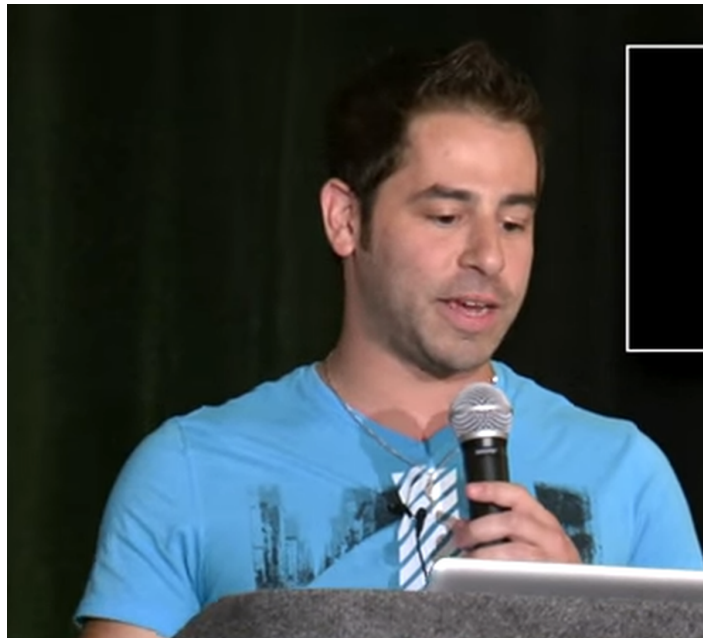


7.1 7.1 Teoria

Step 1

`React` to biblioteka JavaScript służąca do tworzenia interfejsów użytkownika na różnych platformach. `React` zapewnia potężny model do pracy oraz pomaga budować interfejsy użytkownika w sposób deklaratywny i oparty na tzw. komponentach. Innymi znanymi bibliotekami JavaScript mający podobną funkcjonalność jak `React` to `Vue`, `Preact`, `Angular`, `Ember`, `Webpack` czy `Redux`. `React` jest często główną częścią aplikacji frontendowych o ma podobne funkcje z wyżej wymienionymi bibliotekami. W rzeczywistości wiele popularnych technologii frontendowych korzysta obecnie z rozwiązań nowatorskich, jakie przyniósł `React`. Natomiast nadal ustala najlepsze praktyki, których głównym celem jest zapewnienie programistom ekspresyjnego modelu i wydajnej technologii do tworzenia aplikacji z interfejsem użytkownika.

`React` został po raz pierwszy stworzony przez Jordana Walke, inżyniera oprogramowania z Facebooka. Został włączony do kanału informacyjnego Facebooka w 2011 r., A później na Instagramie, kiedy został przejęty przez Facebooka w 2012 r. Na JSConf 2013 `React` stał się oprogramowaniem typu open source i dołączył do zatłoczonej kategorii bibliotek UI, takich jak `jQuery`, `Angular`, `Dojo`, `Meteor` i inni. W tym czasie `React` był opisywany jako „V w MVC”. Innymi słowy, komponenty `React` działały jako warstwa widoku lub interfejs użytkownika dla aplikacji JavaScript.



<https://everipedia-storage.s3-accelerate.amazonaws.com/ProfilePics/6666624538227426824.png>

Od tego momentu adopcja społeczności zaczęła się rozprzestrzeniać. W styczniu 2015 r. Netflix ogłosił, że używa `Reacta` do rozwijania interfejsu użytkownika. Jeszcze w tym samym miesiącu została wydana biblioteka `React Native`, służąca do tworzenia aplikacji mobilnych przy użyciu `React`. Facebook wydał również `ReactVR`, kolejne narzędzie, które umożliwiło `Reactowi` szerszy zakres celów renderowania. W 2015 i 2016 roku na scenie pojawiła się ogromna liczba popularnych narzędzi, takich jak `React Router`, `Redux` i `Mobx` do obsługi zadań, takich jak routing i zarządzanie stanem. W końcu `React` był rozliczany jako biblioteka: zajmująca się implementacją określonego zestawu funkcji, a nie dostarczaniem narzędzia dla każdego przypadku użycia. Kolejnym ważnym wydarzeniem na osi czasu było wydanie `React Fibre` w 2017 roku. `Fibre` było przeróbką algorytmu renderowania `Reacta`, który był w pewnym sensie magiczny w wykonaniu. To było pełne przepisanie wewnętrznych elementów `Reacta`, które prawie nic nie zmieniło w publicznym API. Był to sposób na uczynienie `React` bardziej nowoczesnym i wydajnym bez wpływu na jego użytkowników. Niedawno w 2019 roku wydano tzw. `hooks`, które są nowym sposobem dodawania i udostępniania logiki stanu między komponentami. W przyszłości nieuchronnie zobaczymy więcej zmian, ale jednym z powodów sukcesu `React` jest silny zespół, który pracował nad projektem przez lata. Zespół jest ambitny, ale ostrożny, forsuje optymalizacje z myślą o przyszłości, jednocześnie stale biorąc pod uwagę wpływ, jaki wszelkie zmiany w bibliotece wywierają kaskadowo na społeczność.

