ujrzy światło dzienne". [Dowiedz się więcej na temat użycia TypeScriptu w parze z Reactem.](#)

Aby móc używać TypeScriptu, musisz:

- Dodać TypeScript jako zależność twojego projektu
- Skonfigurować opcje kompilatora typescriptowego
- Użyć odpowiednich rozszerzeń nazw plików
- Dodać definicje dla używanych w projekcie bibliotek

Zacznijmy po kolei.

Używanie TypeScriptu w Create React App

Create React App domyślnie obsługuje TypeScript.

Aby stworzyć **nowy projekt** mający wsparcie dla TypeScriptu, uruchom:

```
npx create-react-app my-app --template typescript
```

Możesz go również [dodać do istniejącego projektu stworzonego z Create React App](#).

Uwaga:

Jeśli używasz Create React App, możesz **pominąć resztę instrukcji**, ponieważ opisują one ręczną konfigurację, której nie muszą przeprowadzać użytkownicy Create React App.

Dodawanie TypeScriptu do projektu

Wszystko zaczyna się od uruchomienia jednego polecenia w terminalu.

Jeśli używasz [Yarna](#), uruchom:

```
yarn add --dev typescript
```

Jeśli używasz [npma](#), uruchom:

```
npm install --save-dev typescript
```

Gratulacje! Udało ci się zainstalować w projekcie najnowszą wersję TypeScriptu. Dało nam to dostęp do komendy `tsc`. Zanim zaczniesz konfigurowanie, dodaj `tsc` do sekcji `"scripts"` w pliku `package.json`:

```
{
  // ...
  "scripts": {
    "build": "tsc",
    // ...
  },
  // ...
}
```

Konfigurowanie kompilatora TypeScriptu

Kompilator na nic się nie zda, jeśli nie powiemy mu, co ma robić. W TypeScriptie reguły dla kompilatora definiuje się w specjalnym pliku `tsconfig.json`. Aby wygenerować taki plik:

Jeśli używasz [Yarna](#), uruchom:

```
yarn run tsc --init
```

Jeśli używasz [npma](#), uruchom:

```
npx tsc --init
```

Kiedy zajrzesz do nowo wygenerowanego pliku `tsconfig.json`, zobaczyś, jak wiele jest opcji konfiguracji kompilatora. Szczegółowy [opis wszystkich opcji](#) znajdziesz tutaj.

Spośród licznych opcji interesować nas będzie `rootDir` i `outDir`. Jak można się domyślić, kompilator odczyta pliki typescriptowe i wygeneruje z nich pliki javascriptowe. Jednak chcemy jasno określić, które z nich są plikami źródłowymi, a które wygenerowanym automatycznie kodem.

Zajmiemy się tym w dwóch krokach:

- Najpierw dostosujmy strukturę naszego projektu do poniższego schematu. Umieścmy wszystkie pliki źródłowe w folderze `src`.

```

└── package.json
└── src
  └── index.ts
└── tsconfig.json
```

- Następnie skonfigurujmy kompilator tak, by wiedział, gdzie są pliki źródłowe, a gdzie folder docelowy na wygenerowane pliki.

```
// tsconfig.json
{
  "compilerOptions": {
    // ...
    "rootDir": "src",
```

```

    "outDir": "build"
    // ...
},
}

```

Świetnie! Teraz, gdy tylko uruchomimy skrypt budujący, kompilator wygeneruje pliki z kodem javascriptowym do folderu `build`. Projekt [TypeScript React Starter](#) zawiera plik `tsconfig.json` z rozsądnią konfiguracją początkową dla nowych projektów.

Zwykle nie trzyma się wygenerowanych plików javascriptowych w repozytorium, więc upewnij się, że folder ten dodany jest do listy `.gitignore`.

Rozszerzenia plików

W Reakcie kod komponentów zwykle umieszcza się w plikach `.js`. W TypeScriptie mamy do wyboru dwa rozszerzenia:

`.ts` jest domyślnym rozszerzeniem plików, natomiast `.tsx` służy do oznaczania plików, które zawierają składnię `JSX`.

Uruchamianie TypeScriptu

Jeśli udało ci się wykonać powyższe instrukcje, możesz śmiało uruchomić TypeScript po raz pierwszy.

```
yarn build
```

Lub jeśli używasz npma:

```
npm run build
```

Jeśli nie wyświetlił się żaden komunikat, to znaczy, że komplikacja przebiegła pomyślnie.

Definicje typów

Aby móc wyświetlać błędy i sugestie dla innych paczek, kompilator polega na plikach z deklaracjami. Plik z deklaracjami dostarcza wszystkie informacje na temat typów w danej bibliotece. Pozwala to na używanie w naszym projekcie bibliotek javascriptowych z rejestru npm.

Istnieją dwa główne miejsca, z których odczytamy typy dla biblioteki:

Działające - Niektóre biblioteki dodają własne typy z deklaracjami. Jest to korzystne dla nas, ponieważ będziemy mogli zacząć korzystać z paczki zaraz po jej zainstalowaniu. Aby upewnić się, że biblioteka posiada działające typy, poszukaj w niej pliku `index.d.ts`. Niektóre biblioteki wyszczególniają ten plik w `package.json` pod kluczem `typings` lub `types`.

DefinitelyTyped - DefinitelyTyped to ogromne repozytorium deklaracji typów dla bibliotek, które nie dotyczą swoich własnych typów. Deklaracje te są crowd-source'owane i zarządzane przez firmę Microsoft oraz przez kontrybutorów open source'owych. Dla przykładu, React nie posiada własnego pliku z deklaracjami typów. Zamiast tego możemy pobrać go z DefinitelyTyped. Aby to zrobić, wpisz w terminalu następujące polecenie.

```
# yarn
yarn add --dev @types/react

# npm
npm i --save-dev @types/react
```

Deklaracje lokalne Czasami paczka, której chcesz użyć, nie posiada własnych typów ani nie ma ich w repozytorium DefinitelyTyped. W takim przypadku możesz stworzyć własny, lokalny plik z deklaracjami `declarations.d.ts` w głównym katalogu projektu. Przykładowa deklaracja może wyglądać tak:

```
declare module 'querystring' {
  export function stringify(val: object): string
  export function parse(val: string): object
}
```

Możesz już zacząć programować! Po więcej informacji o TypeScriptie zajrzyj do poniższych źródeł:

- [Dokumentacja TypeScripta: Typy podstawowe](#)
- [Dokumentacja TypeScripta: Migracja z JavaScriptu](#)
- [Dokumentacja TypeScripta: React i Webpack](#)

ReScript

ReScript jest typowanym językiem, który kompliuje się do JavaScriptu. Jego głównymi właściwościami są: gwarancja 100% pokrycia typami, [pierwszoklasowe](#) (*ang. first-class*) wsparcie dla JSX oraz dedykowane [wiązania dla Reacta](#), które pozwalają na zintegrowanie z istniejącymi projektami napisanymi w JS/TS.

Aby dowiedzieć się więcej na temat integracji ReScriptu z istniejącym projektem w JS/React, [przeczytaj dokumentację](#).

Kotlin

Kotlin jest statycznie typowanym językiem rozwijanym przez JetBrains. Docelowe platformy dla niego to JVM, Android, LLVM i [JavaScript](#).

JetBrains rozwija i utrzymuje kilka narzędzi dedykowanych dla społeczności Reacta: [API dla React](#), jak również [Create React Kotlin App](#). To ostatnie narzędzie pozwala na wystartowanie aplikacji reactowej z Kotlinem bez żadnego konfigurowania projektu.

Inne języki

Zauważ, że istnieją jeszcze inne statycznie typowane języki, komplikowane do JavaScriptu i przez to kompatybilne z Reactem. Na przykład [F#/Fable](#) z [elmish-react](#). Zajrzyj na ich strony internetowe po więcej informacji. Jeśli znasz jeszcze jakieś inne statycznie typowane języki, które współpracują z Reactem, nie wahaj się dodać ich do tej strony!

[Go to TOC](#)

Tryb rygorystyczny

`StrictMode` jest narzędziem podkreślającym potencjalne problemy w aplikacji. Tak samo jak `Fragment`, `StrictMode` nie renderuje żadnego widocznego UI. Służy natomiast do aktywacji dodatkowych sprawdzeń i ostrzeżeń dla swoich potomków.

Uwaga:

Sprawdzenia dla trybu rygorystycznego są uruchamiane wyłącznie w trybie deweloperskim; *nie mają one wpływu na build produkcyjny*.

Możesz uruchomić tryb rygorystyczny dla jakiejkolwiek części swojej aplikacji. Dla przykładu:

`embed:strict-mode/enabling-strict-mode.js`

W powyższym przykładzie sprawdzenia trybu rygorystycznego *nie zostaną* uruchomione dla komponentów `Header` i `Footer`. Zadziałały one natomiast dla `ComponentOne` i `ComponentTwo` oraz wszystkich ich potomków.

`StrictMode` aktualnie pomaga w:

- [Identyfikacji komponentów używających niebezpiecznych metod cyklu życia komponentu](#)
- [Ostrzeganiu o użyciu przestarzałego API tekstowych referencji](#)
- [Ostrzeganiu o użyciu przestarzałego `findDOMNode`](#)
- [Wykrywaniu nieoczekiwanych efektów ubocznych](#)
- [Wykrywaniu użycia przestrzałego API kontekstów](#)
- [Zapewnianiu wielokrotności użycia stanu](#)

Dodatkowe funkcjonalności zostaną dodane w przyszłych wydaniach Reacta.

