O'REILLY®

Helion

React

Opanuj do perfekcji tworzenie aplikacji internetowych nowej generacji

Tejas Kumar

Tytuł oryginału: Fluent React: Build Fast, Performant, and Intuitive Web Applications

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-1634-0

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/reacto
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

Przedmowa **11**

Wprowadzenie **13**

1. Podstawy **17**
Skąd bierze się popularność Reacta? 17
Świat aplikacji internetowych przed pojawieniem się Reacta 18
jQuery 26
Backbone.js 29
Knockout 34
AngularJS 38
Poznaj Reacta 43
Wartość propozycji Reacta 44
Wydanie Reacta 50
Architektura Flux 51
Zalety architektury Flux 52
Skąd tak duże znaczenie Reacta? 53
Podsumowanie 54
Pytania 54
Co dalej? 54

2. JSX **55**
JavaScript XML? 55
Zalety kodu w JSX 57
Wady kodu JSX 58
Mechanizm działania składni JSX 59
Jak działa kod? 59
Rozszerzanie składni JavaScriptu za pomocą JSX 63
JSX Pragma 64
Wyrażenia 65

Podsumowanie 66
Pytania 66
Co dalej? 66

3. Wirtualny model DOM 67
Wprowadzenie do wirtualnego modelu DOM 67
Rzeczywisty model DOM 68
Problemy podczas pracy z rzeczywistym modelem DOM 72
Fragmenty dokumentu 80
Jak działa wirtualny model DOM? 81
Elementy Reacta 82
Wirtualny kontra rzeczywisty model DOM 85
Efektywne uaktualnienia 87
Podsumowanie 89
Pytania 90
Co dalej? 90

4. Rekoncyliacja 91
Rekoncyliacja 91
Przetwarzanie wsadowe 93
Poprzednie rozwiązanie 94
Stary stos rekoncyliacji 94
Mechanizm rekoncyliacji Fiber 97
Fiber jako struktura danych 98
Podwójne buforowanie 100
Proces rekoncyliacji Fiber 100
Podsumowanie 106
Pytania 106
Co dalej? 107

5. Zaawansowane wzorce i najczęściej pojawiające się pytania 108
Memoizacja za pomocą wywołania React.memo() 108
Nabycie biegłości w posługiwaniu się wywołaniem React.memo() 112
Wykorzystujące technikę memoizacji komponenty, które nadal są ponownie generowane 112
To zalecenie, a nie reguła 115
Memoizacja za pomocą useMemo 120
Użycie zaczepu useMemo uznawane za szkodliwe 122
Zapomnij o tym wszystkim 128
Wczytywanie z opóźnieniem 129
Większa kontrola nad interfejsem użytkownika dzięki komponentowi Suspense 132

Zaczepy useState i useReducer 134
Immer i ergonomia 137
Zaawansowane wzorce 139
Komponenty prezentacyjne/kontenery 140
Komponent wyższego rzędu 141
Właściwości generowania 148
Właściwości kontrolne 149
Kolekcje właściwości 150
Komponent złożony 153
Reduktor stanu 156
Podsumowanie 158
Pytania 158
Co dalej? 159

6. React po stronie serwera 160
Ograniczenia generowania po stronie klienta 160
SEO 160
Wydajność działania 161
Zapewnienie bezpieczeństwa 164
Popularność generowania po stronie serwera 166
Zalety generowania po stronie serwera 166
Wypełnianie 167
Wypełnianie uznawane za szkodliwe 168
Tworzenie serwera generującego 169
Ręczne dodawanie serwera generującego do aplikacji działającej tylko po stronie klienta 170
Wypełnianie 172
API Reacta do generowania po stronie serwera 172
renderToString() 172
renderToPipeableStream() 175
renderToReadableStream() 185
Kiedy używać poszczególnych API? 186
Nie twórz własnej implementacji 188
Podsumowanie 190
Pytania 191
Co dalej? 191

7. Współbieżność 192
Problem z generowaniem synchronicznym 193
Architektura Fiber 194
Szeregowanie i odkładanie uaktualnień 194

Dokładne omówienie mechanizmów 197
Zarządca procesów 197
Tory generowania 201
Na czym polega działanie toru generowania? 202
Przetwarzanie torów 204
Etap zatwierdzania 204
useTransition 205
Prosty przykład 205
Przykład zaawansowany 206
Dokładniejsze omówienie mechanizmu 208
useDeferredValue 208
Przeznaczenie zaczepu useDeferredValue 209
Kiedy używać useDeferredValue? 211
Kiedy nie używać useDeferredValue? 212
Problemy związane z generowaniem współbieżnym 213
Tearing 213
Podsumowanie 220
Pytania 222
Co dalej? 222

8. Frameworki 223
Dlaczego potrzebny jest framework? 223
Generowanie po stronie serwera 226
Routing 227
Pobieranie danych 229
Zalety stosowania frameworków 232
Wady stosowania frameworków 232
Popularne frameworki Reacta 233
Remix 234
Next.js 241
Wybór frameworka 248
Określenie potrzeb projektu 248
Next.js 249
Remix 250
Wady i zalety 250
Wrażenia programisty 251
Wydajność działania 251
Podsumowanie 252
Pytania 253
Co dalej? 253

9. Komponenty serwerowe Reacta 254
Korzyści 255
Generowanie po stronie serwera 256
Pod maską 258
Wprowadzanie uaktualnień 265
Niuanse 270
Reguły dotyczące komponentów serwerowych 271
Serializacja ma znaczenie 271
Brak skutecznych zaczepów 272
Stan nie jest stanem 272
Komponenty klienckie nie mogą importować komponentów serwerowych 272
Komponenty klienckie nie są złe 274
Akcje serwerowe 274
Formularze i mutacje 274
Poza formularzami 275
Przyszłość komponentów serwerowych Reacta 276
Podsumowanie 277
Pytania 277
Co dalej? 277

10. Alternatywy Reacta 279
Vue.js 279
Sygnały 281
Prostota 281
Angular 282
Wykrywanie zmian 282
Sygnały 282
Svelte 283
Runy 284
SolidJS 287
Qwik 288
Najczęściej spotykane wzorce 290
Architektura oparta na komponencie 290
Składnia deklaratywna 290
Uaktualnienia 291
Metody cyklu życiowego 291
Ekosystem i narzędzia 291
React nie jest reaktywny 292
Przykład — Wartości zależne 295
Przyszłość Reacta 296
React Forget 297

Podsumowanie 299
Pytania 299
Co dalej? 300

11. Zakończenie 301
Wnioski 301
Oś czasu 303
Mechanizm kryjący się za magią 304
Zagadnienia zaawansowane 304
Bądź na bieżąco 305

ROZDZIAŁ 1.

Podstawy

Rozpocznę od ważnego zastrzeżenia: biblioteka React została opracowana w taki sposób, aby mógł jej używać każdy. Dlatego ktoś może nigdy nie sięgnąć po tę książkę i mimo to bez żadnych problemów korzystać z Reacta. W książce dokładnie omówiłem bibliotekę dla tych, którzy chcą dowiedzieć się więcej o kryjących się za nią mechanizmach, zaawansowanych wzorcach użycia oraz najlepszych praktykach. Większy pożytek przynosi poznanie sposobu działania Reacta niż nauka, jak z niego korzystać. Na rynku jest dostępnych wiele różnych książek, których autorzy obrali sobie za cel nauczenie czytelników sposobu korzystania z Reacta z perspektywy użytkownika końcowego. Natomiast moim celem jest umożliwienie Ci poznania Reacta na poziomie autora biblioteki lub frameworka. Zgodnie z tym założeniem będziemy razem zagłębiać się w tematykę Reacta, począwszy od zagadnień ogólnych na poziomie podstawowym. Najpierw omówię podstawy Reacta, a później zagłębię się w szczegóły i dokładnie wyjaśnię sposób działania tej biblioteki.

Z lektury tego rozdziału dowiesz się, dlaczego React w ogóle istnieje, jak działa i jakie problemy pomaga rozwiązywać. Przedstawię początkowe założenia przyświecające twórcom biblioteki oraz jej projekt. Następnie przejdę od jej skromnych początków w Facebooku do praktycznie wszechobecnego zastosowania w aktualnie dostępnych rozwiązaniach. Ten rozdział można po części traktować jako metarozdział (gra słów niezamierzona[1]), ponieważ bardzo ważne jest zrozumienie kontekstu Reacta, zanim będzie można zagłębić się w szczegóły.

Skąd bierze się popularność Reacta?

Jeżeli odpowiedź na to pytanie miałaby być zawarta w jednym słowie, byłoby nim *uaktualnienia*. W początkowym okresie istnienia sieć WWW składała się z wielu statycznych stron internetowych. W celu wyświetlenia zupełnie nowej strony trzeba było wypełnić formularz i kliknąć wysyłający go przycisk. Przez jakiś czas takie rozwiązanie się sprawdzało. Jednak w pewnym momencie te ograniczenia zaczęły mieć poważny wpływ na możliwości internetu. Wraz z opracowywaniem nowych technologii wzrastało także zapotrzebowanie na poprawę wrażeń użytkowników internetu. Jednym z wymagań było natychmiastowe uaktualnianie treści, bez konieczności oczekiwania, aż nowa

[1] Meta to firma będąca właścicielem serwisu Facebook, którego jeden z programistów opracował bibliotekę React — *przyp. tłum.*

strona zostanie wczytana i wygenerowana. Użytkownicy życzyli sobie, aby internet i znajdujące się w nim strony sprawiały wrażenie „żwawszych" i bardziej *natychmiastowych*. Problem polegał jednak na tym, że z wielu powodów te natychmiastowe uaktualnienia były niezwykle trudne do zaimplementowania na *dużą skalę*:

Wydajność działania

Częste uaktualnianie stron internetowych prowadziło do wąskich gardeł wydajności działania, ponieważ taka operacja wymagała od przeglądarki WWW ponownego wygenerowania układu strony internetowej (określano to mianem *reflow*) i jej ponownego wyświetlenia.

Niezawodność

Monitorowanie informacji o stanie i zagwarantowanie ich spójności w rozbudowanej witrynie internetowej okazywało się trudne, ponieważ wiązało się ze śledzeniem tych informacji w wielu miejscach i wymagało zapewnienia ich spójności w tych wszystkich miejscach. Przygotowanie takiego rozwiązania było szczególnie trudne, gdy wiele osób musiało pracować nad tą samą bazą kodu.

Bezpieczeństwo

Konieczne było weryfikowanie całego kodu w HTML i JavaScripcie wstrzykiwanego na stronie internetowej, aby w ten sposób chronić aplikację przed atakami typu XSS (ang. *cross-site scripting*) i CSRF (ang. *cross-site request forgery*).

Aby w pełni zrozumieć i docenić, jak React rozwiązuje te problemy za programistów, trzeba poznać kontekst, w którym został utworzony React, oraz świat aplikacji internetowych istniejący przed pojawieniem się tej biblioteki. Tym się zajmę w następnym podrozdziale.

Świat aplikacji internetowych przed pojawieniem się Reacta

Zanim na rynku pojawiła się biblioteka React, programiści tworzący aplikacje internetowe napotykali ogromne problemy. Konieczne było ustalenie, co zrobić, aby aplikacje były postrzegane jako działające szybko i natychmiastowo. Jednocześnie trzeba było zapewnić możliwość ich skalowania milionom użytkowników w sposób pozwalający na niezawodne i bezpieczne działanie rozwiązania. Na przykład rozważ kliknięcie przycisku: gdy użytkownik kliknie przycisk, interfejs użytkownika ma zostać uaktualniony i odzwierciedlić kliknięcie tego przycisku. To oznacza konieczność uwzględnienia co najmniej czterech różnych stanów, w których może znajdować się interfejs użytkownika:

Przed kliknięciem przycisku

Przycisk znajduje się w swoim stanie domyślnym i nie został kliknięty.

Oczekiwanie po kliknięciu przycisku

Wprawdzie przycisk został kliknięty, ale akcja, którą powinien wykonywać, nie jest jeszcze ukończona.

Kliknięcie przycisku zakończone powodzeniem

Przycisk został kliknięty i akcja, którą powinien wykonywać, jest już ukończona. W tym momencie oczekiwane może być przywrócenie przycisku do jego stanu sprzed kliknięcia. Ewentualnie być może należy zmienić kolor przycisku (na zielony) w celu wskazania na pomyślne zakończenie operacji.

Kliknięcie przycisku zakończone niepowodzeniem

Przycisk został kliknięty, a akcja, którą powinien wykonywać, zakończyła się niepowodzeniem. W tym momencie oczekiwane może być przywrócenie przycisku do jego stanu sprzed kliknięcia. Ewentualnie być może należy zmienić kolor przycisku (na czerwony) w celu wskazania na zakończenie operacji niepowodzeniem.

Po ustaleniu wymienionych stanów konieczne będzie określenie, jak można uaktualnić interfejs użytkownika, aby je odzwierciedlić. Czasami uaktualnienie interfejsu użytkownika będzie wymagało wykonania następujących kroków:

1. Odszukanie przycisku w środowisku hosta (często jest to przeglądarka WWW), korzystając w tym celu z API lokalizującego element, np. `document.querySelector()` lub `document.getElementById()`.
2. Dołączenie do przycisku komponentu nasłuchującego zdarzeń kliknięcia.
3. Przeprowadzenie niezbędnego uaktualnienia stanu w reakcji na zdarzenia.
4. Po usunięciu przycisku ze strony konieczne jest usunięcie również komponentu nasłuchującego zdarzeń kliknięcia oraz uporządkowanie informacji o stanie.