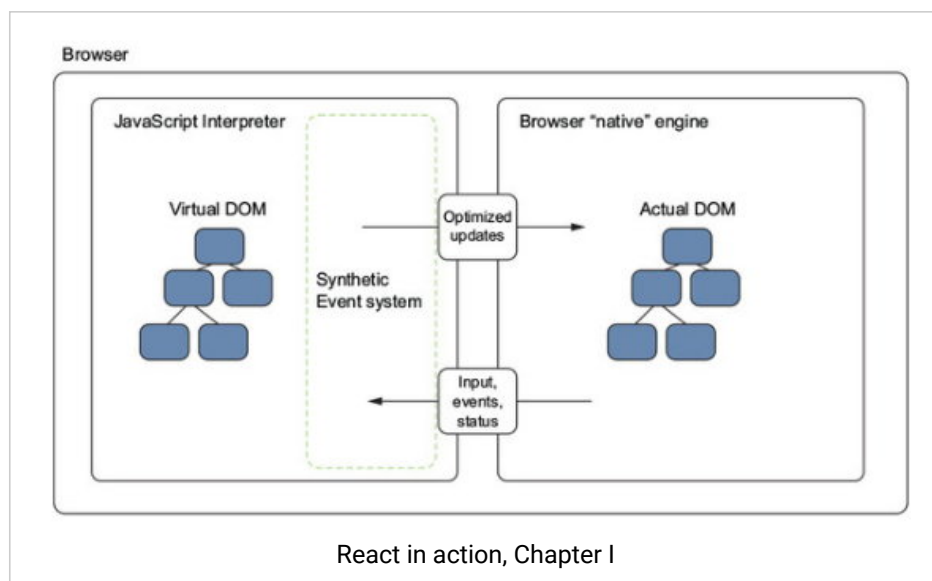
Koncepcja wprowadzona w `React` opiera się na głębokich obszarach informatyki i technik inżynierii oprogramowania. Czerpie on w szerokim zakresie z funkcjonalnych i obiektowych koncepcji programowania i koncentruje się na komponentach jako podstawowych jednostkach, służących do budowania interfejsu użytkownika. W `React` tworzymy interfejsy właśnie za pomocą **komponentów**. Komponenty często odpowiadają aspektom interfejsu użytkownika, takie jak paski nawigacyjne, nagłówki itd. Ponadto dzięki `React` możemy również formatować dane, stylizować je czy przeprowadzać tzw. routing po stronie klienta. Komponenty w `React` powinny być łatwe do zrozumienia oraz zintegrowane z innymi komponentami. Każdy komponent ma przewidywany cykl życia, a to oznacza, że potrafią utrzymać swój własny stan, współpracując ze zwykłym JavaScript.

Strona projektu React: <https://pl.reactjs.org/> (<https://pl.reactjs.org/>)

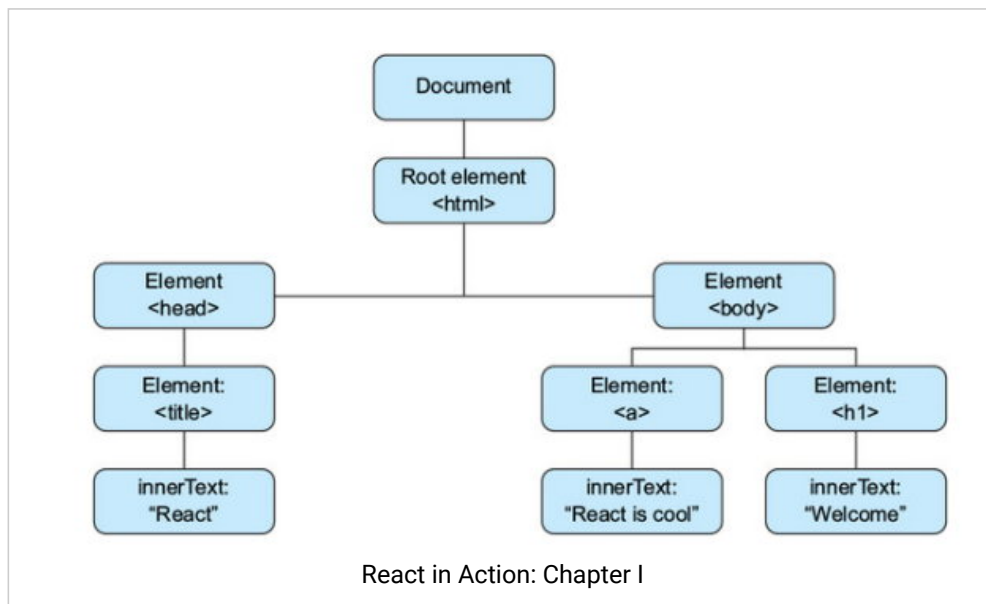


Step 2

Głównym celem w `React` jest dążenie do uproszczenia skomplikowanych zadań i oderwanie od programisty niepotrzebnej złożoności. `React` stara się być wydajny na przykład poprzez zachęcenie do deklaratywności zamiast podejścia imperatywnego. Możemy zadeklarować jak komponenty powinny się zachowywać i wyglądać w różnych stanach, wówczas `React` poradzi sobie ze złożonością, zarządzania aktualizacjami, w wyniku czego będzie aktualizowany interfejs użytkownika w celu odzwierciedlenia zmian. Jedną z głównych technologii `React` jest **wirtualny DOM**. Jest to struktura danych, które naśladują lub odzwierciedlają Model Obiektu Dokumentu (DOM), który istnieje w przeglądarkach. Ogólnie wirtualny DOM będzie służył jako warstwa pośrednia między kodem aplikacji a DOM przeglądarki. Pozwala na ukrycie złożoności wykrywania zmian i zarządzania nimi przez programistę i przeniesienie do wyspecjalizowanej warstwy o nazwie abstrakcja.



