# Wprowadzenie

W dobie wszechobecnego dostępu do internetu coraz większą popularność zyskują strony WWW oraz aplikację internetowe. Szybki rozwój internetu oraz powszechny dostęp do mediów wpływa na to, że użytkownicy coraz częściej wybierają łatwo dostępne aplikacje webowe. Coraz więcej ludzi decyduje się, porzucić instalację aplikacji pulpitowej(dektopowej) na rzecz rejestracji w serwisie internetowym, który pozwala wykonać to co 10 lat temu pozwalały nam aplikacje instalowane na komputerze. Aby skorzystać z tych dobrodziejstw użytkownik potrzebuje jedynie 2 rzeczy: komputera z przeglądarką internetową oraz dostępu do internetu.

Internet stał się miejscem pracy, spotkań oraz rozrywki. Aktualnie wiele ludzi pracuje zdalnie z własnego mieszkania. Wiele osób, szczególnie z branży IT świadczy usługi nawet dla klientów, którzy żyją w miejscach znacznie oddalonych. Dzięki takim aplikacją jak Skype czy Google Hangouts nie ma problemów z komunikacją, kod umieszczony jest na ogólnodostępnym repozytorium a o funkcjach języka programowania możemy się dowiedzieć z dokumentacji online. Platformy takie jak YouTube to nie tylko filmy, które oglądamy w wolnym czasie, ale także materiały do nauki i rozwoju. Materiały te są przygotowywane przez hobbystów, ale także przez ludzi, dla których platforma ta jest głównym źródłem zarobku. Swoją pracę mogą monetyzować w postaci reklam, które wyświetlają się podczas oglądania.

Świat, w którym obecnie żyjemy w znacznym stopniu różni się od świata sprzed 20 lat. Monitory CRT zamieniły się w panoramiczne wyświetlacze LED, przewodowe myszki kulkowe zostały zastąpione bezprzewodowymi laserowymi, a powolny internet sieci telefonicznej został wyparty przez bardzo szybką sieć WiFi.

Jednak nie tylko sprzęt komputerowy się zmienił. Wraz z rozwojem musiały również ewoluować programy komputerowe. Zaczynając od aplikacji konsolowych, poprzez programy z graficznym interfejsem użytkownika, aż do wysoce nowoczesnych aplikacji internetowych. Aby taki rozwój był możliwy, języki programowania również zostały udoskonalane. Kolejne wersje języków takich jak Java, Python czy C++, nie tylko podnoszą wydajność, ale również dostarczają programistom coraz to więcej narzędzi i udogodnień aby napisany przez nich kod był łatwiejszy do utrzymania, a sama aplikacja powstawała w szybszym tempie.

# 1.1 Opis problemu

Aplikacja internetowa składa się głównie z trzech podmitów. Języka HTML(Hyper text markoup language), CSS(Cascade stylesheet) i JavaScript. Pierwszy z nich jest odpowiedzialny za strukturę i umiejscowienie elementów na stronie. Drugi z nich odpowiada za wygląd strony, animację i efekty wizualne. Trzeci zaś służy do obsłużenia logiki strony, przetwarzania danych, operacji asynchronicznych i HTTP. Utworzymy w nim funkcje, która wykona się jak użytkownik naciśnie na przycisk. Przeprowadzi on również walidacje pól z formularza, tak aby nie dopuścić do błędnych informacji, oraz wykona zapytanie do serwera aby pobrać aktualne dane.

Podczas gdy języki HTML oraz CSS przez czas swojego istnienia, nie wprowadziły zmian, przez które z wersji na wersje używanie tych języków znacznie by się zmieniło, to JavaScript wypuszczając nową wersję w roku 2015 znacznie odmieniła podejście programistów do programowania w tym języku. ECMAScript 2015 znany również pod nazwą ECMAScript 6 (skrót ES6) zrobił to dla JavaScriptu czego dokonała Java 8 dla programistów języka, którym rozwojem zajmuje się Oracle. Wprowadził on masę użytecznych narzędzi takich jak, stałe, funkcje strzałkowe, moduły, klasy czy też inne elementy zapewniające szybsze i prostsze powstawanie aplikacji.

Dzięki takim programom jak NPM czy Yarn, programiści mogą pobrać i podłączyć do projektu nową bibliotekę w kilka sekund. Działa to również w drugą stronę, udostępnienie swoich funkcjonalności JavaScriptowych to kwestia 2 komend. Dzięki tak prostemu dostępowi, ogromie bibliotek i funkcji oraz społeczności, która chętnie udziela się w ramach projektów OpenSource, język ten zyskał bardzo dużą popularność.

Wraz z dużą popularnością języka powstało wiele bibliotek i frameworków. Największe firmy z branży IT, również postanowiły wytworzyć swoje rozwiązania: Facebook wydał bibliotekę React.JS a Google framework Angular. Według danych są to 2 aktualnie najpopularniejsze języki do wytwarzania aplikacji internetowych opartych na komponentach. Tuż za nimi plasuje się Vue.JS opracowany przez Evan You. TU WKLEJ JAKIEŚ DANE

Dostępność tak wielu narzędzi i języków to ogromne dobrodziejstwo, ale również pułapka. Architekt oprogramowania przystępując do wybory odpowiedniego frameworka staje przed nie łatwym zadaniem. Musi zebrać wymagania co do aplikacji oraz posiadać dużą wiedzę na temat działania i sposoby użycia danych bibliotek, aby w jak najefektywniejszy sposób przeprowadzić proces inżynierii oprogramowania.

# 1.2 Uzasadnienie aktualności tematu

Wybór odpowiedniej biblioteki czy frameworka to bardzo ważna decyzja, którą należy podjąć już na samym starcie projektu. Wybór ten ciągnie za sobą wiele konsekwencji, a najważniejszą jest to, że próba zmiana biblioteki może być bardzo kosztowne a czasami wręcz niemożliwe.

Architekt oprogramowania musi posiadać bardzo dużą wiedzę, oraz podążać za aktualnymi trendami, ponieważ te zmieniają się dynamicznie. Musi on wiedzieć co jest dla jego aplikacji ważne, a co można przesunąć na drugi plan. Aby to osiągnąć musi on albo posiadać doświadczenie w tych technologiach lub zapoznać się z często rozległymi dokumentacjami. Zadanie to nie jest proste i wymaga czasu, którego w dzisiejszym świecie IT nie ma wiele. Osoba decydująca musi przeanalizować dzięki, któremu frameworkowi osiągnie największą wydajność procesu wytwórczego.

Uważam, że temat jest jak najbardziej aktualny. Każdego dnia powstają nowe biblioteki i frameworki, a ich dobór w dużej mierze zależy od powodzenia ukończenia projektu. W pracy uwzględniłem 3 najpopularniejsze biblioteki, które są najczęściej wybierane przez programistów. Uważam, że warto przyjrzeć się temu jak one działają oraz jak wybór biblioteki może wpłynąć na inżynierię oprogramowania.

# Cel i zakres pracy

Celem pracy jest przeprowadzenie analizy porównawczej w badaniu wpływu zastosowania frameworków Angular i Vue.js oraz biblioteki ReactJS do tworzenia komponentów aplikacji internetowych na efektywność procesu wytwórczego z uwzględnieniem wymagań funkcjonalnych i wydajnościowych oprogramowania na przykładzie realizacji wybranego projektu. Aby analiza była możliwa, zdefiniowane zostaną kryteria porównania w badaniach efektywności procesu wytwarzania aplikacji internetowych. Dla każdego z ww. frameworka i biblioteki powstanie aplikacja, na której będą przeprowadzone badania. Aby mogła ona powstać zostanie zdefiniowany model wymagań oraz model projektowy. Na powstałych aplikacjach zostaną również przeprowadzone badania wydajnościowe.

W zakres pracy wchodzi również przedstawienie tematu pracy oraz jego aktualność. Zostaną również opisane użyte frameworki i biblioteka oraz pokrewne technologie, które również są istotne z punktu analizy efektywności procesu wytwórczego. Zostaną przeprowadzone badania, po których dostarczą odpowiednich metryk. Na koniec zostanie przeprowadzona analiza porównawcza a zebrane wyniki zostaną opracowane w podsumowaniu.

# 1.4 Struktura pracy

Praca została podzielona na 2 części. Pierwsze trzy rozdziały to część teoretyczna dotycząca pracy. Od rozdziału 4 opisywane są badania i część praktyczna, które zostały wykonane w ramach pracy.

Pierwszy rozdział to przedstawienie problemu – dlaczego jest on istotny i dlaczego ta praca powstała. Została również uzasadniona aktualność tematu. Następnie zdefiniowany został cel oraz jak wygląda struktura pracy.W drugim rozdziale opisane zostały wykorzystywane w pracy technologie, narzędzia oraz metodologie. W rozdziale trzecim opisane zostały planowane badania oraz model aplikacji.

Część praktyczną rozpoczyna opis aplikacji oraz implementacje rozwiązań w każdym z trzech języków. Następnie opisywane jest proces testowania oraz zbierania metryk. Posiadając te dane, następnym krokiem jest przeprowadzenie analizy porównawczej na podstawie otrzymanych wyników. Pracę kończy rozdział podsumowujący całą pracę i badania.

# Przegląd wykorzystywanych technologii i metodologii

W poniższym rozdziale zostały opisane technologie, narzędzia i metodologie, które zostały użyte podczas wytwarzania aplikacji oraz badań.

# 2.1 Biblioteka a framework

…Cośtam cośtam. Pomimo tego, że istnieją różnice pomiędzy tymi definicjami, w pracy będę używał ich zamiennie.

# 2.1 JavaScript

JavaScript jest językiem, który jest nieodłącznym elementem programowania aplikacji webowych. Od momentu jego powstania był on przeznaczony do wykonywanie kodu w przeglądarce użytkownika. Jednak coraz częściej używa się go też w technologiach serwerowych oraz w zastosowaniach w chmurze(serverless).

JavaScript został zaprezentowany w 1995 roku jako sposób na to aby dodać do stron internetowych aplikacje otwieranych w przeglądarce Netscape Navigator. Język został również zaadaptowany przez wszystkie inne duże przeglądarki. Język ten pozwolił na tworzenia bogatych witryn internetowych, z którymi użytkownik może prowadzić interakcje bez zbędnego odświeżania strony. JavaScript jest również używany do bardziej tradycyjnych stron internetowych, np. do obsługi formularzy i walidacji.[1]

Należy podkreślić to, że JavaScript nie jest w żaden sposób związany z językiem Java. Podobna nazwa wzięła się ze względów marketingowych, gdyż w tamtym czasie Java stawała się bardzo popularna. Twórcy pomyśleli, że to dobry pomysł aby wybić na fali popularności. 25 lat później JavaScript jest jednym z najbardziej popularnych języków na świecie, a nazwa dalej pozostała[1].

Po tym jak język został zaadaptowany na zewnątrz Netscape został napisany dokument, który opisuje sposób w jaki JavaScript powinien działać, po to aby wiele różnych przeglądarek potrafiły w taki sam sposób interpretować ten sam kod. Dokument ten nazywany jest standardem ECMAScript, po tym jak organizacja Ecma International dokonała standaryzacji. W praktyce ECMAScript oraz JavaScript, mogą być używane zamiennie.[1]

Fakt, że język ewoluuje oznacza, że przeglądarki nieustannie muszą podążać za tymi zmianami. Jeżeli użytkownik ma starszą wersję przeglądarki, to ta może nie wspierać wszystkich funkcjonalności języka. Projektanci języka są ostrożni aby nie zrobić zmiany, która spowodowałaby zepsucie aktualnie działających programów.[1]

Przeglądarki internetowe to nie jedyne miejsce gdzie JavaScript znalazł zastosowanie. Niektóre bazy danych takie jak MongoDB czy CouchDB, używają języka do kwerend. Kilka platform dla programowania serwerowego, a najbardziej popularny projekt Node.js, dostarcza środowisko, które pozwala na używanie JavaScript spoza przeglądarki.[1]

# 2.2 ECMAScript

TC39 (Ecma Technical Committee 39) to komisja która rozwija język JavaScript, a jej członkami są firmy(między innymi główni dostawcy przeglądarek). TC39 spotyka się regularnie, aby planować rozwój języka – minutki z tych spotkań są dostępne online.[2]

JavaScript to nazwa, którą używa każdy, jednak jest ona znakiem towarowym firmy Oracle(firma rozwijająca m.in. język Java). Z tego też powodu oficjalną nazwą języka JavaScript jest ECMAScript. Nazwa ta pochodzi od organizacji Ecma, która obsługuje standardy tego języka.[2]

Wersje JavaScript są zdefiniowane w specyfikacjach, które noszą oficjalną nazwę języka. Dlatego pierwszą standaryzowaną wersją JavaScript jest ECMAScript 1. ECMAScript x jest często skracana do ESx.[2]

# 2.3 TypeScript

TypeScript jest to typowany nadzbiór języka JavaScript, który kompiluje się prosto do kodu JavaScript. Pliki TypeScript często noszą rozszerzenie .ts. Wiele środowisk programistycznych wspiera TypeScript bez dodatkowej konfiguracji. TypeScript, może być również kompilowany za pomocą linii komend.[3]

Każdy kod języka JavaScript jest poprawnym kodem TypeScript. TypeScript natomiast dodaje dużo dodatkowej funkcjonalności. Powoduje on, że JavaScript staje się silnie typowanym, zorientowanym obiektowo językiem takim jak C# czy Java. Oznacza to, że kod TypeScript ma tendencje do bycia łatwiejszym do używania w dużych projektach oraz łatwiejszym do zrozumienia, czytania i utrzymania. Silne typowanie oznacza również, że język jest prekompilowany a do zmiennych nie może być przypisywana wartość, która wykracza poza zakres deklarowanego typu. Dla przykładu, jeżeli zmienna jest zadeklarowana jako numer, to nie można do niej przypisać wartości tekstowej.[3]

Silne typowanie i zorientowanie obiektowe, powoduje że TypeScript jest łatwiejsze do znajdywania błędów i utrzymania. Te dwie rzeczy były najsłabszymi punktami standardowego JavaScriptu.[3]

Typy mają udowodnioną zdolność do poprawiania jakości i zrozumiałości kodu. Wielke zespoły takie jak Google, Microsoft, Facebook ciągle dochodzą do tego wniosku. Typy zwiększają elastyczność i pewność podczas refaktoryzacji kodu. Lepszym rozwiązaniem jest gdy kompilator złapie błąd, niż użytkownik końcowy podczas przeglądania witryny. Typy są najlepszym typem dokumentacji jaki może być. Wskazują na to co jakie dane funkcja dostaje na wejście oraz co jest wynikiem jej działania. Jednak czasami typy są tylko niepotrzebnymi ozdobami, na szczęście TypeScript potrafi utrzymywać tą barierę tak nisko jak możliwe. [4]

# Angular

Angular jest frameworkiem języka JavaScript, który dostarcza narzędzi do budowania bogatych aplikacji, w sposób szybszy i łatwiejszy. Aplikacje Angular są zbudowane w oparciu o wzorzec Model-Widok-Kontroler(Model-View-Controller, MVC), który kładzie nacisk na tworzenie aplikacji, które są rozszerzalne, utrzymywalne, testowalne oraz ustandaryzowane.[5]