To jest dobry przykład i świetnie sprawdzi się na początek. Załóżmy, że przycisk zawiera etykietę *Polub*, która po jego kliknięciu zmienia wartość na *Polubiony*. W jaki sposób można to zrobić? Trzeba zacząć od utworzenia elementu HTML:

```
<button>Polub</button>
```

Potrzebna będzie możliwość odwołania się do tego przycisku z poziomu kodu w JavaScripcie, więc elementowi `<button>` trzeba przypisać atrybut `id`:

```
<button id="likeButton">Polub</button>
```

Doskonale! Skoro element ma atrybut `id`, można z nim pracować za pomocą JavaScriptu, a tym samym zapewnić interaktywność elementowi. Do takiego elementu można się odnieść za pomocą wywołania `document.getElementById()` i następnie dodać przyciskowi komponent nasłuchujący zdarzeń kliknięcia:

```
const likeButton = document.getElementById("likeButton");
likeButton.addEventListener("click", () => {
  // Dowolne operacje
});
```

W ten sposób przyciskowi został przypisany komponent nasłuchujący zdarzeń kliknięcia. Można więc zdefiniować zadania wykonywane po kliknięciu przycisku. Załóżmy, że kliknięcie ma spowodować uaktualnienie etykiety przycisku do postaci *Polubiony*. To będzie wymagało uaktualnienia kontekstu tekstowego przycisku:

```
const likeButton = document.getElementById("likeButton");
likeButton.addEventListener("click", () => {
  likeButton.textContent = "Polubiony";
});
```

Świetnie! W ten sposób mamy przycisk o etykiecie *Polub*, która po kliknięciu zmienia się na *Polubiony*. Problem polega na tym, że nie można odwrócić tego sposobu działania przycisku. Teraz to poprawimy, aby ponowne kliknięcie przycisku spowodowało zmianę etykiety *Polubiony* na *Polub*. Konieczne jest dodanie do przycisku kolejnych informacji o stanie pozwalających sprawdzić, czy został on kliknięty. W tym celu do elementu przycisku można dodać atrybut `data-liked`:

```
<button id="likeButton" data-liked="false">Polub</button>
```

Nowy atrybut można wykorzystać do sprawdzenia czy przycisk został kliknięty. Kontekst tekstowy przycisku będzie można uaktualniać na podstawie wartości tego atrybutu:

```
const likeButton = document.getElementById("likeButton");
likeButton.addEventListener("click", () => {
  const liked = likeButton.getAttribute("data-liked") === "true";
  likeButton.setAttribute("data-liked", !liked);
  likeButton.textContent = liked ? "Polub" : "Polubiony";
});
```

Po prostu zmieniamy tutaj wartość właściwości `textContent` przycisku. Informacje o jego stanie nie są zapisywane w bazie danych. Normalnie to wymagałoby prowadzenia komunikacji przez sieć z użyciem rozwiązania takiego jak tu:

```
const likeButton = document.getElementById("likeButton");
likeButton.addEventListener("click", () => {
  var liked = likeButton.getAttribute("data-liked") === "true";

  // Komunikacja przez sieć
  var xhr = new XMLHttpRequest();
  xhr.open("POST", "/like", true);
  xhr.setRequestHeader("Content-Type", "application/json;charset=UTF-8");

  xhr.onload = function () {
    if (xhr.status >= 200 && xhr.status < 400) {
      // Sukces!
      likeButton.setAttribute("data-liked", !liked);
      likeButton.textContent = liked ? "Polub" : "Polubiony";
    } else {
      // Wprawdzie udało się nawiązać połączenie z serwerem docelowym, ale zwrócił on komunikat błędu
      console.error("Serwer wygenerował komunikat błędu:", xhr.statusText);
    }
  };

  xhr.onerror = function () {
    // To był pewnego rodzaju błąd połączenia
    console.error("Błąd sieci");
  };

  xhr.send(JSON.stringify({ liked: !liked }));
});
```

Oczywiście używamy obiektu `XMLHttpRequest` i polecenia `var`, aby uwzględnić czas. Biblioteka React została wydana jako oprogramowanie otwartoźródłowe w 2013 roku. Większa liczba API typu `fetch` pojawiła się dopiero w 2015 roku. W okresie między dostępnością obiektu `XMLHttpRequest` i pojawieniem się API `fetch` bardzo często używano biblioteki jQuery, aby w ten sposób nieco uprościć rozwiązanie za pomocą wywołań typu `$.ajax()`, `$.post()` itd.

Gdyby poprzedni fragment kodu miał zostać utworzony dzisiaj, prawdopodobnie jego postać prezentowałaby się następująco:

```
const likeButton = document.getElementById("likeButton");
likeButton.addEventListener("click", () => {
  const liked = likeButton.getAttribute("data-liked") === "true";

  // Komunikacja przez sieć
  fetch("/like", {
    method: "POST",
    body: JSON.stringify({ liked: !liked }),
  }).then(() => {
    likeButton.setAttribute("data-liked", !liked);
    likeButton.textContent = liked ? "Polub" : "Polubiony";
  });
});
```

Bez zbędnych dygresji: w tym fragmencie kodu chodzi o to, że prowadzi on komunikację przez sieć. Co się stanie w sytuacji, gdy wykonanie żądania sieciowego zakończy się niepowodzeniem? Konieczne jest uaktualnienie tekstu przycisku i odzwierciedlenie tego niepowodzenia. W tym celu do elementu przycisku można dodać atrybut `data-failed`:

```
<button id="likeButton" data-liked="false" data-failed="false">Polub</button>
```

Teraz tekst przycisku można uaktualniać na podstawie wartości nowego atrybutu:

```
const likeButton = document.getElementById("likeButton");
likeButton.addEventListener("click", () => {
  const liked = likeButton.getAttribute("data-liked") === "true";

  // Komunikacja przez sieć
  fetch("/like", {
    method: "POST",
    body: JSON.stringify({ liked: !liked }),
  })
    .then(() => {
      likeButton.setAttribute("data-liked", !liked);
      likeButton.textContent = liked ? "Polub" : "Polubiony";
    })
    .catch(() => {
      likeButton.setAttribute("data-failed", true);
      likeButton.textContent = "Niepowodzenie";
    });
});
```

Pozostał jeszcze jeden przypadek do obsłużenia: przetwarzanie aktualnie „polubionego” elementu — chodzi o stan oczekiwania. Aby go modelować w kodzie, do elementu przycisku można dodać następny atrybut, o nazwie `data-pending`. Spójrz na kolejny fragment kodu:

```
<button
  id="likeButton"
  data-pending="false"
  data-liked="false"
  data-failed="false"
>
  Polub
</button>
```

Nowy atrybut pozwala zablokować przycisk, jeśli żądanie sieciowe jest aktualnie przetwarzane. Dzięki temu wiele kliknięć nie spowoduje kolejkowania żądań sieciowych i wystąpienia stanu wyścigu, który ostatecznie mógłby zwiększyć obciążenie serwera. Kod w zmodyfikowanej postaci przedstawia się następująco:

```
const likeButton = document.getElementById("likeButton");
likeButton.addEventListener("click", () => {
  const liked = likeButton.getAttribute("data-liked") === "true";
  const isPending = likeButton.getAttribute("data-pending") === "true";

  likeButton.setAttribute("data-pending", "true");
  likeButton.setAttribute("disabled", "disabled");

  // Komunikacja przez sieć
  fetch("/like", {
    method: "POST",
    body: JSON.stringify({ liked: !liked }),
  })
    .then(() => {
      likeButton.setAttribute("data-liked", !liked);
      likeButton.textContent = liked ? "Polub" : "Polubiony";
      likeButton.setAttribute("disabled", null);
    })
    .catch(() => {
      likeButton.setAttribute("data-failed", "true");
      likeButton.textContent = "Niepowodzenie";
    })
    .finally(() => {
      likeButton.setAttribute("data-pending", "false");
    });
});
```

Można jeszcze zastosować oferujące potężne możliwości techniki określane mianem *debouncing* i *throttling*, aby uniemożliwić użytkownikom wykonywanie zbędnych bądź nadmiarowych akcji.

W tekście wspomniałem o dwóch technikach, a mianowicie debouncingu i throttlingu. Pierwsza z nich powoduje opóźnienie wykonywania funkcji o podany czas, który musi upłynąć od ostatniego wywołania zdarzenia (np. przetwarzanie danych wejściowych rozpocznie się dopiero po zakończeniu ich wprowadzania przez użytkownika). Druga ogranicza możliwość wykonania funkcji do najwyżej jednego razu w przedziale czasu, więc gwarantuje, że funkcja nie będzie wykonywana zbyt często (np. przetwarzanie zdarzeń przewijania w określonych odstępach czasu). Obie wymienione techniki optymalizują wydajność działania poprzez kontrolowanie częstotliwości, z jaką jest wykonywana funkcja.

W obecnej postaci nasz przykładowy przycisk jest niezawodny i może obsługiwać wiele stanów. Mimo to kilka pytań wciąż pozostaje aktualnych:

- Czy atrybut `data-pending` naprawdę jest potrzebny? Czy nie można po prostu sprawdzić, czy przycisk nie został zablokowany? Prawdopodobnie nie można zastosować takiego rozwiązania, ponieważ przycisk mógł zostać zablokowany z różnych powodów: użytkownikowi nie udało się zalogować lub nie ma uprawnień pozwalających na kliknięcie przycisku.
- Czy bardziej sensowne nie byłoby utworzenie atrybutu `data-state` przyjmującego jedną z wartości: `pending`, `liked` lub `unliked`, zamiast tak wielu innych atrybutów danych? Prawdopodobnie nie można zastosować takiego rozwiązania, ponieważ wówczas potrzebna byłaby ogromna konstrukcja typu `switch-case` bądź podobny blok kodu w celu obsłużenia poszczególnych przypadków. Ostatecznie ilość kodu niezbędnego do obsługi obu podejść jest nieporównywalna: niezależnie od rozwiązania będzie ono charakteryzowało się złożonością i rozwlekłością.
- W jaki sposób ten przycisk można przetestować oddzielnie? Czy w ogóle istnieje taka możliwość?
- Dlaczego na początku przycisk został utworzony w kodzie w HTML, a następnie używamy go w kodzie w JavaScripcie? Czy nie byłoby lepszym rozwiązaniem po prostu utworzenie przycisku w JavaScripcie za pomocą wywołania `document.createElement('button')`, a później użycie `document.appendChild(likeButton)`? Takie podejście ułatwiłoby testy i zapewniło większą niezależność kodu. Jednak z drugiej strony oznaczałoby również konieczność monitorowania elementu nadrzędnego, jeśli nie byłby nim `document`. W rzeczywistości być może trzeba byłoby monitorować *wszystkie* elementy nadrzędne na stronie.

React pomaga w rozwiązywaniu tego rodzaju problemów, choć nie wszystkich. Na przykład nie odpowie na pytanie, jak podzielić stan na osobne opcje (`isPending`, `hasFailed` itd.) ani jak używać pojedynczej zmiennej stanu (takiej jak `state`). To są pytania, na które programiści sami muszą znaleźć odpowiedzi. Jednak React pomaga poradzić sobie z problemem skali: tworzeniem wielu przycisków interaktywnych i uaktualnianiem interfejsu użytkownika w odpowiedzi na zdarzenia. Te zadania wykonuje w sposób nie tylko minimalny i efektywny, ale jednocześnie testowalny, powtarzalny, deklaratywny, wydajny, przewidywalny i niezawodny.

Ponadto dzięki Reactowi stan staje się bardziej przewidywalny, ponieważ biblioteka jest w pełni właścicielem informacji o stanie interfejsu użytkownika i generuje treść na podstawie tych informacji. Mamy więc wyraźny kontrast względem stanu, którego właścicielem jest przeglądarka WWW, i to ona nim zarządza. Wówczas stan może być zawodny ze względu na pewną liczbę czynników, takich jak inne skrypty działające po stronie klienta uruchomione na stronie, zainstalowane rozszerzenia przeglądarki WWW, ograniczenia urządzenia itd.

Wprawdzie omówiony tutaj przykład przycisku *Polub* jest bardzo prosty, ale świetnie nadaje się na początek. Dotychczas pokazałem, jak dzięki użyciu kodu w JavaScripcie można zapewnić interaktywność przycisku. Ten proces jednak należy będzie wykonać ręcznie, jeśli ma być zrobiony *dobrze*: trzeba odszukać przycisk w przeglądarce WWW, dodać komponent nasłuchujący zdarzeń, uaktualnić treść tekstową przycisku oraz uwzględnić wiele przypadków skrajnych. To oznacza wiele pracy, a samo rozwiązanie nie skaluje się zbyt dobrze. Co się stanie w sytuacji, gdy na stronie znajduje się wiele przycisków? Co w sytuacji, gdy wiele przycisków musi być interaktywnych? Co w sytuacji, gdy wiele przycisków musi być interaktywnych i trzeba uaktualniać interfejs użytkownika w odpowiedzi

na zdarzenia? Czy trzeba będzie zastosować delegowanie zdarzeń (lub nawet propagację zdarzeń) i dołączać komponent nasłuchujący zdarzeń do znajdujących się wyżej w hierarchii elementów `document`? A może należy dołączać takie komponenty do wszystkich przycisków?

Jak już wspomniałem we wprowadzeniu, w książce przyjąłem założenie, że w pełni rozumiesz następujące zdanie: przeglądarki WWW generują strony internetowe. Strona internetowa to dokument HTML, w którym arkusze stylów CSS odpowiadają za jego wygląd, natomiast JavaScript zapewnia możliwość interaktywnego działania. Takie rozwiązanie sprawdzało się przez dekady i nadal działa. Jednak zbudowanie nowoczesnej aplikacji internetowej przeznaczonej do obsługi znacznej (mam tutaj na myśli miliony) liczby użytkowników z wykorzystaniem wymienionych technologii będzie wymagało dużej abstrakcji, aby rozwiązanie było bezpieczne i niezawodne, a przy tym charakteryzowało się jak najmniejszą awaryjnością. Niestety na podstawie przedstawionego wcześniej przykładu przycisku *Polub* wiemy, że budowa wspomnianej aplikacji internetowej będzie wymagała pewnej pomocy.

Zapoznaj się teraz z innym przykładem, nieco bardziej złożonym niż wcześniej zaprezentowany przycisk *Polub*. Rozpoczynamy od czegoś prostego: listy elementów. Powiedzmy, że mamy listę elementów, do której chcemy dodać nowy. W tym celu można skorzystać z formularza HTML o takiej oto postaci:

```
<ul id="list-parent"></ul>

<form id="add-item-form" action="/api/add-item" method="POST">
  <input type="text" id="new-list-item-label" />
  <button type="submit">Dodaj element</button>
</form>
```

JavaScript zapewnia dostęp do API modelu DOM (ang. *document object model*). Warto w tym miejscu przypomnieć, że DOM to istniejący w pamięci model struktury dokumentu strony internetowej. To po prostu drzewo obiektów, które przedstawia elementy znajdujące się na stronie, i umożliwia pracę z nimi za pomocą kodu w JavaScripcie. Problem polega na tym, że model DOM w urządzeniu użytkownika przypomina obcą planetę: nie ma możliwości sprawdzenia, jaka jest używana przeglądarka WWW, jaka jest konfiguracja i stan sieci, jaki jest używany system operacyjny itd. Wynik? Konieczność utworzenia kodu źródłowego, który będzie odporny na te wszystkie czynniki.

Jak już wcześniej wspomniałem, informacje o stanie aplikacji są trudne do przewidzenia, gdy będą uaktualniane bez użycia jakiegokolwiek mechanizmu rekoncyliacji stanu pozwalającego na monitorowanie informacji. Kontynuujemy przykład listy i przystępujemy do analizy kodu w JavaScripcie odpowiedzialnego za dodanie nowego elementu do listy:

```
(function myApp() {
  var listItems = ["I love", "React", "and", "TypeScript"];
  var parentList = document.getElementById("list-parent");
  var addForm = document.getElementById("add-item-form");
  var newListItemLabel = document.getElementById("new-list-item-label");

  addForm.onsubmit = function (event) {
    event.preventDefault();
    listItems.push(newListItemLabel.value);
    renderListItems();
  };
```

```
  function renderListItems() {
    for (i = 0; i < listItems.length; i++) {
      var el = document.createElement("li");
      el.textContent = listItems[i];
      parentList.appendChild(el);
    }
  }

  renderListItems();
})();
```

Ten fragment kodu został utworzony w sposób jak najbardziej przypominający wczesne aplikacje internetowe. Dlaczego wraz z upływem czasu stał się wadliwy? Przede wszystkim dlatego, że tworzenie w taki sposób aplikacji, które powinny być skalowane, prowadzi do różnych problemów:

Podatność na błędy

Atrybut `onsubmit` elementu `addForm` mógłby został łatwo przepisany przez inny fragment kodu w JavaScripcie działającego na stronie. Wprawdzie można użyć wywołania `addEvent ↪Listener()`, ale to rodzi kilka dodatkowych pytań:

- Gdzie i kiedy należy przeprowadzić operacje porządkowe za pomocą wywołania `removeEventListener()`?
- Czy jeśli nie będzie zachowana ostrożność, na stronie zostanie zakumulowana duża liczba komponentów nasłuchiwania zdarzeń?
- Jakie mogą być tego konsekwencje?
- Czy wobec tego delegowanie zdarzeń będzie dobrym rozwiązaniem?