DOM (dla przypomnienia Document Object Model), to interfejs programistyczny, który umożliwia programom JavaScript współdziałanie różnych typów dokumentów (HTML, XML, czy SVG). DOM jest w większości synonimem przeglądarek internetowych. Zapewnia on uporządkowany sposób uzyskiwania dostępu, przechowywania i manipulowania różnymi częściami dokumentu. Na wysokim poziomie, DOM jest strukturą drzewiastą, która bardzo odzwierciedla hierarchię dokumentu XML. Struktura ta składa się z poddrzew, które z kolei składają się z węzłów. Za każdym razem, kiedy korzystamy z metod w JavaScript (w przeglądarce internetowej), które uzyskują dostęp, modyfikują czy przetwarzają dane, na pewno korzystają z budowy drzewa DOM.



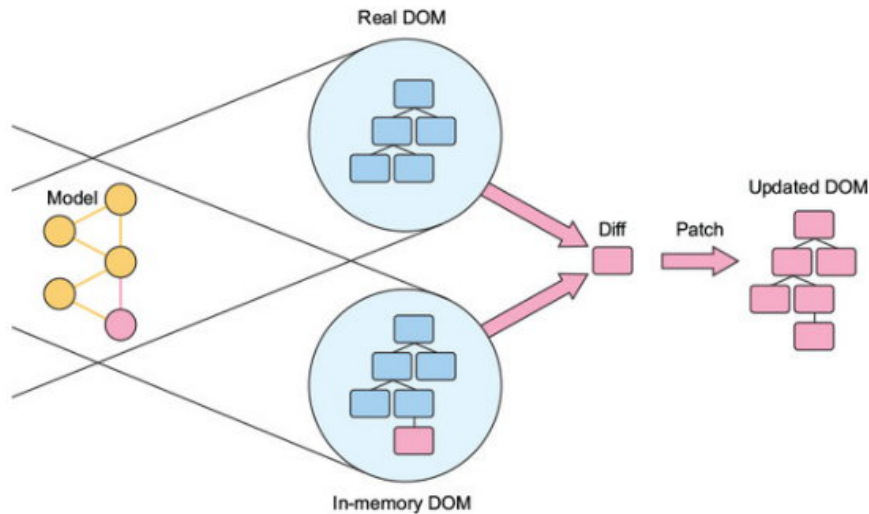
Praca z DOM jest zwykle prosta, ale może być skomplikowana w przypadku dużej strony internetowej. Nie ma potrzeby wchodzenia w bezpośrednią interakcję z DOM podczas tworzenia aplikacji w React.

Step 3

Internetowe interfejsy API w przeglądarkach pozwalają nam na komunikację z dokumentami sieciowymi poprzez JavaScript oraz DOM. Przeglądarka często uzyskuje dostęp do elementu DOM, modyfikuje go lub tworzy wykonanie zapytania w drzewie strukturalnym w celu znalezienia danego elementu do dokonania aktualizacji. Najczęściej jednak może być konieczne ponowne wykonanie układu, rozmiaru i innych działań podczas aktualizacji. Wszystko to może być kosztowne obliczeniowo. React stworzył tzw. wirtualny DOM, który nie obejście tego problemu, ale może pomóc w optymalizacji aktualizacji DOM w celu uwzględnienia tych ograniczeń. Podczas tworzenia i zarządzania dużą aplikacją, która obsługuje dane, które zmieniają się w czasie, może być wymaganych wiele zmian w DOM, a często te zmiany mogą powodować konflikty lub są wykonywane w sposób mniej niż optymalny. Może to skutkować nadmiernie skomplikowanym systemem, nad którym inżynierowie będą mieli trudności z pracą, a użytkownicy prawdopodobnie będą mieli słabe wrażenia. Dlatego wydajność jest kolejnym kluczowym czynnikiem w projektowaniu i wdrażaniu React. Wdrożenie wirtualnego DOM pomaga rozwiązać ten problem, ale należy zauważyć, że został zaprojektowany tak, aby był po prostu „wystarczająco szybki”. Solidny interfejs API i inne rzeczy, takie jak kompatybilność z różnymi przeglądarkami, są ważniejszymi rezultatami wirtualnego DOM Reacta niż ekstremalne skupienie się na wydajności.

Wirtualny DOM Reacta ma kilka podobieństw do innego świata oprogramowania: gier 3D. Gry 3D czasami wykorzystują proces renderowania, który działa mniej więcej w następujący sposób: pobierz informacje z serwera gry, wyślij je do świata gry (wizualna reprezentacja, którą widzi użytkownik), określ, jakie zmiany należy wprowadzić w świecie wizualnym oraz następnie pozwól karcie graficznej określić minimalne niezbędne zmiany. Jedną z zalet tego podejścia jest to, że potrzebujemy tylko zasobów do radzenia sobie ze zmianami przyrostowymi i ogólnie możesz robić rzeczy znacznie szybciej niż gdybyś musiał wszystko aktualizować. Słabo wykonana aktualizacja DOM może być kosztowna, dlatego React stara się skutecznie aktualizować interfejs użytkownika i stosuje metody podobne do gier 3D. React tworzy i utrzymuje wirtualny DOM w pamięci, a renderer, taki jak ReactDOM, obsługuje aktualizację DOM przeglądarki na podstawie zmian. React może przeprowadzać inteligentne aktualizacje i pracować tylko na częściach, które uległy zmianie, ponieważ może używać różnicowania heurystycznego do obliczania, które części pamięci DOM wymagają zmian w DOM.