Angular jest nowoczesną platformą internetową, która obiecuje dostarczyć programiście kompleksowy zestaw narzędzi i możliwości do tworzenia dużych i niezawodnych aplikacji. Podstawową zaletą Angulara jest umożliwienie tworzenie aplikacji, które działają na prawie każdej platformie – mobilnej, internetowej czy pulpitowej. Zespół Angulara skoncentrował się na budowaniu czegoś więcej niż solidnej struktury aplikacji – zbudowali również cały ekosystem.[7]

Kod źródłowy Angulara jest otwarcie dostępny, a framework jest utrzymywany i sponsorowany przez Google. Framework ten był użyty w największych i najbardziej skomplikowanych aplikacjach internetowych.[5]

Angular jest następcą frameworka AngularJS. Nowa wersja używa TypeScript zamiast JavaScript, zatem zdecydowano się na usunięcie końcówki JS z nazwy. Przez długi okres na ten framework, mówiło się potocznie nowy Angular albo Angular 2. W roku 2017 Google zmieniło wersjonowanie i co pół roku nową wersję, która nazywa się po prostu Angular. Zrezygnowano zatem z nazywania wersji Angular 5, Angular 6, na rzecz krótszego Angular z semantyczną wersją.[6]

Aplikacje stworzone za pomocą Angulara oparte są na komponentach, które komunikują się ze sobą. Komponent przyjmuje dane od rodzica, a sam może wytwarzać zdarzenia, na które nasłuchuje rodzic. Jest to normalne dla języka HTML, dlatego Angular jest łatwy do zrozumienia i nauczenia.[7]

Każda Angularowa aplikacja ma przynajmniej jeden komponent, komponent-rodzic, który łączy hierarchie komponentów z DOM-em. Każdy komponent definiuje klasę, która zawiera dane , logikę oraz połączony z nim szablon dokumentu HTML, który definiuje widok, który ma zostać wyświetlony w docelowym środowisku.[17]

Szablony łączą HTML ze znacznikami Angulara, które mogą modyfikować elementy HTML zanim zostaną wyświetlone. Dyrektywy szablonów zapewniają logikę programu, a znaczniki łączą dane aplikacji i DOM. Zanim widok jest wyświetlony, Angular wykonuje logikę schowaną za dyrektywami i rozwiązuje składnie wiązania w szablonie, aby zmodyfikować elementy HTML i DOM, zgodnie z danymi i logiką programu. Angular wspiera dwustronne wiązanie danych(two-way data binidng). Funkcjonalność ta zapewnia, że każda zmiana dokonana przez użytkownika wpływa na zmianę danych, a zmiana danych wpływa na to co zobaczy użytkownik.[17]

# ReactJS

React jest popularną biblioteką używaną do tworzenia aplikacji internetowych. Została utworzona przez firmę Facebook, aby poradzić sobie z trudnościami, które napotykali. Kiedy React został wydany w roku 2013, do projektu podchodzono sceptycznie ponieważ konwencje, które zostały tam użyte są dosyć wyjątkowe[8].

React jest małą biblioteką i jest używana jedynie do małej części pracy. React nie dostarcza wielu narzędzi, których można by było się spodziewać po tak popularnej bibliotece. Wiele decyzji, co do używanych narzędzi należy, należy do dewelopera. Dodatkowo cały czas pojawiają się nowe narzędzia, które wypierają stare. Każdego dnia pojawiają się nowe narzędzia, przez które niemożliwe jest aby nadążyć za wszystkimi trendami.[8]

Zespół utrzymujący bibliotekę React również wydał JavaScript XML(JSX), który dostarcza funkcjonalność do tworzenia skomplikowanych drzew DOM z atrybutami. JSX również ułatwia czytanie i pisanie kodu.[8]

W JSX, typ elementu jest określony za pomocą znacznika. Atrybuty znacznika reprezentują właściwości. Dziecko elementu może być dodane pomiędzy znacznik otwierający i zamykający.

Aby utworzyć komponent Reactowy, należy posłużyć się funkcją createElement.

1. React.createElement(
2. typ,
3. [właściwości],
4. [...dziecko]
5. )

gdzie:

* Typ – typ elementu np. div, span albo inny komponent React
* Właściwości - czyli cechy przekazywane do utworzonego elementu np.

1. {
2. onClick: () => console.alert('Kliknięto mnie!')
3. }

* Dziecko – to co zostanie umieszczone pomiędzy znacznikiem otwierającym a zamykającym

Zatem aby prosty komponent, należy użyć funkcji w następujący sposób:

1. **const** komponent = React.createElement(
2. 'div',
3. {
4. onClick: () => console.alert('Kliknięto mnie!')
5. },
6. 'Witaj Świecie!'
7. )

Natomiast z użyciem rozszerzenia JSX można to zapisać o wiele krócej, a kod jest dużo czytelniejsza:

**const** komponent = <div onClick={() => console.alert('Kliknięto mnie!')}>Witaj Świecie!</div>

Aby móc pracować z React, należy załadować dwie biblioteki: React i ReactDOM. React jest biblioteką do tworzenia widoków. ReactDOM jest biblioteką używaną do renderowania interfejsu użytkownika w przeglądarce.[8]

React i ReactDOM zostały podzielone na dwa pakiety we wczesnych fazach życia projektu.[8] Podział ten pozwolił na utworzenie biblioteki ReactNative, która pozwala pisać aplikacje w React na platformy mobilne. Wystarczy jedynie wymienić bibliotekę ReactDOM na ReactNative. Dzięki takiemu rozwiązaniu biblioteka React nie jest zależna, od konkretnego środowiska uruchomieniowego, można zarówno ją użyć w przeglądarce jak i na Androidzie czy iOS. Ale nie tylko na tych platformach można użyć React. Dzięki pakietowi react-reconciler, programista może utworzyć swój własny silnik renderujący, który może użyć do swojego specyficznego przypadku np. zarządzanie matrycą diod led.

Biblioteka React stała się niezwykle popularna ze względu na bardzo niski próg wejścia. Sama biblioteka jest bardzo prosta w zrozumieniu, a nowe komponenty powstają w ekspresowym tempie. Prostość biblioteki wpływa też na to, że jest bardzo surowa. Operacje zmiany wartości w polu czy walidacji formularza programista musi obsłużyć ręcznie. Daje to dużą swobodę w działaniu, ale również wpływa na to, że jeżeli chce się dostarczyć dojrzałą aplikację spełniającą wymogi biznesowe to należy użyć zewnętrznych bibliotek, których ilość może czasami przytłoczyć osobę, która dołącza projektu.

# Vue.js

Vue jest progresywnym frameworkiem do budowania interfejsów użytkownika. W przeciwieństwie do innych monolitycznych frameworków, Vue jest zaprojektowany od podstaw do tego aby stopniowo go przystosowywać do aplikacji. Rdzeń biblioteki jest skoncentrowany tylko na warstwie widoku i z tego powodu jest łatwy do wdrożenia i integracji z innymi bibliotekami albo istniejącymi projektami. Z drugiej strony Vue perfekcyjnie pasuje jako framework, do utworzenia Single-Page-Application(SPA), kiedy jest używany ze współczesnymi narzędziami oraz wspierającymi bibliotekami.[9]

Framework został wymyślony przez Evana You, który pracował w Google Creative Labs nad jednym z projektów, gdzie potrzebowali szybkiego prototypowania niż dużego interfejsu użytkownika. Pisanie wielu powtarzalnych dokumentów HTML było czaso-i-zasobożerne. Na tamtejszy czas nie znaleziono rozwiązania, które by pasowało do istniejących potrzeb. Z tego powodu Evan utworzył bibliotekę, która pomogła w szybkim prototypowaniu interfejsów użytkownika na podstawie modelu danych.[10]

Jak każda dobra biblioteka, Vue rosło i rozwijało się, dostarczając coraz to więcej funkcjonalności niż zapowiadało się na to na początku. Obecnie, framework dostarcza prosty sposób do dołączania i tworzenia wtyczek oraz generalnie dodawania własnej logiki i zachowań.[10]

Vue.js pozwala w prosty sposób połączyć modele danych z warstwą reprezentacji. Pozwala również na łatwą re-używalność utworzonych komponentów. Nie ma potrzeby tworzenia specjalnych modeli lub rejestrowania wydarzeń, nie trzeba również znać specjalnej składni. Modele to czyste obiekty JavaScriptowe[10].

W vue w łatwy sposób można odseparować warstwę widoku, od stylów oraz logiki kodu JavaScript lub też umieścić wszystkie te rzeczy w jednym pliku. Istnieje wtyczka do Vue, która wspiera wszystkie nowoczesne i często używane IDE.[10]

# 2.7 npm

Node Package Manager(npm) jest największym na świecie rejestrem oprogramowania. Programiści oprogramowania z wolnym dostępem używają npm do dzielenie się oraz pożyczania pakietów. Wiele organizacji również używa npm do zarządzania prywatnymi paczkami.[11]

npm składa się z trzech komponentów:

* ze strony, gdzie użytkownicy mogą odkrywać paczki, ustawiać swoje profile i zarządzać innymi aspektami np. stworzyć własną organizację to zarządzania dostępem do publicznych i prywatnych pakietów,
* z interfejsu linii komend, który uruchamiany jest w terminalu. W ten sposób większość programistów, korzysta z npm,
* z rejestru, który jest ogromną bazą publicznych bibliotek JavaScript oraz związanymi z nimi metadanymi.[11]

npm jest bardzo dobrym narzędziem do zarządzania projektem. Za pomocą krótkiej komendy, programista może dodać nową bibliotekę do swojego projektu, ustawić predefiniowane skrypty, a także wpisać określić autora paczki czy też jej licencje. Wszystkie definicje zależności są trzymane w jednym pliku, a zewnętrzne zasoby są pobierane do jednego folderu node\_modules.

# 2.8 Yarn

Yarn podobnie jak npm jest menadżerem pakietów. Pozwala na użycie i dzielenie się kodem z innymi programistami z całego świata. Yarn robi to szybko, bezpiecznie i niezawodnie. Yarn pozwala używać rozwiązań innych developerów, aby w łatwiejszy sposób dostarczać aplikację. Kod jest współdzielony przez paczkę. Paczka zawiera cały kod, który jest dzielony razem z plikiem package.json, który opisuje paczkę[12]

# 2.9 Git i GitHub

Git jest rozproszonym systemem kontroli wersji. Pozwala on w łatwy sposób zarządzać i udostępniać kod swojej aplikacji. Dzięki historii, którą tworzy git jest możliwość w łatwy sposób prześledzić zmiany w pliku i dotrzeć do autora zmian. Git pozwala na tworzenie gałęzi, które można potem dowolnie łączyć. Funkcjonalność ta jest bardzo ważna w procesie wytwarzania oprogramowania ze względu na to, że projekty najczęściej powstają w zespołach, a praca na tym samym kodzie przez wielu programistów, bez narzędzia zarządzającego była by praktycznie nie możliwa.

Największą różnicą pomiędzy Gitem a innymi systemami kontroli wersji jest fakt w jaki sposób Git zarządza danymi. Konceptualnie, wiele innych systemów przechowuje informacje jako listę zmian w plikach. Systemy te myślą o informacji, które przechowują jako zestaw plików i zmiany wprowadzonej w każdym pliku w czasie. Git działa w inny sposób. Git traktuje dane bardziej jak serię migawek miniaturowego systemu plików. Z każdą zatwierdzoną zmianą kodu, Git tworzy obraz wszystkich plików – jak wyglądają w danym momencie – i zapisuje referencje do tej migawki. Aby być wydajnym, jeżeli plik się nie zmienił, Git nie zapisuje tego pliku ponownie, po prostu łączy go do pliku, który już ma zapisany.[13]

Praktycznie każdą operację można wykonać lokalnie, bez dostępu do innego komputera czy sieci. Pobierając na komputer repozytorium, pobieramy kopię oprogramowania, na której można wykonywać operację. Po skończonej pracy, kod może zostać wysłany do repozytorium skąd został pobrany, a następnie połączony z istniejącym.[13]

Wszystko w Git jest podawane operacji sumy kontrolnej przed zapisem, a następnie określane przez tę sumę. Oznacza to, że jest to niemożliwe aby zmienić zawartość dowolnego pliku albo folderu bez wiedzy Git o tym. Funkcjonalność ta jest wbudowana w Git na najniższych poziomach i jest integralną części jego filozofii. Nie można stracić informacji w trakcie przesyłania lub uszkodzić plików beż wykrycia tego przez Git.[13]

GitHub jest platformą developerską, na której można wykonywać przegląd kodu, zarządzać projektami oraz budować oprogramowanie bazując na Git. GitHub nie jest jedynie miejscem gdzie można przetrzymywać kod, można go również używać do planowania oraz śledzenia rozwoju oprogramowania. [14] Na platformie GitHub można również znaleźć mnóstwo projektów z otwartym źródłem, gdzie programiści mogą podpatrzeć najlepsze praktyki lub kontrybuować. Oprogramowanie posiada również funkcję do tworzenia i śledzenia zadań tak aby ułatwić pracę w zespole. Dodatkowo posiada również funkcjonalność integrowania się z systemami CI/CD, aby proces wdrożenia aplikacji był jeszcze szybszy.