Identyfikacja niebezpiecznych metod cyklu życia komponentu

Tak jak zostało to wytłumaczone [w tym poście](#), niektóre stare metody cyklu życia komponentu nie są bezpiecznie dla asynchronicznych aplikacji reactowych. Jednakże jeżeli Twoja aplikacja używa bibliotek firm trzecich, upewnienie się, że nie używa ona niebezpiecznych metod cyklu życia komponentu może być trudne. Na szczęście, tryb rygorystyczny może z tym pomóc!

Gdy tryb rygorystyczny jest włączony, React tworzy listę wszystkich komponentów klasowych, które używają niebezpiecznych metod cyklu życia, i loguje komunikaty ostrzegawcze z informacją o tych komponentach, jak na przykładzie:

```
▶ Warning: Unsafe lifecycle methods were found within a strict-mode tree:
  in div (created by ExampleApplication)
  in ExampleApplication

componentWillMount: Please update the following components to use componentDidMount instead: ThirdPartyComponent

Learn more about this warning here:
https://fb.me/react-strict-mode-warnings
```

Rozwiążanie teraz problemów zidentyfikowanych przez tryb rygorystyczny ułatwi użycie współbieżnego renderowania w przyszłych wydaniach Reacta.

Ostrzeganiu o użyciu przestarzałego API tekstowych referencji

Poprzednio React umożliwiał zarządzanie referencjami na dwa sposoby: przestarzałe referencje tekstowe i funkcje zwrotne. Pomimo że referencja tekstowa była z tych dwóch wygodniejsza, miała [kilka wad](#), dlatego też naszą oficjalną rekomendacją było [używanie formy funkcji zwrotnej](#).

React 16.3 wprowadził trzecią opcję, która oferuje wygodę jak w przypadku referencji tekstowej, bez żadnych wad: [embed:16-3-release-blog-post/create-ref-example.js](#)

Z uwagi na fakt, iż referencje obiektowe zostały dodane głównie jako zamiennik za referencje tekstowe, tryb rygorystyczny obecnie ostrzega w przypadku użyciu referencji tekstowych.

Uwaga:

Referencje w formie funkcji zwrotnej nadal będą wspierane wraz z nowym interfejsem `createRef`.

Nie musisz zamieniać referencji w postaci funkcji zwrotnej w swoich komponentach. Są one nieco bardziej elastyczne, więc pozostaną jako zaawansowana funkcjonalność.

Dowiedz się więcej o nowym API `createRef` tutaj.

Ostrzeganie o użyciu przestarzałego `findDOMNode`

React wspierał `findDOMNode`, aby umożliwić wyszukanie węzła powiązanego z daną instancją klasy w drzewie DOM. Zwykle jest to zbędna funkcjonalność, gdyż możesz [podczepić referencję bezpośrednio do węzła DOM](#).

`findDOMNode` mógł również być używany na komponencie klasy, jednak powodowało to zakłócenie poziomów abstrakcji poprzez umożliwienie rodzicowi, aby wymagał wyrenderowania się pewnego potomka. Tworzyło to ryzyko podczas tzw. "refactoringu", ponieważ nie można było zmienić implementacji komponentu, z węzłów którego mógłby korzystać rodzic. `findDOMNode` zawsze zwraca pierwsze dziecko, ale w przypadku użycia fragmentów jest możliwe, że komponent wyrenderuje wiele węzłów DOM. `findDOMNode` pozwala na jednorazowy odczyt, zwracając wynik tylko na żądanie. Jeżeli komponent potomny wyrenderuje inny węzeł, nie ma sposobu na obsłużenie tej zmiany. Innymi słowy `findDOMNode` działa wyłącznie gdy komponenty zwracają zawsze pojedynczy węzeł DOM, który nigdy się nie zmienia.

Zamiast tego do komponentu możesz jawnie przekazać referencję, którą następnie przekażesz do drzewa DOM przy użyciu [przekierowania referencji](#).

Możesz również dodać do swojego komponentu opakowujący węzeł DOM i bezpośrednio do niego dowiązać referencję.

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

Uwaga:

W CSS możesz użyć atrybutu `display: contents`, jeżeli nie chcesz, aby węzeł nie był częścią szablonu.

Wykrywanie nieoczekiwanych efektów ubocznych

Zasadniczo, React działa w dwóch fazach:

- Faza **renderowania** określa, jakie zmiany należy zaaplikować w np. drzewie DOM. Podczas tej fazy React wywołuje metodę `render` i porównuje jej wynik z poprzednim.
- Faza **aktualizacji** następuje wtedy, gdy React aplikuje zmiany. (W przypadku React DOM następuje to, gdy React dodaje, aktualizuje i usuwa węzły DOM.) Podczas tej fazy React wywołuje również metody cyklu życia komponentu tj. `componentDidMount` czy `componentDidUpdate`.

Faza aktualizacji jest zazwyczaj bardzo szybka, jednak renderowanie może być powolne. Z tego powodu nadchodzący tryb współbieżny (który nie jest jeszcze domyślnie włączony), rozbija pracę związaną z renderowaniem na części, zatrzymując i wznowiając pracę, aby uniknąć blokowania przeglądarki. To oznacza, że React może wywołać metody cyklu życia w fazie renderowania więcej niż raz przed aktualizacją lub może je wywołać nawet bez aktualizacji (z powodu błędu lub przerwania o wyższym priorytecie).

Cykł życia fazy renderowania odnosi się do poniższych metod z komponentu klasowego:

- `constructor`
- `componentWillMount` (or `UNSAFE_componentWillMount`)
- `componentWillReceiveProps` (or `UNSAFE_componentWillReceiveProps`)
- `componentWillUpdate` (or `UNSAFE_componentWillUpdate`)
- `getDerivedStateFromProps`
- `shouldComponentUpdate`
- `render`
- funkcje aktualizujące dla `setState` (pierwszy argument)

Ponieważ powyższe metody mogą być wywołane więcej niż raz, ważne jest, aby nie zawierały efektów ubocznych (ang. *side-effects*). Zignorowanie tej zasady może prowadzić do różnych problemów, włączając w to wycieki pamięci i niepoprawny stan aplikacji. Niestety, może być ciężko wykryć problemy tego typu, ponieważ są one często [niedeterministyczne](#).

Tryb rygorystyczny nie zapewni automatycznego wykrywania efektów ubocznych, ale może pomóc w ich zauważeniu poprzez sprawienie, by były trochę bardziej deterministyczne. Dzieje się to za sprawą celowego podwójnego wywoływania poniższych funkcji:

- Metod `constructor`, `render` oraz `shouldComponentUpdate` komponentu klasowego
- Metody statycznej `getDerivedStateFromProps` komponentu klasowego
- Ciała komponentu funkcyjnego
- Funkcji aktualizującej stanu (pierwszy argument `setState`)
- Funkcji przekazywanych do `useState`, `useMemo` oraz `useReducer`

Uwaga:

Ma to zastosowanie tylko w trybie deweloperskim. *Metody życia komponentu nie będą podwojnie wywoływane w trybie produkcyjnym.*

Dla przykładu, rozważ poniższy kod: `embed:strict-mode/side-effects-in-constructor.js`

Na pierwszy rzut oka kod nie wydaje się problematyczny. Ale jeżeli `SharedApplicationState.recordEvent` nie jest [idempotentna](#), to stworzenie wielu instancji tego komponentu może prowadzić do niepoprawnego stanu aplikacji. Ten subtelny błąd może się nie ukazać podczas dewelopmentu lub może występować sporadycznie i zostać przeoczony.

Przez celowe podwójne wywołanie metod, takich jak konstruktor komponentu, tryb rygorystyczny sprawia, że wzorce tego typu są łatwiejsze do zauważenia.

Uwaga:

Począwszy od wersji 17, React automatycznie modyfikuje metody takie jak `console.log()`, aby uciśczyć logi przy powtórnym wywołaniu funkcji odpowiedzialnych za cykl życia komponentów. Jednak w niektórych przypadkach może to prowadzić do niepożądanych zachowań, [na co można zaradzić w ten sposób](#).

Od wersji React 18, React nie ucisza żadnych logów. Jeśli jednak korzystasz z React DevTools, logi z drugiego wywołania będą nieco bardziej wyszarzone. React DevTools posiada również opcję (domyślnie wyłązioną), która pozwala całkowicie je uciszyć.

Wykrywanie przestarzałego API kontekstów

Przestarzałe API kontekstów jest podatne na błędy i może zostać usunięte w przyszłych wersjach. Nadal będzie działać dla wszystkich wydań 16.x, ale w trybie rygorystycznym będzie w nich wyświetlany poniższy komunikat ostrzegawczy:

```
✖ ► Warning: Legacy context API has been detected within a strict-mode tree:  
  in div (at App.js:32)  
  in App (at index.js:7)  
  
Please update the following components: LegacyContextConsumer, LegacyContextProvider  
  
Learn more about this warning here:  
https://fb.me/react-strict-mode-warnings
```

Przeczytaj [dokumentację nowego API kontekstów](#), aby dowiedzieć się, jak zmigrować do nowej wersji.

Zapewnienie wielorazowego stanu

W przyszłości planujemy dodać do Reacta funkcjonalność, która pozwalałaby na dodawanie i usuwanie fragmentów interfejsu, jednocześnie zachowując ich stan. Na przykład, kiedy użytkownik zmienia zakładkę i za chwilę wraca z powrotem, React powinien być w stanie natychmiast pokazać poprzedni widok. Aby to było możliwe, React musi móc ponownie zamontować poddrzewo z takim samym stanem, jaki był przed jego odmontowaniem.

Taka funkcjonalność dałaby Reactowi lepszą wydajność bez żadnej dodatkowej konfiguracji, lecz wymaga ona, by komponenty były odporne na efekty podczas wielokrotnego montowania i odmontowywania. Większość efektów będzie działać bez żadnych zmian, jednak niektóre nie sprzątają po sobie poprawnie subskrypcji w funkcji zwracanej przez efekt albo po prostu z góry zakładają, że komponent będzie zamontowany lub odmontowany tylko jeden raz.