Figure 1.5. React's diffing and update procedure. When a change happens, React determines differences between the actual and in-memory DOMs. Then it performs an efficient update to the browser's DOM. This process is often referred to as a *diff* ("what changed?") and *patch* ("update only what changed") process.



React in action, Chapter I, p. 21

Wirtualny DOM to coś więcej niż szybkość. Jest wydajny z założenia i generalnie daje w wyniku zgrabne, szybkie aplikacje, które są wystarczająco szybkie, aby sprostać potrzebom nowoczesnych aplikacji internetowych. Inżynierowie tak docenili wydajność i lepszy model myślenia, że wiele popularnych bibliotek JavaScript tworzy własne wersje lub odmiany wirtualnego DOM. Nawet w takich przypadkach ludzie mają skłonność do myślenia, że wirtualny DOM koncentruje się głównie na wydajności. Wydajność jest kluczową cechą Reacta, ale jest drugorzędna w stosunku do prostoty. Wirtualny DOM jest częścią tego, co pozwala odłożyć myślenie o skomplikowanej logice stanu i skupić się na innych, ważniejszych częściach aplikacji.



<https://reactkungfu.com/assets/images/the-difference-between-virtual-dom-and-dom/meme.jpg>

W pracy z Reactem, jest więcej niż prawdopodobne, że będziemy tworzyć aplikacje w JSX. JSX to składnia JavaScript oparta na tagach, która bardzo przypomina HTML. Aby móc pracować z Reactem w przeglądarce, musimy dołączyć dwie biblioteki: `React` i `ReactDOM`. `React` to biblioteka do tworzenia widoków. `ReactDOM` to biblioteka używana do rzeczywistego renderowania interfejsu użytkownika w przeglądarce. Obie biblioteki są dostępne jako skrypty z unpkg CDN (linki znajdują się w poniższym kodzie). Poniżej znajduje się szablon HTML, dzięki któremu możemy działać w React:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Samples</title>
  </head>
  <body>
    <!-- Target container -->
    <div id="root"></div>

    <!-- React library & ReactDOM (Development Version)-->
    <script
src="https://unpkg.com/react@16/umd/react.development.js">
</script>
    <script
src="https://unpkg.com/react-dom@16/umd/react-dom.development.js">
</script>

    <script>
      // Pure React and JavaScript code
    </script>
  </body>
</html>
```

Oto minimalny szablon wymagany do pracy z Reactem w przeglądarce. Możemy umieścić swój JavaScript w osobnym pliku, ale musi on zostać załadowany gdzieś na stronie po załadowaniu Reacta. Będziemy używać wersji rozwojowej Reacta, aby wyświetlać wszystkie komunikaty o błędach i ostrzeżenia w konsoli przeglądarki. Możesz użyć zminimalizowanej wersji produkcyjnej, używając `aware.production.min.js` i `espons-dom.production.min.js`, co spowoduje brak komunikatów ostrzeżeń w konsoli.

HTML to po prostu zestaw instrukcji, których przeglądarka postępuje podczas konstruowania DOM. Elementy składające się na dokument HTML stają się elementami DOM, gdy przeglądarka ładuje HTML i renderuje interfejs użytkownika. Powiedzmy, że dla przepisu musimy stworzyć hierarchię HTML:

```
<section id="baked-salmon">
  <h1>Baked Salmon</h1>
  <ul class="ingredients">
    <li>2 lb salmon</li>
    <li>5 sprigs fresh rosemary</li>
    <li>2 tablespoons olive oil</li>
    <li>2 small lemons</li>
    <li>1 teaspoon kosher salt</li>
    <li>4 cloves of chopped garlic</li>
  </ul>
  <section class="instructions">
    <h2>Cooking Instructions</h2>
    <p>Preheat the oven to 375 degrees.</p>
    <p>Lightly coat aluminum foil with oil.</p>
    <p>Place salmon on foil</p>
    <p>Cover with rosemary, sliced lemons, chopped garlic.</p>
    <p>Bake for 15-20 minutes until cooked through.</p>
    <p>Remove from oven.</p>
  </section>
</section>
```

W HTML elementy odnoszą się do siebie w hierarchii, która przypomina drzewo genealogiczne. Można powiedzieć, że element główny (w tym przypadku sekcja) ma troje dzieci: nagłówek, nieuporządkowaną listę składników i sekcję z instrukcjami. W przeszłości strony internetowe składały się z niezależnych stron HTML. Gdy użytkownik nawigował po tych stronach, przeglądarka żądała i łądowała różne dokumenty HTML. Wynalazek AJAX (asynchroniczny JavaScript i XML) przyniósł nam jednostronicową aplikację, czyli SPA. Ponieważ przeglądarki mogły żądać i łądować małe fragmenty danych za pomocą AJAX, całe aplikacje internetowe mogą teraz łądować się

na jednej stronie i polegać na JavaScript w celu zaktualizowania interfejsu użytkownika. W SPA przeglądarka początkowo ładuje jeden dokument HTML. Gdy użytkownicy poruszają się po witrynie, w rzeczywistości pozostają na tej samej stronie. JavaScript niszczy i tworzy nowy interfejs użytkownika, gdy użytkownik współdziała z aplikacją. Może się wydawać, że przeskakujemy w tym wypadku ze strony na stronę, ale w rzeczywistości nadal znajdujemy się na tej samej stronie HTML.

`DOM API` to zbiór obiektów, których JavaScript może używać do interakcji z przeglądarką w celu modyfikowania `DOM`. Do metod modyfikowania DOM należą między innymi `document.createElement` lub `document.appendChild`.

`React` to biblioteka zaprojektowana do aktualizowania modelu `DOM` przeglądarki za nas. Nie musimy już martwić się o zawłościzwiązane z budowaniem wysokowydajnych SPA, ponieważ `React` może to zrobić za nas. Dzięki `Reactowi` nie współpracujemy bezpośrednio z `DOM API`. Zamiast tego podajemy instrukcje dotyczące tego, co ma zbudować `React`, a `React` zajmie się renderowaniem i uzgadnianiem elementów, które mu zleciliśmy. W przeglądarce, strona składa się z elementów DOM. Podobnie, `React DOM` składa się z elementów `React`. Elementy DOM i elementy `React` mogą wyglądać tak samo, ale w rzeczywistości są zupełnie inne. Element `React` to opis tego, jak powinien wyglądać rzeczywisty element DOM. Innymi słowy, elementy `React` to instrukcje dotyczące tworzenia DOM przeglądarki.

Możemy stworzyć element `React` reprezentujący `h1` używając `React.createElement`:

