React

Konfiguracja środowiska, narzędzia i struktura projektu

infoShare Academy

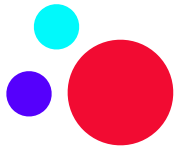
infoShareAcademy.com

HELLO

Jakub Wojtach

Senior **full stack** developer

infoShare ACADEMY



Rozgrzewka i agenda

Zacznijmy ten dzień z przytupem!

Rozgrzewka

Chłopaki nie płaczą

Poranek kojota

Rozgrzewka

Robert Lewandowski

Adam Małysz

Rozgrzewka

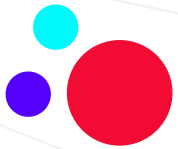
Jogurt naturalny

Kefir

Rozgrzewka

NFT

Obraz Da Vinci

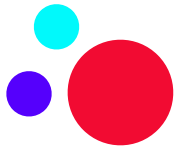


Agenda

- **Podstawy teorii**
- **Podstawy biblioteki React** (wprowadzenie)
- **Narzędzia wykorzystywane w toolingu frontendowym**
- **Wykorzystanie popularnych toolchainów do budowania aplikacji SPA**

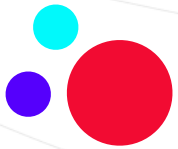
Współpraca

- Zadajemy pytania w dowolnym momencie - kanał **merytoryka**
- Krótkie przerwy (**5 min**) co godzinę
- Długa przerwa (**20 min**) po ostatnim bloku



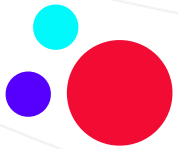
Podstawy teorii

infoShareAcademy.com



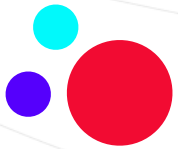
Samodzielne zdobywanie wiedzy

- Dokumentacja React
- Awesome React



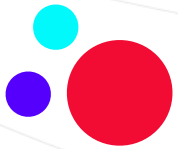
React - jakie problemy rozwiązuje

- React to biblioteka Javascript wykorzystywana przy tworzeniu interfejsów użytkownika dla aplikacji.
- Dostępna na zasadzie open source, opiera się na modularności (komponenty) tworzących większą całość.
- Stworzony przez pracownika Facebook, jedna z najpopularniejszych bibliotek JS
- Wykorzystuje nowoczesny sposób renderowania stron internetowych, pozwala na tworzenie dynamicznych interakcji użytkownika.
- Jest wygodny zarówno dla programisty i jak i dla końcowego użytkownika



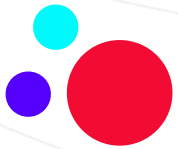
React vs **Vanillajs**

- React rozdziela interfejs użytkownika do mniejszych, reużywalnych komponentów. React ma przewagę nad Vanilla JS właśnie z racji na tę modularność.
- Z racji na to, iż UI zmienia się bardzo często w dużych aplikacjach, kod napisany w VanillaJS który pozwoli na takie zachowanie jest bardzo skomplikowany i ciężki do utrzymania. Aby tego dokonać trzeba najpierw znaleźć element DOM, który chcemy zaktualizować. W przypadku pojedynczego elementu jest to dość proste, ale w przypadku złożonych kolekcji elementów staje się to dość karkołomne.
- Świetnym rozwiązaniem tego problemu jest VirtualDOM stworzony przez React. Aby nie robić wymienionych wyżej procesów ręcznie mechanizm przechowuje obraz prawdziwego DOM. Z jego wykorzystaniem jedynie niezbędne elementy są aktualizowane, bardzo szybko.
- Nowoczesne frameworki stworzone zostały celem rozwiązania problemu, jakim jest reagowanie na zmiany UI wraz ze zmieniającym się stanem aplikacji.



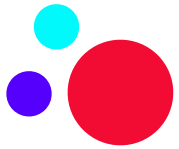
Frameworki, biblioteki - zalety

- **Czas to pieniądz** - klient często nie dba o środki, dla niego ważny jest produkt końcowy w odpowiednim czasie, najlepiej jak najkrótszym, aby płacił jak najmniej
- **Społeczność** - duże frameworki i biblioteki to duża społeczność i duże możliwości na znalezienie odpowiedzi na problem, który akurat rozwiązujemy
- **Standardy** - dzięki standardom nasz kod zyskuje na czytelności, zwłaszcza dla osób z zewnątrz. Gdyby nie standardy kod byłby czytelny tylko dla twórcy, a duże projekty byłyby skazane na porażkę.