Aby uwypuklić te problemy, w Reakcie 18 do trybu rygorystycznego dodaliśmy nowy test, działający tylko w środowisku deweloperskim. Ów test automatycznie odmontuje i zamontuje ponownie każdy komponent renderowany po raz pierwszy, przywracając jego poprzedni stan podczas drugiego montowania.

Aby zademonstrować zachowanie aplikacji w takiej sytuacji, zastanówmy się, co się dzieje, gdy React montuje nowy komponent. Jeśli na chwilę zapomnimy o tym teście, to podczas montowania komponentu React tworzy efekty:

- * React montuje komponent.
- * Tworzone są efekty układu interfejsu (ang. *layout effects*).
- * Tworzone są zwykłe efekty.

W trybie rygorystycznym w Reakcie 18 podczas montowania komponentu React natychmiast zasymuluje jego odmontowanie i ponowne zamontowanie:

- * React montuje komponent.
 - * Tworzone są efekty układu interfejsu.
 - * Tworzone są zwykłe efekty.
- * React symuluje zniszczenie efektów zamontowanego komponentu.
 - * Niszczone są efekty układu interfejsu.

- * Niszczzone są zwykłe efekty.
- * React symuluje ponownie utworzenie efektów zamontowanego komponentu.
 - * Tworzone są efekty układu interfejsu.
 - * Tworzone są zwykłe efekty.

Przy drugim montowaniu React przywróci stan z pierwszego montowania. Symuluje to zachowanie użytkownika, np. w przypadku przejścia na inną zakładkę i z powrotem na aktualną, upewniając się, że kod poprawnie obsługuje przywrócenie stanu komponentu.

Kiedy komponent zostaje odmontowany, efekty są usuwane standardowo:

- * React odmontowuje komponent.
- * Niszczone są efekty układu interfejsu.
- * Niszczone są zwykłe efekty.

Do cyklu odmontowywania i ponowne montowania zalicza się:

- `componentDidMount`
- `componentWillUnmount`
- `useEffect`
- `useLayoutEffect`
- `useInsertionEffect`

Uwaga:

Zachowanie opisane powyżej dotyczy tylko trybu deweloperskiego, *zachowanie na produkcji pozostaje bez zmian*.

Opis najczęstszych błędów znajdziesz na:

- [Jak obsłużyć wielorazowy stan w efektach](#)

Środowiska testujące

W tym rozdziale opisujemy czynniki wpływające na środowisko testujące i nasze rekomendacje dla niektórych scenariuszy.

Narzędzia uruchamiające testy (ang. *test runners*)

Narzędzia uruchamiające testy, jak np. [Jest](#), [mocha](#) czy [ava](#), pozwalają tworzyć zestawy testowe przy użyciu samego JavaScriptu, a także uruchamiać je jako część procesu tworzenia oprogramowania. Dodatkowo, testy mogą być uruchamiane w ramach procesu "ciągłej integracji" (ang. *Continuous Integration, CI*).

- Jest ma wysoką kompatybilność z projektami reactowymi i obsługuje wiele przydatnych funkcjonalności, jak [mockowanie modułów](#) czy [sztuczne timery](#). Dobrze współpracuje również z [jsdom](#). **Jeśli używasz Create React App, domyślnie masz już dostęp do Jesta z odpowiednią konfiguracją.**
- Biblioteki takie jak [mocha](#) świetnie spisują się w środowiskach przeglądarkowych, dzięki czemu mogą okazać się pomocne w przypadku niektórych testów.
- Testy kompleksowe end-to-end, stosowane w przypadku dłuższych ścieżek rozciągających się na wiele stron aplikacji, wymagają [innej konfiguracji](#).

Mockowanie warstwy renderującej

Testy często uruchamiane są w środowiskach niemających dostępu do prawdziwej warstwy renderującej, np. przeglądarki. Zalecamy więc symulowanie zachowań przeglądarki za pomocą [jsdom](#), niewielkiej implementacji przeglądarki działającej na Node.js.

W większości przypadków [jsdom](#) zachowuje się jak prawdziwa przeglądarka, lecz nie posiada niektórych funkcjonalności, jak np. [generowanie układu strony](#) czy [nawigacja](#). Mimo tego paczka z powodzeniem sprawdza się dla większości komponentów pisanych pod przeglądarkę; działa szybciej niż uruchamianie przeglądarki dla każdego testu z osobna. Ponadto, uruchamia ona testy w tym samym procesie, umożliwiając pisanie kodu sprawdzającego wyrenderowane drzewo DOM.

Podobnie jak prawdziwa przeglądarka, [jsdom](#) pozwala na modelowanie interakcji użytkownika; testy mogą wywoływać zdarzenia na węzłach DOM, a następnie obserwować i sprawdzać wyniki tych akcji ([przykład](#)).

Przy takiej konfiguracji można śmiało napisać większość testów dla UI: Jest jako narzędzie uruchamiające testy, [jsdom](#) służący do renderowania, interakcje użytkownika określone jako sekwencje zdarzeń przeglądarkowych - a to wszystko "spięte" za pomocą funkcji pomocniczej [act\(\)](#) ([przykład](#)). Spora część testów samego Reacta jest napisana przy użyciu powyższej kombinacji.

Jeśli piszesz bibliotekę, która testuje głównie zachowania charakterystyczne dla przeglądarki, a w dodatku wymaga natywnych mechanizmów przeglądarki, jak generowanie układu strony, zalecamy skorzystanie z framework'a [mocha](#).

W środowisku, które *uniemożliwia* symulowanie modelu DOM (np. podczas testowania komponentów napisanych w React Native na Node.js), możesz skorzystać z [narzędzi do symulowania zdarzeń](#) do symulowania interakcji z elementami. Alternatywnie możesz skorzystać z funkcji `fireEvent` dostarczonej przez `@testing-library/react-native`.

Frameworki jak [Cypress](#), [puppeteer](#) czy [webdriver](#) służą do uruchamiania testów [end-to-end](#).

Mockowanie funkcji

Podczas pisania testów czasami chcemy podmienić części naszego kodu, które nie posiadają odpowiedników w używanym przez nas środowisku (np. sprawdzanie statusu `navigator.onLine` w Node.js). Testy mogą również śledzić niektóre funkcje i obserwować, jak pozostałe części kodu wchodzą z nimi w interakcje. Pomocna okazuje się wtedy możliwość wybiórczego zastąpienia niektórych funkcji wersjami odpowiednimi dla testów.

Szczególnie przydatne okazuje się to przy pobieraniu danych. Zazwyczaj lepiej w testach używać "sztucznych" danych, aby uniknąć spowolnień czy niestabilności z powodu odwołań do prawdziwego API ([przykład](#)). Dzięki takiemu zabiegowi testy są przewidywalne. Biblioteki typu [Jest](#) czy [sinon](#) wspierają mockowanie funkcji. W przypadku testów end-to-end, mockowanie sieci może okazać się trudniejsze, choć należy tego unikać i zamiast tego testować korzystając z prawdziwego API.

Mockowanie modułów

Niektóre komponenty mają zależności w modułach, które mogą nie działać w środowisku testowym lub które zwyczajnie nie są istotne z punktu widzenia naszych testów. Warto wtedy zastąpić te moduły czymś odpowiednim dla danego przypadku ([przykład](#)).

W Node.js [mockowanie modułów](#) jest wspierane np. przez bibliotekę Jest. Można to również osiągnąć z pomocą paczki `mock-require`.

Mockowanie timerów

Komponenty mogą korzystać z funkcji opartych na czasie, np. `setTimeout`, `setInterval` czy `Date.now`. W środowisku testowym warto zamieniać tego typu funkcje na ich zastępniki, które pozwalają ręcznie "stutować czasem". To świetny sposób na znaczne przyspieszenie działania testów. Testy korzystające z timerów nadal będą wykonywać się w odpowiedniej kolejności, ale zdecydowanie szybciej ([przykład](#)). Większość frameworków, również [Jest](#), [sinon](#) oraz [lolex](#), pozwalają na mockowanie timerów w testach.

Niekiedy jednak możesz chcieć skorzystać z prawdziwych timerów, na przykład, gdy testujesz animację lub interakcję z endpointem, który zależy od czasu (np. ogranicza częstotliwość odpytywania API). Biblioteki zawierające sztuczne timery pozwalają na łatwe włączanie i wyłączanie tego mechanizmu dla każdego zestawu testowego lub pojedynczego testu. Dzięki temu możesz zdecydować, jak poszczególne testy mają być uruchamiane.

Testy end-to-end

Testy end-to-end są efektywne przy testowaniu dłuższych sekwencji interakcji, zwłaszcza jeśli są one krytyczne dla twojego produktu (np. płatność czy rejestracja). W takich przypadkach konieczne jest przetestowanie, jak przeglądarka renderuje całą aplikację, jak pobiera dane z API, korzysta z sesji i ciasteczek lub nawiguje pomiędzy poszczególnymi stronami. Możesz w nich sprawdzać nie tylko stan drzewa DOM, lecz także sterujące nim dane (np. weryfikując, czy dane zostały zapisane w bazie danych).

Do takich scenariuszy możesz skorzystać z framework'a [Cypress](#), [Playwright](#) lub biblioteki [Puppeteer](#), które pozwalają nawigować pomiędzy stronami i sprawdzać rezultaty nie tylko w samej przeglądarce, ale potencjalnie również na backendzie.

Przykłady i dobre praktyki

Wzorce często używane przy testowaniu komponentów reactowych.