```
React.createElement("h1", { id: "recipe-0" }, "Baked Salmon");
```

Pierwszy argument definiuje typ elementu, który chcemy stworzyć. W tym przypadku chcemy stworzyć element `h1`. Drugi argument reprezentuje właściwości elementu. W naszym przypadku `h1` ma mieć identyfikator `recipe-0`. Trzeci argument reprezentuje elementy potomne elementu: wszelkie węzły wstawione między znacznikiem otwierającym a zamykającym (w tym przypadku tylko tekst). Podczas renderowania `React` przekonwertuje ten element na rzeczywisty element DOM:

```
<h1 id="recipe-0">Baked Salmon</h1>
```

Właściwości są podobnie stosowane do nowego elementu DOM: właściwości są dodawane do tagu jako atrybuty, a tekst podrzędny jest dodawany jako tekst w elemencie. Element `React` to po prostu literał JavaScript, który mówi `Reactowi`, jak zbudować element DOM. Gdybyśmy chcieli podejrzeć ten element, wyglądałoby to tak:

```
{
  $$typeof: Symbol(React.element),
  "type": "h1",
  "key": null,
  "ref": null,
  "props": {id: "recipe-0", children: "Baked Salmon"},
  "_owner": null,
  "_store": {}
}
```

To jest struktura elementu `React`. Istnieją pola używane przez `Reacta`: `_owner`, `_store` i `$$typeof`. Właściwość `type` elementu `React` mówi `Reactowi`, jaki typ elementu HTML lub SVG ma utworzyć. Właściwość `props` reprezentuje dane i elementy potomne wymagane do skonstruowania elementu DOM. Właściwość `children` służy do wyświetlania innych zagnieżdżonych elementów jako tekstu.

Gdy utworzymy element `React`, będziemy chcieli go zobaczyć w przeglądarce. `ReactDOM` zawiera narzędzia niezbędne do renderowania elementów `Reacta` w przeglądarce. W `ReactDOM` znajdziemy metodę renderowania, w tym jego dzieci, do DOM za pomocą `ReactDOM.render`. Element, który chcemy renderować, jest przekazywany jako pierwszy argument, a drugi argument to węzeł docelowy, w którym powinniśmy renderować element:

```
const dish = React.createElement("h1", null, "Baked Salmon");

ReactDOM.render(dish, document.getElementById("root"));
```

Renderowanie elementu do DOM spowodowałoby dodanie elementu `h1` do `div` z identyfikatorem `root`, który już zdefiniowalibyśmy w naszym HTML (wszystko co było w `div` zostanie nadpisane przez dodany element `h1`).

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>React Samples</title>
</head>
<body>
<!-- Target container -->
<div id="root">
  <section id="baked-salmon">
    <h1>Baked Salmon</h1>
    <ul class="ingredients">
      <li>2 lb salmon</li>
      <li>5 sprigs fresh rosemary</li>
      <li>2 tablespoons olive oil</li>
      <li>2 small lemons</li>
      <li>1 teaspoon kosher salt</li>
      <li>4 cloves of chopped garlic</li>
    </ul>
    <section class="instructions">
      <h2>Cooking Instructions</h2>
      <p>Preheat the oven to 375 degrees.</p>
      <p>Lightly coat aluminum foil with oil.</p>
      <p>Place salmon on foil</p>
      <p>Cover with rosemary, sliced lemons, chopped garlic.</p>
      <p>Bake for 15-20 minutes until cooked through.</p>
      <p>Remove from oven.</p>
    </section>
  </section>
</div>

<!-- React library & ReactDOM (Development Version)-->
<script
  src="https://unpkg.com/react@16/umd/react.development.js">
</script>
<script
  src="https://unpkg.com/react-dom@16/umd/react-dom.development.js">
</script>

<script>
  const dish = React.createElement("h1", null, "Baked Salmon");
  ReactDOM.render(dish, document.getElementById("root"));
</script>
</body>
</html>
```