Frameworki, biblioteki - problemy i ograniczenia

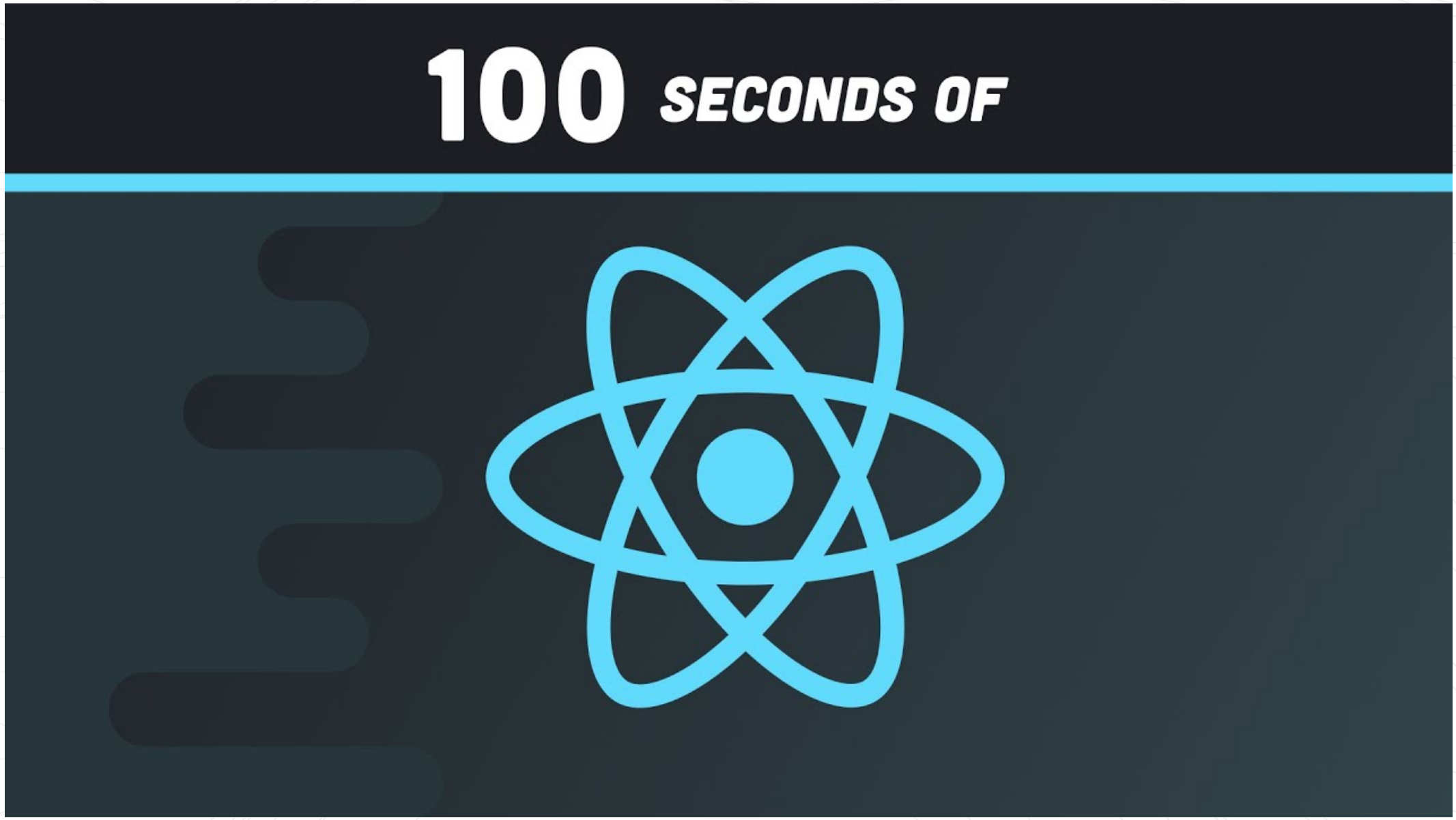
- **Narzucenie konkretnego ekosystemu** - częsty problem, gdy chcemy użyć biblioteki do naszego frameworku, a ta nie jest do niego przystosowana. Coś może nie działać z czymś jak bez tego itd.
- **Bycie na bieżąco** - świat front endu zmienia się bardzo dynamicznie, każdego dnia powstaje nowy framework, standardy zmieniają się bardzo dynamicznie, trendy również. Aby być na bieżąco trzeba poświęcić bardzo dużo czasu, co często prowadzi do wypalenia. Niestety.
- **Próg wejścia** - niestety często ilość wiedzy, jaka jest potrzebna do opanowania danej biblioteki i frameworku to długie godziny z nosem w dokumentacji.

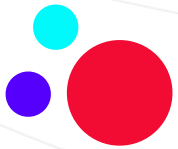


Podstawy biblioteki React

infoShareAcademy.com

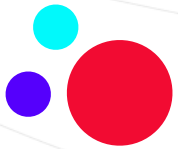
React





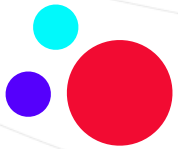
React - zalety

- Szybkość implementacji.
- Dynamiczny interfejs z natychmiastowymi zmianami na stronie.
- **Virtual DOM** - aktualizacja najmniejszych zmian bez wpływania na cały interfejs.
- Modularność
- Jednokierunkowy przepływ danych - co może być zaletą, ale w przypadku trudniejszych przypadków może być wadą.
- Duża społeczność z racji na jego popularność, co daje dużo zmian
- Na jego podstawie powstało wiele pobocznych produktów, jak np. React Native



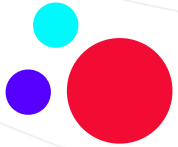
React - wady

- Czasami starszy kod jest nieużywalny wraz z nadchodzącymi zmianami.
- **JSX** - złożony i trudny w obsłudze dla mniej wprawionych programistów.
- **SEO** - dynamika strony niekoniecznie wpływa dobrze na odczytywanie strony przez roboty, można to rozwiązać przez SSR, ale generalnie jest to problematyczne bez dodatkowych środków.