Uwaga:

Ten rozdział zakłada, że do uruchamiania testów używasz [Jesta](#). Jeśli używasz innego narzędzia, konieczne będzie dostosowanie kodu do jego interfejsu API, jednak ogólny schemat rozwiązania powinien być taki sam. Aby dowiedzieć się więcej na temat konfiguracji środowiska testowego, przeczytaj rozdział pt. "[Środowiska testowe](#)".

W tym rozdziale będziemy głównie używać komponentów funkcyjnych. Mimo to z powodzeniem możesz zastąpić je komponentami klasowymi, ponieważ opisane tu rozwiązania nie zależą od sposobu implementacji.

- [Setup/Teardown \(pol. Ustawienie/Zniszczenie\)](#)
- `act()`
- [Renderowanie](#)
- [Pobieranie danych](#)
- [Mockowanie modułów](#)
- [Zdarzenia](#)
- [Timery](#)
- [Testowanie snapshotowe](#)
- [Wiele silników renderujących](#)
- [Brakuje czegoś?](#)

Setup/Teardown (pol. *Ustawienie/Zniszczenie*)

Zwykle w każdym teście chcemy wyrenderować nasze drzewo reactowe do elementu DOM umieszczonego w globalnym obiekcie `document`. Tylko wtedy nasze komponenty będą otrzymywać zdarzenia DOM. Po zakończonym teście musimy "posprzątać", odmontowując drzewo od obiektu `document`.

Popularnym sposobem na to jest użycie pary bloków `beforeEach` (pol. *przed każdym*) oraz `afterEach` (pol. *po każdym*) uruchamianych przy każdym teście, które gwarantują pełną izolację wszelkich efektów ubocznych w naszych testach:

```
import { unmountComponentAtNode } from "react-dom";

let container = null;
beforeEach(() => {
  // ustaw element DOM jako cel renderowania
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // posprzątaj po zakończeniu
```

```
    unmountComponentAtNode(container);
    container.remove();
    container = null;
});
```

Możesz skorzystać z innego podejścia. Pamiętaj jednak, że należy posprzątać, *nawet jeśli test nie przejdzie pomyślnie*. W przeciwnym wypadku testy staną się "dziurawe" i będą mogły wpływać na działanie innych testów. Znacznie utrudni to szukanie błędów.

act() {#act}

Podczas pisania testów interfejsu UI, zadania takie jak: renderowanie, obsługa zdarzeń czy pobieranie danych można traktować jako "jednostki" interakcji z interfejsem użytkownika. Moduł `react-dom/test-utils` dostarcza funkcję pomocniczą `act()`, która upewnia się, że przed wykonaniem asercji zostają przetworzone wszelkie zmiany dotyczące tych "jednostek", a drzewo DOM jest aktualne.

```
act(() => {
  // wyrenderuj komponenty
});
// wykonaj sprawdzenia
```

Pozwala to na uruchamianie testów w sposób zbliżony do faktycznego zachowania podczas interakcji użytkownika z aplikacją. Pozostałe przykłady w tym rozdziale używają funkcji `act()`, aby to zagwarantować.

Używanie `act()` bezpośrednio może wydawać się jednak zbyt rozwlekłe. Aby uniknąć pisania wciąż tego samego kodu, możesz użyć biblioteki takiej jak [React Testing Library](#), której to funkcje pomocnicze są "opakowane" w `act()`.

Uwaga:

Nazwa `act` pochodzi od zasady [Arrange-Act-Assert](#) (pol. *Przygotuj-Wykonaj-Sprawdź*).

Renderowanie

Komponenty zazwyczaj testuje się poprzez sprawdzenie, czy renderują się one poprawnie dla konkretnych danych wejściowych. Rozważmy następujący komponent wyświetlający wiadomość:

```
// hello.js
import React from "react";

export default function Hello(props) {
  if (props.name) {
    return <h1>Witaj, {props.name}!</h1>;
  } else {
    return <span>Cześć, nieznajomy</span>;
  }
}
```

Test dla takiego komponentu może wyglądać tak:

```
// hello.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // ustaw element DOM jako cel renderowania
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // posprzątaj po zakończeniu
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("renders with or without a name", () => {
  act(() => {
    render(<Hello />, container);
  });
  expect(container.textContent).toBe("Cześć, nieznajomy");

  act(() => {
    render(<Hello name="Janusz" />, container);
  });
  expect(container.textContent).toBe("Witaj, Janusz!");

  act(() => {
    render(<Hello name="Grażyna" />, container);
  });
  expect(container.textContent).toBe("Witaj, Grażyna!");
});
```

Pobieranie danych

Zamiast odpytywania prawdziwego API w każdym z testów, możesz zamockować żądania przy użyciu sztucznych danych. Dzięki temu testy będą odporne na czasową niedostępność backendu, a także będą wykonywały się szybciej. Uwaga: mimo wszystko warto wydzielić kilka testów do frameworka testującego "end-to-end", który sprawdzi, czy cała aplikacja działa poprawnie.

```
// user.js

import React, { useState, useEffect } from "react";

export default function User(props) {
  const [user, setUser] = useState(null);

  async function fetchUserData(id) {
    const response = await fetch(`/${id}`);
    setUser(await response.json());
  }

  useEffect(() => {
    fetchUserData(props.id);
  });
}
```

```

}, [props.id]);

if (!user) {
  return "Wczytywanie...";
}

return (
<details>
  <summary>{user.name}</summary>
  <strong>{user.age}</strong> lat
  <br />
  adres: {user.address}
</details>
);
}

```

Testy dla takiego komponentu mogłyby wyglądać następująco:

```

// user.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import User from "./user";

let container = null;
beforeEach(() => {
  // ustaw element DOM jako cel renderowania
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // posprzątaj po zakończeniu
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("wyświetla dane użytkownika", async () => {
  const fakeUser = {
    name: "Jan Kowalski",
    age: "32",
    address: "ul. Zimna 12, Pcim Dolny"
  };

  jest.spyOn(global, "fetch").mockImplementation(() =>
    Promise.resolve({
      json: () => Promise.resolve(fakeUser)
    })
  );

  // Użyj asynchronicznej wersji funkcji act, aby poczekać na zakończenie efektu
  await act(async () => {
    render(<User id="123" />, container);
  });

  expect(container.querySelector("summary").textContent).toBe(fakeUser.name);
  expect(container.querySelector("strong").textContent).toBe(fakeUser.age);
  expect(container.textContent).toContain(fakeUser.address);
})

```

```
// usuń mock, aby zapewnić izolację testu
global.fetch.mockRestore();
});
```

Mockowanie modułów

Niektóre moduły mogą nie działać poprawnie w środowisku testowym lub mogą być nieistotne z perspektywy danego testu. Mockowanie takich modułów przy użyciu "udawanych" (ang. *dummy*) modułów może znacznie ułatwić pisanie testów do własnego kodu.

Rozważmy komponent `Contact`, który zawiera w sobie komponent `GoogleMap` z biblioteki zewnętrznej:

```
// map.js

import React from "react";

import { LoadScript, GoogleMap } from "react-google-maps";
export default function Map(props) {
  return (
    <LoadScript id="script-loader" googleMapsApiKey="TWOJ_KLUCZ_API">
      <GoogleMap id="example-map" center={props.center} />
    </LoadScript>
  );
}

// contact.js

import React from "react";
import Map from "./map";

export default function Contact(props) {
  return (
    <div>
      <address>
        Skontaktuj się z {props.name} za pomocą{" "}
        <a data-testid="email" href={"mailto:" + props.email}>
          e-maila
        </a>
        lub <a data-testid="site" href={props.site}>
          strony internetowej
        </a>.
      </address>
      <Map center={props.center} />
    </div>
  );
}
```

Jeśli nie chcemy ładować tego komponentu w ramach naszych testów, możemy go zamockować przy użyciu udawanego komponentu:

```
// contact.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Contact from "./contact";
import MockedMap from "./map";
```

```

jest.mock("./map", () => {
  return function DummyMap(props) {
    return (
      <div data-testid="map">
        {props.center.lat}:{props.center.long}
      </div>
    );
  };
});

let container = null;
beforeEach(() => {
  // ustaw element DOM jako cel renderowania
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // posprzątaj po zakończeniu
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("wyświetla informacje kontaktowe", () => {
  const center = { lat: 0, long: 0 };
  act(() => {
    render(
      <Contact
        name="Jan Kowalski"
        email="test@example.com"
        site="http://test.com"
        center={center}
      />,
      container
    );
  });

  expect(
    container.querySelector("[data-testid='email']").getAttribute("href")
  ).toEqual("mailto:test@example.com");

  expect(
    container.querySelector('[data-testid="site"]').getAttribute("href")
  ).toEqual("http://test.com");

  expect(container.querySelector('[data-testid="map"]').textContent).toEqual(
    "0:0"
  );
});
});

```

Zdarzenia

Sugerujemy, aby do elementów DOM przesyłać prawdziwe zdarzenia DOM, a następnie sprawdzać wynik ich działania. Rozważmy następujący komponent `Toggle`:

```

// toggle.js

import React, { useState } from "react";

```

```

export default function Toggle(props) {
  const [state, setState] = useState(false);
  return (
    <button
      onClick={() => {
        setState(previousState => !previousState);
        props.onChange(!state);
      }}
      data-testid="toggle"
    >
      {state === true ? "Wyłącz" : "Włącz"}
    </button>
  );
}

```

Testy dla niego mogłyby wyglądać następująco:

```

// toggle.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Toggle from "./toggle";

let container = null;
beforeEach(() => {
  // ustaw element DOM jako cel renderowania
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // posprzątaj po zakończeniu
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("zmienia wartość po kliknięciu", () => {
  const onChange = jest.fn();
  act(() => {
    render(<Toggle onChange={onChange} />, container);
  });

  // zlokalizuj przycisk i wywołaj na nim zdarzenie kliknięcia
  const button = document.querySelector("[data-testid=toggle]");
  expect(button.innerHTML).toBe("Włącz");

  act(() => {
    button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
  });

  expect(onChange).toHaveBeenCalledTimes(1);
  expect(button.innerHTML).toBe("Wyłącz");

  act(() => {
    for (let i = 0; i < 5; i++) {
      button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
    }
  });
})

```

```
expect(onChange).toHaveBeenCalledTimes(6);
expect(button.innerHTML).toBe("Włącz");
});
```

Poszczególne zdarzenia DOM wraz z ich właściwościami zostały opisane na [MDN](#). Zwróć uwagę, że do każdego zdarzenia musisz przekazać `{ bubbles: true }`, aby mogło ono dotrzeć do Reacta, który z kolei przekaże je do komponentu głównego całej struktury.