# 2.10 Wirtualny DOM a DOM

HTML jest prostym zbiorem instrukcji, za którymi podąża przeglądarka konstruując obiektowy model dokumentu(document object model – DOM). Elementy, które tworzą dokument HTML stają się elementami DOM kiedy przeglądarka wczytuje HTML i tworzy widok użytkownika. W HTML elementy zależą od siebie w hierarchii, która przypomina drzewo. Można powiedzieć, że główny tag HTML posiada dwoje dzieci: head oraz body.[8]

Tradycyjnie strony internetowe składają się z niezależnych stron HTML. Kiedy użytkownik przegląda te strony, przeglądarka załaduje różne dokumenty HTML. Dzięki wynalezieniu AJAX, mogły powstać jednostronne aplikacje(single-page applications - SPA). Od tego momentu przeglądarki mogły ładować dane używając AJAX, co zapewniło możliwość umieszczenia wszystkiego na jednej stronie i użycie JavaScript do odświeżania interfejsu użytkownika.[8]

W SPA przeglądarka początkowo ładuje jeden dokument HTML. Podczas nawigowania po aplikacji przez użytkownika, zostaje on ciągle na tej samej stronie. JavaScript niszczy i tworzy nowy interfejs użytkownika podczas gdy użytkownik prowadzi interakcję z aplikacją. Tworzy to wrażenie jakby użytkownik przeskakiwał ze strony na stronę, ale właściwie zostaje na tej samej stronie HTML, a JavaScript wykonuje ciężką pracę.[8]

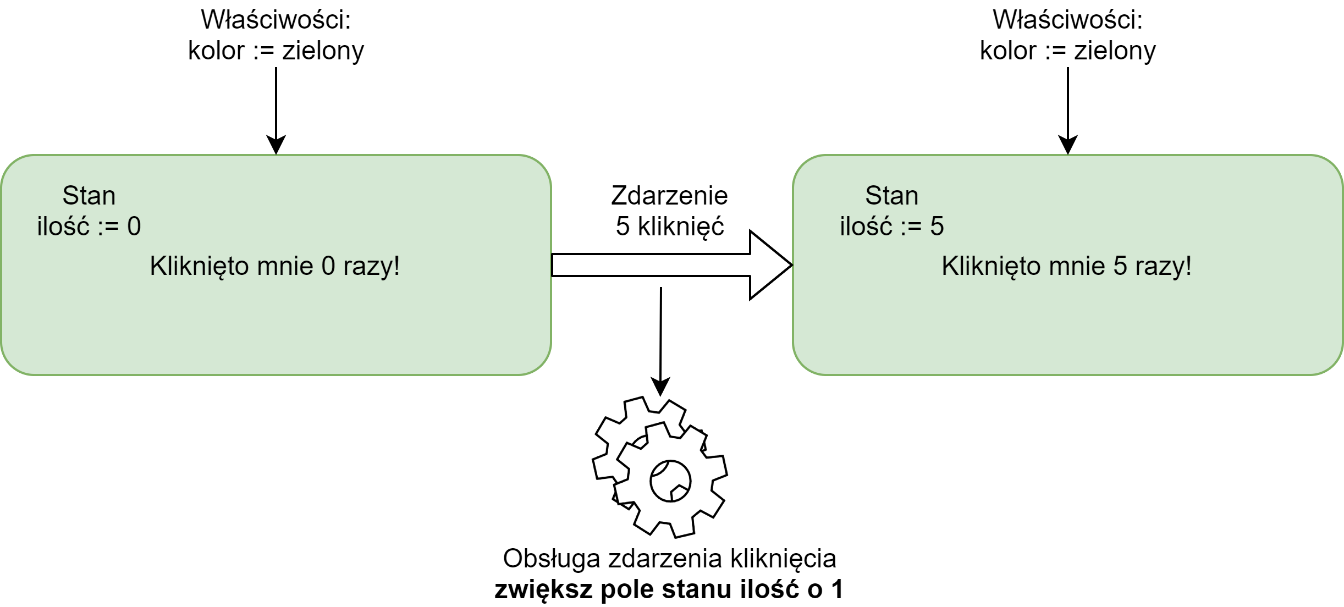
DOM API to kolekcja obiektów, których JavaScript może użyć do interakcji z przeglądarką do modyfikowania drzewa DOM. Metoda document.createElement utworzy nowy element DOM a document.appendChild dołączy dziecko do istniejącego elementu. Tworzenie lub łączenie elementów poprzez JavaScript jest relatywnie proste. Jednakże proces dodawania nowych elementów jest bardzo wolny. Oznacza to że jeżeli programiści stron internetowych nie zadbają o to jakie zmiany wykonują w interfejsie użytkownika, mogą w znaczącym stopniu pogorszyć efektywność działania.[8]

Efektywne zarządzanie zmianami DOM z JavaScript może stać się bardzo skomplikowane i czasochłonne. Z punktu widzenia kodu, łatwiej jest wyczyścić wszystkie dzieci konkretnego elementu i stworzyć je na nowo niż starać się efektywnie je odświeżyć.[8]

Wirtualny DOM jest stworzony z elementów, które konceptualnie przypominają elementy HTML, ale w rzeczywistości są obiektami JavaScript. Praca z obiektami JavaScript jest dużo szybsza i łatwiejsza niż z DOM API. Zmiany wykonywane są na wirtualnym DOM a silnik obsługujący dany framework zajmuje się nakładaniem zmian używając DOM API tak efektywnie jak to tylko możliwe.[8]

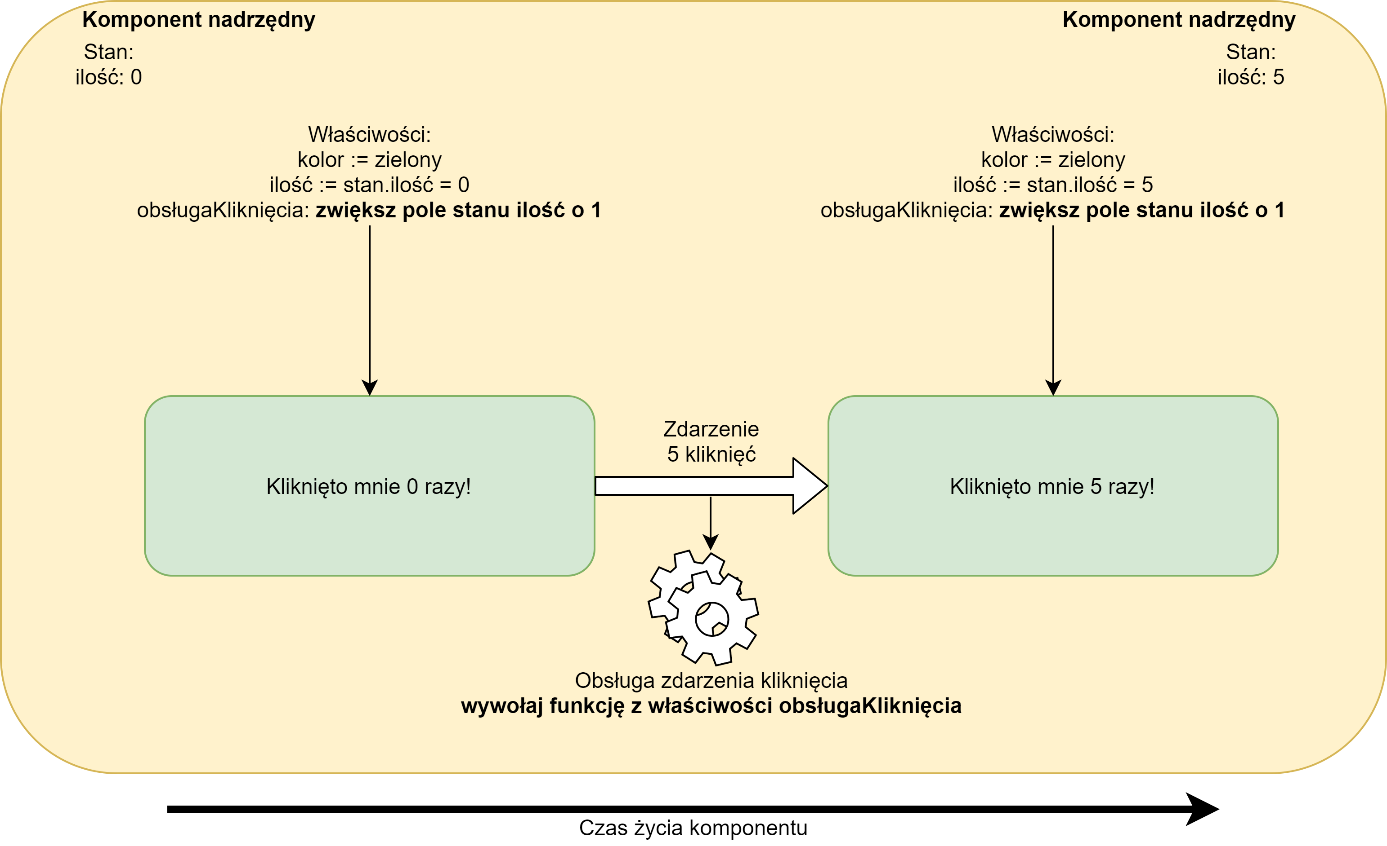
# 2.11 Programowanie komponentowe

Programowanie komponentowe jest to podejście, w którym najmniejszą jednostką, z których składa się aplikacja to komponent. Komponent w swojej budowie, może zawierać inne komponenty. Dodatkowo komponent posiada stan oraz właściwości. Z założenia właściwości są tylko do odczytu(nie można ich modyfikować). Stan komponentu to pewne cechy, które mogą zostać zmieniane podczas czasu życia komponentu. Różnica pomiędzy właściwościami a stanem została pokazana na schemacie poniżej.



Na schemacie można zauważyć przykład komponentu, który posiada właściwości jak i stan. Właściwością komponentu jest jego kolor, który nie może zostać modyfikowany przez komponent. W stanie komponentu przetrzymywana jest ilość kliknięć na niego. Za każdym razem gdy występuje zdarzenie kliknięcia w komponent, uruchamiana jest funkcja, która obsługuje to zdarzenie. Funkcja ta z każdym kliknięciem zwiększa ilość o 1. Po pięciu kliknięciach komponent zmienił stan 5cio krotnie, natomiast właściwości zostały cały czas takie same. Stan jest zmieniany wewnątrz komponentu, a właściwości mogą być zmienione jedynie z zewnątrz.

Komponenty dzielimy również na stanowe i bezstanowe. Komponenty stanowe, nazywane również mądrymi komponentami, posiadają stan, którym zarządza. W odróżnieniu od komponentów stanowych, komponenty bezstanowe, zwane również prezentacyjnymi, nie posiadają stanu – opierają się jedynie na właściwościach. Komponent stanowy został przedstawiony na schemacie X, natomiast na schemacie poniżej zostanie przedstawiony równoważny komponent bezstanowy.



Na powyższym schemacie widać komponent bezstanowy oraz komponent nadrzędny. Komponent bezstanowy nie posiada stanu, a jego wygląd i zachowanie zależy tylko od jego właściwości, które są dostarczane przez komponent nadrzędny. Komponent nadrzędny przekazuje kolor, ilość oraz w jaki sposób komponent ma się zachować na zdarzenie kliknięcia. Gdy następuje zdarzenie kliknięcia uruchamiana jest funkcja z właściwości komponentów, która to zmienia stan w komponencie nadrzędnym. Technika wynoszenia stanu do góry[15] jest często używana przez programistów, gdyż sprawia że komponent jest zależny tylko od tego co dostanie z zewnątrz. Powoduje to, że komponent jest łatwiejszy w utrzymaniu i testowaniu.

# Metodyka badań

W poniższym rozdziale opisana zostało jakie badania zostały wykonane, jakie zostały zdefiniowane kryteria porównanie oraz jak te rzeczy, mogą wpływać na efektywność procesy wytwórczego.

# 3.1 Rodzaje badań

Badania zostały podzielone na 2 kategorie. Pierwsza kategoria to dane statyczne, które wymagają lektury literatury bądź analizy dokumentacji technicznej. Drugą kategoria to dane, które zostały pobrane podczas eksperymentów z bibliotekami i frameworkami.

# 3.2 Kryteria porównań

W pierwszej kolejności zostały porównane dane statyczne, które odpowiadają za poniższe rzeczy:

* obsługa żądań HTTP,
* obsługa formularzy,
* użycie wirtualnego DOM,
* obsługa wyrażeń warunkowych,
* obsługa pętli,
* licencja,
* wsparcie przeglądarek,
* wsparcie platform mobilnych,
* dostępność CLI (command line interface),
* dostępność i jakość dokumentacji,
* podatność na rozszerzenia,
* zabezpieczenia przed podstawowymi podatnościami,
* społeczność.

Następnie zostały przeprowadzone badania, dzięki którym zbadano następujące właściwości.

* rozmiar biblioteki,
* rozmiar podstawowej aplikacji dostarczanej przez bibliotekę/framework,
* ilość linii potrzebnych do napisania podstawowej aplikacji (TODO app),
* test wydajności przy:
  + dodawaniu komponentów,
  + zastępowaniu komponentów,
  + zastępowaniu komponentów w wybranym interwale,
  + wyróżnienie jednego komponentu po kliknięciu,
  + zamiana miejscami 2 komponentów,
  + wyczyszczenie komponentów.

1. **Wytworzenie oprogramowania**

W poniższym rozdziale opisany został proces wytworzenia aplikacji, wymaganej do pobrania metryk w każdej z bibliotek. Zostały również opisane aplikacje oraz wymagania co do nich.

# 4.1 Opis i wymagania aplikacji

W ramach badań zostały stworzone 3 aplikacje. Pierwsza aplikacja to prosty przykład aplikacji biznesowej, w której użytkownik może tworzyć zadania do wykonania. Dodatkowo aplikacja posiada funkcjonalność, oznaczania czynności jako wykonaną. Jest to aplikacja, którą bardzo często się wykonuje podczas poznawania frameworka czy biblioteki, ponieważ wymaga użycia podstawowych konceptów. Na podstawie tej aplikacji będzie możliwość pobrania takich miar jak ilość linii kodu czy trudność i efektywność w implementacji danej biblioteki. Aplikacja zostanie zaimplementowana we wszystkich badanych bibliotekach.

Druga aplikacja jest systemem, który pozwala wykonywać modyfikacje drzewa DOM. Posiada on takie funkcjonalności jak dodawanie, usuwanie i zamienianie dużej ilości komponentów. W ten sposób będzie możliwość zbadania jak aplikację w różnych środowiskach poradzą sobie z tym problemem obciążeniowym. Aplikacja również zostanie zaimplementowana we wszystkich badanych bibliotekach.

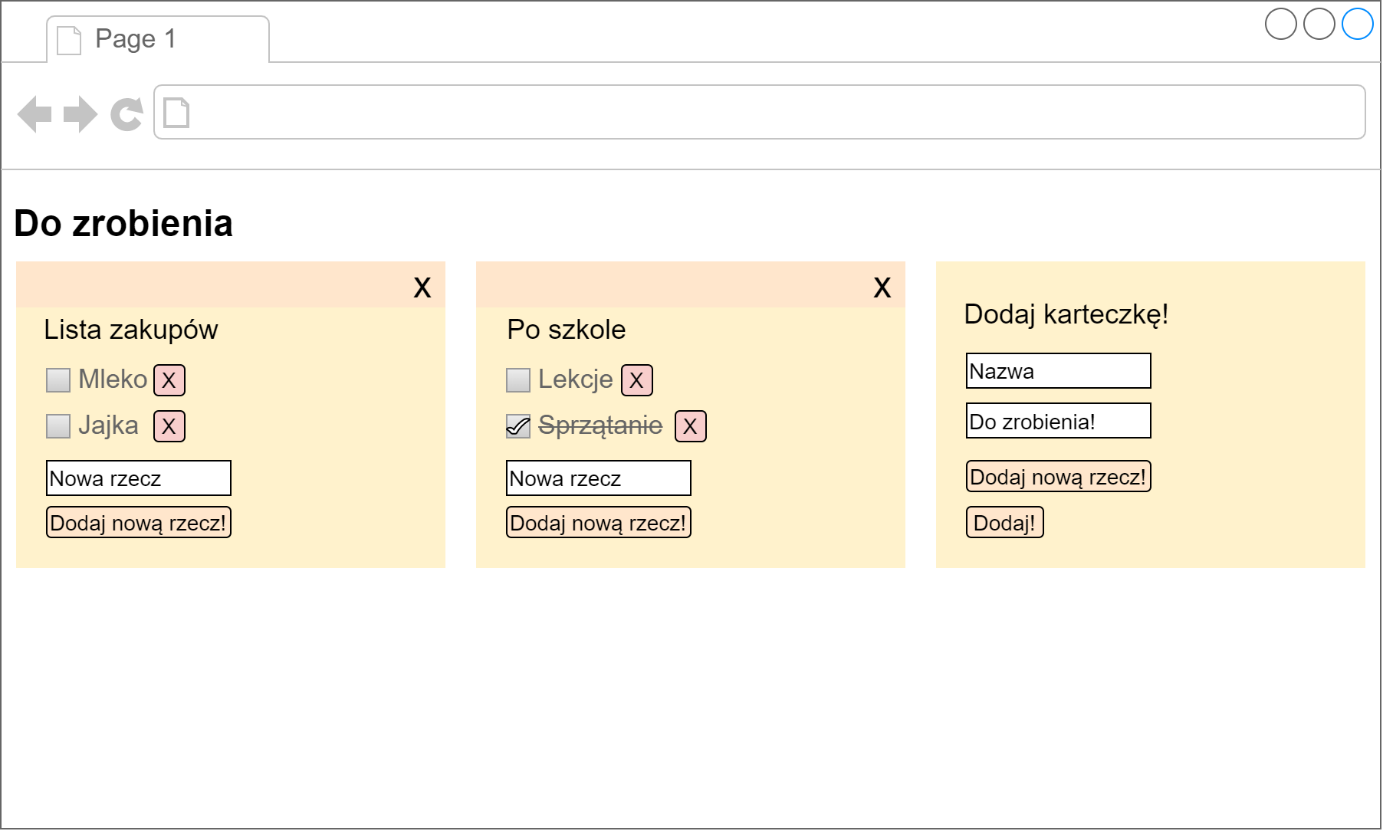
Ostatnią aplikacją, jest system który potrafi przetestować i zebrać metryki czasowe, aplikacji do modyfikacji drzewa DOM. System ten potrafi automatycznie wykonać funkcjonalność, która jest dostarczana przez badaną aplikację. System ten potrafi zmierzyć czas działania modyfikacji oraz potrafi wykonać zadaną ilość iteracji.