Nieprzewidywalność

W tym przykładzie źródła prawdy są wymieszane: lista jest przechowywana w postaci tablicy w JavaScripcie, przy czym w celu ukończenia aplikacji bazujemy na istniejących elementach w modelu DOM (np. element o atrybucie `id="list-parent"`). Z powodu takich mieszanych zależności między kodem w JavaScripcie i HTML konieczne jest uwzględnienie kilku kwestii:

- Co można zrobić w sytuacji, gdy pomyłkowo kilka elementów będzie miało ten san atrybut `id`?
- Co można zrobić w sytuacji, gdy element w ogóle nie istnieje?
- Co można zrobić w sytuacji, gdy elementem nie będzie `<ul>`? Czy element listy `<li>` można dołączyć do innego elementu nadrzędnego?
- Co można zrobić w sytuacji, gdy zamiast atrybutu będzie użyta nazwa klasy?

Nasze źródła prawdy są wymieszane (JavaScript i HTML), a tym samym będą zawodne. Znacznie korzystniejsze byłoby istnienie tylko pojedynczego źródła prawdy. Ponadto elementy są przez cały czas dodawane i usuwane z modelu DOM przez kod w JavaScripcie działający po stronie klienta. Jeżeli będziemy polegać na istnieniu określonych elementów, nie ma gwarancji niezawodnego działania aplikacji podczas uaktualnień interfejsu użytkownika. W takim przypadku nasza aplikacja będzie pełna „efektów ubocznych", gdy sukces bądź niepowodzenie będzie zależało od czynnika znajdującego się po stronie użytkownika. React eliminuje ten problem poprzez promowanie modelu inspirowanego programowaniem funkcyjnym, w którym efekty uboczne są wyraźnie oznaczane i izolowane.

Nieefektywność

Wywołanie funkcji `renderListItems()` powoduje sekwencyjne wygenerowanie elementów na ekranie. Każda modyfikacja modelu DOM może być kosztowna (może wymagać wielu obliczeń), zwłaszcza gdy wiąże się z przesunięciem układu oraz ponownym wygenerowaniem i wyświetleniem elementów strony internetowej. Skoro jesteśmy na obcej planecie o nieznanej mocy obliczeniowej, dla ogromnych list ta operacja może okazać się niebezpieczna z perspektywy wydajności działania. Pamiętaj, że nasza aplikacja internetowa o ogromnej skali jest przeznaczona dla milionów użytkowników. Niektórzy z nich mogą mieć urządzenia o niewielkiej mocy obliczeniowej, znacznie odbiegającej od najnowszych i najszybszych procesorów typu Apple M3 Max. Wobec tego znacznie lepszym rozwiązaniem zamiast sekwencyjnego uaktualniania modelu DOM dla poszczególnych elementów listy będzie hurtowe przeprowadzenie tych operacji w jakiś sposób, a następnie ich jednoczesne zastosowanie w modelu DOM. Być może nie warto sięgać po takie rozwiązanie, ponieważ kiedyś może się zmienić sposób działania przeglądarki WWW i będzie ona automatycznie i wsadowo przeprowadzała szybkie modyfikacje modelu DOM.

To tylko niektóre z problemów, jakie napotykali programiści przez lata tworzący aplikacje internetowe przed pojawieniem się Reacta oraz innych abstrakcji. Niemożność umieszczania kodu w sposób pozwalający później na jego łatwą obsługę techniczną, wielokrotne użycia i przewidywalność na dużą skalę była poważnym problemem, dla którego branża nie wypracowała zbyt wielu standardów. Wiele firm napotykało trudności związane z tworzeniem niezawodnych i skalowanych interfejsów użytkownika. To był czas, w którym można było dostrzec pojawienie się wielu produktów opartych na JavaScripcie przeznaczonych do rozwiązywania tego problemu. Przykładami takich produktów były Backbone.js, KnockoutJS, AngularJS i jQuery. Warto się im po kolei przyjrzeć i przekonać się, jak rozwiązywały ten problem. To pomoże lepiej zrozumieć, dlaczego React różni się od tych rozwiązań, a nawet może być od nich lepszy.

jQuery

Wyjaśnię teraz, jak niektóre z wymienionych wcześniej problemów związanych z aplikacjami internetowymi były rozwiązywane przez narzędzia istniejące przed pojawieniem się Reacta. To pozwoli przekonać się, dlaczego React jest tak ważny. Rozpocznę od biblioteki jQuery i pokażę, jak można ją wykorzystać do przygotowania omówionego wcześniej przycisku *Polub*.

Mamy zatem przycisk *Polub* wyświetlony na stronie internetowej i chcemy, aby stał się on interaktywny:

```
<button id="likeButton">Polub</button>
```

W przypadku jQuery interaktywność przycisku można zapewnić podobnie, jak zostało to zrobione wcześniej:

```
$("#likeButton").on("click", function () {
  this.prop("disabled", true);
  fetch("/like", {
    method: "POST",
    body: JSON.stringify({ liked: this.text() === "Polub" }),
  })
    .then(() => {
      this.text(this.text() === "Polub" ? "Polubiony" : "Polub");
    })
```

```
      .catch(() => {
        this.text("Niepowodzenie");
      })
      .finally(() => {
        this.prop("disabled", false);
      });
});
```

Na podstawie tego przykładu można zobaczyć, że dołączamy dane do interfejsu użytkownika i wykorzystujemy dane wiązanie do uaktualnienia tego interfejsu. jQuery to narzędzie dość aktywne w bezpośrednim operowaniu interfejsem użytkownika.

jQuery działa w sposób, o którym można powiedzieć, że ma istotne skutki uboczne. Nieustannie ma do czynienia z informacjami o stanie pozostającym poza kontrolą tej biblioteki i je modyfikuje. Użyłem określenia „skutki uboczne", ponieważ jQuery pozwala na bezpośrednie i globalne modyfikowanie struktury strony internetowej przeprowadzane z dowolnego miejsca w kodzie, w tym również z innych zaimportowanych modułów, a nawet ze zdalnie wykonywanych skryptów. To może prowadzić do nieprzewidywalnego działania oraz złożonych interakcji, które będą trudne do monitorowania i analizowania. Zmiany wprowadzone w jednym fragmencie strony mogą w nieprzewidziany sposób wpływać na inne fragmenty strony. Tego rodzaju rozproszone i nieuporządkowane operacje powodują, że kod staje się trudny w obsłudze technicznej oraz debugowaniu.

W nowoczesnych frameworkach rozwiązano te problemy poprzez dostarczenie uporządkowanych i przewidywalnych mechanizmów uaktualniania interfejsu użytkownika bez konieczności odwoływania się do bezpośredniego operowania modelem DOM. Swego czasu ten wzorzec był powszechnie spotykany. Trudno go było przeanalizować i przetestować, ponieważ środowisko związane z kodem, czyli stan aplikacji, ulega nieustannym zmianom. W pewnym momencie trzeba było się zatrzymać i zadać sobie pytanie, jaki jest obecnie stan aplikacji w przeglądarce, a wraz ze wzrostem poziomu złożoności aplikacji na tego rodzaju pytanie coraz trudniej jest udzielić odpowiedzi.

Ponadto nasz przycisk utworzony z wykorzystaniem jQuery jest trudny do przetestowania, ponieważ jest to jedynie procedura obsługi zdarzeń. Gdybyśmy mieli utworzyć test, prawdopodobnie przedstawiałby się następująco:

```
test("LikeButton", () => {
  const $button = $("#likeButton");
  expect($button.text()).toBe("Polub");
  $button.trigger("click");
  expect($button.text()).toBe("Polubiony");
});
```

Jedyny problem polega na tym, że wywołanie `$('#likeButton')` zwraca wartość `null` w środowisku testowym, ponieważ nie jest to prawdziwa przeglądarka WWW. W celu przetestowania tego kodu trzeba byłoby przygotować imitację środowiska przeglądarki, co wymaga wiele pracy. Jest to problem powszechnie napotykany podczas pracy z jQuery: kod jest trudny do przetestowania z powodu trudności w odizolowaniu dodawanego przez niego sposobu działania. Ponadto jQuery w ogromnym stopniu zależy od środowiska przeglądarki. jQuery współdzieli z przeglądarką własność interfejsu użytkownika, co utrudnia jego analizę i testowanie: przeglądarka jest właścicielem interfejsu, a jQuery to jedynie gość. Takie odejście od paradygmatu „jednokierunkowego przepływu danych" było w owym czasie powszechnym problemem związanym z bibliotekami.

Ostatecznie biblioteka jQuery zaczynała tracić na znaczeniu wraz z rozwojem internetu oraz wyraźnie pojawiającą się potrzebą dysponowania znacznie bardziej niezawodnymi i skalowanymi rozwiązaniami. Wprawdzie biblioteka ta nadal jest stosowana w wielu aplikacjach produkcyjnych, ale nie stanowi już najlepszego rozwiązania podczas tworzenia nowoczesnych aplikacji internetowych. Oto wybrane powody, które przyczyniły się do spadku popularności jQuery:

Wielkość danych i czas ich wczytywania

Jedno z najpoważniejszych zastrzeżeń dotyczących biblioteki jQuery wiązało się z jej wielkością. Integracja pełnej biblioteki z projektem internetowym prowadziła do jego zwiększenia, co mogło być szczególnie niekorzystne w przypadku witryn, które miały być wczytywane w bardzo krótkim czasie. W obecnych czasach dostępności internetu w urządzeniach mobilnych, gdy wielu użytkowników może korzystać z wolnych bądź ograniczonych połączeń, każdy kilobajt przekazywanych danych ma znaczenie. Dołączenie całej biblioteki jQuery mogło więc mieć negatywny wpływ na wydajność działania aplikacji i wrażenia jej użytkowników mobilnych.

Przed pojawieniem się Reacta powszechną praktyką było oferowanie konfiguracji dla bibliotek takich jak jQuery i Mootools, co pozwalało użytkownikom wybierać niezbędną im funkcjonalność. Wprawdzie dzięki temu można było zmniejszyć ilość przekazywanych danych, ale jednocześnie takie podejście komplikowało decyzje podejmowane przez programistów oraz ogólnie sposób pracy nad projektem.

Nadmiarowość w przypadku nowoczesnych przeglądarek WWW

Gdy pojawiła się biblioteka jQuery, pomagała wyeliminować niespójności występujące między wieloma przeglądarkami WWW. Ponadto zapewniała programistom ujednolicony sposób obsługi tych różnic w kontekście wyboru elementów w przeglądarce oraz ich późniejszego modyfikowania. Wraz z rozwojem internetu zmieniły się również przeglądarki. Wiele funkcji jQuery, z powodu których była ona czymś niezbędnym i niezastąpionym (np. spójne operowanie modelem DOM oraz funkcjonalność pobierania danych z sieci), obecnie jest wbudowanych w nowoczesne przeglądarki i są one przez nie spójnie obsługiwane. Dlatego stosowanie jQuery podczas tworzenia aplikacji internetowych może być postrzegane jako zbędne i powoduje niepotrzebne zwiększanie poziomu złożoności rozwiązania.

Na przykład wywołanie `document.querySelector()` w przeglądarce może z łatwością zastąpić API $ wbudowane w bibliotece jQuery.

Problemy z wydajnością działania

Wprawdzie jQuery ułatwiała wykonywanie wielu zadań, ale bardzo często odbywało się to kosztem wydajności działania. Wraz z każdym kolejnym wydaniem przeglądarki usprawniane są jej metody JavaScriptu dostępne w środowisku uruchomieniowym. Dlatego w pewnym momencie korzystający z nich kod może działać szybciej niż jego odpowiednik wykorzystujący jQuery. W przypadku małych projektów ta różnica może być niezauważalna. Natomiast w większych i bardziej skomplikowanych aplikacjach internetowych te złożoności mogą się kumulować, prowadząc do zauważalnych opóźnień i mniejszej responsywności.

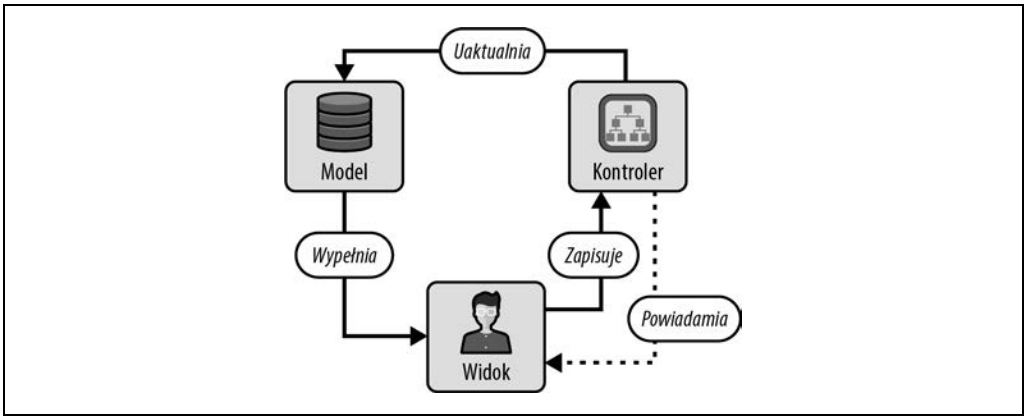
Z wymienionych powodów, choć biblioteka jQuery odegrała ważną rolę w rozwoju internetu i uprościła wiele zadań stojących przed programistami, obecne środowisko tworzenia aplikacji internetowych oferuje wiele rozwiązań, które sprawiły, że biblioteka ta straciła na znaczeniu. Programista

musi wziąć pod uwagę zarówno wygodę oferowaną przez jQuery, jak i jej potencjalne wady, zwłaszcza w kontekście aktualnych projektów aplikacji internetowych.

Pomimo swoich wad biblioteka jQuery stanowiła absolutną rewolucję w sposobie pracy z modelem DOM. Na jej bazie powstało wiele innych bibliotek, które zapewniały większą przewidywalność i możliwość wielokrotnego użycia kodu. Jedną z takich bibliotek była Backbone.js, której twórcy próbowali rozwiązywać dokładnie te same problemy, które obecnie rozwiązuje React. Jednak biblioteka Backbone.js pojawiła się znacznie wcześniej niż React. Z następnego punktu dowiesz się nieco więcej na jej temat.

Backbone.js

Backbone.js to opracowana w 2010 roku biblioteka, która była jednym z rozwiązań problemów, jakie w dziedzinie tworzenia aplikacji internetowych istniały przed pojawieniem się Reacta, takich jak dysonans stanu między przeglądarką WWW i JavaScriptem, możliwość wielokrotnego użycia kodu, możliwość przetestowania kodu itd. To było elegancko proste rozwiązanie — biblioteka umożliwiająca tworzenie modeli i widoków. W Backbone.js zastosowano inne podejście do tradycyjnego wzorca MVC (ang. *model-view-controller*), czyli model – widok – kontroler, które możesz zobaczyć na rysunku 1.1. Warto nieco dokładniej poznać ten wzorzec, ponieważ pomoże to w lepszym zrozumieniu Reacta oraz zapewni solidne podstawy do dalszych analiz i dyskusji.



Rysunek 1.1. Tradycyjny wzorzec MVC

Wzorzec MVC

Wzorzec MVC można uznać za filozofię projektową, w ramach której aplikacja zostaje podzielona na trzy powiązane ze sobą komponenty. To pozwala oddzielić wewnętrzny sposób reprezentowania informacji od sposobu, w jaki są one przedstawiane bądź pobierane od użytkownika. Oto dokładniejsze omówienie poszczególnych komponentów:

Model

Model (ang. *model*) odpowiada za dane i reguły biznesowe aplikacji. Jednocześnie nie ma żadnych informacji odnośnie do widoku i kontrolera, co gwarantuje, że logika biznesowa zostaje oddzielona od interfejsu użytkownika.