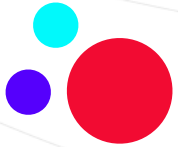
React bez JSX

- JSX nie jest wymagany do używania React
- React bez JSX jest dobrym rozwiązaniem, gdy nie chcemy dokładać kolejnej warstwy komplikującej życie w naszym środowisku developerskim
- Każdy element JSX to tzw. *synthetic sugar* dla wykorzystania komendy **React.createELement(component, props, ...children).**
- Oznacza to nic innego jak fakt, iż wszystko co robimy z użyciem JSX może być wykonane również bez Niego



Kod napisany **w JSX**

```
class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.toWhat}</div>;
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Hello toWhat="World" />);
```

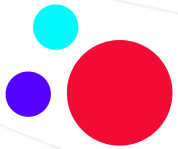


Kod napisany **w JSX**

```jsx
class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.toWhat}</div>;
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Hello toWhat="World" />);
```

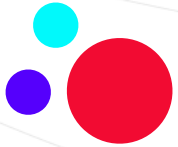
Kod napisany **bez JSX**

```jsx
class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Hello ${this.props.toWhat}`);
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(React.createElement(Hello, {toWhat: 'World'}, null));
```



Komendy top level API

- **React.createElement()**
- **ReactDom.createRoot() i root.render()**



Komponenty

- Komponenty pozwalają podzielić interfejs użytkownika na niezależne, pozwalające na ponowne użycie części i myśleć o każdej z nich osobno.
- Komponenty możemy zdefiniować jako funkcję lub klasę
- Reactowy element stworzony przez developera pozwalający na przyjęcie parametrów, tzw **właściwości.**

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

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

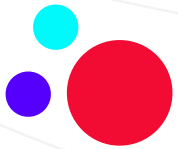
Przykładowy komponent

Kompozycja komponentów - zagnieżdżona struktura

- Komponenty przy zwracaniu wyniku mogą odwoływać się do innych komponentów.
- Przycisk, formularz, okno dialogowe, ekran - w aplikacjach reactowych tego typu składniki są zwykle reprezentowane przez dedykowane komponenty.

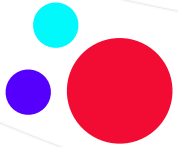
```
function Welcome(props) {
  return <h1>Cześć, {props.name}</h1>;
}

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

Przykład złożonego



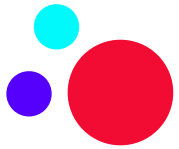
Wyodrębnianie komponentów

- Nie bój się dzielenia komponentów na mniejsze części.
- Nazwy właściwości przekazywanych do komponentu powinny być przekazywać w jego kontekście, a nie kontekstu, w którym jest używany.
- Wyodrębnianie komponentów może z początku wydawać się żmudnym zajęciem, ale posiadanie palety pozwalających na ponowne użycie komponentów jest opłacalne w większych aplikacjach.
- Dobrą praktyczną zasadą jest to, że jeśli część twojego interfejsu użytkownika jest używana wielokrotnie (np. **Button, Panel, Avatar**) lub jest ona dostatecznie skomplikowana sama w sobie (np. **App, FeedStory, Comment**), jest ona dobrym kandydatem do stania się oddzielnym komponentem.



Props

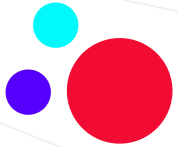
- Stosujemy je do tego, by przekazać dane do komponentu.
- Bez względu na to, czy zadeklarujesz komponent jako **funkcję** czy **klasę**, nie może on **nigdy** modyfikować swoich właściwości (**props**).
- Główna zasada Reacta mówi o tym, że
  Wszytkie komponenty muszą zachowywać się jak czyste funkcje w odniesieniu do ich właściwości



Tooling frontendowy

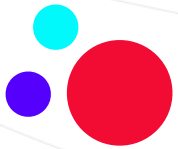
infoShareAcademy.com

infoShare ACADEMY



Babel

- Babel przekształca kod JS do takiej postaci, aby był kompatybilny ze wszystkimi przeglądarkami oraz działał również na starszych wersjach przeglądarek.
- W procesie kompilacji Bąbel zmienia składnię kodu do kompatybilnej postaci.
- Sam w sobie jest **transpilatorem** kodu JavaScript
- Powstał na potrzeby nowej wersji JS ES6 i wyżej.
- Pozwala na wykorzystywanie najnowszych mechanizmów JS, które są następnie tłumaczone na starsze wersje kodu.
- Babel tak naprawdę zajmuje się jedynie transformacją kodu do czytelnej postaci przez przeglądarki. Nie zajmuje się dodawaniem funkcjonalności, które w ogóle nie są wspierane. Za to zadanie odpowiadają skrypty **polyfill**. Jednak sam Babel posiada dodatkowe rozszerzenie i **może** dodawać polyfill do naszego kodu.

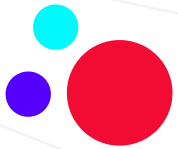


Babel – przykład

```
[1, 2, 3].map((n) => n + 1);
```

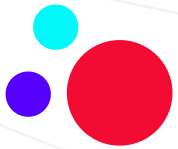
Taka funkcja strzałowa nie jest wspierana przez starsze wersje Internet Explorer, Babel kod ten zmieni do takiej postaci:

```
[1, 2, 3].map(function(n) {
  return n + 1;
});
```



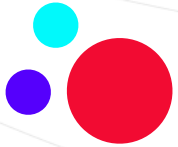
Babel - playground

- https://babeljs.io/repl#?browsers=defaults%2C%20not%20ie%2011%2C%20not%20ie_mob%2011&build=&builtIns=false&corejs=3.21&spec=false&loose=false&code_lz=Q&debug=false&forceAllTransforms=false&shippedProposals=false&circleciRepo=&evaluate=false&fileSize=false&timeTravel=false&sourceType=module&lineWrap=true&presets=env%2Creact%2Cstage-2&prettier=false&targets=&version=7.20.6&externalPlugins=&assumptions=%7B%7D



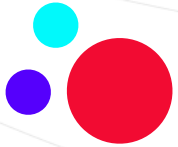
Babel - szybkie użycie Reacta

- https://reactjs.org/docs/add-react-to-a-website.html#quickly-try-jsx



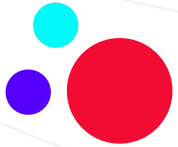
Package manager

- Z definicji jest to zestaw narzędzi pozwalających na instalowanie, usuwanie, modyfikowanie, upgrade’owanie i konfigurowanie programów komputerowych.
- Z ich wykorzystaniem można również dokonać audytu zależności celem sprawdzenia potencjalnych luk w zabezpieczeniach itd.
- Korzystanie z package managera różni się od instalowania pojedynczych pakietów ręcznie tym, iż manager obsługuje zależności i potrafi ściągnąć z Internetu wymagane pakiety.



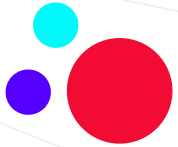
NPM

- Node Package Manager, czyli standardowe narzędzie konsolowe przeznaczone do tego, by za jego pomocą instalować zależności Node.js
- Jest to jednocześnie publiczna baza danych, w której umieścić można swoją paczkę, swój plugin
- NPM został niedawno wykupiony przez Microsoft.
- Jest entry pointed do ekosystemu modułów JS o otwartym kodzie źródłowym, oraz całym ekosystemie stworzonym do pracy z tymi modułami i zarządzaniem nimi.



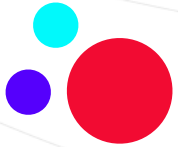
NPM Registry

- Jest to bardzo ważna część ekosystemu NPM, będąc publiczną bazą danych kodu JS.
- Możemy w nim znaleźć kod JS, narzędzia, biblioteki, frameworki, które możemy pobrać i wstrzyknąć do naszej aplikacji
- Jest to największy rejestr tego typu w internecie. W tym momencie jest to około miliona paczek dostępnych online.
- Każdy może tworzyć i dodawać paczki do rejestru, pozwalając na tworzenie relacji między milionami developerów na świecie, gdyż każdy może ją wykorzystać, zgłosić błąd i tworzyć kod razem z Nami!
- Moduły tego typu mają za zadanie uprościć i skrócić czas developowania - nie musimy za każdym razem na nowo wynajdywać koła!



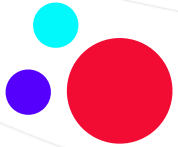
NPM Registry

- Każda dodana paczka otrzymuje swoją stronę internetową w rejestrze.
- Możemy przejrzeć informacje o niej, zobaczyć ile osób wykorzystuje dane paczki, znaleźć linki do repozytorium paczki, znaleźć listę zgłoszonych błędów i inne informacje o każdej paczce.
- Z racji na to, iż każdy może publikować paczkę do rejestru nie ma gwarancji jakości na paczkę. Z racji na ten fakt lepiej korzystać z popularnych i aktywnie developowanych paczek skupionych na problemie, który dana paczka ma dla nas rozwiązywać.



NPM Command Line

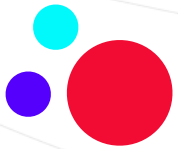