# 4.2 Model wymagań

Pierwsza aplikacja została nazwana „Do zrobienia”. Jej wymagania prezentują się następująco:

* aplikacja pozwala dodać grupę zadań do wykonania i nadać jej tytuł,
* aplikacja pozwala usunąć grupę zadań do wykonania
* aplikacja pozwala dodać zadanie do zrobienia,
* aplikacja pozwala oznaczyć zadanie jako wykonane,
* aplikacja pozwala usunąć zadanie.

Na poniższym schemacie zaprezentowano makietę strony:



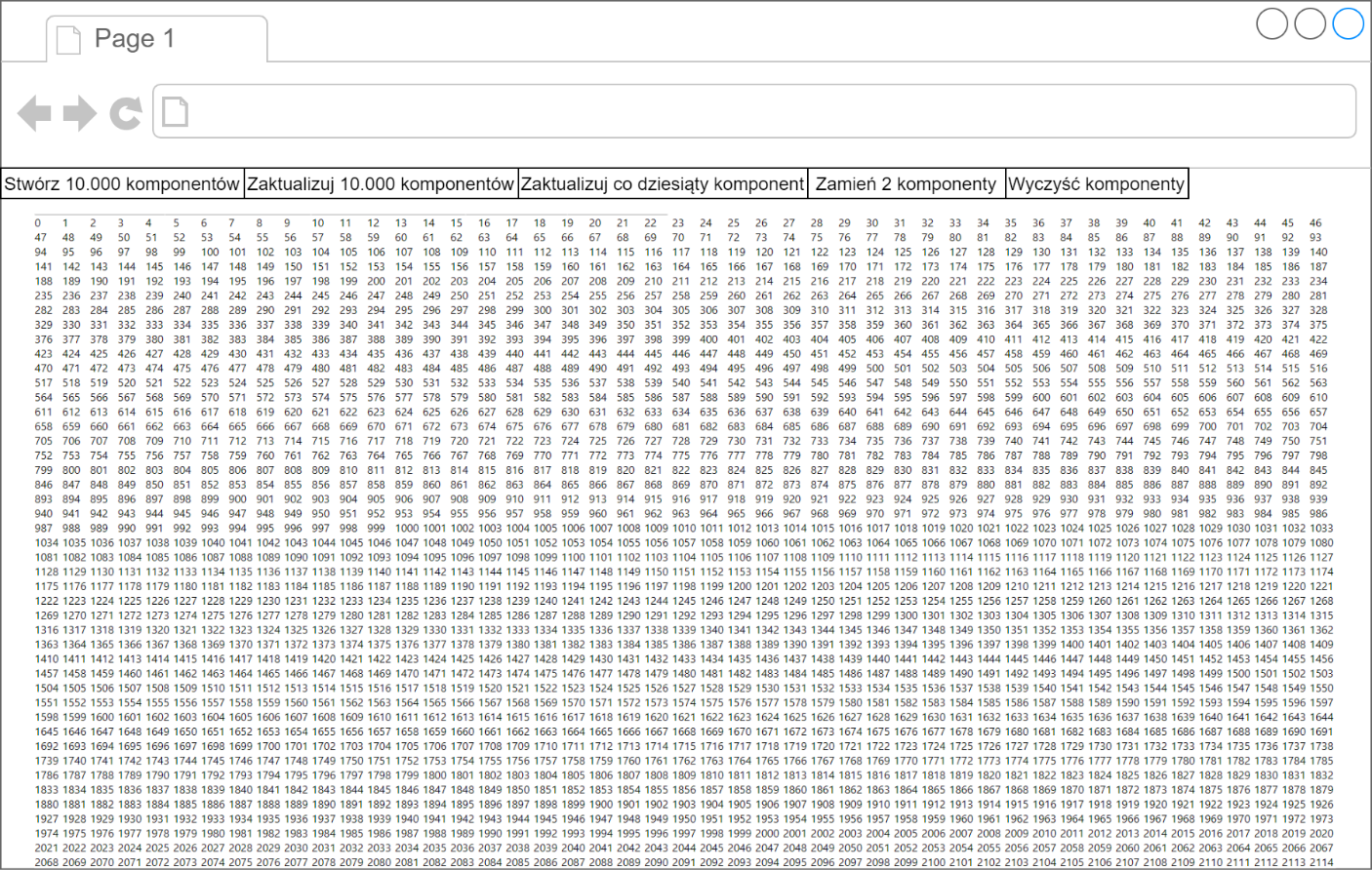
Aplikacja składa się z następujących komponentów:

* nazwy aplikacji,
* pojedynczej grupy zadań do zrobienia, która zawiera następujące komponenty:
  + nazwę grupy zadań,
  + listę zadań do wykonania z polem wyboru, który informuje o statusie zadania oraz przycisk do usunięcia zdania,
  + pole tekstowe wraz z przyciskiem do dodania nowego zadania do aktualnej grupy,
  + przycisk do usunięcia grupy.
* komponent, który pozwala na dodanie nowej grupy zadań, który składa się z:
  + nagłówka,
  + pola tekstowego, w którym umieszczona jest nazwa grupy,
  + pół tekstowych, w których umieszczone są nazwy zadań do wykonania,
  + przycisk do utworzenia kolejnego pola tekstowego z nazwą zadania,
  + przycisk do dodania grupy zadań.

Druga aplikacja została nazwana „Benchmark Item”. Jej wymagania prezentują się następująco:

* aplikacja pozwala dodać 10.000 komponentów,
* aplikacja pozwala zastąpić 10.000 komponentów,
* aplikacja pozwala zastąpić co 10 komponent w grupie 10.000 komponentów,
* aplikacja pozwala wyróżnić kliknięty komponent w grupie 10.000 komponentów,
* aplikacja pozwala zamienić miejscami 2 komponenty,
* aplikacja pozwala wyczyścić 10.000 komponentów,
* aplikacja posiada prosty interfejs graficzny do wykonania tych czynności.

Makieta aplikacji prezentuje się w następujący sposób:



Na samej górze istnieją przyciski, dzięki którym możliwa jest modyfikacja drzewem DOM. Poniżej wyświetlone są komponenty, które są efektem wywołań powyższych funkcjonalności.

Ostatnia aplikacja została nazwana „Benchmark Engine”. Jej zadaniem jest przetestowanie i zebranie odpowiednich metryk. Jej wymagania prezentują się następująco:

* aplikacja powinna udostępniać interfejs dostępny z poziomu wiersza polecenia,
* aplikacja potrafi w sposób rzetelny zmierzyć czas wykonania akcji, zaimplementowanych w aplikacjach „Benchmark Item”,
* aplikacja potrafi przeprowadzić badania z zadaną liczbą iteracji,
* aplikacja potrafi zebrać wszystkie czasy, wartość średnią, maksymalną oraz minimalną,
* aplikacja potrafi udostępnić zebrane dane w formacie JSON,
* aplikacja potrafi współpracować niezależnie od platformy implementacji,
* aplikacja potrafi zebrać wszystkie dane podczas jednego uruchomienia.

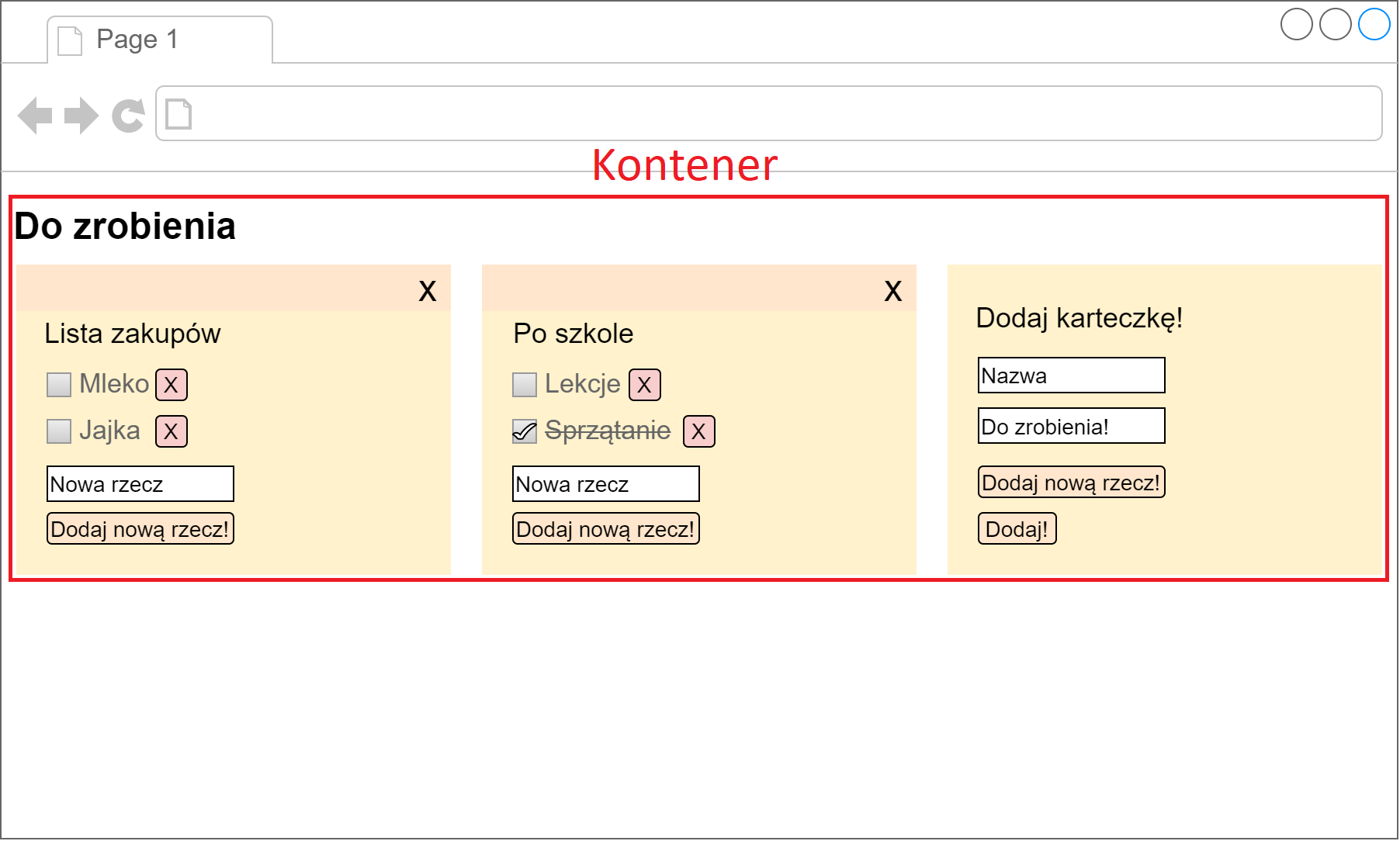
# 4.3 Implementacja rozwiązania

W poniższym rozdziale został opisany proces wytworzenia oprogramowania wraz z fragmentami kodu i opisem słownym.

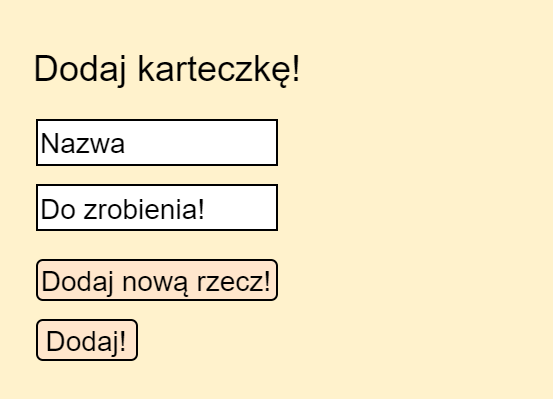
# 4.3.1 Do zrobienia

Niezależnie od implementacji, można wyróżnić części wspólne dla wszystkich rozwiązań. Aplikacja „Do zrobienia” jest doskonałym przykładem aplikacji opartej na komponentach. Aplikacja składa się głównie z 4 rzeczy:

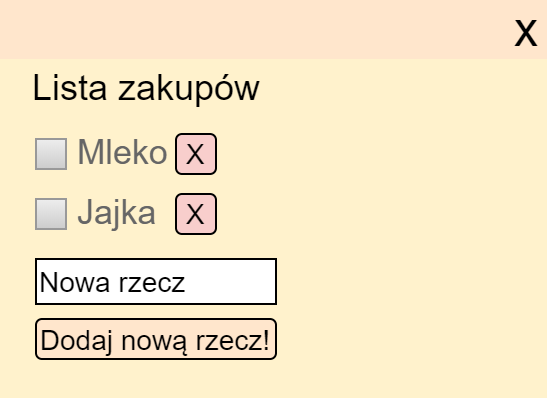
* kontener, który opakowuje wszystkie inne komponenty



* formularz, który pozwala dodać nową grupę zadań



* grupę zadań do wykonania



* pojedyncze zdanie, które jest enkapsulowane przez grupę zadań



Wszystkie te komponenty mają swoje zadania, które realizują. Zaczynając od najmniejszego składnika jakim jest pojedyncze zadanie. Jego funkcjonalnością jest wyświetlanie jednego zadania oraz jego stanu. Daje on również możliwość zaznaczenia danego zadania jako wykonane lub nie. Dodatkowo użytkownik końcowy posiada funkcjonalność usunięcia go z grupy. Kolekcja zadań to grupa. Oprócz funkcji agregacyjnej dostarcza również możliwość dodania nowego zadania do już istniejącej grupy. Jeżeli użytkownik chce utworzyć nową grupę, powinien skorzystać z komponentu dodawania nowej grupy, który posiada taką możliwość. Ostatnim komponentem jest kontener, który opakowuje wszystkie komponenty w aplikacji oraz w zależności od implementacji, może posiadać również inne funkcje np. przechowywanie stanu.