Wszystko, co jest związane z renderowaniem elementów do DOM, znajduje się w pakiecie `ReactDOM`. W wersjach `Reacta` wcześniejszych niż `React 16` można było renderować tylko jeden element w modelu DOM. Obecnie możliwe jest również renderowanie tablic, co pokazane jest poniżej.

```
const dish = React.createElement("h1", null, "Baked Salmon");
const dessert = React.createElement("h2", null, "Coconut Cream Pie");

ReactDOM.render([dish, dessert], document.getElementById("root"));
```

Step 6

React renderuje elementy potomne za pomocą `props.children`. W poprzedniej sekcji wyrenderowaliśmy element tekstowy jako element potomny elementu `h1`, a zatem `props.children` został ustawiony na `Baked Salmon`. Moglibyśmy również renderować inne elementy Reacta jako dzieci, tworząc drzewo elementów. Dlatego używamy terminu drzewo elementów: drzewo ma jeden element główny, z którego wyrasta wiele gałęzi. Rozważmy nieuporządkowaną listę zawierającą składniki:

```
<ul>
  <li>2 lb salmon</li>
  <li>5 sprigs fresh rosemary</li>
  <li>2 tablespoons olive oil</li>
  <li>2 small lemons</li>
  <li>1 teaspoon kosher salt</li>
  <li>4 cloves of chopped garlic</li>
</ul>
```

W tym przykładzie nieuporządkowana lista jest elementem głównym i ma sześcioro dzieci. Możemy reprezentować ten `ul` i jego dzieci za pomocą `React.createElement`:

```
const list = React.createElement(
  "ul",
  null,
  React.createElement("li", null, "2 lb salmon"),
  React.createElement("li", null, "5 sprigs fresh rosemary"),
  React.createElement("li", null, "2 tablespoons olive oil"),
  React.createElement("li", null, "2 small lemons"),
  React.createElement("li", null, "1 teaspoon kosher salt"),
  React.createElement("li", null, "4 cloves of chopped garlic")
);

ReactDOM.render(list, document.getElementById("root"));
```

Każdy dodatkowy argument wysłany do funkcji `createElement` jest kolejnym elementem potomnym. `React` tworzy tablicę tych elementów potomnych i ustawia wartość właściwości `props.children` na tę tablicę. Gdybyśmy mieli sprawdzić wynikowy element `React`, zobaczylibyśmy każdy element listy reprezentowany przez element `React` i dodany do tablicy o nazwie `props.children`. Możemy podejrzeć zawartość tej tablicy dzięki konsoli:

```
const list = React.createElement(
  "ul",
  null,
  React.createElement("li", null, "2 lb salmon"),
  React.createElement("li", null, "5 sprigs fresh rosemary"),
  React.createElement("li", null, "2 tablespoons olive oil"),
  React.createElement("li", null, "2 small lemons"),
  React.createElement("li", null, "1 teaspoon kosher salt"),
  React.createElement("li", null, "4 cloves of chopped garlic")
);
ReactDOM.render(list, document.getElementById("root"));
console.log(list);
```

W wyniku czego otrzymamy:

```
{
  "type": "ul",
  "props": {
    "children": [
      { "type": "li", "props": { "children": "2 lb salmon" } ... },
      { "type": "li", "props": { "children": "5 sprigs fresh rosemary" } ... },
      { "type": "li", "props": { "children": "2 tablespoons olive oil" } ... },
      { "type": "li", "props": { "children": "2 small lemons" } ... },
      { "type": "li", "props": { "children": "1 teaspoon kosher salt" } ... },
      { "type": "li", "props": { "children": "4 cloves of chopped garlic" } ... }
    ]
  }
}
```


Widzimy teraz, że każdy element powyższej listy jest dzieckiem. Teraz utworzymy wcześniej zdefiniowaną przez nas sekcję w HTML za pomocą wywołań React:

```
const sec = React.createElement(
  "section", {
    id: "baked-salmon"
  },
  React.createElement("h1", null, "Baked Salmon"),
  React.createElement(
    "ul", {
      className: "ingredients"
    },
    React.createElement("li", null, "2 lb salmon"),
    React.createElement("li", null, "5 sprigs fresh rosemary"),
    React.createElement("li", null, "2 tablespoons olive oil"),
    React.createElement("li", null, "2 small lemons"),
    React.createElement("li", null, "1 teaspoon kosher salt"),
    React.createElement("li", null, "4 cloves of chopped garlic")
  ),
  React.createElement(
    "section", {
      className: "instructions"
    },
    React.createElement("h2", null, "Cooking Instructions"),
    React.createElement("p", null, "Preheat the oven to 375 degrees."),
    React.createElement("p", null, "Lightly coat aluminum foil with oil."),
    React.createElement("p", null, "Place salmon on foil."),
    React.createElement(
      "p",
      null,
      "Cover with rosemary, sliced lemons, chopped garlic."
    ),
    React.createElement(
      "p",
      null,
      "Bake for 15-20 minutes until cooked through."
    ),
    React.createElement("p", null, "Remove from oven.")
  )
);
ReactDOM.render(sec, document.getElementById("root"));
```

Każdy element, który ma atrybut klasy HTML, używa `className` dla tej właściwości zamiast `class`. Ponieważ `class` jest słowem zastrzeżonym w JavaScript, musimy użyć `className` do zdefiniowania atrybutu class elementu HTML. Ta próbka pokazuje, jak wygląda czysty React, który jest ostatecznie tym, co działa w przeglądarce. W aplikacji tworzonej przez React mamy drzewo elementów, które pochodzą z jednego elementu głównego. Natomiast utworzone elementy przez React to instrukcje, których React użyje do zbudowania interfejsu użytkownika w przeglądarce.