- NPM command-line (CLI) jest domyślnym menadżerem paczek preinstalowany wraz z Node.js. Pozwala na instalowanie i zarządzanie paczek dla projektów JS.
- Ma za zadanie pobrać i umieścić paczki zapisane w liście, którą sporządzamy dla danego projektu i umieszczenie ich w folderze **node_modules**
- Przy instalacji, w momencie gdy w odpowiedni sposób oznaczymy wersję paczki w naszej liście, paczka zostanie **zaktualizowana** do najnowszej



NPM Command Line - komendy

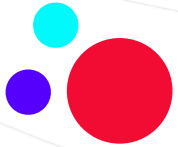
- **`npm install`** - Jego użycie pozwala na dodanie paczki do naszego projektu
- **`npm init`** - Za jego pomocą generujemy **package.json**
- **`npm audit`** - Daje możliwość wygenerowania raportu o znanych zagrożeniach związanych z paczkami w projekcie
- **`npm update`** - Pozwala zaktualizować zainstalowane zależności
- **`npm uninstall`** - Usuwanie zależności z folderu **node_modules** i z **package.json**
- **`npm run`** - Pozwala na uruchamianie skryptów ustawionych w **package.json**
- **`npm start`** - Uruchamia Twój skrypt **start**
- **`npm publish`** - Umieszczenie Twojej paczki do rejestru NPM

Test komend



Yarn

- Nowszy odpowiednik NPM, nie posiadający własnego repozytorium pakietów
- Pozwala na używanie innych, istniejących już repozytorium, w tym tego powiązanego z NPM
- Jest bardzo szybki, z racji na inną metodę instalacji pakietów. NPM skanuje drzewo zależności i następnie zaciąga wszystkie niezbędne pakiety, YARN sprawdza katalog globalnego cache'u (kiedyś pobranych tych samych paczek) i sprawdza czy odpowiada ona pakietom z listy. Jeśli znajdzie - użyje jej, jeśli nie - zainstaluje i doda do cache, przerzucając jednocześnie paczki z cache do node_modules.
- Jest rownież bezpieczny bo sprawdza sumy kontrolne, bo żadne dane nie będą utracone podczas pobierania pakietów.

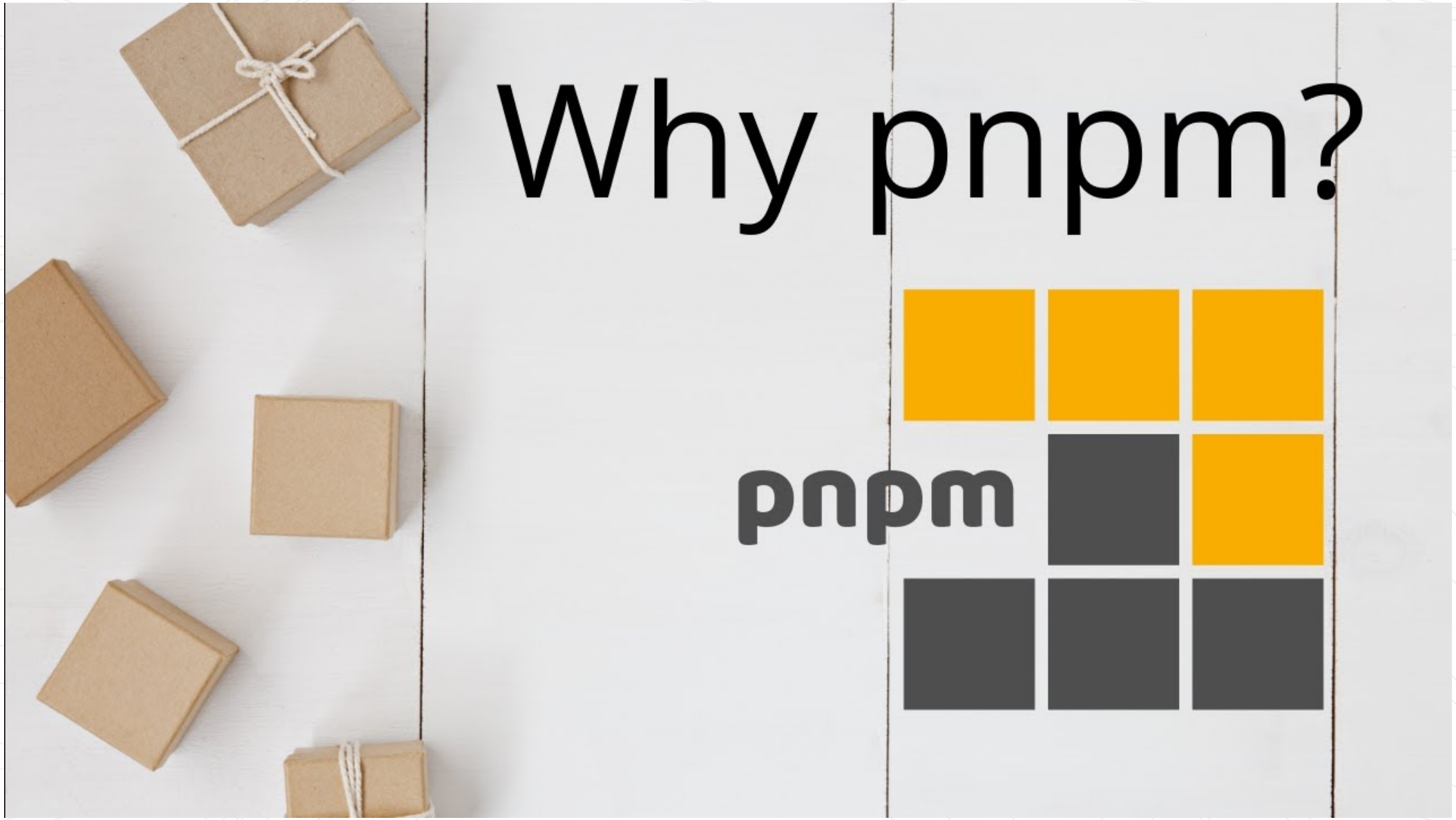


Yarn - polecenia CLI

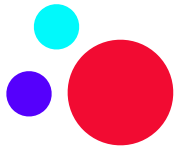
- **`yarn add`** - Instaluje zależność w naszym repozytorium.
- **`yarn init`** - Za jego pomocą generujemy **package.json**
- **`yarn install`** - Instaluje wszystkie paczki zapisane w **package.json**
- **`yarn publish`** - Umieszczenie Twojej paczki do rejestru w jakim ma być opublikowana paczka
- **`yarn remove`** - Usuwanie zależności z folderu **node_modules** i z **package.json**

Test komend

PNPM



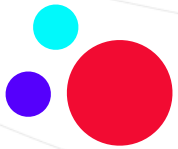
NPM vs Yarn - video



package.json, package-lock.json

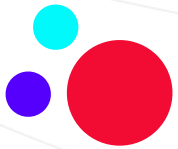
infoShareAcademy.com

infoShare ACADEMY



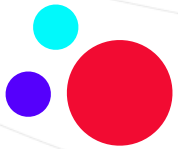
package.json

- Jest plikiem w formacie JSON, który istnieje w każdym projekcie opartym na Node.js
- Zawiera informacje o projekcie (wersja, opis, autorzy, tagi, git url itd.)
- Przechowuje informacje o dodatkowych zależnościach, jakie zostały wykorzystane do stworzenia danego projektu.
- Zależności zapisane są za pomocą nazwy i wersji.
- Gdy inicjujemy nowy projekt - musimy zacząć od **npm init/yarn init**, gdy wykorzystujemy już istniejący - zaczniemy najprawtopodbnioej od **npm install/yarn install**