Warto wspomnieć o tym, że pomimo że komponenty posiadają taką samą funkcjonalność w każdym z poniższych rozwiązań, mogą one posiadać różne interfejsy. Podyktowane jest to tym, że języki te różnią się od siebie i mogą oferować różne sposoby do rozwiązania tego problemu biznesowego.

Dodatkowo wspólną częścią dzieloną między aplikacjami jest model danych. W każdej implementacji prezentuje on się w następujący sposób.

1. **interface** State {
2. groups: Group[]
3. }
5. **interface** Group {
6. name: string;
7. todos: Todo[];
8. }
10. **interface** Todo {
11. name: string;
12. done: **boolean**;
13. }

Nadrzędnym elementem modelu jest tutaj State, który odpowiada za przechowywanie stanu aplikacji, czyli kolekcji grup. Pojedyncza grupa składa się z nazwy oraz z tablicy, rzeczy do zrobienia. Najmniejszym obiektem jest pojedyncza rzecz, która zawiera jej nazwę i status. Przykładowy stan aplikacji znajduje się poniżej.

1. [
2. {
3. name: 'Lista zakupów',
4. todos: [
5. {
6. name: 'Mydło',
7. done: **true**,
8. },
9. {
10. name: 'Szampon',
11. done: **false**,
12. },
13. ],
14. },
15. {
16. name: 'Po szkole',
17. todos: [
18. {
19. name: 'Lekcje',
20. done: **true**,
21. },
22. {
23. name: 'Sprzątanie',
24. done: **false**,
25. },
26. ],
27. },
28. ]

# 4.3.1.1 React

Opis implementacji w bibliotece ReactJS zostanie rozpoczęty od najmniejszej jednostki czyli komponentu pojedynczego zadania.

1. **export** **const** Todo = ({ name, done, onCheck, onRemove }) => (
2. <div>
3. <input
4. type={'checkbox'}
5. checked={done}
6. onChange={(e) => onCheck(e.target.checked)}
7. />
8. <span className={done ? 'done' : ''}>{name}</span>
9. <input className={'remove'} type={'button'} value={'x'} onClick={onRemove}/>
10. </div>
11. )

Pierwsza linia prezentuje interfejs komponentu – posiada on 4 różne właściwości

* name – nazwa zadania,
* done – status zadania,
* onCheck – funkcja, która ma się wykonać gdy użytkownik kliknie na przycisk wyboru,
* onRemove – funkcja, która wykona się gdy użytkownik kliknie na przycisk usunięcia zadania.

Zapis zmiennych pomiędzy nawiasami klamrowymi to destrukturyzacja, która została dodana wraz z ES6. Pozwala on na bezpośrednie użycie pól obiektu. Zapis ten jest ekwiwalentny z poniższym:

1. **export** **const** Todo = (props) => {
2. let { name, done, onCheck, onRemove } = props;
3. **return** (
4. <div>
5. ...
6. </div>
7. )
8. }

Kod ten z kolei przekłada się na:

1. **export** **const** Todo = (props) => {
2. let name = props.name;
3. let done = props.done;
4. let onCheck = props.onCheck;
5. let onRemove = props.onRemove;
6. **return** (
7. <div>
8. ...
9. </div>
10. )
11. }

Jak widać ten prosty zapis pozwolił zaoszczędzić kilku linii kodu.

W powyższym komponencie nie ma żadnej logiki, jest on typowym komponentem prezentacyjnym. Przekazywana do niego nazwa oraz status są bezpośrednio wyświetlane. Przekazywane są również uchwyty do funkcji, które są wykonywane podczas zaznaczenia/odznaczenia zadania oraz gdy zadanie ma zostać usunięte.

Kolejnym komponentem jest komponent grupy.

1. **export** **const** TodoGroup = ({
2. name,
3. todos,
4. groupIndex,
5. onCheck,
6. onRemove,
7. onTodoRemove,
8. onTodoAdd,
9. }) => {
10. **const** [newTodo, setNewTodo] = useState('')
11. **const** onNewTodoAdd = () => {
12. onTodoAdd(newTodo)
13. setNewTodo('')
14. }
15. **return** (
16. <div>
17. <div className={'todo-group\_\_header'}>
18. <span className={'pointer'} onClick={onRemove}>X</span>
19. </div>
20. <div className={'todo-group\_\_body'}>
21. <h3>{name}</h3>
22. {todos.map((todo, i) => (
23. <Todo
24. key={todo.name}
25. {...todo}
26. onCheck={onCheck(groupIndex, i)}
27. onRemove={onTodoRemove(groupIndex, i)}
28. groupIndex={groupIndex}
29. todoIndex={i}
30. />
31. ))}
32. <input
33. type={'text'}
34. placeholder={'Nowa rzecz'}
35. value={newTodo}
36. onChange={(e) => setNewTodo(e.target.value)}
37. />
38. <input
39. className={'btn'}
40. type={'button'}
41. value={'Dodaj nową rzecz!'}
42. onClick={onNewTodoAdd}
43. />
44. </div>
45. </div>
46. )
47. }

W skład jego interfejsu wchodzą:

* name – nazwa grupy,
* todos – kolekcja zadań do wykonania,
* groupIndex – indeks grupy,
* onCheck, onTodoRemove – funkcje, która jest przekazywana z komponentu nadrzędnego do komponentu podrzędnego (opisane w komponencie pojedynczego zadania),
* onRemove – funkcja, która zostaje wywołana w momencie usunięcia całej grupy,
* onTodoAdd – funkcja, która zostaje wywołana w momencie kliknięcia w przycisk dodania nowego zadania.

Komponent ten w przeciwieństwie od poprzedniego nie jest tylko komponentem prezentacyjnym – posiada on własną logikę. W linii 10. użyta została funkcjonalność, która pozwala na utrzymywanie stanu w komponencie funkcyjnym. Dodatkowo w następnej linii zdefiniowana jest funkcja, która jest użyta w momencie dodania nowego zadania. Używa ona funkcji przekazanej z zewnątrz i wywołuje ją z argumentem, który jest reprezentacją nowego zadania. Następnie pole tekstowe jest czyszczone przy pomocy funkcji setNewTodo(). Funkcja ta jest również używana w momencie zmiany pola tekstowego, w którym jest wpisywane nowe zadanie.

1. <input
2. type={'text'}
3. placeholder={'Nowa rzecz'}
4. value={newTodo}
5. onChange={(e) => setNewTodo(e.target.value)}
6. />

Zabieg ten pozwala na ścisłą kontrolę i zarządzanie przez programistę, co ma się w polu tekstowym wyświetlać i w jaki sposób ma reagować na zmiany. Pole te przyjmuje 2 własności:

* value – która odpowiada za bieżącą wartość pola tekstowego,
* onChange – funkcja, która wykonuje się podczas jakiejkolwiek zmiany w polu.

Mogłoby się wydawać że funkcja onChange jest tu zbędna. Jednak w podejściu jakie dostarcza React jest ona niezbędna do prawidłowego funkcjonowania witryny. Gdyby funkcji tej zabrakło, pole newTodo nigdy by się nie zmieniało, a co za tym idzie cały czas wyświetlona by była ta sama wartość. Można tą funkcjonalność traktować dwojako. Z jednej strony obciąża to programistę i do każdego pola formularza należy dostarczyć taką funkcjonalność, co na przestrzeni dużej ilości wartości może być zadaniem żmudnym i powtarzającym się. Z drugiej strony rozwiązanie to dostarcza bardzo dużo swobody, dzięki której programista sam może zdecydować jak dane wydarzenia mogą wpływać na zachowanie się komponentu.

Kolejną ciekawą sprawą jest fakt w jaki sposób w ReactJS, buduje się tablicę komponentów. Otóż zazwyczaj robi się to za pomocą funkcji map() z interfejsu tablicy Javascriptowej:

1. {todos.map((todo, i) => (
2. <Todo
3. key={todo.name}
4. {...todo}
5. onCheck={onCheck(groupIndex, i)}
6. onRemove={onTodoRemove(groupIndex, i)}
7. groupIndex={groupIndex}
8. todoIndex={i}
9. />
10. ))}

todos jest tablicą obiektów, które reprezentują pojedyncze zadanie. Funkcja map() z interfejsu tablicy pozwala na zamianę każdego elementu z kolekcji w dowolny sposób. W taki sposób jest tworzona tablica komponentów w ReactJS. Każdy obiekt, który reprezentuje pojedyncze zadanie, zamieniany jest w odpowiedni sposób na komponent Todo. Dodatkowo przy zastosowaniu tej techniki bardzo ważne jest aby do każdego komponentu przypisać wartość klucza. Klucze pomagają Reactowi identyfikować zmiany w komponentach, dlatego bardzo ważnym jest to, żeby zatroszczyć się oto aby te wartości były unikalne.

Następnym użytecznym skrótem programistycznym użytym wyżej jest zastosowanie wyrażenia {…todo}. Operator ten pozwala wypakować pola obiektu todo. Wyrażenie to jest równe:

1. <Todo
2. key={todo.name}
3. name={todo.name}
4. done={todo.done}
5. onCheck={onCheck(groupIndex, i)}
6. onRemove={onTodoRemove(groupIndex, i)}
7. groupIndex={groupIndex}
8. todoIndex={i}
9. />

Jak można zauważyć, dzięki ES6, kod kolejny raz stał się krótszy i czytelniejszy.

Nadrzędnym komponentem w hierarchii jest komponent, który opakowuje całą aplikację:

1. **export** **const** App = () => {
2. **const** [groups, setGroups] = useState(mock)
3. **const** onCheck = (groupIndex, todoIndex) => (done) =>  …
4. **const** onTodoRemove = (groupIndex, todoIndex) => () =>  …
5. **const** onTodoAdd = (groupIndex) => (todo) => …
6. **const** onRemove = (groupIndex) => () =>  …
7. **return** (
8. <>
9. <h1>Do zrobienia</h1>
10. <div className={'todo-group-wrapper'}>
11. {groups.map((group, i) => (
12. <TodoGroup
13. key={group.name}
14. {...group}
15. onCheck={onCheck}
16. onTodoRemove={onTodoRemove}
17. onRemove={onRemove(i)}
18. onTodoAdd={onTodoAdd(i)}
19. groupIndex={i}
20. />
21. ))}
22. <AddNewTodo onAdd={(group) => setGroups([...groups, group])} />
23. </div>
24. </>
25. )
26. }

Komponent ten przetrzymuje stan wszystkich grup zadań oraz posiada funkcję, które są przekazywane do odpowiednich komponentów. Jego zadaniem jest wyświetlenie wszystkich grup zadań oraz komponentu który służy do dodawania nowej grupy zadań.

Stan w Reacie nie powinien być modyfikowany bezpośrednio. Powinno się to wykonywać za pomocą odpowiednich metod. Z tego też powodu modyfikacja stanu, który jest mocno zagnieżdżony staje się trudna do realizacji. Przykładowo kod funkcji zmieniający status pojedynczego zadania został przedstawiony poniżej:

1. **const** onCheck = (groupIndex, todoIndex) => (done) =>
2. setGroups(
3. groups.map((group, i) =>
4. groupIndex === i
5. ? {
6. ...group,
7. todos: group.todos.map((todo, j) =>
8. todoIndex === j ? { ...todo, done } : todo
9. ),
10. }
11. : group
12. )
13. )

Funkcja onCheck przyjmuje 2 argumenty:

* groupIndex – indeks grupy, w której należy zmienić stan zadania,
* todoIndex – indeks konkretnego zadania, który należy zmienić,

a w rezultacie zwraca kolejną funkcję, która przyjmuje status wykonania danego zadania. Funkcja ta wykonuje jedną prostą rzecz, aczkolwiek kod realizujący to zadanie jest dosyć obszerny i nie do końca czytelny. Jako, że w React stan nie powinien być bezpośrednio modyfikowany użyto funkcji map(), która nie modyfikuje stanu tablicy tylko zwraca nową. Mapowanie grup odbywa się w następujący sposób: jeżeli indeks grupy jest równy aktualnemu indeksowi to następuje modyfikacja tablicy zadań w tej grupie, w innym wypadku zwracana jest ta sama grupa bez zmian. Modyfikacja tablicy zadań jest realizowana w podobny sposób, z użyciem funkcji map(). Jeżeli indeks zadania jest równy indeksowi aktualnego elementu to stan zadania zmieniany jest na ten przekazany z argumentu, w przeciwnym razie zwracane jest zadanie bez zmian.

W powyższym fragmencie została zastosowana kolejna funkcjonalność z EcmaScript 6. Jest to skrócony zapis tworzenia obiektów. Poniżej został przedstawiony przykład:

1. {
2. name: name,
3. age: age
4. }

Utworzony obiekt posiada dwa pola. Należy zwrócić uwagę, że zarówno pole jak i jego wartość mają tą samą nazwę. W takim przypadku można użyć skróconego zapisu:

1. {
2. name, age
3. }

W taki sposób została przypisana wartość pola done w linii 8.

Ostatni komponent jaki został stworzony w ramach aplikacji „Do zrobienia”, to komponent, w którym można stworzyć nową grupę zadań. Jego kod prezentuje się następująco:

1. **export** **const** AddNewTodo = ({ onAdd }) => {
2. **const** [name, setName] = useState('')
3. **const** [todos, setTodos] = useState([])
4. **const** onNameChange = (e) => setName(e.target.value)
5. **const** onTodoChange = (i) => (e) =>
6. setTodos(
7. todos.map((todo, index) => (i === index ? e.target.value : todo))
8. )
9. **const** onNewTodoAdd = () => setTodos([...todos, ''])
10. **const** onGroupAdd = () => {
11. onAdd({
12. name,
13. todos: todos.map(todo => ({
14. name: todo,
15. done: **false**
16. }))
17. });
18. setTodos([]);
19. setName("");
20. }
21. **return** (
22. <div className={'todo-group\_\_body'}>
23. <h3>Dodaj karteczkę!</h3>
24. <input type={'text'} value={name} onChange={onNameChange} placeholder={"Nazwa"}/>
25. {todos.map((todo, i) => (
26. <input
27. key={i}
28. type={'text'}
29. value={todos[i]}
30. placeholder={"Do zrobienia!"}
31. onChange={onTodoChange(i)}
32. />
33. ))}
34. <input
35. className={'btn'}
36. type={'button'}
37. value={'Dodaj nową rzecz!'}
38. onClick={onNewTodoAdd}
39. />
40. <input
41. className={'btn'}
42. type={'button'}
43. value={'Dodaj!'}
44. onClick={onGroupAdd}
45. />
46. </div>
47. )
48. }

Znajduje się tu kilka rzeczy na które warto zwrócić uwagę. Pola w których możemy wpisać nowe zadania są generowane po raz kolejny za pomocą funkcji map(). Początkowo stan tablicy zadań jest pusty, ale po kliknięciu w przycisk „Dodaj nową rzecz!” do tablicy dodawany jest pusty łańcuch znaków. Jest to realizowane w funkcji onNewTodoAdd().