Widok

Widok (ang. *view*) reprezentuje interfejs użytkownika aplikacji. Wyświetla użytkownikowi dane pochodzące z modelu oraz przekazuje do kontrolera polecenia otrzymane od użytkownika. Widok jest komponentem pasywnym, co oznacza, że czeka, aż model przekaże dane przeznaczone do wyświetlenia. Bezpośrednio nie pobiera i nie zapisuje danych. Widok nie zajmuje się również obsługą interakcji użytkownika, lecz deleguje to zadanie do następnego komponentu, czyli do kontrolera.

Kontroler

Kontroler (ang. *controller*) działa w charakterze interfejsu między modelem i widokiem. Pobiera z widoku dane wejściowe pochodzące od użytkownika, przetwarza je (potencjalnie uaktualniając przy tym model), a następnie zwraca do widoku dane wyjściowe przeznaczone do wyświetlenia. Kontroler pozwala zachować rozdzielność modelu od widoku, dzięki czemu architektura systemu jest znacznie elastyczniejsza.

Podstawową zaletą wzorca MVC jest podział zadań, co oznacza, że za obsługę logiki biznesowej, interfejsu użytkownika oraz danych wejściowych użytkownika odpowiadają inne fragmenty bazy kodu. W ten sposób aplikacja nie tylko jest bardziej modułowa, ale również łatwiejsza w późniejszej obsłudze technicznej, skalowaniu i testowaniu. Ten wzorzec jest powszechnie stosowany w aplikacjach internetowych, a jego obsługa została wbudowana w wiele frameworków, np. Django, Ruby on Rails oraz ASP.NET MVC.

Wzorzec MVC przez wiele lat stanowił filar w zakresie tworzenia oprogramowania, zwłaszcza aplikacji internetowych. Jednak te aplikacje ewoluowały, a oczekiwania użytkowników w zakresie interaktywności i dynamiczności interfejsów również się zwiększyły. Wobec tego pewne ograniczenia tradycyjnego wzorca MVC stały się wyraźnie widoczne. Oto wybrane ograniczenia wzorca MVC i sposoby, w jakie React jest w stanie je pokonać:

Skomplikowane zarządzanie interaktywnością i stanem

Tradycyjne architektury MVC często zmagają się z zarządzaniem skomplikowanymi interfejsami użytkownika, w których znajduje się wiele interaktywnych elementów. Wraz z rozbudową aplikacji zarządzanie zmianami stanu i ich efektem na wielu różnych obszarach interfejsu użytkownika może stać się uciążliwym zadaniem. Tak się dzieje w miarę gromadzenia się kontrolerów, które czasami mogą prowadzić do konfliktów z innymi kontrolerami. Część kontrolerów może kontrolować widoki, które ich nie reprezentują, bądź rozdział między komponentami MVC nie został odpowiednio przeprowadzony w kodzie produktu.

Dzięki wykorzystaniu architektury opartej na komponentach i wirtualnemu modelowi DOM React ułatwia analizowanie zmian stanu i ich wpływu na interfejs użytkownika. Jest to możliwe z powodu traktowania komponentów interfejsu użytkownika jak funkcji: otrzymują one dane wejściowe (właściwości) i zwracają dane wyjściowe wygenerowane na podstawie danych

wejściowych (elementy). Taki model mentalny pozwolił znacznie uprościć wzorzec MVC, ponieważ funkcje są wszechobecne w kodzie w JavaScripcie i dużo łatwiej dostępne w porównaniu do zewnętrznego modelu mentalnego, który nie był rodzimie związany z językiem programowania.

Dwukierunkowe dołączanie danych

Niektóre frameworki MVC używają dwukierunkowego dołączania danych. W przypadku braku starannego zarządzania rozwiązaniem może to prowadzić do niezamierzonych efektów ubocznych, zwłaszcza jeśli widok nie będzie zsynchronizowany z modelem bądź na odwrót. Ponadto w przypadku dwukierunkowego dołączania danych odpowiedź na pytanie dotyczące własności była jedynie przybliżona z powodu niejasnego podziału zadań. To jest szczególnie interesujące — wprawdzie wzorzec MVC jest sprawdzonym modelem dla zespołów w pełni rozumiejących znaczenie podziału zadań, ale reguły związane z tą separacją rzadko są wymuszane. To dotyczy zwłaszcza sytuacji, w której mamy do czynienia z generowaniem dużej ilości danych wyjściowych oraz z szybkim rozwojem na początkowym etapie istnienia projektu. Dlatego wspomniana separacja zadań, która jest jedną z największych zalet wzorca MVC, często okazuje się też jego największą słabością, wynikającą z braku egzekwowania reguł.

React wykorzystuje wzorzec licznika do zapewnienia dwukierunkowego dołączania danych, które jest określane mianem jednokierunkowego przepływu danych (do tego zagadnienia jeszcze powrócę). To pozwala nadawać priorytet jednokierunkowemu przepływowi danych, a nawet go wymuszać, za pomocą systemu takiego jak React Forget (jego omówienie znajdziesz w dalszej części książki). Dzięki takiemu podejściu uaktualnienia interfejsu użytkownika stają się bardziej przewidywalne, a separacja zadań może być wyraźniej zastosowana. To ostatecznie będzie sprzyjało zespołom w szybkim rozwoju oprogramowania.

Ścisłe powiązanie komponentów

W niektórych implementacjach MVC komponenty modelu, widoku i kontrolera mogą być ze sobą ściśle powiązane, co będzie znacznie utrudniało zmianę lub refaktoryzację dowolnego z nich bez wpływania na inne. Dzięki modelowi opartemu na komponentach React zachęca do stosowania bardziej modularnego podejścia oraz umożliwia i wspomaga umieszczanie zależności blisko ich reprezentacji w interfejsie użytkownika.

Ponieważ niniejsza książka jest poświęcona Reactowi, nie będę w tym miejscu zbytnio się zagłębiać w przedstawiony wzorzec. Jednak na potrzeby omawianych tutaj zagadnień warto dodać, że koncepcyjnie modele były źródłami danych, natomiast interfejsy użytkownika korzystały z tych danych i je wyświetlały. Biblioteka Backbone.js udostępniała wygodne API przeznaczone do pracy z modelami i widokami, a także umożliwiała ich połączenie ze sobą. W owym czasie takie rozwiązanie okazywało się niezwykle elastyczne i dawało potężne możliwości. To również często było rozwiązanie możliwe do skalowania oraz pozwalające programistom na oddzielne przetestowanie kodu źródłowego.

W kolejnym fragmencie kodu przedstawiłem wcześniejszy przykład przycisku *Polub*, tym razem utworzonego z wykorzystaniem Backbone.js:

```
const LikeButton = Backbone.View.extend({
  tagName: "button",
  attributes: {
    type: "button",
  },
```

```
  events: {
    click: "onClick",
  },
  initialize() {
    this.model.on("change", this.render, this);
  },
  render() {
    this.$el.text(this.model.get("liked") ? "Polubiony" : "Polub");
    return this;
  },
  onClick() {
    fetch("/like", {
      method: "POST",
      body: JSON.stringify({ liked: !this.model.get("liked") }),
    })
      .then(() => {
        this.model.set("liked", !this.model.get("liked"));
      })
      .catch(() => {
        this.model.set("failed", true);
      })
      .finally(() => {
        this.model.set("pending", false);
      });
  },
});

const likeButton = new LikeButton({
  model: new Backbone.Model({
    liked: false,
  }),
});

document.body.appendChild(likeButton.render().el);
```

Zwróć uwagę na dwie kwestie: element `LikeButton` stanowi rozszerzenie `Backbone.View`, a metoda `render()` zwraca `this`. Podobna metoda `render()` znajduje się w bibliotece React, ale do tego powrócę w dalszej części rozdziału. Trzeba w tym miejscu dodać, że biblioteka Backbone.js nie zawierała rzeczywistej implementacji metody `render()`, lecz programista musiał ręcznie zmieniać model DOM za pomocą jQuery bądź skorzystać z systemu szablonów, takiego jak Handlebars.

Biblioteka Backbone.js udostępniała możliwe do łączenia API, które pozwalało programistom przygotowywać logikę w postaci właściwości obiektów. W porównaniu do poprzedniego przykładu można dostrzec, że dzięki Backbone.js znacznie bardziej komfortowe stało się tworzenie przycisku, który jest interaktywny i uaktualnia interfejs użytkownika w odpowiedzi na zdarzenia.

Ponadto rozwiązanie ma bardziej strukturalną postać z powodu stosowania funkcjonalności grupowania logiki. Zwróć także uwagę na znaczne ułatwienie przetestowania tego przycisku oddzielnie od pozostałego kodu, ponieważ można utworzyć egzemplarz `LikeButton`, a następnie wywołać jego metodę `render()`.

Oto jak można przetestować nasz przycisk utworzony za pomocą biblioteki Backbone.js:

```
test("Stan początkowy przycisku LikeButton", () => {
  const likeButton = new LikeButton({
```

```
    model: new Backbone.Model({
      liked: false, // Stan początkowy zdefiniowany jako Polub
    }),
  });
  likeButton.render(); // Upewnienie się o wywołaniu metody render() w celu odzwierciedlenia stanu
początkowego
  // Sprawdzenie, czy tekst ma postać "Polub", odzwierciedlając tym samym stan początkowy
  expect(likeButton.el.textContent).toBe("Polub");
});
```

Istnieje nawet możliwość przetestowania sposobu działania przycisku po zmianie stanu, jak ma to miejsce w przypadku wystąpienia zdarzenia typu `click`:

```
test("LikeButton", async () => {
  // Oznaczenie funkcji jako asynchronicznej w celu umożliwienia obsługi obietnicy
  const likeButton = new LikeButton({
    model: new Backbone.Model({
      liked: false,
    }),
  });
  expect(likeButton.render().el.textContent).toBe("Polub");

  // Imitowanie wywołania fetch, aby uniknąć wykonania rzeczywistego żądania HTTP
  global.fetch = jest.fn(() =>
    Promise.resolve({
      json: () => Promise.resolve({ liked: true }),
    })
  );
  // Oczekiwanie na metodę onClick(), aby mieć pewność o zakończeniu operacji asynchronicznych
  await likeButton.onClick();

  expect(likeButton.render().el.textContent).toBe("Polubiony");

  // Opcjonalnie można przywrócić pierwotną implementację wywołania fetch, jeśli zachodzi potrzeba
  global.fetch.mockRestore();
});
```

Z tego powodu w owym czasie biblioteka Backbone.js była niezwykle popularnym rozwiązaniem. Alternatywą było samodzielne tworzenie ogromnej ilości kodu trudnego do przetestowania i przeanalizowania, w dodatku bez żadnych gwarancji, że będzie on działał niezawodnie i zgodnie z oczekiwaniami. Dlatego biblioteka Backbone.js była bardzo mile widzianym rozwiązaniem. Wprawdzie na początku swojego istnienia zyskała popularność z powodu prostoty i elastyczności, ale nie była idealnym rozwiązaniem. Oto kilka słabych stron Backbone.js:

Rozwlekły kod wymagany do obsługi rozwiązania

Jeden z często pojawiających się zarzutów pod adresem Backbone.js dotyczył ilości kodu wymaganego do obsługi rozwiązania, jaki musieli utworzyć programiści. W przypadku prostych aplikacji to niekoniecznie był duży problem, ale wraz z rozbudową aplikacji, a tym samym wzrostem ilości kodu potrzebnego do jej obsługi, mogło to prowadzić do nadmiarowości i powstania kodu, który okazywał się trudny w późniejszej obsłudze technicznej.

Brak dwukierunkowego dołączania danych

W przeciwieństwie do innych technologii biblioteka Backbone.js nie oferowała wbudowanych możliwości w zakresie dwukierunkowego dołączania danych. To oznaczało, że w przypadku zmiany danych nie następowało automatyczne uaktualnienie modelu DOM i na odwrót. Programiści często musieli samodzielnie tworzyć kod bądź używać wtyczek, aby osiągnąć taką funkcjonalność.

Architektura oparta na zdarzeniach

Uaktualnienie modelu danych mogło pociągnąć za sobą wiele zdarzeń w aplikacji. Zarządzanie tego rodzaju zdarzeniami kaskadowymi może być praktycznie niemożliwe, prowadząc tym samym do sytuacji, w której nie wiadomo, czy zmiana pewnych danych będzie miała wpływ na pozostałą część aplikacji. W efekcie znacznie utrudnione było debugowanie kodu źródłowego i jego późniejsza obsługa techniczna. W celu rozwiązania tego problemu programiści często musieli uciekać się do ostrożnego zarządzania zdarzeniami, aby uniknąć propagowania uaktualnień w całej aplikacji.

Brak możliwości tworzenia kompozycji

Backbone.js nie oferuje funkcji pozwalających łatwo zagnieżdżać widoki, co utrudnia tworzenie skomplikowanych interfejsów użytkownika. Natomiast React umożliwia bezproblemowe zagnieżdżanie komponentów we właściwościach potomnych, co niezwykle ułatwia budowanie złożonych hierarchii interfejsów użytkownika. Marionette.js, czyli rozszerzenie dla Backbone.js, próbowała wyeliminować niektóre z problemów Backbone.js związanych z brakiem możliwości tworzenia kompozycji, ale nie zapewnia tak zintegrowanego rozwiązania, jakie otrzymujemy w przypadku modelu komponentu Reacta.

Wprawdzie Backbone.js ma pewne wady, ale trzeba koniecznie pamiętać, że żadne narzędzie nie jest idealne. Najlepsze rozwiązanie często zależy od konkretnych wymagań danego projektu oraz preferencji tworzącego go zespołu programistów. Warto w tym miejscu dodać, że możliwości narzędzi programistycznych w ogromnym stopniu są uzależnione od istnienia silnej społeczności. Niestety w ostatnich latach można dostrzec spadek popularności biblioteki Backbone.js, zauważalny zwłaszcza po pojawieniu się Reacta. Ktoś mógłby powiedzieć, że React zabił Backbone.js, ale ocenę tego stwierdzenia pozostawiam Tobie.

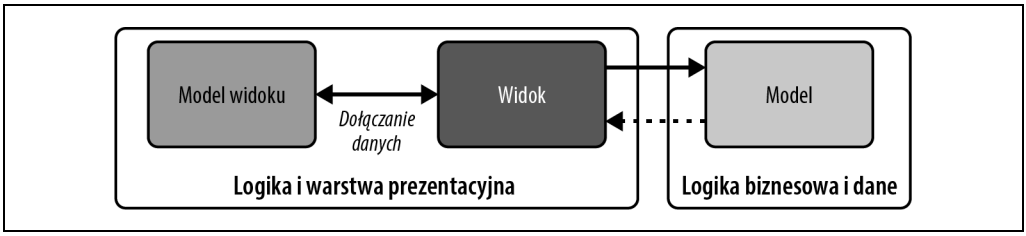
Knockout

Przedstawione wcześniej podejście porównam teraz z innym popularnym niegdyś rozwiązaniem otwartoźródłowym: biblioteką Knockout. Ta opracowana w 2010 roku biblioteka umożliwiała tworzenie komponentów „obserwowanych" i „wiązań", wykorzystując śledzenie zależności, gdy dochodziło do zmiany stanu.