Step 7

Główną zaletą korzystania z `React` jest możliwość oddzielenia danych od elementów interfejsu użytkownika. Ponieważ `React` to tylko `JavaScript`, możemy dodać logikę `JavaScript`, która pomoże nam zbudować drzewo komponentów `React`. Na przykład składniki mogą być przechowywane w tablicy i możemy odwzorować tę tablicę na elementy `React`. Wróćmy i pomyślmy przez chwilę o nieuporządkowanej wcześniej przez nas liście:

```
React.createElement(
  "ul",
  null,
  React.createElement("li", null, "2 lb salmon"),
  React.createElement("li", null, "5 sprigs fresh rosemary"),
  React.createElement("li", null, "2 tablespoons olive oil"),
  React.createElement("li", null, "2 small lemons"),
  React.createElement("li", null, "1 teaspoon kosher salt"),
  React.createElement("li", null, "4 cloves of chopped garlic")
);
```

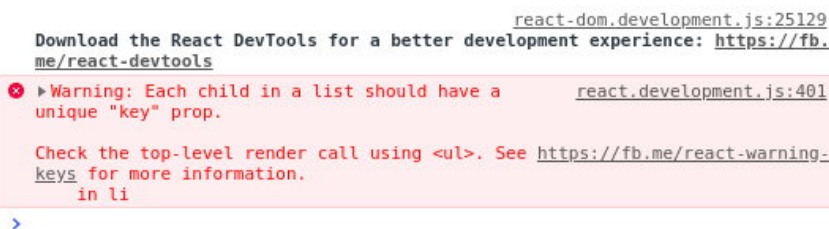
Dane użyte na tej liście składników można łatwo przedstawić za pomocą tablicy JavaScript:

```
const items = [
  "2 lb salmon",
  "5 sprigs fresh rosemary",
  "2 tablespoons olive oil",
  "2 small lemons",
  "1 teaspoon kosher salt",
  "4 cloves of chopped garlic"
];
```

Chcemy wykorzystać te dane do wygenerowania prawidłowej liczby elementów listy bez konieczności kodowania każdego z nich na stałe. Możemy skorzystać z funkcji `map` na tablicy i utworzyć pozycje listy dla tylu składników, ile jest ich w tablicy:

```
const items = [
  "2 lb salmon",
  "5 sprigs fresh rosemary",
  "2 tablespoons olive oil",
  "2 small lemons",
  "1 teaspoon kosher salt",
  "4 cloves of chopped garlic"
];
const list = React.createElement(
  "ul", {
    className: "ingredients"
  },
  items.map(ingredient => React.createElement("li", null, ingredient))
);
ReactDOM.render(list, document.getElementById("root"));
```

Ta składnia tworzy element `React` dla każdego składnika w tablicy. Każdy ciąg jest wyświetlany w elementach potomnych elementu listy jako tekst. Wartość każdego składnika jest wyświetlana jako pozycja na liście. Ale w konsoli dla powyżej odpalonego kodu dostaniemy następujące ostrzeżenie:



```
react-dom.development.js:25129
Download the React DevTools for a better development experience: https://fb.
me/react-devtools
Warning: Each child in a list should have a unique "key" prop.
react.development.js:401
Check the top-level render call using <ul>. See https://fb.me/react-warning-
keys for more information.
in li
```

Kiedy tworzymy listę elementów potomnych poprzez iterację po tablicy, `React` lubi, aby każdy z tych elementów miał właściwość klucza (`key`). Właściwość `key` jest używana przez `Reacta`, aby pomóc mu wydajnie aktualizować DOM. Możemy usunąć to ostrzeżenie, dodając unikalną właściwość klucza do każdego elementu pozycji listy. Możemy użyć indeksu tablicy dla każdego składnika jako tej unikalnej wartości:

```
const items = [
  "2 lb salmon",
  "5 sprigs fresh rosemary",
  "2 tablespoons olive oil",
  "2 small lemons",
  "1 teaspoon kosher salt",
  "4 cloves of chopped garlic"
];
const list = React.createElement(
  "ul", {
    className: "ingredients"
  },
  items.map((ingredient, i) => React.createElement("li", {
    key: i
  }, ingredient))
);
ReactDOM.render(list, document.getElementById("root"));
```

Step 8

Bez względu na jego rozmiar, zawartość lub technologie użyte do jego stworzenia, interfejs użytkownika składa się z różnych części między innymi z przycisków, list czy nagłówków. Wszystkie te części razem tworzą interfejs użytkownika. Napišemy teraz aplikację składającą się z kilku przepisów. Każdy z nich będzie oddzielony od drugiego w osobnej sekcji. A więc każdy przepis będzie w osobnym pudełku, co będziemy nazywać **komponentem**. Komponenty pozwalają nam ponownie używać tej samej struktury, a następnie możemy zapełniać te struktury różnymi zestawami danych.