Do obsługi wielu wartości pól tekstowych została użyta funkcjonalność useState(), w której zamiast jednej wartości została umieszczona tablica. Zabieg ten pozwolił na modyfikowanie i przekazywanie dalej wielu elementów. Gdy dane pole się zmienia wywoływana jest funkcja onTodoChange(), do której przekazywany jest odpowiedni indeks pola, a następnie jedynie odpowiadająca mu wartość jest modyfikowana.

Gdy użytkownik kliknie na przycisk „Dodaj!” nowa grupa zadań zostaje dodana. Dzieje się to z wykorzystaniem funkcji dostarczonej z komponentu nadrzędnego, który zarządza stanem aplikacji. Do funkcji tej przekazywany jest obiekt, który posiada pole z nazwą grupy oraz zadania. Jednak zanim zadania zostaną przekazane dalej, muszą zostać zmodyfikowane, ponieważ komponent nadrzędny oczekuje aby pojedyncze zadanie składało się z nazwy oraz jego statusu, jednakże w komponencie występują jedynie same nazwy. Z tego też powodu, każda nazwa jest przemapowana do obiektu w następujący sposób: utworzony zostaje obiekt a do jego pola nazwa przypisywana jest nazwa zadania do zrobienia, a wartość statusu ustawiana jest domyślnie na fałsz. Następnie stan komponentu wraca do ustawień domyślnych i jest gotowy na to aby móc utworzyć kolejną grupę.

# 4.3.1.2 Angular

W odróżnieniu od implementacji używającej biblioteki ReactJS, w Angularze kod HTML oddzielony jest od kodu Typescript, który kontroluje dany komponent. Komponent pojedynczego zadania w tej technologii prezentuje się w następujący sposób.

1. **<div>**
2. **<input** type="checkbox" [(ngModel)]="todo.done" **/>**
3. **<span** [class.done]="todo.done"**>**{{ todo.name }}**</span>**
4. **<input** class="remove" type="button" value="x" (click)="remove()" **/>**
5. **</div>**
6. @Component({
7. selector: 'todo',
8. templateUrl: './todo.component.html',
9. })
10. **export** **class** TodoComponent {
11. @Input()
12. todo: Todo
14. @Output()
15. onRemove = **new** EventEmitter<Todo>()
17. remove = () => **this**.onRemove.emit(**this**.todo)
18. }

W kodzie HTML występuje odmienne podejście do obsługi stanu niż w ReactJS. Stan danej kontrolki jest obsługiwany za pomocą dyrektywy ngModel, która automatycznie łączy wszystkie zmiany wykonane przez użytkownika do modelu. Dużą swobodę w działaniu dostarczana jest przez warunkowe dodawanie klas css. Dzieje się to za pomocą składni [class.nazwaKlasy]=”warunek”. Pozwala to na szybki i czytelny zapis.

Przechodząc do kontrolera komponentu, można zauważyć dekorator @Component, w której programista może umieścić takie informacje jak selektor, po którym można wywołać później komponent oraz miejsce, w którym znajduje się szablon. Dodatkowo znajdują się tu jeszcze 2 inne dekoratory. @Input informuje, że poniższy obiekt jest wejściem komponentu, które jest podawane z zewnątrz. @Output natomiast informuje o tym co wychodzi z danego komponentu czyli jakie zdarzenia mogą być obsługiwane na zewnątrz komponentu. Podejście to jest nazywane programowaniem sterowanym zdarzeniami. Przykład wysłania zdarzenia znajduje się w funkcji remove, która dodatkowo dodaje argument. Na to zdarzenie, może nasłuchiwać rodzic tego komponentu i następnie może podjąć kroki w celu jego obsłużenia. Tak też się dzieje w komponencie grupy zadań do zrobienia, który został przedstawiony poniżej:

1. **<div>**
2. **<div** class="todo-group\_\_header"**>**
3. **<span** class="pointer" (click)="removeGroup()"**>**X**</span>**
4. **</div>**
5. **<div** class="todo-group\_\_body"**>**
6. **<h3>**{{ group.name }}**</h3>**
7. **<todo**
8. \*ngFor="let todo of group.todos"
9. [todo]="todo"
10. (onRemove)="removeTodo($event)"
11. **></todo>**
12. **<input** type="text" placeholder="Nowa rzecz" [(ngModel)]="newTodo" **/>**
13. **<input**
14. class="btn"
15. type="button"
16. value="Dodaj nową rzecz!"
17. (click)="addNewTodo()"
18. **/>**
19. **</div>**
20. **</div>**
21. @Component({
22. selector: 'todo-group',
23. templateUrl: './todo-group.component.html',
24. })
25. **export** **class** TodoGroupComponent {
26. newTodo: string
28. @Input()
29. group: TodoGroup
31. @Output()
32. onGroupRemove = **new** EventEmitter<TodoGroup>()
34. removeTodo = (todo: Todo) =>
35. (**this**.group.todos = **this**.group.todos.filter((el) => el !== todo))
37. removeGroup = () => **this**.onGroupRemove.emit(**this**.group)
39. addNewTodo = () => {
40. **this**.group.todos.push({
41. name: **this**.newTodo,
42. done: **false**,
43. })
44. **this**.newTodo = ''
45. }
46. }

Powyższy komponent posiada jedno wejście i jedno wyjście. Na wejściu przekazywana jest mu grupa zadań, a na wyjściu jest wytwarzane zdarzenie usunięcia grupy.

W Angularze za pomocą podwójnych nawiasów klamrowych dokonywana jest operacja interpolacji, która podmienia zawartość na odpowiadającą mu wartość z modelu danych. Aby wygenerować kolekcje komponentów, należy użyć dyrektywy \*ngFor, która umożliwia dostęp do pojedynczego elementu kolekcji. Zabieg ten został zastosowany w powyższym przykładzie, gdzie pojedyncze zadania są tworzone na podstawie tablicy obiektów. Każdy obiekt pojedynczego zadania jest przekazywany do odpowiadającego mu komponentu, który został opisany wcześniej.

Wartym zaznaczenia jest również sposób w jakim zostało zrealizowane nasłuchiwanie na zdarzenie usunięcia pojedynczego zadania, które jest wytwarzane przez komponent potomny. Do nasłuchiwanie na zdarzenie jest użyte metoda onRemove, która wykona się za każdym razem kiedy nastąpi to wydarzenie. Metoda ta następnie przefiltruje wszystkie pojedyncze zadania z grupy i usunie te, które zostało przesłane w parametrze wydarzenia.

Funkcja dodawania zadania do istniejącej grupy jest realizowana w bardzo prosty sposób, poprzez użycie metody push() z interfejsu tablicy, która doda nowy element. Zmiana ta zostanie zauważona i dzięki wcześniej opisanej dyrektywie \*ngFor zostanie od razu dodana do interfejsu użytkownika.

Najwyższym w hierarchii komponentów, jest kontener który zarządza stanem całej aplikacji. Jego kod prezentuje się w następujący sposób:

1. **<h1>**Do zrobienia**</h1>**
2. **<div** class="todo-group-wrapper"**>**
3. **<todo-group**
4. \*ngFor="let group of groups"
5. [group]="group"
6. (onGroupRemove)="removeGroup($event)"
7. **></todo-group>**
8. **<add-new-todo** (onGroupAdd)="addGroup($event)"**></add-new-todo>**
9. **</div>**
10. @Component({
11. selector: 'app-root',
12. templateUrl: './app.component.html',
13. })
14. **export** **class** AppComponent {
15. groups: TodoGroup[] = [
16. {
17. name: 'Lista zakupów',
18. todos: [
19. {
20. name: 'Mydło',
21. done: **true**,
22. },
23. {
24. name: 'Szampon',
25. done: **false**,
26. },
27. ],
28. },
29. {
30. name: 'Po szkole',
31. todos: [
32. {
33. name: 'Lekcje',
34. done: **true**,
35. },
36. {
37. name: 'Sprzątanie',
38. done: **false**,
39. },
40. ],
41. },
42. ]
44. removeGroup = (group: TodoGroup) =>
45. (**this**.groups = **this**.groups.filter((el) => el !== group))
47. addGroup = (group: TodoGroup) => **this**.groups.push(group)
48. }

Komponent ten za pomocą dyrektywy \*ngFor wyświetla wszystkie grupy zadań, a także przekazuje do odpowiedzialnego komponentu wymagane wejście oraz nasłuchuje na zdarzenie usunięcia grupy. Dodatkowo zawiera on jeszcze jeden komponent, który pozwala na dodanie nowej grupy.

Komponent ten ma bardzo prostą budowę. Posiada on stan, który jest kolekcją grup zadań. Kolekcja ta jest w odpowiedni sposób propagowana, w dół komponentów przez odpowiednie wejścia, a zmiany w stanie są obsługiwane przez funkcje na różnych poziomach hierarchii. Dodatkowo komponent posiada 2 metody do zarządzania grupami. Jedna z nich nasłuchuje na zdarzenie usunięcia grupy i obsługuje je w podobny sposób do tego jak obsługiwane jest usunięcie pojedynczego zadania. Druga metoda obsługuje zdarzenie dodania nowej grupy, która korzysta z argumentu grupy przesłanego w zdarzeniu i dodaje go do istniejących grup.

Ostatnim komponentem jest wspomniany wyżej, komponent odpowiedzialny za dodanie nowej grupy a jego kod został przedstawiony poniżej:

1. **<div** class="todo-group\_\_body"**>**
2. **<h3>**Dodaj karteczkę!**</h3>**
3. **<input** type="text" placeholder="Nazwa" [(ngModel)]="group.name" **/>**
4. **<input**
5. \*ngFor="let todo of group.todos"
6. type="text"
7. placeholder="Do zrobienia!"
8. [(ngModel)]="todo.name"
9. **/>**
10. **<input**
11. class="btn"
12. type="button"
13. value="Dodaj nową rzecz!"
14. (click)="addTodo()"
15. **/>**
16. **<input** class="btn" type="button" value="Dodaj!" (click)="addGroup()" **/>**
17. **</div>**
18. @Component({
19. selector: 'add-new-todo',
20. templateUrl: './add-new-todo.component.html',
21. })
22. **export** **class** AddNewTodoComponent {
23. group: TodoGroup = {
24. todos: [],
25. name: '',
26. }
28. @Output()
29. onGroupAdd = **new** EventEmitter<TodoGroup>()
31. addTodo = () =>
32. **this**.group.todos.push({
33. name: '',
34. done: **false**,
35. })
37. addGroup = () => {
38. **this**.onGroupAdd.emit(**this**.group)
39. **this**.group = {
40. todos: [],
41. name: '',
42. }
43. }
44. }

Powyższy komponent nie ma żadnego wejścia, ale za to posiada jedno wyjście. Jest to zdarzenie powstania nowej grupy.

Komponent ten nie różni się specjalnie od poprzednio opisywanych poza jedną bardzo użyteczną funkcjonalnością. Połączenie dyrektywy ngFor oraz ngModel sprawia, że pola tekstowe są generowane w pętli a ich stan jest modyfikowany za pomocą dwustronnego połączenia. Warto również zwrócić uwagę na metodę addTodo, która dodaje nowe zadanie do tablicy zadań, co spowoduje, że wcześniej opisana dyrektywa ngFor wygeneruje nowe pole, które zostanie obsłużone za pomocą ngModel. Dodatkowo znajduje się przycisk, który po kliknięciu wygeneruje zdarzenie dodania nowej grupy, a następnie ustawi stan komponentu na domyślny, tak aby można było od początku dodać nową grupę.

Do poprawnego działania Angular potrzebuje jeszcze jednego pliku. Plik ten nosi nazwę modułu a jego kod prezentuje się w następujący sposób:

1. @NgModule({
2. declarations: [
3. AppComponent,
4. TodoGroupComponent,
5. AddNewTodoComponent,
6. TodoComponent,
7. ],
8. imports: [
9. BrowserModule,
10. FormsModule,
11. ],
12. providers: [],
13. bootstrap: [AppComponent]
14. })
15. **export** **class** AppModule { }

Dekorator @NgModule to funkcja, która przyjmuje obiekt z metadanymi, które opisują ten moduł. Najważniejsze właściwości modułu to:

* declarations – komponenty, które należą do modułu,
* exports – deklaracje, które powinny być widoczne na zewnątrz modułu,
* imports – inne moduły, których wyeksportowane klasy są potrzebne w szablonach tego modułu,
* providers – serwisy, które są wytworzone przez ten moduł i są dostępne przez we wszystkich miejscach w aplikacji,
* bootstrap – główny komponent modułu.

# 4.3.1.3 Vue.js

Ostatnia implementacja została zrealizowana przy pomocy frameworka Vue.js. W odróżnieniu od dwóch poprzednich bibliotek, framework ten dostarcza własne rozszerzenie plików .vue. Sam układ kodu w pliku też jest unikalny. Poniżej przedstawiony został kod komponentu pojedynczego zadania.

1. <template>
2. <div>
3. <input type="checkbox" :checked="todo.done" @change="changeTodo" />
4. <span
5. :**class**="{
6. done: todo.done,
7. }"
8. >{{ todo.name }}</span
9. >
10. <input **class**="remove" type="button" value="x" @click="removeTodo" />
11. </div>
12. </template>
14. <script>
15. **export** **default** {
16. name: 'Todo',
17. props: {
18. todo: Object,
19. onTodoChange: Function,
20. onTodoRemove: Function,
21. },
22. methods: {
23. changeTodo: **function** () {
24. **this**.$emit('on-todo-change', **this**.todo, !**this**.todo.done)
25. },
26. removeTodo: **function** () {
27. **this**.$emit('on-todo-remove', **this**.todo)
28. },
29. },
30. }
31. </script>

W pierwszej części pliku występuje kod HTML, który znajduje się pomiędzy znacznikami template. Następnie znajduje się kod Javascript, który występuje pomiędzy znacznikami script. Dodatkowo mogą również wystąpić znaczniki style, który odpowiedzialne by były za ostylowanie za pomocą CSS.

Obsługa stanu pojedynczego zadania jest podobna do React. Do pola wyboru dostarczana jest wartość poprzez atrybut :checked a na zmianę nasłuchuje metoda, która jest oznaczona przez @change. Dyrektywy :[atrybut] i @[zdarzenie] jest są skrótami, a ich pełne rozwinięcie to v-bind:[atrybut] i v-on:[zdarzenie].