Knockout był jedną z pierwszych, o ile nie pierwszą, reaktywną biblioteką JavaScriptu, której reaktywność była definiowana jako wartości uaktualniane w odpowiedzi na zmiany stanu w obserwowanych komponentach. Nowoczesne podejścia do tego stylu reaktywności są czasami określane mianem sygnałów i powszechnie dominują w bibliotekach takich jak Vue.js, SolidJS, Svelte, Qwik, nowoczesny Angular itd. Więcej informacji na ten temat znajdziesz w rozdziale 10.

Pod względem koncepcyjnym komponenty obserwowalne to źródła danych, wiązania zaś to interfejsy użytkownika, które używają tych danych i je wyświetlają. Dlatego komponenty obserwowalne można porównać do modeli, a wiązania do widoków.

Jednak na skutek omówionej wcześniej ewolucji wzorca MVC biblioteka Knockout działa bardziej w stylu MVVM (ang. *model-view-viewmodel*), czyli model – widok – model widoku, jak pokazałem na rysunku 1.2. Warto nieco dokładniej poznać ten wzorzec.



Rysunek 1.2. Graficzna prezentacja wzorca MVVM

Wzorzec MVVM

MVVM to architekturalny wzorzec projektowy, szczególnie popularny w przypadku aplikacji wyposażonych w rozbudowane interfejsy użytkownika, takie jak opracowane za pomocą platform typu WPF i Xamarin. MVVM można uznać za ewolucję tradycyjnego wzorca MVC, dopasowanego do nowoczesnych platform tworzenia interfejsów użytkownika, w których dołączanie danych jest ważną funkcjonalnością. Oto krótkie omówienie komponentów wzorca MVVM:

Model

- Przedstawia logikę biznesową i dane aplikacji.
- Odpowiada za pobieranie, przechowywanie i przetwarzanie danych.
- Zwykle komunikuje się z bazami danych, usługami lub innymi źródłami danych i operacjami.
- Nie ma żądnych informacji odnośnie do widoku i modelu widoku.

Widok

- Reprezentuje interfejs użytkownika aplikacji.
- Wyświetla informacje użytkownikowi oraz otrzymuje dane wejściowe pochodzące od użytkownika.
- We wzorcu MVVM widok jest komponentem pasywnym i nie zawiera żadnej logiki aplikacji, lecz w sposób deklaratywny jest dołączany do modelu widoku i automatycznie odzwierciedla zmiany za pomocą mechanizmu dołączania danych.

Model widoku

- Działa jako łącznik między modelem i widokiem.
- Udostępnia dane i polecenia, do których może się dołączać widok. Dane są tutaj często w formacie gotowym do wyświetlenia.

- Obsługuje dane wejściowe pochodzące od użytkownika, często wykorzystując do tego wzorzec polecenia.
- Zawiera logikę prezentacyjną i przekształca dane modelu na format, który może być łatwo wyświetlony przez widok.
- Nie ma żadnych dokładnych informacji odnośnie do specyfikacji używającego go widoku, co pozwala uzyskać architekturę, w której nie istnieje ścisłe powiązanie między komponentami.

Kluczową cechą wzorca MVVM jest separacja zadań, podobnie jak w MVC, która ma następujące skutki:

Możliwość przetestowania kodu

Brak ścisłego powiązania między widokiem i modelem widoku znacznie ułatwia tworzenie testów jednostkowych dla logiki prezentacyjnej bez angażowania do tego celu interfejsu użytkownika.

Możliwość wielokrotnego użycia kodu

Model widoku może być wielokrotne używany w różnych widokach bądź na różnych platformach.

Łatwa obsługa techniczna

Dzięki wyraźnej separacji zadań znacznie łatwiej można zarządzać, rozbudowywać i refaktoryzować kod źródłowy.

Dołączanie danych

Omawiany wzorzec sprawdza się doskonale na platformach obsługujących dołączanie danych. To pozwala zmniejszyć ilość kodu wymaganego do uaktualnienia interfejsu użytkownika.

Skoro omówiłem wzorce projektowe MVC i MVVM, warto je porównać i poznać zachodzące między nimi różnice (tabela 1.1).

Tabela 1.1. Porównanie wzorców MVC i MVVM

Kryterium	MVC	MVVM
Podstawowe przeznaczenie	Przede wszystkim dla aplikacji internetowych, aby zapewnić rozdzielenie interfejsu użytkownika od logiki.	Dla aplikacji o rozbudowanych interfejsach użytkownika, zwłaszcza wykorzystujących dwukierunkowe dołączanie danych, np. aplikacji biurkowych lub typu SPA (jednostronicowe aplikacje internetowe).
Komponenty	Model: dane i logika biznesowa. Widok: interfejs użytkownika. Kontroler: zarządzanie interfejsem użytkownika i uaktualnianie widoku.	Model: dane i logika biznesowa. Widok: elementy interfejsu użytkownika. Model widoku: łącznik między modelem i widokiem.
Przepływ danych	Pochodzącymi od użytkownika danymi wejściowymi zarządza kontroler, który następnie uaktualnia model, a później widok.	Widok jest dołączany bezpośrednio do modelu widoku. Zmiany wprowadzone w widoku zostają automatycznie odzwierciedlone w modelu widoku i na odwrót.
Brak powiązania komponentów	Widok jest często ściśle powiązany z kontrolerem.	Model widoku nie ma żadnych szczegółowych informacji odnośnie do używającego go widoku.

Tabela 1.1. Porównanie wzorców MVC i MVVM (ciąg dalszy)

Kryterium	MVC	MVVM
Współpraca z użytkownikiem	Obsługiwana przez kontroler.	Obsługiwana przez mechanizm dołączania danych i polecenia w modelu widoku.
Zastosowanie	Powszechnie używany podczas tworzenia aplikacji internetowych (np. Ruby on Rails, Django, ASP.NET MVC).	Świetnie sprawdza się na platformach obsługujących niezawodne dołączanie danych (np. WPF, Xamarin).

Na podstawie tego krótkiego porównania można zauważyć, że rzeczywiste różnice między wzorcami MVC i MVVM wiążą się z dwiema kwestiami: ścisłym połączeniem komponentów i dołączaniem danych. Gdy nie istnieje kontroler między modelem i widokiem, kwestia własności danych jest oczywista i bliższa użytkownikowi. React jeszcze bardziej usprawnia wzorzec MVVM dzięki jednokierunkowemu przepływowi danych, do którego powrócę w dalszej części rozdziału, jeszcze *bardziej zawężając* kwestię własności — właścicielem informacji o stanie jest konkretny komponent, który ich potrzebuje. Powróćmy teraz do biblioteki Knockout i jej powiązania z Reactem.

Knockout udostępnia API przeznaczone do pracy z komponentami obserwowanymi i wiązaniami danych. Zobacz teraz, jak za pomocą tej biblioteki można zaimplementować nasz przykładowy przycisk *Polub*. To pomoże zrozumieć, dlaczego React jest nieco lepszym rozwiązaniem. Oto wersja przycisku *Polub* utworzona z użyciem biblioteki Knockout:

```
function createViewModel({ liked }) {
  const isPending = ko.observable(false);
  const hasFailed = ko.observable(false);
  const onClick = () => {
    isPending(true);
    fetch("/like", {
      method: "POST",
      body: JSON.stringify({ liked: !liked() }),
    })
      .then(() => {
        liked(!liked());
      })
      .catch(() => {
        hasFailed(true);
      })
      .finally(() => {
        isPending(false);
      });
  };
  return {
    isPending,
    hasFailed,
    onClick,
    liked,
  };
}

ko.applyBindings(createViewModel({ liked: ko.observable(false) }));
```

W Knockoucie „model widoku” to obiekt JavaScriptu zawierający klucze i wartości, pozwalające łączyć różne elementy na stronie internetowej z wykorzystaniem atrybutu `data-bind`. W bibliotece tej nie istnieją „komponenty” bądź „szablony”, lecz jedynie model widoku i sposób na jego dołączanie do elementu w przeglądarce.

Nasza funkcja `createViewModel()` pokazuje, jak odbywa się tworzenie modelu widoku za pomocą biblioteki Knockout. Następnie wywołanie funkcji `ko.applyBindings()` pozwala połączyć model widoku ze środowiskiem hosta (przeglądarka WWW). Ta funkcja pobiera model widoku, a potem wyszukuje w przeglądarce WWW wszystkie elementy mające atrybut `data-bind`, który przez Knockouta jest używany w celu połączenia elementów z modelem widoku.

W przeglądarce przycisk zostanie dołączony do właściwości modelu widoku:

```
<button
  data-bind="click: onClick, text: liked ? 'Polubiony' : isPending ? [...]
></button>
```

Zauważ, że ten fragment kodu został skrócony dla zachowania prostoty.

Dołączenie elementu HTML do utworzonego „modelu widoku” odbywa się za pomocą naszej funkcji `createViewModel()`, a następnie witryna staje się interaktywna. Jak można sobie wyobrazić, samodzielne definiowanie subskrypcji zmian w komponentach obserwowanych, a następnie uaktualnianie interfejsu użytkownika w odpowiedzi na te zmiany oznaczało konieczność wykonania ogromnej pracy. W swoim czasie biblioteka Knockout była doskonałym produktem, ale jednocześnie wymagała tworzenia ogromnej ilości kodu, aby rozwiązanie działało zgodnie z oczekiwaniami.

Ponadto modele widoku często rozrastały się do postaci ogromnej i skomplikowanej, co z kolei prowadziło do większej niepewności związanej z refaktoryzacją i optymalizacją kodu źródłowego. Ostatecznie powstawały rozwlekłe i monolityczne modele widoku, które były trudne do przetestowania i przeanalizowania. Mimo to w swoim czasie Knockout był bardzo popularnym rozwiązaniem i świetną biblioteką. Ponadto dość łatwo pozwalał testować kod w izolacji, co niewątpliwie było jego plusem.

Dla potomności zamieszam tutaj fragment kodu pokazujący, jak można przetestować nasz przykładowy przycisk *Polub* utworzony za pomocą biblioteki Knockout:

```
test("LikeButton", () => {
  const viewModel = createViewModel({ liked: ko.observable(false) });
  expect(viewModel.liked()).toBe(false);
  viewModel.onClick();
  expect(viewModel.liked()).toBe(true);
});
```

AngularJS

AngularJS został opracowany przez Google'a w 2010 roku. Był to pionierski framework JavaScriptu, który wywarł ogromny wpływ na branżę tworzenia aplikacji internetowych. Wyraźnie się wyróżnia spośród omówionych dotychczas bibliotek i frameworków, ponieważ oferuje obsługę wielu innowacyjnych funkcjonalności, których wpływ można zauważyć w produktach pojawiających się

później, np. w bibliotece React. Przedstawię teraz nieco bardziej szczegółowe porównanie frameworka AngularJS z innymi bibliotekami oraz omówię jego najważniejsze cechy — to powinno pomóc w dostrzeżeniu szlaku, jaki AngularJS przetarł dla Reacta.

Mechanizm dwukierunkowego dołączania danych

Mechanizm dwukierunkowego dołączania danych był cechą charakterystyczną frameworka AngularJS, która znacznie upraszczała współpracę między interfejsem użytkownika i danymi. Jeżeli model (czyli dane) uległ zmianie, wówczas widok (czyli interfejs użytkownika) był automatycznie uaktualniany w celu odzwierciedlenia zmiany, i na odwrót. To stanowiło wyraźny kontrast względem bibliotek takich jak jQuery, których programiści musieli ręcznie operować modelem DOM, aby odzwierciedlać wszelkie zmiany w danych oraz przechwytywać dane wejściowe pochodzące od użytkownika i uaktualniać dane modelu.

Oto prosta aplikacja utworzona z użyciem frameworka AngularJS, w której mechanizm dwukierunkowego dołączania danych odgrywa kluczową rolę:

```
<!DOCTYPE html>
<html>
  <head>
    <script
    src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js">
    </script>
  </head>
  <body ng-app="">
    <p>Imię: <input type="text" ng-model="name" /></p>
    <p ng-if="name">Witaj, {{name}}!</p>
  </body>
</html>
```

W tej aplikacji dyrektywa `ng-model` powoduje dołączenie do zmiennej o nazwie `name` wartości pola danych wejściowych. Po wpisaniu czegokolwiek w polu danych wejściowych model `name` zostanie uaktualniony, podobnie jak widok (w omawianym przykładzie powitanie `"Witaj, {{name}}!"`), który również zostanie uaktualniony w czasie rzeczywistym.

Architektura modularna

AngularJS wprowadził architekturę modularną, umożliwiającą programistom logiczne separowanie komponentów aplikacji. Każdy moduł mógł hermetyzować funkcjonalność oraz być niezależnie opracowywany, testowany i obsługiwany. Wprawdzie można pokusić się o stwierdzenie, że był to prekursor modelu komponentu stosowanego w bibliotece React, ale jest to kwestia dyskusyjna.

Oto krótki przykład:

```
var app = angular.module("myApp", [
  "ngRoute",
  "appRoutes",
  "userCtrl",
  "userService",
]);
```

```
var userCtrl = angular.module("userCtrl", []);
userCtrl.controller("UserController", function ($scope) {
  $scope.message = "Witaj z komponentu UserController";
});

var userService = angular.module("userService", []);
userService.factory("User", function ($http) {
  //...
});
```

W przedstawionym fragmencie kodu moduł `myApp` jest zależny od wielu innych modułów: `ngRoute`, `appRoutes`, `userCtrl` i `userService`. Każdy z tych modułów zależnych może być zdefiniowany w oddzielnym pliku JavaScriptu i opracowywany niezależnie od modułu głównego `myApp`. Taka koncepcja okazała się znacząco odmienna od stosowanych w bibliotekach jQuery i Backbone.js, które nie miały koncepcji „modułu" w tym sensie.

Zależności (`appRoutes`, `userCtrl` itd.) są wstrzykiwane do modułu głównego `app` za pomocą wzorca nazywanego *wstrzykiwaniem zależności* (ang. *dependency injection*), który został spopularyzowany przez nowoczesnego Angulara. Nie trzeba dodawać, że ten wzorzec był powszechnie stosowany jeszcze przed ustandaryzowaniem modułów JavaScriptu. Następnie został szybko wyparty przez polecenia `import` i `export`. Mamy tutaj wyraźny kontrast dla zależności komponentów Reacta. Warto więc nieco dokładniej omówić mechanizm wstrzykiwania zależności.

Mechanizm wstrzykiwania zależności

Wstrzykiwanie zależności to wzorzec projektowy, w którym obiekt otrzymuje zależności zamiast je tworzyć. Na bazie tego wzorca projektowego zbudowano framework AngularJS. W tamtym czasie nie była to funkcjonalność, którą można było spotkać w innych bibliotekach JavaScriptu. Miało to ogromny wpływ na sposób tworzenia modułów i komponentów, na zarządzanie nimi oraz na promowanie wysokiego poziomu modułowości i wielokrotnego użycia kodu źródłowego.

Oto przykład pokazujący działanie mechanizmu wstrzykiwania zależności w aplikacji AngularJS:

```
var app = angular.module("myApp", []);

app.controller("myController", function ($scope, myService) {
  $scope.greeting = myService.sayHello();
});

app.factory("myService", function () {
  return {
    sayHello: function () {
      return "Witaj, świecie!";
    },
  };
});
```

W tym przykładzie `myService` to usługa, która za pomocą mechanizmu wstrzykiwania zależności zostanie wstrzyknięta do kontrolera `myController`. Kontroler nie musi wiedzieć, w jaki sposób ma być utworzona usługa. Deklaruje jedynie usługę jako zależność, a framework AngularJS zajmuje się jej utworzeniem i wstrzyknięciem. To ułatwia zarządzanie zależnościami oraz udoskonala możliwości w zakresie testowania komponentów i ich wielokrotnego użycia.