Uwaga:

Funkcje pomocnicze w [React Testing Library](#) umożliwiają przesyłanie zdarzeń w sposób bardziej zwięzły.

Timery

Twój kod może korzystać z funkcji opartych na timerach, np. `setTimeout` w celu zaplanowania wykonania jakiejś akcji w przyszłości. W tym przykładzie stworzyliśmy panel wielokrotnego wyboru, który czyści zaznaczenie, jeśli użytkownik nie wybierze niczego w ciągu 5 sekund:

```
// card.js

import React, { useEffect } from "react";

export default function Card(props) {
  useEffect(() => {
    const timeoutID = setTimeout(() => {
      props.onSelect(null);
    }, 5000);
    return () => {
      clearTimeout(timeoutID);
    };
  }, [props.onSelect]);

  return [1, 2, 3, 4].map(choice => (
    <button
      key={choice}
      data-testid={choice}
      onClick={() => props.onSelect(choice)}
    >
      {choice}
    </button>
  ));
}
```

W pisaniu testów do tego komponentu pomocne mogą okazać się [sztuczne timery z biblioteki Jest](#). Umożliwiają sprawdzenie stanu komponentu w różnych punktach w czasie.

```
// card.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
```

```
import Card from "./card";

let container = null;
beforeEach(() => {
  // ustaw element DOM jako cel renderowania
  container = document.createElement("div");
  document.body.appendChild(container);
  jest.useFakeTimers();
});

afterEach(() => {
  // posprzątaj po zakończeniu
  unmountComponentAtNode(container);
  container.remove();
  container = null;
  jest.useRealTimers();
});

it("powinien zaznaczyć 'null' po upływie czasu", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  // przesuń timer o 100 milisekund
  act(() => {
    jest.advanceTimersByTime(100);
  });
  expect(onSelect).not.toHaveBeenCalled();

  // a teraz dodaj kolejne 5 sekund
  act(() => {
    jest.advanceTimersByTime(5000);
  });
  expect(onSelect).toHaveBeenCalledWith(null);
});

it("powinien posprzątać po sobie po usunięciu", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  act(() => {
    jest.advanceTimersByTime(100);
  });
  expect(onSelect).not.toHaveBeenCalled();

  // odmontuj aplikację
  act(() => {
    render(null, container);
  });

  act(() => {
    jest.advanceTimersByTime(5000);
  });
  expect(onSelect).not.toHaveBeenCalled();
});

it("powinien umożliwiać zaznaczanie", () => {
  const onSelect = jest.fn();
  act(() => {
```

```

    render(<Card onSelect={onSelect} />, container);
});

act(() => {
  container
    .querySelector("[data-testid='2']")
    .dispatchEvent(new MouseEvent("click", { bubbles: true }));
});

expect(onSelect).toHaveBeenCalledWith(2);
});

```

Możesz używać sztucznych timerów w wybranych przez siebie testach. Na powyższym przykładzie aktywowałeś je poprzez wywołanie `jest.useFakeTimers()`. Główną ich zaletą jest brak konieczności czekania 5 sekund na wykonanie się testu. Ponadto, nie trzeba było wprowadzać na potrzeby testu zawiłych zmian w kodzie komponentu.

Testowanie snapshotowe

Frameworki takie jak Jest pozwalają również na zapisywanie "snapshotów" (pol. zrzutów) danych przy użyciu funkcji `toMatchSnapshot` / `toMatchSnapshot`. Z ich pomocą możemy zapisać wynik renderowania komponentu i zagwarantować, że dowolna jego zmiana będzie musiała być każdorazowo potwierdzana podczas generowania snapshota.

W poniższym przykładzie renderujemy komponent i formatujemy powstały w ten sposób kod HTML za pomocą paczki `pretty`, zanim ostatecznie wynik trafi do snapshota:

```

// ponownie hello.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import pretty from "pretty";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // ustaw element DOM jako cel renderowania
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // posprzątaj po zakończeniu
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("powinien wyrenderować powitanie", () => {
  act(() => {
    render(<Hello />, container);
  });

  expect(
    pretty(container.innerHTML)
  );
});

```

```

).toMatchSnapshot(); /* ... zostanie automatycznie zastąpione przez Jesta
... */

act(() => {
  render(<Hello name="Janusz" />, container);
});

expect(
  pretty(container.innerHTML)
).toMatchSnapshot(); /* ... zostanie automatycznie zastąpione przez Jesta
... */

act(() => {
  render(<Hello name="Grażyna" />, container);
});

expect(
  pretty(container.innerHTML)
).toMatchSnapshot(); /* ... zostanie automatycznie zastąpione przez Jesta
... */
);

```

Zazwyczaj jednak, zamiast porównywać snapshoty, zaleca się wykonywać bardziej szczegółowe sprawdzenia. Tego typu testy zawierają zbyt wiele szczegółów implementacyjnych, więc łatwo się psują. Ponadto usypanią czujność zespołu na błędy w snapshotach. Wybiórce [mockowanie niektórych komponentów potomnych](#) może pomóc zredukować rozmiar snapshotów, a co za tym idzie zwiększyć ich czytelność dla osoby sprawdzającej kod.

Wiele silników renderujących

W rzadkich przypadkach możesz natknąć się na test komponentu, który korzysta z wielu silników renderujących. Na przykład, test snapshotowy może używać paczki `react-test-renderer`, która wewnętrznie korzysta z `render` z pakietu `react-dom` dla komponentu potomnego w celu wyrenderowania jakiejś treści. W takim przypadku możesz opakować aktualizacje drzewa komponentów w funkcje `act()` właściwe odpowiednim silnikom renderującym.

```

import { act as domAct } from "react-dom/test-utils";
import { act as testAct, create } from "react-test-renderer";
// ...
let root;
domAct(() => {
  testAct(() => {
    root = create(<App />);
  });
});
expect(root).toMatchSnapshot();

```

Czegoś brakuje?

Jeśli nie opisaliśmy tu jakiegoś przypadku, [daj nam o tym znać poprzez zgłoszenie](#) dotyczące strony z dokumentacją.

[Go to TOC](#)

Ogólne informacje

- "community/testing.html"

Komponenty reactowe można testować podobnie jak pozostały kod javascriptowy.

Istnieje kilka sposobów na przetestowanie komponentów reactowych. W dużym uproszczeniu, dzielą się one na dwie kategorie:

- **Renderowanie drzew komponentów** w uproszczonym środowisku testowym oraz sprawdzanie wyniku renderowania.
- **Uruchamianie pełnej aplikacji** w realistycznym środowisku przeglądarkowym (znane również jako testy "end-to-end").

Ten rozdział dokumentacji skupia się na strategiach testowania w pierwszy sposób. Mimo iż pełne testy "end-to-end" często zapobiegają regresji w kluczowych ścieżkach aplikacji, nie przywiązuje one zbyt dużej uwagi do komponentów reactowych. Z tego powodu pominęliśmy je w tej sekcji.

Kompromisy

Podczas wybierania narzędzia testującego warto zastanowić się nad kilkoma decyzjami:

- **Szybkość iteracji czy realistyczne środowisko:** Niektóre narzędzia oferują szybkie sprzężenie zwrotne pomiędzy wprowadzeniem zmiany a otrzymaniem wyniku, lecz nie odwzorowują dokładnie zachowania przeglądarki. Inne z kolei używają realistycznego środowiska przeglądarkowego, lecz zmniejszają szybkość iteracji i działają topornie na serwerach CI (ang. *Continuous Integration*).
- **Ile powinniśmy zamockować:** W przypadku komponentów, granica pomiędzy testami "jednostkowymi" a "integracyjnymi" może się zacierać. Kiedy testujesz formularz, czy testy powinny także sprawdzić działanie znajdujących się w nim przycisków? Czy może przycisk powinien mieć dedykowany zestaw testowy? Czy zmiany w kodzie przycisku powinny wpływać na testy formularza?

Do każdego zespołu i każdego produktu pasują inne odpowiedzi na powyższe pytania.

Zalecane narzędzia

Jest to javascriptowy "test runner" (pol. *narzędzie uruchamiające testy*), które pozwala uzyskać dostęp do DOM dzięki paczce `jsdom`. Mimo iż `jsdom` tylko w przybliżeniu działa jak prawdziwa przeglądarka, zwykle wystarcza do przetestowania komponentów reactowych. Biblioteka Jest gwarantuje szybką iterowalność połączoną z praktycznymi funkcjonalnościami, jak mockowanie **modułów** czy **timerów**. Dzięki temu masz większą kontrolę nad tym, jak wykonywany jest twój kod.

React Testing Library jest zestawem funkcji pomocniczych, które pozwalają nad testowanie komponentów reactowych bez polegania na ich szczegółach implementacyjnych (ang. *implementation details*). Takie podejście sprawia, że refactoring kodu staje się niezwykle prosty, a także "popycha" cię w kierunku dobrych

praktyk dotyczących dostępności (ang. *accessibility*). Mimo iż ta biblioteka nie umożliwia "pływkiego" renderowania (ang. *shallow rendering*) komponentów bez ich potomków, doskonale sprawdza się w połączeniu z Jestem i jego funkcjonalnością [mockowania modułów](#).