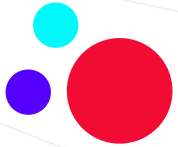
package.json

- Zależności projektu podzielone są przeważnie na dwie grupy:
  - **dependencies** - są to zależności niezbędne do działania Twojej aplikacji. Przykład - paczka UUID służąca do generowania unikalnych id.
  - **devDependencies** - zależności wspomagające proces tworzenia aplikacji, czyli narzędzia do sprawdzania, budowania kodu, zarządzania projektem, pisaniem testów.
- Kolejna sekcja paczki to sekcja **scripts**. Wewnątrz tego obiektu przechowywane są komendy, które są uruchamiane przez narzędzie linii komend. Komendy mogą dotyczyć wywołania innych skryptów, zbudowanie projektu, uruchomienie projektu w wersji dev/prod itd.



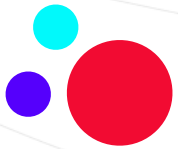
package.json - przykład

https://gitlab.com/sognare/squash4us/-/raw/master/package.json



package-lock.json

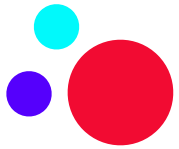
- Jest to automatycznie generowany plik, służący przechowywaniu informacji o całym drzewie zależności, które zostało stworzone podczas instalacji z użyciem **npm install**.
- Dzięki jego obecności jesteśmy w stanie za każdym razem zreprodukować drzewo zależności w identycznym stanie podczas instalacji, co znacznie przyspiesza sam proces i zapewnia bezpieczeństwo w kwestii zachowania tych samych wersji paczek u innych developerów.



package-lock.json - przykład

https://gitlab.com/sognare/squash4us/-/raw/master/package-lock.json

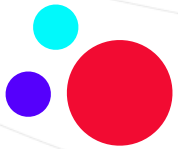
infoShare ACADEMY



node_modules

infoShareAcademy.com

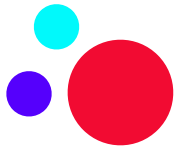
infoShare ACADEMY



node_modules

- Zainstalowane zależności zlokalizowane są w folderze **node_modules**.
- Zazwyczaj jest to największy i najcięższy folder w naszym projekcie
- Zawartości folderu node_modules nie umieszczamy ani na serwerze produkcyjnym, ani w repozytorium - stosujemy do tego .**gitignore**.
- Nie przejmuj się też rozmiarem tego folderu. Potrafi on mieć rozmiary kilkuset MB, a czasem nawet kilku GB.
- Istnieją rozwiązania takie jak pnpm, które pozwalają optymalizować przestrzeń zajmowaną przez node_modules poprzez mechanizmy linkowania i reużywania paczek.

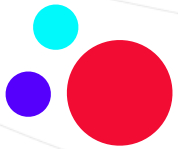
Przykład



Webpack

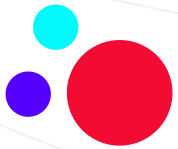
infoShareAcademy.com

infoShare ACADEMY



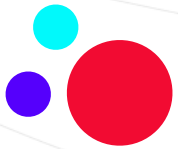
Webpack

- Bundler, czyli narzędzie potrafiące spakować wiele różnych formatów do jednego pliku JavaScript.
- Webpack za pomocą przygotowanej konfiguracji zajmie się zbudowaniem grafu zależności między modułami naszej aplikacji.
- Zajmuje się nie tylko plikami **JS**, ale również innymi plikami **CSS, HTML**, **PNG**, **SVG.**
- Zachowuje kolejność importowanych modułów i zależności, zadba o wszelkie zależności dla każdej linii kodu.
- Z tak przygotowanego kodu powstaje tak zwany bundle, czyli jeden lub więcej plików, które są ze sobą połączone.
  Bundle to **zoptymalizowany pakiet dla przeglądarki.**
- Wyjściowo pliki są jak najmniejsze, aby proces ich pobierania i ładowania był jak najkrótszy