Porównanie z bibliotekami Backbone.js i Knockout

Backbone.js i Knockout były popularnymi bibliotekami używanymi mniej więcej w czasie pojawienia się frameworka AngularJS. Obie miały swoje zalety, ale również brakowało im pewnych funkcjonalności, które były wbudowane w AngularJS.

Na przykład biblioteka Backbone.js zapewniała programistom większą kontrolę nad ich kodem oraz charakteryzowała się większą elastycznością niż AngularJS. Ta elastyczność była zarówno wadą, jak i zaletą: pozwalała na lepsze dostosowanie do własnych potrzeb, choć jednocześnie wymagała więcej kodu do obsługi rozwiązania. Z kolei framework AngularJS razem z mechanizmami dwukierunkowego dołączania danych i wstrzykiwania zależności umożliwiał stosowanie bardziej strukturalnego podejścia. Dawał więcej możliwości, co pozwalało programistom osiągać większą produktywność — to jest coś, co można dostrzec w nowoczesnych frameworkach takich jak Next.js, Remix itd. Był to jeden z aspektów, dzięki któremu AngularJS wyprzedzał swoje czasy.

Biblioteka nie dawała odpowiedzi na pytanie dotyczące bezpośredniej modyfikacji widoku (model DOM) i często tę kwestię pozostawiała programistom do rozwiązania. Z kolei AngularJS zajmował się modyfikacjami modelu DOM, wykorzystując do tego mechanizm dwukierunkowego dołączania danych, co na pewno jest ogromnym plusem.

Biblioteka koncentrowała się przede wszystkim na dołączaniu danych i nie zapewniała innych potężnych narzędzi, jakie oferował AngularJS, takich jak mechanizm wstrzykiwania zależności i architektura modularna. AngularJS jako w pełni wyposażony framework oferował znacznie bardziej rozbudowane rozwiązanie do budowania jednostronicowych aplikacji internetowych (SPA). W pewnym momencie dalsze prace nad frameworkiem AngularJS zostały zakończone, a jego miejsce zajęła nowsza wersja, o nazwie Angular. W porównaniu do poprzednika Angular ma te same, choć znacznie bardziej rozbudowane, możliwości, dzięki którym stał się doskonałym wyborem dla aplikacji działających na ogromną skalę.

Słabe strony frameworka AngularJS

Wprowadzenie frameworka AngularJS (1.x) oznaczało ogromny postęp w praktykach związanych z tworzeniem aplikacji internetowych. Jednak branża tworzenia aplikacji internetowych nieustannie i szybko ewoluowała, więc określone aspekty tego frameworka okazały się ograniczeniem bądź słabością, które ostatecznie przyczyniły się do spadku jego popularności i zaniechania dalszych prac nad nim. Oto niektóre z tych słabych stron:

Problemy z wydajnością działania

AngularJS miał problemy z wydajnością działania, zwłaszcza w przypadku aplikacji działających na ogromną skalę i ze skomplikowanymi operacjami dołączania danych. W ogromnych aplikacjach mechanizm *digest cycle* w AngularJS, czyli podstawowa funkcjonalność wykrywania zmian, mógł prowadzić do wolnych uaktualnień oraz opóźnień w interfejsie użytkownika. Mechanizm dwukierunkowego dołączania danych, choć innowacyjny i użyteczny w wielu sytuacjach, również miał niekorzystny wpływ na wydajność działania.

Złożoność

AngularJS wprowadził wiele nowatorskich koncepcji, takich jak dyrektywy, kontrolery, usługi, wstrzykiwanie zależności, fabryki itd. Wprawdzie dzięki nim framework oferował tak potężne możliwości, ale jednocześnie z ich powodu był skomplikowany i trudny do opanowania, zwłaszcza przez początkujących. Często były prowadzone dyskusje typu „czy należy użyć fabryki, czy raczej usługi", które okazywały się zaskoczeniem dla wielu programistów w zespołach.

Problemy z migracją do wydania Angular 2+

Po zaprezentowaniu frameworka Angular 2 okazało się, że nie zapewnia on wstecznej zgodności z wydaniem AngularJS 1.x, a migracja wymaga ponownego utworzenie kodu w języku Dart lub TypeScript. To oznaczało, że programiści musieli ponownie tworzyć ogromne fragmenty kodu, aby uaktualnić rozwiązanie do frameworka Angular 2, co niewątpliwie było postrzegane jako duża niedogodność. Wprowadzenie frameworka Angular 2+ doprowadziło dosłownie do podziału społeczności Angulara, wywołało dezorientację programistów i utorowało drogę dla Reacta.

Skomplikowana składnia w szablonach

AngularJS pozwalał używać skomplikowanych wyrażeń JavaScriptu w atrybutach szablonu, np. `on-click="$ctrl.some.deeply.nested.field = 123"`. To okazało się problematyczne, ponieważ prowadziło w kodzie znaczników do zacierania się granicy między warstwą prezentacyjną i logiką biznesową. Takie rozwiązanie powodowało także trudności w jego późniejszej obsłudze technicznej, ponieważ odszyfrowanie takiego kodu i zarządzanie nim było uciążliwe.

Co więcej, debugowanie również stało się znacznie trudniejsze, gdyż warstwy szablonu nie zostały zaprojektowane do obsługi skomplikowanej logiki, a znalezienie i usunięcie wszelkich błędów pojawiających się w tych osadzonych wyrażeniach mogło być prawdziwym wyzwaniem. Ponadto takie praktyki oznaczały złamanie reguły separacji zadań, która stanowiła podstawową filozofię projektową. Zgodnie z nią różne fragmenty aplikacji miały odpowiadać za obsługę poszczególnych aspektów aplikacji, aby w ten sposób poprawić jakość kodu i ułatwić jego późniejszą obsługę techniczną.

Teoretycznie szablon powinien wywoływać metodę kontrolera w celu przeprowadzenia aktualizacji, ale w praktyce nie było mechanizmu egzekwującego taki sposób działania.

Brak zapewnienia bezpieczeństwa typów

Szablony we frameworku AngularJS nie działały ze statycznymi narzędziami sprawdzania typu jak w TypeScripcie, co utrudniało wychwytywanie błędów na wczesnym etapie pracy. To była poważna wada, zwłaszcza w przypadku aplikacji działających na ogromną skalę, w których zapewnienie bezpieczeństwa typów miało znaczenie krytyczne z perspektywy skalowalności i łatwej obsługi technicznej rozwiązania.

Dezorientujący model obiektu $scope

Obiekt `$scope` we frameworku AngularJS był często uznawany za źródło zamieszania ze względu na odgrywaną przez niego rolę podczas dołączania danych, a także z uwagi na jego sposób działania w różnych kontekstach. Ten obiekt był pewnego rodzaju łącznikiem między widokiem i kontrolerem, przy czym ten sposób działania nie zawsze był intuicyjny bądź przewidywalny.

To prowadziło do trudności, zwłaszcza dla początkujących, w zrozumieniu sposobu synchronizacji danych między modelem i widokiem. Ponadto w kontrolerach zagnieżdżonych obiekt `$scope` mógł dziedziczyć właściwości po zasięgach nadrzędnych, utrudniając tym samym monitorowanie tego, gdzie określona właściwość obiektu `$scope` była pierwotnie zdefiniowana bądź zmodyfikowana.

Wspomniane dziedziczenie mogło prowadzić do nieoczekiwanych efektów ubocznych w aplikacji, w szczególności podczas pracy z zagnieżdżonymi zasięgami, w których zasięgi nadrzędny i potomny mogły przypadkowo na siebie wpływać. Koncepcja hierarchii zasięgu i dziedziczenia prototypowego, na którym bazowała ta koncepcja, często były sprzeczne z bardziej tradycyjnymi i znanymi regułami określania zasięgu leksykalnego w JavaScripcie, co tylko jeszcze bardziej utrudniało poznawanie frameworka.

Z kolei React łączy informacje o stanie z wymagającym ich komponentem, a tym samym pozwala zupełnie wyeliminować omówiony problem.

Ograniczona liczba narzędzi programistycznych

AngularJS nie oferował zbyt wielu narzędzi programistycznych do debugowania i profilowania wydajności działania, zwłaszcza w porównaniu do narzędzi dostępnych dla biblioteki React, takich jak Replay.io, które mają rozbudowane możliwości w zakresie debugowania podróży w czasie podczas tworzenia aplikacji za pomocą Reacta.

Poznaj Reacta

Mniej więcej właśnie w tym czasie React zaczął osiągać znaczącą pozycję. Jedną z jego podstawowych cech była architektura oparta na komponentach. Wprawdzie jej implementacja jest odmienna, ale kryjąca się za nią koncepcja pozostaje podobna — to optymalne rozwiązanie w zakresie wykorzystania wielokrotnego użycia komponentów do budowania interfejsów użytkownika na potrzeby internetu oraz na innych platform.

W celu dołączania widoków do modeli AngularJS używał dyrektyw, natomiast w Reaccie wprowadzono składnię JSX oraz znacznie prostszy model komponentu. Można się spotkać z opinią, że bez gruntu przygotowanego przez AngularJS w zakresie promowania architektury opartej na komponentach i modułów Angulara przejście do modelu oferowanego przez Reacta niekoniecznie byłoby takie płynne i bezproblemowe.

Wprowadzony we frameworku AngularJS model dwukierunkowego dołączania danych był standardem przemysłowym. Jednak miał również pewne wady, takie jak potencjalne problemy z wydajnością działania w ogromnych aplikacjach. Twórcy Reacta wyciągnęli z tego wnioski i opracowali wzorzec jednokierunkowego przepływu danych, pozostawiając tym samym programistom większą kontrolę nad aplikacjami oraz ułatwiając zrozumienie, jak dane zmieniają się wraz z upływem czasu.

W Reaccie pojawił się również tzw. wirtualny model DOM, który omówię dokładnie w rozdziale 3. Ta koncepcja pomogła w znacznej poprawie wydajności działania poprzez minimalizację bezpośrednich operacji na modelu DOM. Trzeba w tym miejscu dodać, że AngularJS często uciekał się do bezpośredniego przeprowadzania operacji na modelu DOM, co mogło prowadzić do problemów z wydajnością działania oraz innych związanych z niespójnymi informacjami o stanie, jak to dokładniej wyjaśniłem podczas omawiania biblioteki jQuery.

Mając to wszystko na uwadze, można powiedzieć, że AngularJS spowodował dużą zmianę w praktykach związanych z tworzeniem aplikacji internetowych. Byłoby niedopatrzeniem z mojej strony, gdybym nie wspomniał, że wprowadzenie tego frameworka nie tylko zrewolucjonizowało tworzenie aplikacji internetowych, ale również utorowało drogę do opracowania kolejnych frameworków i bibliotek, z których jedną jest React.

Warto wyjaśnić, jak React się w to wpisuje oraz skąd się pojawił. W owym czasie uaktualnienia interfejsu użytkownika wciąż były stosunkowo trudnym i nierozwiązanym problemem. Obecnie także są dalekie od rozwiązania, ale dzięki pojawieniu się Reacta uaktualnienia stały się znacznie prostsze, sam React zaś zainspirował programistów, czego wynikiem było powstanie innych bibliotek, np. SolidJS i Qwik. Serwis Facebook firmy Meta również doświadcza problemu związanego ze skomplikowanym interfejsem użytkownika aplikacji internetowej używanej na ogromną skalę. W efekcie Meta opracowała wiele wewnętrznych rozwiązań, które uzupełniały istniejącą wówczas technologię. Wśród tych rozwiązań był BoltJS, czyli narzędzie określane przez inżynierów Facebooka jako „łączące" wiele lubianych przez nich produktów. Rozwiązanie składające się z połączonych ze sobą narzędzi zostało opracowane, aby uaktualnianie interfejsu użytkownika aplikacji Facebooka stało się znacznie bardziej intuicyjne.

Mniej więcej w tym samym czasie Jordan Walke, czyli jeden z inżynierów Facebooka, wpadł na zaskakujący pomysł, który zmienił status quo i umożliwił zastępowanie minimalnych fragmentów stron internetowych nowymi, gdy te fragmenty zostały uaktualnione. Jak już wcześniej wyjaśniłem, biblioteki JavaScriptu zarządzają relacjami między widokiem (interfejs użytkownika) i modelem (pod względem koncepcyjnym jest to źródło danych) za pomocą paradygmatu nazywanego dwukierunkowym dołączaniem danych. Wobec omówionych wcześniej ograniczeń takiego modelu Jordan uznał, że znacznie lepszym rozwiązaniem będzie zastosowanie paradygmatu określanego mianem jednokierunkowego przepływu danych. Ten paradygmat był nie tylko znacznie prostszy, ale również ułatwiał synchronizację widoków i modeli. W taki sposób narodziła się architektura jednokierunkowa, która stanowiła podstawy dla powstania biblioteki React.

Wartość propozycji Reacta

Wystarczy historii na dzisiaj. W tym momencie masz już wystarczający kontekst, aby zacząć rozumieć, dlaczego React jest ważnym produktem. Biorąc pod uwagę to, jak łatwo można było wpaść w pułapkę niebezpiecznego, nieprzewidywalnego i nieefektywnego kodu w JavaScripcie w aplikacji działającej na dużą skalę, nic dziwnego, że istniała potrzeba rozwiązania prowadzącego w kierunku sukcesu, który *przypadkowo okazał się zwycięstwem*. Warto więc dokładnie wyjaśnić, jak Reactowi udało się tego dokonać.

Kod deklaratywny kontra imperatywny

React oferuje deklaratywną abstrakcję modelu DOM. Więcej szczegółowych informacji na temat działania tej abstrakcji znajdziesz w dalszej części książki. W tym miejscu chcę jedynie wyjaśnić, że React pozwala tworzyć kod w sposób wyrażający *efekt, który chcemy zobaczyć*, i sam zajmuje się niezbędnymi działaniami w celu jego *osiągnięcia*. Gwarantuje przy tym, że interfejs użytkownika zostanie utworzony i będzie działał w sposób bezpieczny, przewidywalny i efektywny.

Spójrz na utworzoną wcześniej aplikację listy. Korzystając z Reacta, można ją utworzyć w następujący sposób:

```
function MyList() {
  const [items, setItems] = useState(["I love"]);

  return (
    <div>
      <ul>
        {items.map((i) => (
          <li key={i /* Zapewnienie unikatowości elementów */}>{i}</li>
        ))}
      </ul>
      <NewItemForm onAddItem={(newItem) => setItems([...items, newItem])} />
    </div>
  );
}
```

Zauważ, że w poleceniu `return` znajduje się kod przypominający HTML: definiuje efekt, który chcemy uzyskać. W omawianym przykładzie chcę uzyskać pole `NewItemForm` i listę. Jak to zrobić? To już jest zadanie Reacta. Czy trzeba zebrać wszystkie niezbędne elementy listy, aby zostały dodane jednocześnie? A może powinny być dodawane sekwencyjnie, jeden po drugim? Do Reacta należy ustalenie, *jak* osiągnąć efekt końcowy, podczas gdy programista ma zaledwie wskazać, *co* chce otrzymać. W dalszych rozdziałach zagłębimy się w Reacta i dokładnie wyjaśnię, jak ten mechanizm działa w chwili powstawania tej książki.