Dowiedz się więcej

Ten rozdział został podzielony na dwie części:

- [Przykłady i dobre praktyki](#): Wzorce często spotykane przy testowaniu komponentów reactowych.
- [Środowiska](#): Na co należy zwrócić uwagę podczas zestawiania środowiska testującego dla komponentów reactowych.

Myślenie reactowe

- 'blog/2013/11/05/thinking-in-react.html'
- 'docs/thinking-in-react-zh-CN.html'

Naszym zdaniem React dostarcza pierwszorzędnych narzędzi do budowy dużych, szybkich aplikacji webowych. Znakomicie sprawdza się, na przykład, w naszych zastosowaniach na Facebooku i w Instagramie.

Jedną z wielu zalet Reacta jest to, jak praca z tą biblioteką uczy cię myśleć o aplikacjach, które tworzysz. Poniżej przybliżymy ci proces myślący towarzyszący budowie przykładowego programu. Będzie to tabela z danymi o produktach z funkcją wyszukiwania, w całości zbudowana w Reakcie.

Zacznij od projektu

Załóżmy, że mamy już gotowy interfejs JSON API oraz projekt graficzny, który wygląda następująco:

Search...	
<input type="checkbox"/> Only show products in stock	
Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99

Projekt graficzny

Nasz interfejs JSON API dostarcza następujących informacji:

[
];

Krok 1: Podziel interfejs użytkownika na zhierarchizowany układ komponentów

W pierwszej kolejności zakreśl na projekcie wszystkie komponenty (i podkomponenty) oraz nadaj im nazwy. Jeśli współpracujesz z zespołem designerów, możliwe, że oni zrobili to już za ciebie. Koniecznie skontaktuj się z nimi. Nazwy komponentów reactowych często biorą się z nazw nadanych warstwom w Photoshopie.

Skąd wiadomo, co powinno być komponentem? Zastosuj te same metody, których używamy tworząc nowe funkcje lub obiekty. Jedną z takich metod jest [Zasada jednej odpowiedzialności](#), zgodnie z którą każdy komponent powinien być odpowiedzialny za tylko jedną rzecz. Jeśli komponent nie spełnia tej zasady i odpowiada za więcej rzeczy, należy go rozbić na kilka mniejszych komponentów.

Model danych wyświetlanych użytkownikowi często odpowiada modelowi zawartemu w plikach JSON. Dlatego jeśli właściwie skonstruujesz swój model, twój interfejs użytkownika (a co za tym idzie także twój układ komponentów) zostanie właściwie zmapowany. Wiąże się to z faktem, że interfejsy użytkownika i modele danych zwykle stosują się do tych samych zasad *architektury informacji*. Wszystko to zaś oznacza, że zadanie podziału interfejsu użytkownika na komponenty jest zwykle zadaniem dziecinnie prostym. Po prostu podziel go tak, aby jednemu elementowi twojego modelu danych odpowiadał jeden komponent.



Diagram pokazujący zagnieżdżenie komponentów

Zwróć uwagę, że nasza prosta aplikacja składa się z pięciu komponentów. Dane, za które odpowiedzialne są poszczególne komponenty, zaznaczyliśmy kursywą. Numery umieszczone na obrazku odpowiadają numerom poniżej.

1. **FilterableProductTable** (**pol. tabela produktów z wyszukiwaniem; pomarańczowy**): mieszczą się w nim wszystkie pozostałe komponenty
2. **SearchBar** (**pol. pasek wyszukiwania; niebieski**): odbiera wpisane przez użytkownika słowo lub frazę (*szukana fraza*)
3. **ProductTable** (**pol. tabela produktów; zielony**): wyświetla i filuluje *kolekcję danych* na podstawie *szukanej frazy*
4. **ProductCategoryRow** (**pol. wiersz rodzaju produktu; turkusowy**): wyświetla nagłówek dla każdego *rodzaju produktów*
5. **ProductRow** (**pol. wiersz produktu; czerwony**): wyświetla wiersz dla każdego *produkту*

Zauważ, że nagłówek naszej **ProductTable** (zawierający nazwy kolumn "Name" i "Price") nie jest osobnym komponentem, chociaż mógłby nim być. W tym przypadku jest to bardziej kwestią naszych indywidualnych preferencji niż zasada ogólna dla tego typu elementów. W naszej przykładowej aplikacji uznaliśmy ten nagłówek za integralną część komponentu **ProductTable**, ponieważ wyświetlany jest razem z *danymi zebraonymi*, a wyświetlanie *danego zebranych* jest odpowiedzialnością **ProductTable**. Jeśli jednak element ten miałby się w naszej aplikacji rozrosnąć (tzn. gdybyśmy mieli dodać do niego funkcję sortowania), jak najbardziej wskazane byłoby zrobienie z niego osobnego komponentu **ProductTableHeader**.

Teraz, kiedy już określiliśmy, które z elementów projektu graficznego mają być komponentami, ułożymy je w odpowiedniej hierarchii. Nie jest to zbyt trudne. Komponenty występujące wewnątrz innych komponentów przedstawimy w naszej hierarchii jako komponenty potomne.

- FilterableProductTable
 - SearchBar
 - ProductTable
 - ProductCategoryRow
 - ProductRow

Krok 2: Zbuduj wersję statyczną w Reakcie

The screenshot shows a code editor interface with tabs for HTML, CSS, Babel, and Result. The Babel tab is selected, displaying the following code:

```

class ProductCategoryRow extends React.Component {
  render() {
    const category = this.props.category;
    return (
      <tr>
        <th colSpan="2">
          {category}
        </th>
      </tr>
    );
  }
}

class ProductRow extends React.Component {
  render() {
    const product = this.props.product;
    const name = product.stocked ?
      product.name :
      <span style={{color: 'red'}}>
        {product.name}
      </span>;
    return (
      <tr>
        <td>{name}</td>
        <td>{product.price}</td>
      </tr>
    );
  }
}

```

Below the code editor, there are buttons for Resources and View Compiled.

Skoro wiemy już, jak wygląda hierarchia naszych komponentów, możemy zacząć ją wdrażać. Budowę aplikacji najłatwiej jest zacząć od wersji statycznej, tzn. takiej, która wyrenderuje interfejs użytkownika na podstawie naszego modelu danych, ale nie będzie zawierała żadnych elementów interaktywnych. Dobrze jest rozdzielić te procesy, ponieważ budowa wersji statycznej wymaga więcej pisania niż myślenia, podczas gdy dodawanie interaktywności wymaga więcej myślenia niż pisania. Za chwilę zobaczyś dlaczego.

Aby zbudować statyczną wersję aplikacji, która wyrenderuje nasz model danych, musimy stworzyć komponenty, które będą korzystać z innych komponentów i przekazywać dane za pomocą *atributów* (ang. *props*). Atrybuty umożliwiają przekazywanie danych z komponentu nadrzędnego do komponentu potomnego. Jeśli wiesz już, czym jest *stan* w Reakcie, powstrzymaj się przed zastosowaniem go tutaj. **Nie należy używać stanu** do budowy statycznych wersji aplikacji. Stan wiąże się wyłącznie z interaktywnością, tzn. danymi zmieniającymi się w czasie.

Krok 3: Określ minimalne (ale kompletne) odzworowanie stanu interfejsu użytkownika

Aby interfejs użytkownika mógł zawierać elementy interaktywne, musimy mieć możliwość dokonywania zmian w modelu danych, na którym opiera się nasza aplikacja. W Reakcie jest to bardzo łatwe dzięki **stanowi**.

Poprawna budowa aplikacji wymaga w pierwszej kolejności określenia minimalnego zestawu danych dla stanu aplikacji. Kluczowa jest tutaj reguła **DRY: Don't Repeat Yourself** (pol. *Nie powtarzaj się*). Zdecyduj, jak ma wyglądać najprostsze możliwe odzworowanie stanu aplikacji, a wszystko inne generuj dopiero wtedy, gdy pojawi się taka potrzeba. Przykładowo, jeśli tworzysz aplikację, która ma zarządzać "Listą rzeczy do zrobienia", zachowaj "pod ręką" jedynie tablicę z rzeczami do zrobienia; nie ma potrzeby tworzenia osobnej zmiennej stanu przechowującej liczbę tych rzeczy. Kiedy zachodzi potrzeba wyrenderowania liczby rzeczy do zrobienia, po prostu pobierz długość tablicy.

Przyjrzyjmy się wszystkim rodzajom informacji w naszej przykładowej aplikacji. Mamy tutaj:

- Początkową listę produktów
- Frazę wyszukiwania podaną przez użytkownika
- Wartość odznaczonego pola
- Listę produktów spełniających kryteria wyszukiwania

Aby zdecydować, która z powyższych informacji zalicza się do stanu, w przypadku każdej z nich musimy zadać sobie trzy pytania:

1. Czy informacja ta jest przekazywana za pomocą atrybutu? Jeśli tak, to prawdopodobnie nie wchodzi w skład stanu.
2. Czy informacja ta pozostanie niezmienna na przestrzeni czasu? Jeśli tak, to prawdopodobnie nie wchodzi w skład stanu.
3. Czy informację tę można wygenerować na podstawie innego stanu lub atrybutu w danym komponencie. Jeśli tak, to nie należy zaliczyć jej do stanu.

Początkowa lista produktów jest przekazywana jako atrybut, zatem nie jest stanem. Wyszukiwana fraza i wartość odznaczonego pola wydają się wchodzić w skład stanu, ponieważ mogą ulegać zmianom i nie da się ich w żaden sposób wygenerować. Jeśli chodzi o listę produktów spełniających kryteria wyszukiwania, to nie jest ona stanem, ponieważ może być wygenerowana na podstawie wyszukiwanej frazy i wartości odznaczonego pola.