W sekcji odpowiedzialnej za kod Javascript jest tworzony obiekt Vue. Posiada on takie właściwości jak name, props czy methods. Jak nazwa wskazuje name odpowiada za to jak komponent się nazywa i za pomocą jakiego znacznika może zostać wywołany z innego komponentu. Definicja elementów przekazywanych do komponentu znajduje się w wartości pola props. Jako klucz przekazywana jest nazwa właściwości a jako wartość jej typ. Ostatnim elementem jest pole methods, które definiuje wszystkie funkcje, które dany komponent posiada. W tym przypadku są to 2 metody, które nasłuchują na zdarzenia a następnie wysyłają do komponentu wyżej odpowiednie zdarzenia wraz z danymi, które są z nim związane.

Następnym komponentem, który występuje wyżej w hierarchii jest komponent grupy zadań a jego kod został przedstawiony poniżej.

1. <template>
2. <div>
3. <div **class**="todo-group\_\_header">
4. <span **class**="pointer">X</span>
5. </div>
6. <div **class**="todo-group\_\_body">
7. <h3>{{ group.name }}</h3>
8. <Todo
9. v-**for**="todo in group.todos"
10. :todo="todo"
11. :key="todo.name"
12. @on-todo-change="changeTodo"
13. @on-todo-remove="removeTodo"
14. />
15. <input
16. type="text"
17. placeholder="Nowa rzecz"
18. v-model="newTodo.name"
19. />
20. <input
21. **class**="btn"
22. type="button"
23. value="Dodaj nową rzecz!"
24. @click="addNewTodo"
25. />
26. </div>
27. </div>
28. </template>
30. <script>
31. **export** **default** {
32. name: 'TodoGroup',
33. components: { Todo },
34. data: () => ({
35. newTodo: { name: '', done: **false** },
36. }),
37. props: {
38. group: Object,
39. onNewTodoAdd: Function,
40. onTodoChange: Function,
41. onTodoRemove: Function,
42. },
43. methods: {
44. addNewTodo: **function** () {
45. **this**.$emit('on-new-todo-add', **this**.group, **this**.newTodo)
46. **this**.newTodo = { name: '', done: **false** }
47. },
48. changeTodo: **function**(todo, done) {
49. **this**.$emit('on-todo-change', **this**.group, todo, done)
50. },
51. removeTodo: **function**(todo) {
52. **this**.$emit('on-todo-remove', **this**.group, todo)
53. }
54. },
55. }
56. </script>

Podobnie jak Angularze, interpolacja w Vue odbywa się za pomocą podwójnych nawiasów klamrowych. Kolejną podobną rzeczą jest użyta dyrektywa v-model, która działa na zasadzie podwójnego wiązania. Aby utworzyć kolekcję komponentów, w Vue używana jest dyrektywa v-for, dzięki której jest możliwość dostania się do pojedynczego obiektu z tablicy.

Komponent grupy używa komponentu pojedynczego zadania, który został użyty wcześniej. Chęć użycia takiego komponentu, w szablonie HTML deklarowany jest w wartości components w obiekcie Vue. Do komponentu pojedynczego zadania przekazywane są 2 funkcje, które nasłuchują na zdarzenia wywoływane przez ten komponent. Stan komponentu jest zapisany jako funkcja, która jest przypisywana do pola data.

Kolejnym komponentem jest komponent, który jest kontenerem dla reszty. Jego kod został przedstawiony poniżej.

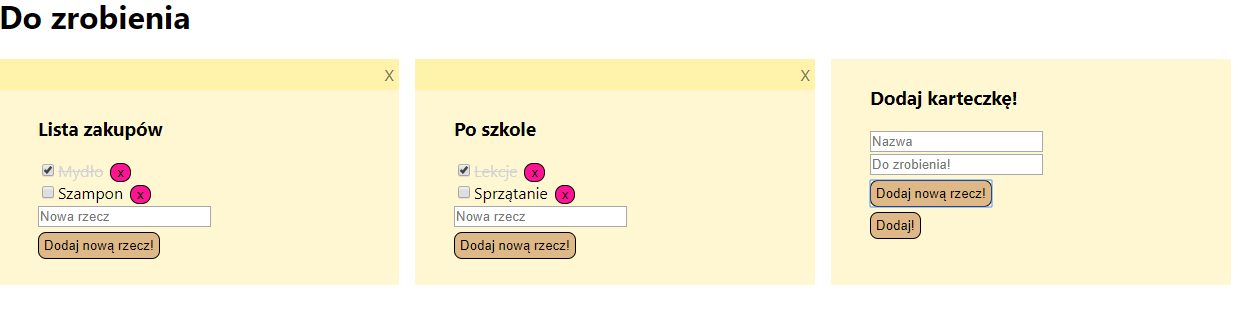
1. <template>
2. <div>
3. <h1>Do zrobienia</h1>
4. <div **class**="todo-group-wrapper">
5. <TodoGroup
6. v-**for**="group in groups"
7. :group="group"
8. :key="group.name"
9. @on-**new**-todo-add="addNewTodo"
10. @on-todo-change="changeTodo"
11. @on-todo-remove="removeTodo"
12. />
13. <AddNewTodo @on-group-add="addGroup" />
14. </div>
15. </div>
16. </template>
18. <script>
19. **export** **default** {
20. name: 'App',
21. components: { AddNewTodo, TodoGroup },
22. data: () => ({
23. groups: [...],
24. }),
25. methods: {
26. addGroup: **function** (group) {
27. **this**.groups.push(group)
28. },
29. addNewTodo: **function** (group, todo) {
30. **this**.groups = **this**.groups.map((gr) =>
31. gr === group ? { ...gr, todos: [...gr.todos, todo] } : gr
32. )
33. },
34. changeTodo: **function** (group, todo, done) {
35. **this**.groups = **this**.groups.map((gr) =>
36. gr === group
37. ? {
38. ...gr,
39. todos: gr.todos.map((td) =>
40. td === todo ? { ...td, done } : td
41. ),
42. }
43. : gr
44. )
45. },
46. removeTodo: **function** (group, todo) {
47. **this**.groups = **this**.groups.map((gr) =>
48. gr === group
49. ? { ...gr, todos: gr.todos.filter((td) => td !== todo) }
50. : gr
51. )
52. },
53. },
54. }
55. </script>

Komponent ten używa komponentu grupy, aby wyświetlać kolekcję grup zadań, przekazując do niego odpowiednie właściwości oraz metody nasłuchujące. W komponencie tym znajduje się również stan aplikacji, który jest propagowany w dół hierarchii komponentów. Dodatkowo posiada on również metody, które na tym stanie operują. Są one dosyć zawiłe ze względu na to, że stan w Vue nie powinien być mutowany. Taką samą własność posiada biblioteka ReactJS.

Ostatnim komponentem stworzonym w ramach realizacji implementacji w frameworku Vue.js jest komponent, który pozwala na dodanie pojedynczego zadania.

1. <template>
2. <div **class**="todo-group\_\_body">
3. <h3>Dodaj karteczkę!</h3>
4. <input type="text" placeholder="Nazwa" v-model="group.name" />
5. <input
6. type="text"
7. placeholder="Do zrobienia!"
8. v-**for**="(todo, i) in group.todos"
9. v-model="todo.name"
10. :key="i"
11. />
12. <input
13. **class**="btn"
14. type="button"
15. value="Dodaj nową rzecz!"
16. @click="addNewTodo"
17. />
18. <input **class**="btn" type="button" value="Dodaj!" @click="addGroup"/>
19. </div>
20. </template>
22. <script>
23. **export** **default** {
24. name: 'AddNewTodo',
25. data: () => ({
26. group: {
27. name: '',
28. todos: [],
29. },
30. }),
31. methods: {
32. addNewTodo: **function** () {
33. **this**.group.todos.push({ name: '', done: **false** })
34. },
35. addGroup: **function**() {
36. **this**.$emit('on-group-add', **this**.group)
37. **this**.group = {
38. name: '',
39. todos: []
40. }
41. }
42. },
43. props: {
44. onGroupAdd: Function
45. }
46. }
47. </script>

Podobnie jak w frameworku Angular została tutaj użyta technika, która pozwala na połączenie dyrektyw v-for oraz v-model. Dzięki temu istnieje możliwość podwójnego łączenia danych podczas wyświetlania kolekcji danych. Komponent ten przyjmuje tylko jedną właściwość, którą jest funkcja nasłuchująca na dodanie nowej grupy.

We wszystkich wyżej opisanych implementacjach aplikacja działa w taki sam sposób, a jej wygląd został przedstawiony na obrazie poniżej.

# 4.3.2 Benchmark Item

Następną aplikacją, która została zaimplementowana we wszystkich badanych aplikacjach jest „Benchmark Item”. Jest to aplikacja, którym głównym zadaniem jest zarządzanie dużą ilością komponentów. Dzięki implementacji aplikacji, w każdym z 3 frameworków, będzie istniała możliwość przetestowania aplikacji pod kątem wydajnościowym i obciążeniowym.

Aplikacja w każdej z 3 implementacji będzie składać się z 2 komponentów. Testowy komponent, który jako wejście będzie przyjmował następująco strukturę:

1. **interface** Obj {
2. id: number
3. str: string
4. }

gdzie:

* id – atrybut id dla komponentu,
* str – łańcuch znaków wyświetlany przez komponent.

Komponent ten posiada również swój własny stan, który decyduje o tym czy komponent powinien zostać wyróżniony (wymaganie co do aplikacji).

Drugim komponentem jest komponent, który zarządza stanem aplikacji, oraz dostarcza funkcjonalności modyfikacji komponentów.

# 4.3.2.1 ReactJS

Implementacja komponentu testowego prezentuje się w następujący sposób.

1. **const** BenchmarkItem = ({ obj }) => {
2. **const** [highlight, setHighlight] = useState(**false**)
3. **return** (
4. <div
5. onClick={() => setHighlight(**true**)}
6. className={highlight ? 'highlight' : ''}
7. id={obj.id}
8. >
9. {obj.str}
10. </div>
11. )
12. }
14. **export** **default** BenchmarkItem

Początkowo stan komponentu(informacja o tym czy komponent ma zostać wyróżniony) jest ustawiana na fałsz, natomiast podczas kliknięcia na komponent informacja ta jest zmieniana naprawdę. Gdy stan zostanie zmieniony, na komponencie ustawia się również nowa klasa CSS. Każdy komponent posiada unikalne id oraz wartość tekstową, która jest wyświetlana.

Części kodu komponentu zewnętrznego zostały zaprezentowany poniżej:

1. **const** [state, setState] = useState([])
3. **const** onCreateClick = () =>
4. setState(range(10\_000).map((num) => ({ id: num, str: String(num) })))
6. **const** onUpdateClick = () =>
7. setState(state.map((s) => ({ ...s, str: s.str + '!' })))
9. **const** onUpdate10thClick = () =>
10. setState(
11. state.map((s) => (s.id % 10 === 0 ? { ...s, str: s.str + '!' } : s))
12. )
14. **const** onSwapClick = () => {
15. **const** first = state[2];
16. **const** second = state[9997]
17. setState(
18. state.map((s ,i) => {
19. **if** (i === 2) {
20. **return** second;
21. } **else** **if** (i === 9997) {
22. **return** first;
23. }
24. **return** s;
25. })
26. )
27. }
29. **const** onClearClick = () => setState([])

Początkowo stan aplikacji reprezentowany jest przez pustą tablicę. Funkcja onCreateClick pozwala na zapisanie w stanie 10.000 elementów. Funkcja onUpdateClick pozwala na aktualizacje wszystkich komponentów, poprzez dodanie do łańcucha znaków pojedynczego znaku wykrzyknika. Następna funkcja robi dokładnie to samo tylko dla co 10 elementu. Funkcja odpowiedzialna za zamianę dwóch elementów, najpierw pobiera 2 elementy a następnie w procesie mapowania zamienia je miejscami. React wymaga aby stan nie był modyfikowany, zatem operacja ta musi się odbyć z pomocą metody map() z interfejsu tablicy. Ostatnia funkcja pozwala wyczyścić istniejące komponenty.

Poniżej zaprezentowano w jaki sposób utworzono przyciski do wyzwalania powyższych metod:

1. <input
2. id={'create/update/update10th/swap/clear'}
3. type={'button'}
4. onClick={onCreateClick/onUpdateClick/onUpdate10thClick/onSwapClick/onClearClick}
5. value={'Opis wykonywanego zadania'}
6. />

Pole wejścia składa się z 4 elementów:

* id – unikalne dla aplikacji, pozwalają na pobranie elementów przez silnik testujący rozwiązanie (id jest unikalne w ramach jednej aplikacji, ale nazwy są wspólne dla wszystkich implementacji),
* type – rodzaj wejścia – zawsze przycisk,
* onClick – metoda która ma zostać wyzwolona po kliknięciu na przycisk,
* value – słowny opis wykonywanej funkcji.

Powyższe pole zostało utworzone 5 razy – dla każdej z funkcjonalności.

Ostatnim istotnym miejscem w aplikacji jest wyświetlanie stanu poprzez jego reprezentację w komponentach testowych.

1. <div className={'wrapper'}>
2. {state.map((s) => (
3. <BenchmarkItem obj={s} key={s.id} />
4. ))}
5. {state.length === 0 && <div id={'empty'}>Pusty</div>}
6. </div>

Każdy element stanu zostaje przekazany do komponentu testowego, który potrafi go wyświetlić. Jeżeli w stanie nie znajduje się żaden element, to jest wyświetlana informacja że stan jest pusty.

# 4.3.2.2 Angular

Kod komponentu testowego w Angularze prezentuje się w następujący sposób:

1. **<div** (click)="onClick()" [class.highlight]="highlight" [id]="obj.id"**>**
2. {{ obj.str }}
3. **</div>**

Po kliknięciu w komponent, wywoływana jest funkcja onClick. Klasa highlight nakładana jest jedynie jeżeli pole highlight w komponencie jest prawdziwe. Do otaczającego bloku przekazywany jest atrybut id, a wartość łańcucha tekstowego jest wyświetlana.

1. **export** **class** BenchmarkItemComponent {
2. highlight: **boolean** = **false**
4. @Input()
5. obj: Obj;
7. onClick() {
8. **this**.highlight = **true**
9. }
10. }

Stan komponentu początkowo jest fałszem, natomiast metoda onClick zmienia jego stan naprawdę. Jako wejście komponent przyjmuje opisany wcześniej obiekt.

Kontroler stanu nadrzędnego wygląda następująco:

1. **export** **class** AppComponent {
2. state: Obj[] = []
4. onCreateClick() {
5. **this**.state = range(10\_000).map((num) => ({ id: num, str: String(num) }))
6. }
8. onUpdateClick() {
9. **for** (let i = 0; i < 10000; i++) {
10. **this**.state[i].str += '!'
11. }
12. }
14. onUpdate10thClick() {
15. **for** (let i = 0; i < 10000; i += 10) {
16. **this**.state[i].str += '!'
17. }
18. }
20. onSwapClick() {
21. **const** first = **this**.state[2]
22. **const** second = **this**.state[9997]
23. **this**.state[9997] = first
24. **this**.state[2] = second
25. }
27. onClearClick() {
28. **this**.state = []
29. }
30. }

Posiada on dokładnie taki sam interfejs jak w implementacji Reactowej. W odróżnieniu od Reacta, w Angularze stan jest mutowalny zatem powyższe funkcję są prostsze i bardziej czytelne. Szablon HTML powyższego komponentu jest taki sam jak w implementacji Reacta (poza zmianami semantycznymi narzucanymi przez framework).