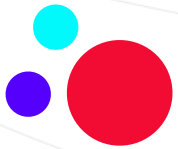
Webpack - konfiguracja

- https://webpack.js.org/guides/getting-started/



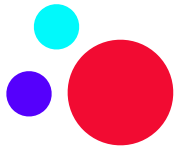
Webpack dev server

- Wykorzystywany do tego, by w procesie tworzenia aplikacji **(nie w wersji produkcyjnej!!!)** utworzyć serwer, który pozwoli nam na dynamiczne i automatyczne odświeżanie zmian w aplikacji bez konieczności przeładowania okna (**hot reload**).
- Konfiguracja i ustawienie takiego modułu:
  - https://webpack.js.org/configuration/dev-server/



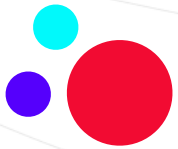
Webpack config

- https://hashinteractive.com/blog/complete-guide-to-webpack-configuration-for-react/



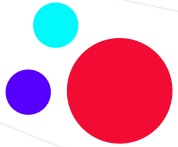
Toolchainy

infoShareAcademy.com



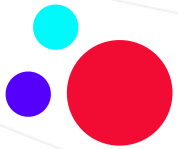
CRA – definicja

- Create React App zapewnia bardzo dogodne środowisko pracy sprzyjające nauce Reacta.
- Jest to najlepszy sposób, aby zacząć tworzyć nową **SPA** w React.
- Środowisko pracy stworzone przez Create React App nie tylko umożliwi ci stosowanie najnowszych funkcjonalności języka **JavaScript**, lecz także zoptymalizuje twój kod przed oddaniem go do użytku i ogólnie znacznie usprawni twoją pracę.
- Aby móc skorzystać z tego rozwiązania na swoim komputerze, będziesz potrzebować **Node >= 14.0.0** i **npm >= 5.6**.



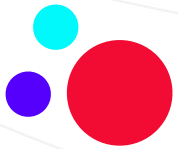
CRA – definicja

- Create React App nie obsługuje ani logiki backendu ani baz danych; tworzy jedynie frontendowy build pipeline. Dzięki temu możesz go używać z dowolnie wybranym przez siebie backendem.
- Create React App zawiera narzędzia takie jak Babel i webpack, ale nie musisz nic o nich wiedzieć, aby z powodzeniem używać ich w swoich projektach.
- Kiedy uznasz, że twoja aplikacja jest gotowa do wdrożenia do środowiska produkcyjnego, zastosuj komendę **npm run build**. Dzięki temu uzyskasz zoptymalizowaną wersję swojej aplikacji.



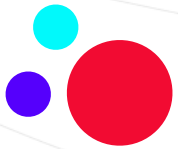
CRA - jak stworzyć

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



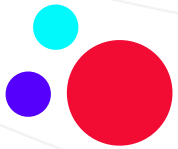
CRA - komendy

- **`npm start`** - Odpalenie aplikacji w trybie developerskim, domyślnie adres localhost:3000. Strona będzie się automatycznie odświeżać, błędy będą widoczne w przeglądarce (jeśli występują).
- **`npm test`** - uruchomienie test runnera w trybie watch oraz interactive. Pozwala to na prześledzenie zmian w testach bez konieczności ponownego uruchamiania.
- **`npm run build`** - Zbudowanie wersji produkcyjnej w folderze **build.** React zostaje wyeksportowany w wersji produkcyjnej, a cała aplikacja zostaje zoptymalizowana pod najlepszy performance.
- **`npm run eject`** - Droga w jedną stronę, wyłączenie wszystkich wbudowanych mechanizmów schowanych za całą aplikacją i przejęcie kontroli w pełni. Następuje doinstalowanie pakietów, dochodzi bardzo dużo dodatkowych plików. Wykorzystywane gdy jest to naprawdę niezbędne, przeznaczone dla **zaawansowanych programistów**.



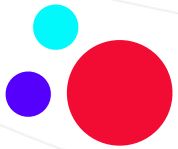
CRA - struktura projektu