Zatem ostatecznie nasz stan przedstawia się następująco:

- Faza wyszukiwania podana przez użytkownika
- Wartość zaznaczonego pola

Mamy zatem określony minimalny zestaw danych stanu aplikacji. Teraz musimy określić, który z naszych komponentów ulega zmianom, czyli do którego komponentu *należy stan*.

Pamiętaj: Dane w Reakcie płyną tylko w jedną stronę - z góry w dół hierarchii komponentów. Przynależność stanu do danego komponentu nie jest sprawą oczywistą i często przysparza problemów na początku pracy z Reactem. Aby to dobrze zrozumieć, postępuj zgodnie z następującymi wskazówkami:

Dla każdego elementu stanu w twojej aplikacji:

- Określ każdy komponent, który coś renderuje na podstawie danego elementu stanu.
- Znajdź wspólnego właściciela stanu (tzn. komponent znajdujący się w hierarchii wyżej od wszystkich pozostałych komponentów wykorzystujących dany element stanu).
- Stan powinien należeć do wspólnego właściciela lub po prostu do innego komponentu znajdującego się wyżej w hierarchii komponentów.
- Jeśli trudno ci określić właściwe miejsce umieszczenia stanu, stwórz nowy komponent, którego jedynym celem będzie przechowywanie stanu. Ulokuj go w hierarchii komponentów gdzieś powyżej komponentu będącego wspólnym właścicielem.

Zastosujmy tę strategię w pracy nad naszą aplikacją:

- `ProductTable` wymaga stanu, aby filtrować listę produktów. `SearchBar` wymaga stanu, aby wyświetlać wyszukiwaną frazę i wartość zaznaczonego pola
- Wspólnym właścicielem jest `FilterableProductTable`.
- Sensownym rozwiązaniem jest umieszczenie wyszukiwanej frazy i wartości zaznaczonego pola w `FilterableProductTable`

No dobra. Zatem ustaliliśmy, że stan umieścimy w `FilterableProductTable`. Teraz do konstruktora tego komponentu dodajemy właściwość instancji `this.state = {filterText: '', inStockOnly: false}`, aby ustalić początkowy stan naszej aplikacji. Następnie za pomocą atrybutów podajemy `filterText` oraz `inStockOnly` do komponentów `ProductTable` i `SearchBar`. Na końcu użyjemy tych atrybutów, aby przefiltrować wiersze `ProductTable` i ustawić wartość pola formularza w `SearchBar`.

Teraz widać już, jak będzie działała nasza aplikacja: ustawiamy `filterText` na ciąg znaków `"ball"` i odświeżamy aplikację. Nasza tabela danych poprawnie wyświetla nowe informacje.

Jeśli spróbujesz wpisać coś do paska wyszukiwania albo zaznaczyć pole wyboru w poprzedniej wersji naszego przykładu (krok 4), React zignoruje dostarczone przez ciebie dane. Jest to działanie zamierzone, które wynika stąd, że wartość atrybutu `value` dla elementu `input` ustawiliśmy jako zawsze równą stanowi `state` podanemu z komponentu `FilterableProductTable`.

Zastanówmy się nad tym, co chcemy żeby się działo. Chcemy, aby stan aktualizował się i odzwierciedlał treść formularza za każdym razem, kiedy użytkownik dokona w formularzu zmian. Ponieważ komponenty powinny aktualizować jedynie własny stan, `FilterableProductTable` poda funkcję zwrotną (ang. *callback*) do paska wyszukiwania `SearchBar`, która ta funkcja zostanie wywołana przy każdej aktualizacji stanu. Jeśli chcemy być o tym za każdym razem poinformowani, możemy dodać zdarzenie `onChange` do elementów naszego formularza. Funkcje zwrotne podane przez `FilterableProductTable` wywołają `setState()` i stan aplikacji zostanie zaktualizowany.

Może wydawać się, że powyższy proces jest skomplikowany, ale w rzeczywistości to tylko kilka linijek kodu, a przepływ danych w całej aplikacji jest naprawdę jawnym i łatwym do prześledzenia.

To byłoby na tyle

Mamy nadzieję, że niniejszy przewodnik przybliżył ci myślenie reactowe, tzn. główne zasady, którymi kierujemy się tworząc komponenty i aplikacje z użyciem React.js. Być może stosowanie tej biblioteki wymaga pisania większej ilości kodu niż inne znane ci biblioteki i frameworki, pamiętaj jednak, że kod czyta się znacznie częściej niż tworzy, a czytanie kodu napisanego w Reakcie nie przysparza problemów ze względu na jego modularność i przejrzystość. Zalety tej przejrzystości i modularności na pewno docenisz tworząc duże biblioteki komponentów. Natomiast wielokrotne stosowanie gotowych kawałków kodu zaoszczędzi ci wiele pracy.
:)

Sprawdzenie typów z wykorzystaniem PropTypes

Uwaga:

Z wersją Reacta v15.5 `React.PropTypes` zostało przeniesione do innej paczki. Zamiast importować z paczki Reacta, używaj [biblioteki prop-types](#).

Dla ułatwienia migracji przygotowaliśmy [skrypt codemod](#).

Wraz ze wzrostem rozmiaru twojej aplikacji, dzięki sprawdzaniu typów możesz wyłapać więcej błędów. W niektórych aplikacjach możesz korzystać z rozszerzeń JavaScriptu do sprawdzenia typów w całej aplikacji, takich jak [Flow](#) lub [TypeScript](#). Nawet jeśli z nich nie korzystasz, możesz skorzystać z mechanizmu sprawdzania typów wbudowanego w Reacta. Aby rozpocząć sprawdzanie typów właściwości w komponencie, możesz dodać do komponentu specjalną właściwość `propTypes`.

```
import PropTypes from 'prop-types';

class Greeting extends React.Component {
  render() {
    return (
      <h1>Witaj, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

W powyższym przykładzie użyliśmy komponentu klasowego, jednak podobna zasada obowiązuje w przypadku komponentów funkcyjnych oraz komponentów tworzonych przy pomocy `React.memo` lub `React.forwardRef`.

`PropTypes` eksportuje validatory, które mogą być używane do sprawdzania poprawności danych wejściowych. W tym przypadku wykorzystujemy `PropTypes.string`. Kiedy wartość przekazanej właściwości będzie nieprawidłowego typu, zostanie wyświetcone ostrzeżenie w konsoli javascriptowej. Ze względu na wydajność, `propTypes` są sprawdzane tylko w trybie deweloperskim.

PropTypes

Oto przykład dokumentujący różne dostarczone validatory:

```
import PropTypes from 'prop-types';

MyComponent.propTypes = {
  // Możesz zadeklarować, że właściwość będzie określonego typu javascriptowego.
```

```

// Domyślnie, wszystkie są opcjonalne.
optionalArray: PropTypes.array,
optionalBool: PropTypes.bool,
optionalFunc: PropTypes.func,
optionalNumber: PropTypes.number,
optionalObject: PropTypes.object,
optionalString: PropTypes.string,
optionalSymbol: PropTypes.symbol,

// Wszystko, co może być wyrenderowane: liczby,łańcuchy znaków, elementy czy
// tablice
// (lub fragmenty) zawierające te typy.
optionalNode: PropTypes.node,

// Element reaktywny.
optionalElement: PropTypes.element,

// Typ komponentu reaktywnego (np. MyComponent).
optionalElementType: PropTypes.elementType,

// Możesz także zadeklarować właściwość, która będzie instancją klasy.
// Wykorzystujemy do tego operator instanceof z JavaScriptu.
optionalMessage: PropTypes.instanceOf(Message),

// Możesz upewnić się, czy właściwość jest ograniczona do określonych wartości.
optionalEnum: PropTypes.oneOf(['News', 'Photos']),

// Właściwość może mieć też wiele typów.
optionalUnion: PropTypes.oneOfType([
  PropTypes.string,
  PropTypes.number,
  PropTypes.instanceOf(Message)
]),

// Tablica zawierająca elementy określonego typu.
optionalArrayOf: PropTypes.arrayOf(PropTypes.number),

// Obiekt zawierający wartości określonego typu.
optionalObjectOf: PropTypes.objectOf(PropTypes.number),

// Obiekt zawierający określone pola.
optionalObjectWithShape: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
}),

// Obiekt zawierający tylko wskazane pola.
optionalObjectWithStrictShape: PropTypes.exact({
  name: PropTypes.string,
  quantity: PropTypes.number
}),

// Możesz dodać do każdego z powyższych `isRequired`,
// aby sprawdzić, czy podana właściwość została zdefiniowana.
requiredFunc: PropTypes.func.isRequired,

// Wymagana wartość dowolnego typu danych.
requiredAny: PropTypes.any.isRequired,

// Możesz też utworzyć niestandardowy validator. Powinien on zwracać obiekt
`Error`,
// jeśli sprawdzenie zakończy się niepowodzeniem. Nie powinien wywoływać

```

```

`console.warn(
  // ani rzucać wyjątku, ponieważ nie będzie działał on wewnątrz `oneOfType`.
  customProp: function(props, propName, componentName) {
    if (!/matchme/.test(props[propName])) {
      return new Error(
        'Niepoprawna właściwość `' + propName + '` przekazana do ' +
        ' `' + componentName + '`. Weryfikacja zakończona niepowodzeniem.'
      );
    }
  },
  // Możesz także przekazać niestandardowy validator do `arrayOf` i `objectOf`.
  // Powinien on zwracać obiekt `Error`, jeśli sprawdzenie zakończy się
  niepowodzeniem.
  // Validator będzie wywoływany dla każdego klucza w tablicy lub obiekcie.
  // Pierwsze dwa argumenty validatorsa to walidowana tablica lub obiekt oraz klucz
  bieżącego elementu.
  customArrayProp: PropTypes.arrayOf(function(propValue, key, componentName,
  location, propFullName) {
    if (!/matchme/.test(propValue[key])) {
      return new Error(
        'Niepoprawna właściwość `' + propFullName + '` przekazana do ' +
        ' `' + componentName + '`. Weryfikacja zakończona niepowodzeniem.'
      );
    }
  })
);
```