# 4.3.2.3 Vue.js

Implementacja w języku Vue.js jest bardzo zbliżona do dwóch poprzednich. Komponent testowy jest reprezentowany poprzez poniższy kod:

1. <template>
2. <div @click="highlight = true" :**class**="{ highlight }" :id="obj.id">
3. {{ obj.str }}
4. </div>
5. </template>
7. <script>
8. **export** **default** {
9. name: 'BenchmarkItem',
10. data: () => ({
11. highlight: **false**,
12. }),
13. props: {
14. obj: Object,
15. },
16. }
17. </script>

Podobnie jak we wcześniejszych implementacjach, komponent posiada domyślny stan, który jest modyfikowany poprzez kliknięcie w komponent. Operacja to powoduje nałożenie klasy highlight. Do bloku otaczającego przekazywane jest id z obiektu wejściowego, a łańcuch tekstowy jest przez komponent wyświetlany.

Komponent nadrzędny prezentuje się w następujący sposób:

1. **export** **default** {
2. name: 'App',
3. components: { BenchmarkItem },
4. data: () => ({
5. state: [],
6. }),
7. methods: {
8. onCreateClick: **function** () {
9. **this**.state = range(10000).map((num) => ({
10. id: num,
11. str: String(num),
12. }))
13. },
14. onUpdateClick: **function** () {
15. **this**.state = **this**.state.map((s) => ({ ...s, str: s.str + '!' }))
16. },
17. onUpdate10thClick: **function** () {
18. **this**.state = **this**.state.map((s) =>
19. s.id % 10 === 0 ? { ...s, str: s.str + '!' } : s
20. )
21. },
22. onSwapClick: **function** () {
23. **const** first = **this**.state[2]
24. **const** second = **this**.state[9997]
25. **this**.state = **this**.state.map((s, i) => {
26. **if** (i === 2) {
27. **return** second
28. } **else** **if** (i === 9997) {
29. **return** first
30. }
31. **return** s
32. })
33. },
34. onClearClick: **function** () {
35. **this**.state = []
36. },
37. },
38. }

Z racji tego, że tak samo jak w ReactJS we Vue.js nie należy modyfikować stanu funkcje te posiadają taki sam kod jak w przypadku React. Kod HTML jest również bliźniaczo podobny do dwóch poprzednich rozwiązań.

# 4.3.3 Benchmark Engine

Najważniejszym zadaniem systemu „Benchmark Engine” jest możliwość automatycznego i niezależnego przetestowania zaimplementowanych aplikacji pod kątem ich wydajności. Istotnym elementem tego systemu jest to, że potrafi on zmierzyć czas wykonywanych operacji oraz wykonać określoną ilość iteracji tak aby wynik był uśredniony i pozbawiony błędów.

Aplikacja została napisana przy użyciu języka Javascript na platformie Node.js. Kod głównej metody prezentuje się w następujący sposób:

1. **const** main = async () => {
2. let driver = await **new** Builder().forBrowser('chrome').build()
3. await benchmarkApp(driver, 'react', 'http://localhost:3000')
4. await benchmarkApp(driver, 'vue', 'http://localhost:8080')
5. await benchmarkApp(driver, 'angular', 'http://localhost:4200')
6. }

Na początku jest tworzony sterownik przeglądarki, który będzie emitował zachowanie użytkownika. Sterownik ten pochodzi z oprogramowania Selenium. Następnie trzykrotnie jest wywoływana funkcja, która wykona test dla każdej z implementacji. Argumentami tej funkcji są sterownik, nazwa aplikacji, która posłuży jako etykieta do zapisanego pliku oraz adres, pod którym znajduje się aplikacja.

1. **const** benchmarkApp = async (driver, label, address) => {
2. **const** benchResults = {
3. create: await createBenchmark(driver, address),
4. update: await updateBenchmark(driver, address),
5. update10th: await update10thBenchmark(driver, address),
6. emphasize: await emphasizeBenchmark(driver, address),
7. swap: await swapBenchmark(driver, address),
8. clear: await clearBenchmark(driver, address)
9. }
10. await saveToFile(benchResults, label)
11. }

Funkcja benchmarkApp, tworzy obiekt, w którym zapisane będą wyniki wszystkich testów dla danego frameworka. Obiekt ten zawiera rezultaty z 6 funkcji. Każda z nich testuje jedną funkcjonalność wytworzonej implementacji. Do każdej z funkcji przekazywany jest sterownik oraz adres aplikacji. Gdy wszystkie dane zostaną zebranie, zapisywane są do formatu JSON za pomocą funkcji saveToFile, która przyjmuje wyniki testów oraz nazwę aplikacji.

Pierwszą opisywaną funkcją, jest funkcja testująca tworzenie 10.000 elementów.

1. **const** createBenchmark = async (driver, address) => {
2. await driver.get(address)
3. **const** times = []
4. **for** (let i = 0; i < ITERATIONS; i++) {
5. **const** start = Date.now()
6. await driver.findElement(By.id('create')).click()
7. await driver.findElement(By.id('9999'))
8. **const** stop = Date.now()
9. times.push(stop - start)
10. await driver.findElement(By.id('clear')).click()
11. await driver.findElement(By.id('empty'))
12. }
13. **return** calculateStats(times)
14. }

Na początku sterownik udaje się pod wskazany adres oraz inicjalizowana jest pusta tablica z czasami. Następnie dla zadanej ilości iteracji wykonywany jest test. Pojedynczy test zaczyna się od pobrania aktualnego znacznika czasowego, który służy jako odnośnik czasowy wystartowanego testu. Poprzez sterownik przeglądarki, pobierany jest element „create”, którym we wszystkich aplikacjach jest przycisk odpowiedzialny za stworzenie 10.000 elementów. Następnie znajdowany jest ostatni komponent po jego id. Wyświetlając 10.000 komponentów ostatni komponent posiada id=9999, ponieważ iteracja rozpoczyna się od 0. Gdy element zostanie znaleziony tworzony jest kolejny znacznik czasowy a następnie do tablicy z czasami dodawana jest różnica pomiędzy stopem a startem wyrażona w milisekundach. Ostatnim elementem jest naciśnięcie przez sterownik przeglądarki przycisku do wyczyszczenia komponentów, a następnie przeczekanie aż przeglądarka na to zareaguje. Zrealizowane jest to dzięki pobraniu elementu o id „empty”, który jest dostępny jedynie w momencie, gdy na stronie nie ma żadnego testowego komponentu. Ostatnim krokiem, który jest wykonywany przez funkcję to przekazanie zebranych czasów do funkcji, która jest odpowiedzialna za wyliczenia na ich podstawie statystyk. Kod funkcji został przedstawiony oraz opisany poniżej:

1. **const** calculateStats = (times) => ({
2. max: Math.max(...times),
3. min: Math.min(...times),
4. average: times.reduce((acc, num) => acc + num, 0) / times.length,
5. times,
6. })

Funkcja ta zwraca obiekt, który zawiera 4 pola:

* max – wartość maksymalna ze wszystkich czasów,
* min – wartość minimalna ze wszystkich czasów,
* average – wartość średnia wszystkich czasów,
* times – wszystkie zanotowane czasy.

Pierwsze 2 wartości zostały wyznaczone za pomocą funkcji z interfejsu Math. Średnia z czasów została wyznaczona z wykorzystaniem funkcji reduce z interfejsu tablicy. Funkcja reduce przyjmuje funkcję, która posiada 2 elementy. Pierwszym jest akumulator, który z każdym elementem się zmienia. Drugim elementem jest bieżący element. Dla pojedynczego cyklu funkcja zwraca sumę akumulatora i aktualnego elementu. Początkowy stan akumulatora jest zdefiniowany jako zero. Następnie tak wyznaczona suma jest dzielona przez ilość elementów w tablicy, co w rezultacie daje średnią arytmetyczną czasów.

Kolejne funkcje testujące, nie różnią się w sposób znaczący zatem zostaną opisane jedynie różnice.

1. **const** updateBenchmark = async (driver, address) => {
2. await driver.get(address);
3. await driver.findElement(By.id('create')).click()
4. await driver.findElement(By.id('9999'));
5. **const** times = [];
6. **for** (let i = 0; i < ITERATIONS; i++) {
7. **const** start = Date.now()
8. await driver.findElement(By.id('update')).click()
9. await driver.findElement(By.id('9999'))
10. **const** stop = Date.now()
11. times.push(stop - start)
12. }
13. **return** calculateStats(times)
14. }

Powyżej został zapisany kod odpowiedzialny za dokonanie testu aktualizacji wszystkich komponentów. Przed samym wejściem do głównej pętli, jednokrotnie jest tworzone 10.000 komponentów, które są później aktualizowane. Dzieje się to poprzez naciśnięcia przez robot sterujący przycisku o id „update” a następnie znalezienie komponentu z id 9999. Warto zauważyć, że po całej operacji komponenty nie są kasowane, ponieważ w tym teście oceniana jest wydajność aktualizacji komponentów, zatem nie ma potrzeby usuwania po każdej iteracji.

Następną bardzo podobną funkcją do poprzedniej jest aktualizacja co dziesiątego komponentu.

1. **const** update10thBenchmark = async (driver, address) => {
2. await driver.get(address);
3. await driver.findElement(By.id('create')).click()
4. await driver.findElement(By.id('9999'));
5. **const** times = [];
6. **for** (let i = 0; i < ITERATIONS; i++) {
7. **const** start = Date.now()
8. await driver.findElement(By.id('update10th')).click()
9. await driver.findElement(By.id('9999'))
10. **const** stop = Date.now()
11. times.push(stop - start)
12. }
13. **return** calculateStats(times)
14. }

Jedyną rzeczą jaka jest zmieniona względem poprzedniego przykładu to element, który jest klikany aby rozpocząć operację. Cała reszta pozostała bez zmian.

Opisywana poniżej funkcja odpowiada za wyróżnienie jednego elementu poprzez kliknięcie na niego.

1. **const** emphasizeBenchmark = async (driver, address) => {
2. await driver.get(address);
3. **const** times = [];
4. **for** (let i = 0; i < ITERATIONS; i++) {
5. await driver.findElement(By.id('create')).click()
6. await driver.findElement(By.id('9999'));
7. **const** start = Date.now()
8. await driver.findElement(By.id('5000')).click()
9. await driver.findElement(By.className('highlight'))
10. **const** stop = Date.now()
11. times.push(stop - start)
12. await driver.findElement(By.id('clear')).click()
13. await driver.findElement(By.id('empty'))
14. }
15. **return** calculateStats(times)
16. }

Na początku każdej pętli tworzone jest 10.000 elementów. Następnie wyszukiwany jest element ze środka wszystkich komponentów a następnie jest kliknięty. Następnie robot testujący szuka elementu, dla którego została nadana odpowiednia klasa CSS. Gdy element zostanie znaleziony, zostaje zapisywany czas wykonania, a następnie tablica komponentów zostaje wyczyszczona. Warto zaznaczyć, że w powyższym teście mierzony jest jedynie czas wyróżnienia jednego komponentu, wszystkie operacje, które przygotowują test nie są brane pod uwagę.

Implementacja testu zamiany dwóch komponentów choć w działaniu inna, jest bliźniaczo podobna do poprzednio opisywanej akcji.

1. **const** swapBenchmark = async (driver, address) => {
2. await driver.get(address);
3. **const** times = [];
4. **for** (let i = 0; i < ITERATIONS; i++) {
5. await driver.findElement(By.id('create')).click()
6. await driver.findElement(By.id('9999'));
7. **const** start = Date.now()
8. await driver.findElement(By.id('swap')).click()
9. await driver.findElement(By.id('9997'))
10. **const** stop = Date.now()
11. times.push(stop - start)
12. await driver.findElement(By.id('clear')).click()
13. await driver.findElement(By.id('empty'))
14. }
15. **return** calculateStats(times)
16. }

Jedyną różnicą jest jaka różnie te dwa testy, to przycisk, który zostaje naciśnięty przez robot oraz element, który jest szukany po naciśnięciu. Reszta procesu pozostaje taka sama.

Ostatnią testowaną funkcjonalnością jest czyszczenie komponentów.

1. **const** clearBenchmark = async (driver, address) => {
2. await driver.get(address);
3. **const** times = [];
4. **for** (let i = 0; i < ITERATIONS; i++) {
5. await driver.findElement(By.id('create')).click()
6. await driver.findElement(By.id('9999'));
7. **const** start = Date.now()
8. await driver.findElement(By.id('clear')).click()
9. await driver.findElement(By.id('empty'))
10. **const** stop = Date.now()
11. times.push(stop - start)
12. }
13. **return** calculateStats(times)
14. }

W głównej pętli jest tworzone 10.000 elementów. Po upewnieniu się, że elementu już istnieją startowany jest zegar oraz kliknięty zostaje przycisk, który uruchamia funkcjonalność wyczyszczenia komponentów. Gdy robot upewni się, że tak się stało(istnieje id = „empty”) to program zapisuje czas wykonania.

Wyniki działania programu zapisywane są do plików za pomocą poniższej funkcji:

1. **const** saveToFile = (stats, label) => {
2. fs.writeFileSync(
3. path.join(\_\_dirname, '../results', `${label}-${Date.now()}.json`),
4. JSON.stringify(stats)
5. )
6. }

Statystyki zapisywane są do podfolderu z wynikami w formacie aplikacja-znacznik-czasowy.json.

# 5.Badania

W poniższym rozdziale opisane zostały badania, który zostały zrealizowane w ramach pracy. Znajdują się tutaj pomiary metryk z aplikacji oraz dane statyczne o frameworkach. Następnie opisane we wcześniejszym rozdziale aplikacje zostaną poddane testem obciążeniowym, a zebrane wyniki zostaną zaprezentowane. Ostatnią czynnością dokonaną w ramach tego rozdziału to analiza porównawcza przedstawianych bibliotek.

# 5.1 Pomiar metryk

# Bibliografia

1. Haverbeke M., Eloquent JavaScript, 2018
2. Rauschmayer A., Exploring ES6, Learnpub, 2015
3. <https://goalkicker.com/TypeScriptBook2/>
4. Syed B., TypeScript Deep Dive, ????
5. Freeman A, Pro Angular 6, Third Edition, 2018
6. Murray N., Coury F., Lerner A., Taborda C., ng-book The Complete Guide to Angular, Fullstack.io, 2018
7. Wilken J., Angular in Action, Manning Publications Co., 2018
8. Banks A., Porcello E., Learning React, O`Reilly Media, 2017
9. <https://vuejs.org/v2/guide/>
10. Filipova O., Learning Vue.js 2, Packt Publishing, 2016
11. <https://docs.npmjs.com/about-npm/>
12. <https://yarnpkg.com/getting-started>
13. Chacon S., Straub B., Pro Git, 2020
14. Tsitoara M., Beginning Git and GitHubm, Apress, 2020
15. <https://reactjs.org/docs/lifting-state-up.html>
16. <https://github.com/facebook/react/tree/master/packages/react-reconciler>
17. <https://angular.io/guide/architecture>