Czy w trakcie odwoływania się do elementów HTML będziemy korzystać z nazw klas? Czy zostanie użyte wywołanie `getElementById()` JavaScriptu? Nie. React tworzy unikatowe „elementy Reacta", z których korzysta w tle, by wykrywać zmiany i wprowadzać przyrostowe uaktualnienia. Dzięki temu nie musimy zajmować się odczytywaniem z kodu użytkownika nazw klas i innych identyfikatorów, których istnienia nie można zagwarantować: jedynym źródłem prawdy staje się wyłącznie JavaScript w połączeniu z Reactem.

Komponent `MyList` zostaje wyeksportowany do Reacta, który następnie umieszcza go na ekranie w sposób bezpieczny, przewidywalny i wydajny — nie trzeba nic więcej zrobić. Zadaniem komponentu jest dostarczenie opisu przedstawiającego wygląd danego fragmentu interfejsu użytkownika. W tym celu jest używany tzw. *wirtualny model DOM* (*vDOM*), czyli lekki opis oczekiwanej struktury interfejsu użytkownika. Następnie React porównuje wirtualny model DOM *po uaktualnieniu* z wirtualnym modelem DOM *przed uaktualnieniem* i na tej podstawie przygotowuje małe, wydajne uaktualnienie rzeczywistego modelu DOM, aby odpowiadał wirtualnemu. W taki sposób React przeprowadza uaktualnienia modelu DOM.

Wirtualny model DOM

Wirtualny model DOM to koncepcja programistyczna, która umożliwia reprezentowanie rzeczywistego modelu DOM, ale w postaci obiektu JavaScriptu. Nie przejmuj się, jeżeli to wyjaśnienie jest niewystarczające — z rozdziału 3. dowiesz się znacznie więcej na temat tej koncepcji. W tym momencie musisz wiedzieć, że wirtualny model DOM pozwala programistom uaktualniać interfejs użytkownika bez bezpośredniego przeprowadzania operacji na rzeczywistym modelu DOM.

React używa wirtualnego modelu DOM do monitorowania zmian w komponencie i ponownie go wyświetla tylko wtedy, gdy jest to konieczne. Takie podejście okazuje się szybsze i znacznie efektywniejsze niż uaktualnianie całego drzewa modelu DOM po wystąpieniu każdej zmiany.

W bibliotece React wirtualny model DOM to lekka reprezentacja rzeczywistego drzewa modelu DOM. Jest to zwykły obiekt JavaScriptu przedstawiający strukturę i właściwości elementów interfejsu użytkownika. React tworzy i uaktualnia wirtualny model DOM w celu jego dopasowania do rzeczywistego drzewa modelu DOM. Wszelkie zmiany w wirtualnym modelu DOM zostają wprowadzone również w rzeczywistym modelu DOM za pomocą procesu *rekoncyliacji*.

W rozdziale 4. dokładnie omówię ten proces. W tym miejscu przedstawię jedynie krótkie wprowadzenie do rekoncyliacji i kilka przykładów. Aby zrozumieć sposób działania wirtualnego modelu DOM, musisz powrócić do naszego wcześniejszego przykładu z przyciskiem *Polub*. Zajmiemy się utworzeniem komponentu Reacta, który będzie odpowiedzialny za wyświetlenie przycisku *Polub* i liczby polubień. Gdy użytkownik kliknie przycisk, liczba polubień powinna się zwiększyć o 1.

Oto kod naszego komponentu:

```
import React, { useState } from "react";

function LikeButton() {
  const [likes, setLikes] = useState(0);

  function handleLike() {
    setLikes(likes + 1);
  }

  return (
    <div>
      <button onClick={handleLike}>Polub</button>
      <p>Liczba polubień: {likes}</p>
    </div>
  );
}

export default LikeButton;
```

W tym fragmencie kodu zaczep `useState` został użyty do utworzenia zmiennej o nazwie `likes` przeznaczonej do przechowywania liczby polubień. Jak być może wiesz, ów zaczep (ang. *hook*) to funkcja specjalna pozwalająca używać funkcji Reacta, takich jak metody związane ze stanem i cyklem życiowym, w komponentach funkcyjnych. Zaczepy umożliwiają wielokrotne użycie logiki zawierającej informacje o stanie bez konieczności zmiany hierarchii komponentu. To niezwykle ułatwia wyodrębnianie zaczepów i ich współdzielenie przez komponenty, a nawet udostępnienie ich społeczności w postaci pakietów otwartoźródłowych.

Zdefiniowana została również funkcja o nazwie `handleLike()`, która inkrementuje o 1 liczbę polubień po kliknięciu przycisku. Na koniec za pomocą składni JSX następuje wygenerowanie przycisku *Polub* i liczby polubień.

Teraz dokładniej omówię sposób działania wirtualnego modelu DOM w tym przykładzie.

Gdy komponent `LikeButton` jest generowany po raz pierwszy, React tworzy drzewo wirtualnego modelu DOM, które odpowiada drzewu rzeczywistego modelu DOM. W tym wirtualnym modelu DOM znajduje się pojedynczy element `<div>` zawierający elementy `<button>` i `<p>`:

```
{
  $$typeof: Symbol.for('react.element'),
  type: 'div',
  props: {},
  children: [
    {
      $$typeof: Symbol.for('react.element'),
      type: 'button',
      props: { onClick: handleLike },
      children: ['Polub']
    },
    {
      $$typeof: Symbol.for('react.element'),
      type: 'p',
      props: {},
      children: ['Liczba polubień ', 0]
    }
  ]
}
```

Właściwość `children` elementu `<p>` zawiera wartość zmiennej stanu `likes`, która początkowo wynosi zero.

Gdy użytkownik kliknie przycisk *Polub*, następuje wywołanie funkcji `handleLike()`, która uaktualnia wartość zmiennej stanu `likes`, Następnie React tworzy nowe drzewo wirtualnego modelu DOM, odzwierciedlające uaktualniony stan:

```
{
  type: 'div',
  props: {},
  children: [
    {
      type: 'button',
      props: { onClick: handleLike },
      children: ['Polub']
    },
    {
      type: 'p',
      props: {},
      children: ['Liczba polubień ', 1]
    }
  ]
}
```

Zauważ, że drzewo wirtualnego modelu DOM zawiera dokładnie te same elementy co wcześniej, przy czym właściwość `children` elementu `<p>` została uaktualniona w celu odzwierciedlenia nowej liczby polubień, która wzrosła z 0 do 1. Później przeprowadzany jest wspomniany już wcześniej proces *rekoncyliacji*, w trakcie którego nowy wirtualny model DOM zostaje porównany z poprzednim. Warto pokrótce przeanalizować ten proces.

Po przygotowaniu drzewa nowego wirtualnego modelu DOM React przeprowadza rekoncyliację w celu ustalenia różnic między nowym i poprzednim drzewem wirtualnego modelu DOM. W trakcie tego procesu odbywa się porównanie poprzedniego drzewa wirtualnego modelu DOM z nowo wygenerowanym oraz ustalenie, które fragmenty rzeczywistego modelu DOM wymagają uaktualnienia. Jeżeli chcesz wiedzieć, *jak* to dokładnie się odbywa, wiele dokładnych informacji na ten temat znajdziesz w rozdziale 4. Teraz przejdę do naszego przycisku *Polub*.

W omawianym przykładzie React porównuje poprzednie i nowe drzewo wirtualnego modelu DOM i ustala, że element `<p>` został zmieniony: a dokładnie zmianie uległy jego właściwości, stan bądź jedno i drugie. To pozwala Reactowi oznaczyć komponent jako zmodyfikowany, czyli przeznaczony do uaktualnienia. Następnie React określa minimalną liczbę uaktualnień, które muszą zostać wprowadzone w rzeczywistym modelu DOM, aby odpowiadał on stanowi przedstawianemu przez połączenie modelu DOM i nowego wirtualnego modelu DOM. Ostatecznie w rzeczywistym modelu DOM są wprowadzane uaktualnienia odpowiadające zmianom w nowym wirtualnym modelu DOM.

React uaktualnia jedynie wymagające tego elementy modelu DOM, aby w ten sposób zmniejszyć liczbę operacji na modelu DOM. Takie podejście okazuje się znacznie szybsze i efektywniejsze niż uaktualnianie całego drzewa modelu DOM za każdym razem, gdy została w nim wprowadzona zmiana.

Wirtualny model DOM okazał się potężnym i ważnym wynalazkiem dla nowoczesnego internetu, a nowe biblioteki takie jak Preact i Inferno zaczęły go wykorzystywać, gdy dowiódł swojej przydatności w bibliotece React. Dokładne omówienie wirtualnego modelu DOM znajdziesz w rozdziale 4., natomiast teraz krótko wyjaśnię model komponentu.

Model komponentu

React zachęca do „myślenia w kategoriach komponentów". To oznacza podział aplikacji na mniejsze fragmenty oraz ich dodawanie do większego drzewa w celu przygotowania aplikacji. Model komponentu jest kluczową koncepcją Reacta, dzięki której ta biblioteka ma tak potężne możliwości. Powody tego są następujące:

- Model komponentu zachęca do wielokrotnego użycia tego samego elementu wszędzie. Jeżeli zostanie uszkodzony i później naprawiony w jednym miejscu, ta zmiana zostanie zastosowana także we wszystkich pozostałych miejscach użycia danego elementu. Mamy tutaj do czynienia z regułą programistyczną „nie powtarzaj się" (ang. *don't repeat yourself*, DRY), która jest istotną koncepcją stosowaną podczas tworzenia oprogramowania. Na przykład jeśli masz komponent `Button`, możesz go wykorzystać w wielu innych miejscach aplikacji. Następnie gdy zajdzie potrzeba zmiany stylu przycisku, modyfikujesz go w jednym miejscu, a zostanie to odzwierciedlone w każdym wystąpieniu danego komponentu.
- React będzie w stanie łatwiej monitorować komponenty i poprawiać wydajność działania za pomocą na przykład memoizacji, przetwarzania wsadowego i innych optymalizacji w tle, jeśli zapewnisz tej bibliotece możliwość nieustannego identyfikowania określonych komponentów i ich uaktualniania na bieżąco. Jest to określane mianem *kluczowania* (ang. *keying*). Na przykład jeśli masz komponent `Button`, możesz w nim zdefiniować właściwość o nazwie `key`, a React będzie w stanie na bieżąco monitorować ten komponent `Button` i będzie „wiedział", kiedy go uaktualnić

bądź pominąć uaktualnienie i kontynuować wprowadzanie minimalnych zmian interfejsu użytkownika. Większość komponentów ma niejawne klucze, choć jeśli zachodzi potrzeba, można również jawnie dostarczyć odpowiednie klucze.

- Pomocne będzie wykorzystanie separacji zadań i umieszczanie logiki bliżej fragmentów interfejsu użytkownika, w których ta logika ma zastosowanie. Na przykład jeśli masz komponent `Register ↪Button`, to logikę odpowiedzialną za działania podejmowane po naciśnięciu przycisku można umieścić w tym samym pliku, w którym znajduje się komponent. Nie trzeba tej logiki umieszczać w oddzielnym pliku i później wyszukiwać po kliknięciu przycisku. Komponent `RegisterButton` będzie opakowaniem dla prostszego komponentu `Button` i stanie się odpowiedzialny za obsługę logiki określającej, co się stanie po kliknięciu przycisku. Nosi to nazwę *kompozycji* (ang. *composition*).

Model komponentu Reacta to podstawowa koncepcja kryjąca się za popularnością i sukcesem tego frameworka. Takie podejście do programowania ma wiele zalet, takich jak m.in. większa modułowość, łatwiejsze debugowanie i znacznie efektywniejsze wielokrotne używanie kodu.

Niezmienny stan

Filozofia projektowa kładzie nacisk na paradygmat, w ramach którego stan aplikacji zostaje opisany za pomocą zbioru niemodyfikowalnych wartości. Każde uaktualnienie stanu jest traktowane jako nowa, oddzielna migawka i odniesienie do pamięci. Takie oparte na niezmienności podejście do zarządzania stanem jest kluczowym aspektem wartości propozycji Reacta oraz oferuje kilka zalet podczas tworzenia niezawodnych, efektywnych i przewidywalnych interfejsów użytkownika.

Dzięki wymuszeniu niezmienności React gwarantuje, że komponenty interfejsu użytkownika odzwierciedlają stan w danej chwili. Gdy stan ulega zmianie, nie zajmujemy się jego bezpośrednią modyfikacją, lecz zwracany jest nowy obiekt reprezentujący ten nowy stan. To znacznie ułatwia śledzenie zmian, debugowanie i rozumienie sposobu działania aplikacji. Skoro zmiana stanu jest samodzielną operacją, która nie ma związku z innymi, znacznie zmniejsza się ryzyko występowania drobnych błędów spowodowanych przez współdzielony stan, który może ulec zmianie.

W dalszych rozdziałach wyjaśnię, jak React przeprowadza hurtowe uaktualnienia stanu i przetwarza je asynchronicznie, aby zoptymalizować wydajność działania. Ponieważ stan musi być traktowany jako niemodyfikowalny, te „transakcje" można bezpiecznie agregować i stosować bez obaw, że dane uaktualnienie może doprowadzić do uszkodzenia stanu innego uaktualnienia. W efekcie mamy do czynienia z przewidywalnym zarządzaniem stanem oraz zyskujemy możliwość poprawy wydajności działania aplikacji, zwłaszcza podczas skomplikowanych zmian stanu.

Unikanie modyfikowania stanu jeszcze bardziej wzmacnia najlepsze praktyki stosowane w branży tworzenia oprogramowania. Zachęca programistów do innego spojrzenia na przepływ danych, do zmniejszenia efektów ubocznych oraz do stosowania rozwiązań ułatwiających zrozumienie kodu źródłowego. Przejrzystość niemodyfikowalnego przepływu danych upraszcza model mentalny pomocny w zrozumieniu sposobu działania aplikacji.

Niemodyfikowalność pozwala również stosować potężne narzędzia programistyczne, takie jak debugowanie podróży w czasie (przykładem może być tutaj Replay.io), w których przypadku programiści mogą poruszać się między poszczególnymi zmianami stanu aplikacji i sprawdzać interfejs

użytkownika w danej chwili. Taka możliwość istnieje tylko wtedy, gdy każde uaktualnienie stanu aplikacji jest przechowywane w postaci unikatowej i niezmienionej migawki.

Dążenie Reacta do niezmiennych uaktualnień stanu to świadoma decyzja projektowa, z którą wiąże się wiele korzyści. Pozostaje w zgodzie z nowoczesnymi regułami programowania funkcyjnego, pozwala na efektywne uaktualnienia interfejsu użytkownika, umożliwia optymalizację wydajności działania, zmniejsza prawdopodobieństwo błędów oraz poprawia ogólne wrażenia programisty. To podejście do zarządzania stanem jest fundamentem dla wielu zaawansowanych funkcji Reacta i nadal będzie jego filarem podczas ewolucji Reacta.

Wydanie Reacta

Jednokierunkowy przepływ danych oznaczał radykalne odejście od stosowanego przez lata sposobu tworzenia aplikacji internetowych i spotkał się ze sceptycyzmem. Facebook to ogromna firma z ogromną ilością zasobów, użytkowników oraz inżynierów z własnymi opiniami i pomysłami, co jeszcze bardziej utrudniło przejście do nowego podejścia. Po wielu analizach React okazał się wewnętrznym sukcesem. Został zastosowany najpierw przez Facebooka, a później przez Instagrama.