```

## Wymaganie dokładne jednego potomka

Wykorzystując `PropTypes.element` możesz sprawdzić, czy do komponentu przekazano dokładnie jednego potomka.

```

import PropTypes from 'prop-types';

class MyComponent extends React.Component {
 render() {
 // Musi zawierać dokładnie jeden element. W przeciwnym wypadku zostanie
 // wyświetlone ostrzeżenie.
 const children = this.props.children;
 return (
 <div>
 {children}
 </div>
);
 }
}

MyComponent.propTypes = {
 children: PropTypes.element.isRequired
};
```

```

Domyślne wartości właściwości

Możesz zdefiniować domyślne wartości dla właściwości przez przypisanie specjalnej właściwości `defaultProps`:

```

class Greeting extends React.Component {
  render() {
    return (
      <h1>Witaj, {this.props.name}</h1>
    );
  }
}

// Definiuje domyślne wartości dla właściwości:
Greeting.defaultProps = {
  name: 'obcy'
};

// Renderuje "Witaj, obcy":
const root = ReactDOM.createRoot(document.getElementById('example'));
root.render(<Greeting />);

```

Od wersji ES2022 możesz także zadeklarować `defaultProps` jako statyczne pole komponentu klasowego. Po więcej informacji na ten temat zatrzymaj się na [dokumentacji dot. publicznych statycznych pól klasy](#). Ta nowoczesna składnia do działania w starszych przeglądarkach będzie wymagać dodatkowej komplikacji.

```

class Greeting extends React.Component {
  static defaultProps = {
    name: 'obcy'
  }

  render() {
    return (
      <div>Witaj, {this.props.name}</div>
    )
  }
}

```

Właściwość `defaultProps` posłuży do zapewnienia wartości dla `this.props.name`, jeśli nie zostanie ona określona przez komponent nadziedzony. Sprawdzanie typu `propTypes` następuje po rozwiązaniu `defaultProps`, więc sprawdzanie typu będzie miało zastosowanie także do `defaultProps`.

Komponenty funkcyjne

Jeśli w swojej aplikacji regularnie korzystasz z komponentów funkcyjnych, musisz dokonać małych zmian, aby `PropTypes` zaczęły działać.

Załóżmy, że masz taki oto komponent:

```

export default function HelloWorldComponent({ name }) {
  return (
    <div>Witaj, {name}</div>
  )
}

```

Aby dodać do niego `PropTypes`, zadeklaruj go osobno przed wyeksportowaniem:

```

function HelloWorldComponent({ name }) {
  return (
    <div>Witaj, {name}</div>
  )
}

```

```
}
```

```
export default HelloWorldComponent
```

Teraz możesz dodać PropTypes bezpośrednio do `HelloWorldComponent`:

```
import PropTypes from 'prop-types'

function HelloWorldComponent({ name }) {
  return (
    <div>Witaj, {name}</div>
  )
}

HelloWorldComponent.propTypes = {
  name: PropTypes.string
}

export default HelloWorldComponent
```

Komponenty niekontrolowane

W większości przypadków zalecamy stosowanie [komponentów kontrolowanych](#) do implementacji formularzy. W komponencie kontrolowanym, dane formularza są obsługiwane przez komponent reactowy. Alternatywą są komponenty niekontrolowane, w których dane formularza są obsługiwane przez sam DOM.

Aby stworzyć komponent niekontrolowany, zamiast pisać funkcję obsługującą każdą zmianę stanu, możesz [użyć właściwości ref](#), aby uzyskać wartości formularza z DOM.

Na przykład, ten kod akceptuje pojedynczą nazwę w komponencie niekontrolowanym:

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

  handleSubmit(event) {
    alert('Podano następujące imię: ' + this.input.current.value);
    event.preventDefault();
  }

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

[Przetestuj kod na CodePen](#)

Ponieważ komponent niekontrolowany zachowuje źródło prawdy w DOM, czasami łatwiej jest zintegrować kod reactowy z kodem niereactowym, gdy używa się komponentów niekontrolowanych. Użycie komponentów niekontrolowanych pozwala zaimplementować formularz "na szybko" dzięki mniejszej ilości kodu. Zwykle jednak powinno używać się komponentów kontrolowanych.

Jeśli nadal nie jest jasne, jakiego rodzaju komponentu należy użyć w konkretnej sytuacji, pomocny może okazać się [ten artykuł o kontrolowanych i niekontrolowanych polach formularza](#).

Wartości domyślne

W reactowym cyklu życia renderowania, wartość atrybutu `value` przypisanego do elementów formularza zastąpi wartość w DOM. W przypadku komponentu niekontrolowanego często zależy nam, aby to React określił wartość początkową, ale by kolejne aktualizacje pozostały niekontrolowane. Aby obsłużyć ten przy-

padek, zamiast atrybutu `value` można podać atrybut `defaultValue`. Warto jednak pamiętać, że zmiana wartości `defaultValue` po tym, jak komponent zostanie już zamontowany, nie będzie miała wpływu na to, co trafi do DOM.

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

Podobnie `<input type="checkbox">` i `<input type="radio">` obsługują atrybut `defaultChecked`, a `<select>` i `<textarea>` obsługują `defaultValue`.

Znacznik `input` dla plików

W HTML-u `<input type="file">` pozwala użytkownikowi wybrać z pamięci urządzenia jeden lub więcej plików, które mają zostać przesłane na serwer lub obsłużone przez JavaScript za pomocą [interfejsu File API](#).

```
<input type="file" />
```

W Reakcie `<input type="file" />` jest zawsze komponentem niekontrolowanym, ponieważ jego wartość może ustawić tylko użytkownik - nie można jej ustawić programowo.

Do interakcji z plikami należy używać interfejsu File API. Poniższy przykład pokazuje, jak utworzyć [odwołanie do węzła DOM](#), aby uzyskać dostęp do plików za pomocą procedury obsługi wysłania formularza:

```
embed:uncontrolled-components/input-type-file.js
```

Komponenty sieciowe

React i [komponenty sieciowe](#) (ang. *Web Components*) zostały stworzone do rozwiązywania różnych problemów. Komponenty sieciowe zapewniają silną enkapsulację w celu wielokrotnego użytku, a React jest deklaratywną biblioteką stworzoną do synchronizacji danych z modelem DOM. Te dwa cele się dopełniają. Jako deweloper możesz wykorzystywać Reacta w swoich komponentach sieciowych lub wykorzystywać komponenty sieciowe w Reakcie.

Większość osób korzystających z Reacta nie używa komponentów sieciowych, ale ty możesz chcieć to zrobić, zwłaszcza jeśli używasz zewnętrznych komponentów UI napisanych z wykorzystaniem komponentów sieciowych.

Używanie komponentów sieciowych w Reakcie

```
class HelloMessage extends React.Component {
  render() {
    return <div> Witaj, <x-search>{this.props.name}</x-search>!</div>;
  }
}
```

Uwaga:

Komponenty sieciowe często udostępniają imperatywne API. Na przykład: komponent sieciowy `video` może udostępniać funkcje `play()` i `pause()`. Aby skorzystać z imperatywnego API komponentu sieciowego, możesz potrzebować użyć referencji (`ref`), aby nawiązać bezpośredni kontakt z węzłem drzewa DOM. Jeśli korzystasz z komponentów sieciowych innych osób, najlepszym rozwiązaniem będzie utworzenie reactowego komponentu służącego jako nakładka dla komponentu sieciowego.

Zdarzenia emitowane przez komponent sieciowy mogą niewłaściwie propagować się przez reactowe drzewo renderowania. Aby obsłużyć te zdarzenia w komponencie reactowym, konieczne może być ręczne dodanie procedur obsługi zdarzeń.

Często mylonym aspektem komponentów sieciowych jest to, że korzystają z atrybutu "class", a nie "className".

```
function BrickFlipbox() {
  return (
    <brick-flipbox class="demo">
      <div>front</div>
      <div>back</div>
    </brick-flipbox>
  );
}
```

Używanie Reacta w komponentach sieciowych

```
class XSearch extends HTMLElement {
  connectedCallback() {
    const mountPoint = document.createElement('span');
    this.attachShadow({ mode: 'open' }).appendChild(mountPoint);

    const name = this.getAttribute('name');
    const url = 'https://www.google.com/search?q=' + encodeURIComponent(name);
    const root = ReactDOM.createRoot(mountPoint);
    root.render(<a href={url}>{name}</a>);
  }
}
customElements.define('x-search', XSearch);
```

Uwaga:

Ten kod **nie** zadziała, jeśli dokonujesz transformacji klas używając narzędzia Babel. Zobacz [dyskusję](#) na ten temat, aby dowiedzieć się więcej. Aby rozwiązać ten problem, użyj [custom-elements-es5-adapter](#) przed załadowaniem komponentów sieciowych.

Colophon

This book is created by using the following sources:

- React - Polski
- GitHub source: [reactjs/pl.reactjs.org/content](https://reactjs.pl.reactjs.org/content)
- Created: 2022-11-28
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- Viewer: <https://read-html-download-pdf.vercel.app/>
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>
- Viewer repo: <https://github.com/shinokada/read-html-download-pdf>