```
moja-aplikacja/
  README.md
  node_modules/
  package.json
  public/
    index.html
    favicon.ico
  src/
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
```



Tryb developerski a produkcyjny

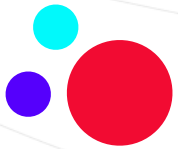
- Domyślnie React dorzuca do kodu wiele ostrzeżeń, które przydają się podczas pisania aplikacji. Niestety z ich powodu React jest cięższy i wolniejszy, dlatego zaleca się wrzucać na produkcję tylko wygenerowany kod produkcyjny.
- Co do zasady, trybu deweloperskiego powinno używać się podczas tworzenia aplikacji, a kod produkcyjny wrzucać tam, gdzie będą z niego korzystać docelowi użytkownicy.



CRA - production build

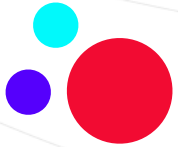
- Pliki JS i CSS będą zbudowane w folderze **build/static**
- Każdy plik ma swoją unikalną nazwę w postaci hasha, pozwalający na wykorzystywanie zaawansowanych technik cache'owania, czyli przechowywania kodu celem szybszego ładowania (w dużym skrócie)
- Podobnie ma się sytuacja z assetami typu obrazki, czcionki itd. One również są optymalizowane celem umożliwienia ich cacheowania w sposób bardziej agresywny, celem unikania dodatkowego pobierania ich przez przeglądarkę - co wizualnie da efekt strony zaczytującej się **OD RAZU**

Vite



Vite vs CRA

- Tworzenie projektu:
- **Vice** - temp latka **react-ts** ~ **1s,** instalacja zależności ~**5s**
- CRA w templatce **typesctipt ~40 sekund**
- **Start projektu:**
  - **Vite** - około **200ms**, build przy załadowaniu aplikacji - natychmiastowy. HMR działa wyśmienicie, niezauważalne przeładowania strony
  - **CRA** - około **7s**, HMR błyskawiczny.
- **Dodanie paczek* i ponowny start:**
  - **Vite** - start bez zmian, bundling od razu, całość **~1s**
  - **CRA** - problem z bibliotekami, po naprawieniu start **~21s(bez cache)**, **~5s z cache**



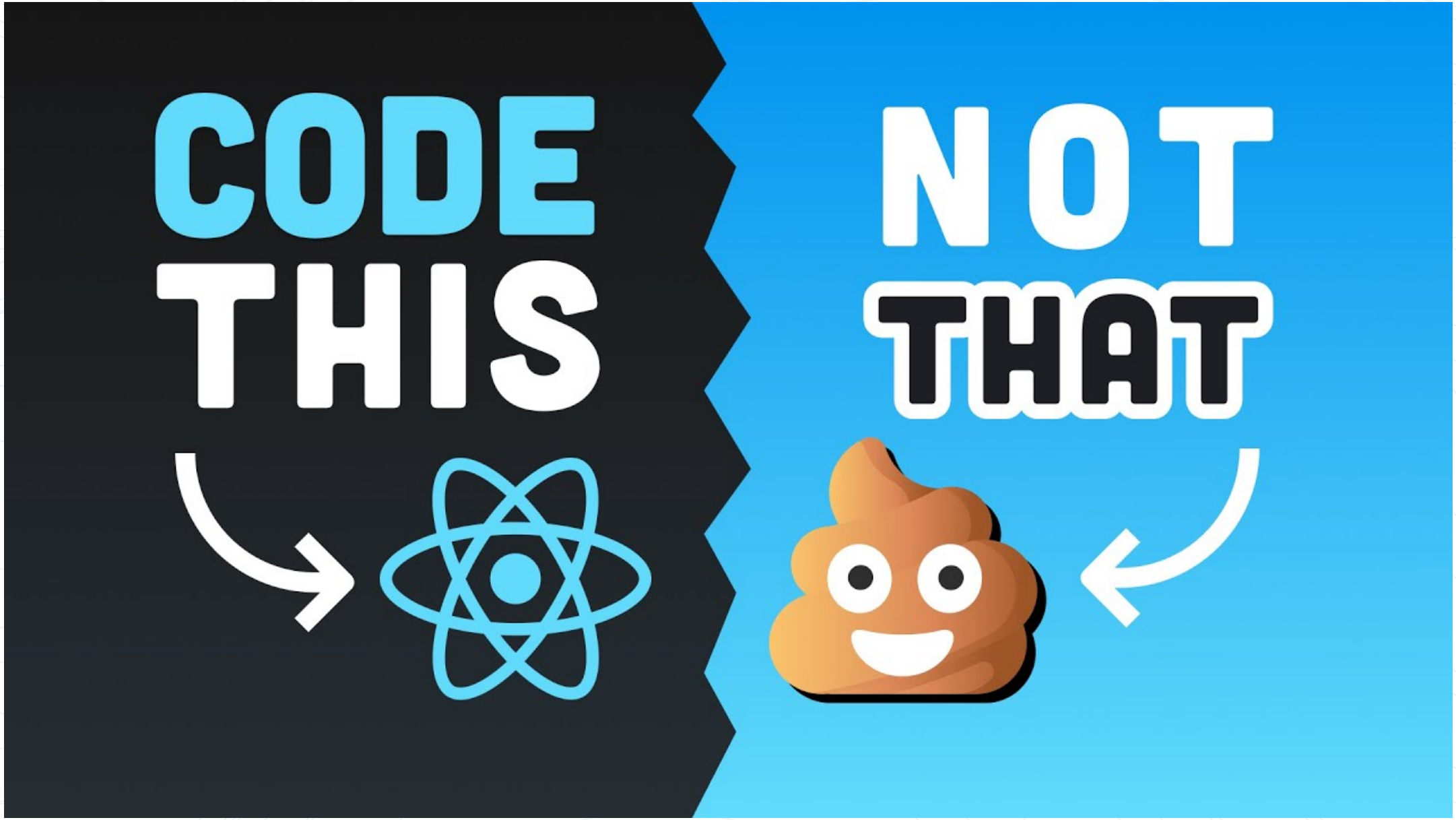
Vite komendy

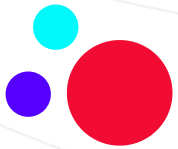
- **vite dev –** włączenie serwera developerskiego w folderze, w jakim odpalamy ten skrypt. **Hot Module Reload** jest w tym wypadku włączony (dynamiczne odświeżanie strony bez konieczności przeładowania czegokolwiek w przeglądarce)
- **vite build** - stworzenie buildu produkcyjnego i wyrzucenie zbudowanych plików do folderu wyjściowego (domyślnie **./dist**)
- **vite preview** - ciekawe polecenie, pozwala na lokalne podejrzenie zbudowanego buildu produkcyjnego (pliki z folderu **./dist**). Brak **HMR.**

Pytania

infoShareAcademy.com

Antypatterny React - ciekawostka





Linki dla chętnych

- https://www.techiediaries.com/vanilla-js-vs-react/

infoShare ACADEMY

Dziękuję za uwagę

Jakub Wojtach