W 2013 roku React został udostępniony światu jako oprogramowanie otwartoźródłowe i wówczas spotkał się z ogromnym sprzeciwem społeczności. Programiści zaciekle krytykowali Reacta za używanie składni JSX i oskarżali Facebooka o „umieszczanie kodu w języku HTML w kodzie w języku JavaScript" i tym samym złamanie zasady podziału zadań. Facebook stał się firmą, która „rewiduje najlepsze praktyki" i psuje internet. Ostatecznie po wolnym i opornym procesie przyjmowania Reacta przez inne firmy, takie jak Netflix, Airbnb i The New York Times Company, biblioteka ta stała się praktycznie standardem podczas budowania interfejsów użytkownika w aplikacjach internetowych.

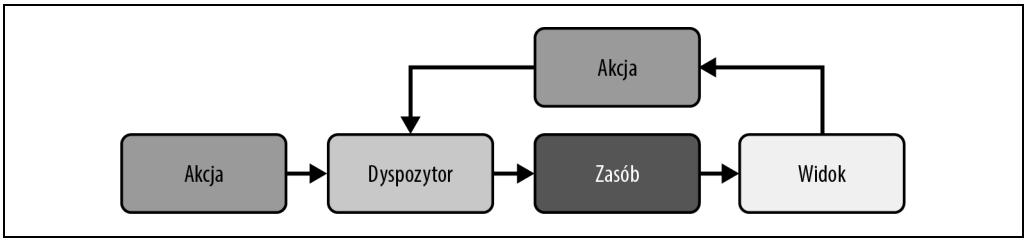
W tej historii pominąłem wiele szczegółów, ponieważ nie mają one znaczenia z perspektywy materiału zamieszczonego w książce. Jednak bardzo ważne jest zrozumienie kontekstu Reacta, zanim zagłębimy się w szczegóły: konkretną klasę problemów technicznych, do których rozwiązywania powstał React. Jeżeli bardziej interesuje Cię historia Reacta, jej dokładne omówienie znajdziesz w filmie *React.js: The Documentary* zamieszczonym przez użytkownika Honeypot w serwisie YouTube (*https://youtu.be/8pDqJVdNa44*).

Biorąc pod uwagę to, że Facebook borykał się z tymi problemami na ogromną skalę, React był pionierem w opartym na komponencie podejściu do budowania interfejsów użytkownika. Umożliwiał rozwiązywanie nie tylko wymienionych wcześniej problemów, ale także innych. W podejściu wykorzystywanym przez Reacta poszczególne komponenty są oddzielnymi jednostkami kodu, które mogą być wielokrotne używanie i składać się z innych komponentów, aby w ten sposób umożliwiać powstawanie znacznie bardziej złożonych interfejsów użytkownika.

Rok po wydaniu Reacta jako oprogramowania otwartoźródłowego Facebook udostępnił Fluxa, czyli wzorzec przeznaczony do zarządzania przepływem danych w aplikacjach tworzonych z wykorzystaniem Reacta. Flux stanowił odpowiedź na wyzwania pojawiające się podczas zarządzania przepływem danych w ogromnych aplikacjach i stanowił kluczowy element ekosystemu Reacta. Warto więc nieco dokładniej poznać ten wzorzec i zobaczyć, jak wpasowuje się w ten ekosystem.

Architektura Flux

Flux to architekturalny wzorzec projektowy przeznaczony do budowania aplikacji internetowych działających po stronie klienta. Został spopularyzowany przez firmę Facebook (obecnie Meta). W sposób graficzny przedstawiłem go na rysunku 1.3. Kładzie nacisk na jednokierunkowy przepływ danych, dzięki któremu przepływ danych w aplikacji jest znacznie bardziej przewidywalny.



Rysunek 1.3. Architektura Flux

Oto kluczowe koncepcje architektury Flux:

Akcja

To prosty obiekt zawierający nowe dane i właściwość identyfikującą typ. Akcje reprezentują zewnętrzne i wewnętrzne dane wejściowe dla systemu, takie jak działania podejmowane przez użytkownika, odpowiedzi udzielane przez serwer oraz dane wejściowe formularzy. Poprzez centralny dyspozytor akcje są przekazywane do różnych magazynów.

```
// Przykład obiektu akcji
{
  type: 'ADD_TODO',
  text: 'Poznaj architekturę Flux'
}
```

Dyspozytor

Dyspozytor jest centralnym hubem w architekturze Flux. Otrzymuje akcje i przekazuje je do zarejestrowanych magazynów w aplikacji. Zarządza listą wywołań zwrotnych, a każdy magazyn rejestruje się i swoje wywołania zwrotne w dyspozytorze. Po wywołaniu akcji jest ona przekazywana do wszystkich zarejestrowanych wywołań zwrotnych.

```
// Przykład przekazywania akcji
Dispatcher.dispatch(action);
```

Magazyn

Magazyn zawiera logikę i informacje o stanie aplikacji. Pod pewnymi względami jest podobny do modelu w architekturze MVC, ale zarządza informacjami o stanie i wieloma obiektami. Magazyn rejestruje się u dyspozytora oraz dostarcza wywołania zwrotne przeznaczone do obsługi akcji. Gdy stan magazynu jest uaktualniany, emituje zdarzenie zmiany w celu ostrzeżenia widoków, że coś uległo zmianie.

```
// Przykład magazynu
class TodoStore extends EventEmitter {
  constructor() {
    super();
```

```
    this.todos = [];
  }

  handleActions(action) {
    switch (action.type) {
      case "ADD_TODO":
        this.todos.push(action.text);
        this.emit("change");
        break;
      default:
      // Brak operacji
    }
  }
}
```

Widok

To komponent Reacta. Nasłuchuje pochodzących z magazynów zdarzeń zmian i uaktualnia się w przypadku zmodyfikowania danych, które są używane w widoku. Widok może również tworzyć nowe akcje w celu uaktualnienia stanu systemu, przygotowując tym samym jednokierunkowy cykl przepływu danych.

Architektura Flux promuje jednokierunkowy przepływ danych poprzez system, co ułatwia monitorowanie zmian zachodzących na przestrzeni czasu. Taka przewidywalność może być później wykorzystywana jako punkt wyjścia dla kompilatorów w celu dalszej optymalizacji kodu źródłowego, jak ma to miejsce w przypadku React Forget (więcej informacji na ten temat znajdziesz w dalszej części rozdziału).

Zalety architektury Flux

Architektura Flux ma wiele zalet oraz pomaga w zarządzaniu złożonością i ułatwia późniejszą obsługę techniczną aplikacji internetowych. Oto niektóre spośród jej zalet:

Pojedyncze źródło prawdy

Architektura Flux kładzie nacisk na istnienie tylko pojedynczego źródła prawdy o stanie aplikacji, który jest przechowywany w magazynach. Scentralizowane zarządzanie informacjami o stanie powoduje, że sposób działania aplikacji jest znacznie bardziej przewidywalny i łatwiejszy do zrozumienia. Eliminuje złożoność związaną z koniecznością obsługi wielu niezależnych źródeł prawdy, która może prowadzić do błędów i niespójnego stanu w aplikacji.

Możliwość przetestowania

Doskonale zdefiniowane struktury architektury Flux oraz przewidywalny przepływ danych powodują, że aplikacja jest możliwa do przetestowania. Separacja zadań między różnymi fragmentami systemu (np. akcje, dyspozytor, magazyny i widoki) pozwala definiować oddzielne testy jednostkowe dla poszczególnych części aplikacji. Ponadto łatwiej można utworzyć testy, gdy przepływ danych jest jednokierunkowy oraz gdy informacje o stanie są przechowywane w konkretnych i przewidywalnych lokalizacjach.

Separacja zadań

Architektura Flux stosuje czytelną separację zadań między różnymi częściami systemu, o czym wspomniałem już wcześniej. Dzięki temu system staje się bardziej modularny oraz łatwiejszy do przeanalizowania i obsługi technicznej. Każdy fragment ma wyraźnie zdefiniowaną rolę, a jednokierunkowy przepływ danych wyraźnie wskazuje, jak poszczególne części ze sobą współdziałają.

Architektura Flux zapewnia solidne podstawy pomocne w budowaniu niezawodnych, skalowalnych i łatwych w późniejszej obsłudze technicznej aplikacji. Kładzie nacisk na jednokierunkowy przepływ danych, istnienie jednego źródła prawdy, a separacja zadań prowadzi do powstawania aplikacji, które są łatwiejsze do zbudowania, przetestowania i debugowania.

Skąd tak duże znaczenie Reacta?

React jest ważną technologią, ponieważ umożliwia programistom tworzenie interfejsów użytkownika charakteryzujących się większą przewidywalnością i niezawodnością oraz deklaratywne wyrażenie oczekiwań tego, *co ma się pojawić na ekranie*, podczas gdy biblioteka samodzielnie ustali, *jak* osiągnąć zdefiniowany cel poprzez efektywne wykonywanie przyrostowych uaktualnień modelu DOM. Zachęca również do myślenia w kategoriach komponentów, co ułatwia separację zadań i wielokrotne używanie kodu źródłowego. React został dokładnie przetestowany w produktach firmy Meta i opracowany do używania na ogromną skalę. Ponadto jest oprogramowaniem otwartoźródłowym, a więc można z niego korzystać bezpłatnie.

React ma również ogromny i aktywny ekosystem obejmujący szeroką gamę narzędzi, bibliotek oraz zasobów dostępnych dla programistów. W tym ekosystemie znajdziesz narzędzia do testowania, debugowania i optymalizowania aplikacji tworzonych z użyciem Reacta, a także wiele bibliotek do najczęściej wykonywanych zadań, takich jak zarządzanie danymi, obsługa routingu i zarządzanie informacjami o stanie. Ponadto społeczność Reacta jest wysoce zaangażowana i pomocna, w internecie dostępnych jest wiele zasobów, forów i społeczności, które pomagają programistom poznawać bibliotekę oraz rozwijać związane z nią umiejętności.

Biblioteka React nie jest związana z żadną konkretną platformą, więc może być używana do budowania aplikacji internetowych dla wielu różnych platform, m.in. desktopowych, mobilnych i rzeczywistości wirtualnej. Ta elastyczność powoduje, że React to atrakcyjny produkt dla programistów, którzy muszą tworzyć aplikacje przeznaczone dla wielu platform, ponieważ pozwala im używać pojedynczej bazy kodu, na podstawie której można budować aplikacje działające na wielu różnych urządzeniach.

Podsumowując: wartość propozycji Reacta jest związana z jego architekturą opartą na komponencie, deklaratywnym modelem programowania, wirtualnym modelem DOM, składnią JSX, obszernym ekosystemem, niezależną od platformy naturą oraz wsparciem ze strony firmy Meta. Połączenie tych wszystkich cech powoduje, że React to atrakcyjna opcja dla programistów, którzy muszą tworzyć szybko działające, skalowalne i łatwe w późniejszej obsłudze technicznej aplikacje internetowe. Niezależnie od tego, czy tworzysz prostą witrynę, czy skomplikowaną aplikację korporacyjną, React może pomóc w osiągnięciu wyznaczonych celów w sposób znacznie efektywniejszy niż wiele innych technologii. Czas na krótkie podsumowanie rozdziału.

Podsumowanie

W tym rozdziale pokrótce przedstawiłem historię biblioteki React i jej początkową proponowaną wartość oraz wyjaśniłem, jak rozwiązuje problemy związane z niebezpiecznym, nieprzewidywalnym i nieefektywnym uaktualnianiem interfejsu użytkownika na ogromną skalę. Wspomniałem również o modelu komponentu i wyjaśniłem, dlaczego okazał się rewolucyjnym podejściem dla interfejsów w internecie. Po lekturze niniejszego rozdziału będziesz mieć dużą wiedzę na temat korzeni Reacta i jego pochodzenia oraz będziesz w stanie wymienić największe zalety i wartość tej biblioteki.

Pytania

Sprawdź, czy rozumiesz zagadnienia omówione w tym rozdziale. Poświęć chwilę, by odpowiedzieć na następujące pytania:

1. Co było katalizatorem do opracowania Reacta?
2. W jaki sposób React stanowi usprawnienie dla wcześniej stosowanych wzorców typu MVC i MVVM?
3. Co jest wyjątkowego w architekturze Flux?
4. Jakie korzyści płyną z abstrakcji programowania deklaratywnego?
5. Jaka jest rola wirtualnego modelu DOM w zapewnieniu efektywnych uaktualnień interfejsu użytkownika?

Jeżeli masz trudności z odpowiedzeniem na nie, warto ponownie przeczytać ten rozdział, a jeżeli Twoje odpowiedzi są bezbłędne, zapraszam do lektury następnego rozdziału.

Co dalej?

W rozdziale 2. zagłębimy się bardziej w abstrakcję deklaratywną, która pozwala wyrazić to, co chcemy zobaczyć na ekranie. Omówię składnię i wewnętrzny sposób działania JSX, czyli języka przypominającego kod w HTML w kodzie w JavaScripcie. Składnia JSX na początku istnienia Reacta przysporzyła mu wiele problemów. Ostatecznie jednak okazała się idealnym sposobem na tworzenie interfejsów użytkownika na potrzeby aplikacji internetowych oraz miała wpływ na wiele kolejnych bibliotek przeznaczonych do tworzenia interfejsów użytkownika.

To jest pozycja obowiązkowa dla każdego, kto chce poznać bibliotekę React i jej nieustannie ewoluujący ekosystem!

Matheus Albuquerque, programista Google

React jest biblioteką języka programowania JavaScript. Służy do tworzenia interfejsów użytkownika różnych aplikacji. Jej twórcą jest Jordan Walke, programista Facebooka, który chciał równocześnie uprościć pracę programisty i poprawić komfort użytkowania gotowej aplikacji. Dziś biegłość w korzystaniu z tej biblioteki oznacza po prostu lepsze aplikacje!

Dzięki tej praktycznej książce zrozumiesz podstawowe koncepcje związane z biblioteką React, takie jak składnia JSX, wzorce zaawansowane, wirtualny model DOM, mechanizm rekoncyliacji Reacta, a także zaawansowane techniki optymalizacji. W rzeczywistości jest to dość złożona biblioteka, jednak tutaj jej tajniki zostały wyjaśnione w wyjątkowo przystępny sposób. A to oznacza, że szybko i dogłębnie zrozumiesz mechanizmy kryjące się za działaniem Reacta, zdobędziesz umiejętności pozwalające na tworzenie intuicyjnego kodu Reacta, zrozumiesz jego niuanse i koncepcje — i przejdziesz na zupełnie nowy poziom biegłości. Efekt? Osiągniesz mistrzostwo w tworzeniu dynamicznych, responsywnych i wydajnych interfejsów!

W książce między innymi:

- jak React działa na niższym poziomie
- tworzenie aplikacji Reacta i ich optymalizowanie
- budowanie niezawodnych i skalowalnych aplikacji Reacta
- mechanizmy udostępniane przez Reacta, takie jak reduktor, stan, odwołanie

Tejas Kumar od dekady pracuje z Reactem, zdobywał doświadczenie, tworząc kod dla wielu startupów. Wielokrotnie był prelegentem na konferencjach, podczas szkoleń i występów gościnnych. Chętnie korzysta ze swojego bogatego doświadczenia w uczeniu efektywnego tworzenia aplikacji Reacta.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-1634-0
9 788328 916340
Cena: 79,00 zł