Rozważając interfejs użytkownika naszej aplikacji, którą chcemy zbudować w React, warto poszukać możliwości rozbicia elementów na części. Na przykład przepisy mają tytuł, listę składników i instrukcje. Wszystkie są częścią większej receptury lub składnika aplikacji. Stworzymy komponent dla każdej wyróżnionej części: składniki, instrukcje i tak dalej. Utworzymy komponent, korzystając z funkcji. Ta funkcja zwróci część interfejsu użytkownika, której można ponownie użyć. Utwórzmy funkcję, która zwraca nieuporządkowaną listę składników.

```
function IngredientsList() {
  return React.createElement(
    "ul", {
      className: "ingredients"
    },
    React.createElement("li", null, "1 cup unsalted butter"),
    React.createElement("li", null, "1 cup crunchy peanut butter"),
    React.createElement("li", null, "1 cup brown sugar"),
    React.createElement("li", null, "1 cup white sugar"),
    React.createElement("li", null, "2 eggs"),
    React.createElement("li", null, "2.5 cups all purpose flour"),
    React.createElement("li", null, "1 teaspoon baking powder"),
    React.createElement("li", null, "0.5 teaspoon salt")
  );
}

ReactDOM.render(
  React.createElement(IngredientsList, null, null),
  document.getElementById("root")
);
```

Nazwa komponentu to `IngredientsList`, a funkcja generuje elementy wyglądające następująco:

```
<IngredientsList>
  <ul className="ingredients">
    <li>1 cup unsalted butter</li>
    <li>1 cup crunchy peanut butter</li>
    <li>1 cup brown sugar</li>
    <li>1 cup white sugar</li>
    <li>2 eggs</li>
    <li>2.5 cups all purpose flour</li>
    <li>1 teaspoon baking powder</li>
    <li>0.5 teaspoon salt</li>
  </ul>
</IngredientsList>
```

Możemy usprawnić nasz komponent dodając argument, albo za pomocą zmiennej lub używając destrukuryzacji obiektu.

```
function IngredientsList(props) {
  return React.createElement(
    "ul", {
      className: "ingredients"
    },
    props.items.map((ingredient, i) =>
      React.createElement("li", {
        key: i
      }, ingredient)
    )
  );
}

const secretIngredients = [
  "1 cup unsalted butter",
  "1 cup crunchy peanut butter",
  "1 cup brown sugar",
  "1 cup white sugar",
  "2 eggs",
  "2.5 cups all purpose flour",
  "1 teaspoon baking powder",
  "0.5 teaspoon salt"
];

ReactDOM.render(
  React.createElement(IngredientsList, {
    items: secretIngredients
  }, null),
  document.getElementById("root")
);
```

```
function IngredientsList({
  items
}) {
  return React.createElement(
    "ul", {
      className: "ingredients"
    },
    items.map((ingredient, i) =>
      React.createElement("li", {
        key: i
      }, ingredient)
    )
  );
}

const secretIngredients = [
  "1 cup unsalted butter",
  "1 cup crunchy peanut butter",
  "1 cup brown sugar",
  "1 cup white sugar",
  "2 eggs",
  "2.5 cups all purpose flour",
  "1 teaspoon baking powder",
  "0.5 teaspoon salt"
];

ReactDOM.render(
  React.createElement(IngredientsList, {
    items: secretIngredients
  }, null),
  document.getElementById("root")
);
```

Zanim pojawiły się komponenty tworzone za pomocą funkcji, istniały inne sposoby tworzenia komponentów. Choć nie spędzimy dużo czasu na tych podejściach, ważne jest, aby zrozumieć historię komponentów `Reacta`, szczególnie gdy mamy do czynienia z tymi interfejsami API w starszych bazach kodu. Kiedy `React` po raz pierwszy stał się open source w 2013, istniał jeden sposób na stworzenie komponentu: `createClass`. Użycie `React.createClass` do stworzenia komponentu wygląda następująco:

```
//Wymaga przestarzałego pakietu w React !!!
const IngredientsList = React.createClass({
  displayName: "IngredientsList",
  render() {
    return React.createElement(
      "ul",
      { className: "ingredients" },
      this.props.items.map((ingredient, i) =>
        React.createElement("li", { key: i }, ingredient)
      )
    );
  }
});
```

Komponenty, które używają `createClass`, mają metodę `render()` opisującą elementy `React`, które powinny być zwracane i renderowane. W `React` wersja 15.5 (kwiecień 2017), `React` zaczął generować ostrzeżenia, jeśli użyto `createClass`. W `React` 16 (wrzesień 2017), `React.createClass` został oficjalnie przestarzały i został przeniesiony do własnego pakietu, `create-aware-class`. Kiedy składnia klas została dodana do JavaScript w ES 2015, `React` wprowadził nową metodę tworzenia komponentów. Interfejs API `React.Component` umożliwił użycie składni klas do utworzenia nowej instancji komponentu:

```
class IngredientsList extends React.Component {
  render() {
    return React.createElement("ul", {className: "ingredients"},
      this.props.items.map((ingredient, i) =>
        React.createElement("li", { key: i }, ingredient)
      )
    )
  }
}

const items = [
  "1 lb Salmon",
  "1 cup Pine Nuts",
  "2 cups Butter Lettuce",
  "1 Yellow Squash",
  "1/2 cup Olive Oil",
  "3 cloves of Garlic"
]

ReactDOM.render(
  React.createElement(IngredientsList, {items}, null),
  document.getElementById('root')
)
```

Nadal jest możliwe utworzenie komponentu `React` przy użyciu składni klas, ale `React.Component` również jest na drodze do wycofania. Chociaż nadal jest obsługiwany, pójdzie drogą `React.createClass`, innego starego przyjaciela. Od teraz będziemy używać funkcji do tworzenia komponentów.