POLITECHNIKA ŚWIĘTOKRZYSKA

Zaawansowane aplikacje frontendowe – wykład 5

Integracje i wdrażanie aplikacji

mgr inż. Mateusz Pawełkiewicz

1.10.2025

Wprowadzenie: Od lokalnego do globalnego

Dotychczasowe wykłady koncentrowały się na budowaniu aplikacji, która działa lokalnie na maszynie dewelopera. W realiach komercyjnych jest to jednak dopiero połowa sukcesu. Aplikacja musi zostać zintegrowana z zewnętrznymi usługami (backendem), być zdolna do bezpiecznego zarządzania sesją użytkownika oraz, co najważniejsze, zostać wdrożona w sposób wydajny i niezawodny dla globalnego odbiorcy.

Ten wykład przeprowadzi nas przez kluczowe etapy transformacji projektu z lokalnego środowiska deweloperskiego (localhost) do w pełni funkcjonalnej, produkcyjnej aplikacji internetowej. Omówimy architekturę tej integracji, strategie bezpieczeństwa oraz zautomatyzowany proces wdrażania i monitorowania.

Integracja z Backendem: REST vs. GraphQL

Dwa podejścia do komunikacji API

Każda dynamiczna aplikacja frontendowa musi komunikować się z backendem w celu pobierania i wysyłania danych. Historycznie, dominującym wzorcem jest **REST (Representational State Transfer)**, architektura oparta na zasobach (np. /users, /products), która wykorzystuje metody HTTP (GET, POST, PUT, DELETE) do manipulacji tymi zasobami.

W ostatnich latach na popularności zyskał **GraphQL**, który jest językiem zapytań dla API Zamiast wielu endpointów definiowanych przez serwer (jak w REST), GraphQL zazwyczaj udostępnia jeden endpoint. Klient wysyła zapytanie (query) precyzyjnie określające, jakie dane są potrzebne, co eliminuje powszechne w REST problemy "nadmiernego pobierania" (over-fetching) lub "niewystarczającego pobierania" (under-fetching) danych.

REST w praktyce: Biblioteka axios

Do komunikacji z API REST w aplikacjach React najczęściej wykorzystuje się bibliotekę axios. Jest to klient HTTP oparty na obietnicach (promise-based), który upraszcza wysyłanie żądań HTTP.Przewagą axios nad wbudowanym w przeglądarkę fetch jest m.in. automatyczna transformacja danych JSON, lepsza obsługa błędów oraz wsparcie dla interceptorów.

Centralizacja konfiguracji axios

Kluczową praktyką przy korzystaniu z axios jest unikanie wywołań axios.get(...) bezpośrednio w komponentach. Taka praktyka narusza zasadę DRY (Don't Repeat Yourself) i utrudnia zarządzanie konfiguracją (np. dodaniem nagłówków autoryzacyjnych).

Znacznie lepszym podejściem jest stworzenie scentralizowanej "instancji" axios. Definiujemy jeden moduł (np. api.js), który eksportuje skonfigurowaną instancję axios.create(). Pozwala to na ustawienie wspólnego baseURL oraz, co najważniejsze, na centralne zarządzanie *interceptorami* (przechwytywaczami) żądań i odpowiedzi.

Przykładowa konfiguracja scentralizowanego klienta API :

```
// src/api.js
import axios from 'axios';

// 1. Utworzenie instancji ze stałą konfiguracją
const API = axios.create({
   baseURL: 'https://api.example.com/v1/',
   timeout: 10000,
   headers: {
      'Content-Type': 'application/json',
   }
});

// 2. Skonfigurowanie interceptora (przechwytywacza) ŻĄDAŃ
// Ten kod uruchomi się PRZED wysłaniem każdego żądania
API.interceptors.request.use(
   (config) => {
      // Przykład: dynamiczne pobranie tokena i dołączenie go do nagłówka
      const token = localStorage.getItem('access_token');
      if (token) {
         config.headers.Authorization = `Bearer ${token}`;
      }
      return config;
   },
   (error) => Promise.reject(error)
);

// 3. Skonfigurowanie interceptora ODPOWIEDZI
// Ten kod uruchomi się PO otrzymaniu odpowiedzi, zanim trafi ona do.then()
API.interceptors.response.use(
   (response) => response, // Zwróć udaną odpowiedź
   async (error) => {
      // Przykład: obsługa wygaśnięcia tokena (np. status 401)
```

```
    if (error.response?.status === 401) {
      // Logika odświeżania tokena...
      // np. przekieruj na stronę logowania
      console.warn('Token wygasł lub jest nieprawidłowy.');
    }
    return Promise.reject(error);
  }
);

export default API;
```

Dzięki temu podejściu, komponenty React importują już skonfigurowaną instancję (import API from './api') i używają jej (API.get('/posts')), a cała logika autoryzacji i obsługi błędów jest zarządzana centralnie.

GraphQL w praktyce: Apollo Client

W przypadku GraphQL, choć można używać axios lub fetch, standardem przemysłowym jest **Apollo Client**. Powodem jest fakt, że Apollo to znacznie więcej niż klient HTTP; to kompleksowa biblioteka zarządzania stanem, stworzona specjalnie dla GraphQL.

Apollo Client automatyzuje zadania, które przy REST i axios musielibyśmy wykonywać ręcznie:

- **Inteligentne buforowanie:** Automatycznie buforuje wyniki zapytań w znormalizowanej pamięci podręcznej. Jeśli zażądamy tych samych danych ponownie, Apollo natychmiast zwróci je z cache'a.
- **Zarządzanie stanem UI:** Zapewnia hooki (np. useQuery), które obsługują stany ładowania (loading), błędu (error) i danych (data).
- **Aktualizacje UI po mutacjach:** Automatycznie aktualizuje powiązane zapytania (queries) po wykonaniu zmiany danych (mutation).

Konfiguracja i użycie Apollo Client

Konfiguracja Apollo Client polega na utworzeniu instancji klienta i udostępnieniu jej całej aplikacji React za pomocą komponentu <ApolloProvider>.

Krok 1: Instalacja zależności
npm install @apollo/client graphql

Krok 2: Konfiguracja w main.jsx

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import { ApolloClient, InMemoryCache, ApolloProvider } from '@apollo/client';

// 1. Inicjalizacja klienta Apollo
const client = new ApolloClient({
  uri: 'https://flyby-router-demo.herokuapp.com/', // Adres URL serwera GraphQL
  cache: new InMemoryCache(), // Uruchomienie buforowania w pamięci
});

// 2. Owiń aplikację komponentem Provider
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <ApolloProvider client={client}>
      <App />
    </ApolloProvider>
  </React.StrictMode>
);
```

Krok 3: Wykonywanie zapytań (Queries) i mutacji (Mutations) w komponencie

Apollo udostępnia dedykowane hooki useQuery do pobierania danych i useMutation do ich modyfikacji.

```
import { gql, useQuery, useMutation } from '@apollo/client';

// 1. Definicja zapytania (Query) w języku GraphQL
const GET_LOCATIONS = gql`
  query GetLocations {
    locations {
      id
      name
      description
    }
  }
`;

// 2. Definicja mutacji
const ADD_TODO = gql`
  mutation AddTodo($type: String!) {
    addTodo(type: $type) {
```

```
    id
    type
   }
  }
`;

function DisplayLocations() {
  // 3. Użycie hooka useQuery
  const { loading, error, data } = useQuery(GET_LOCATIONS);

  // 4. Użycie hooka useMutation
  // Zwraca funkcję 'addTodo' do wywołania oraz stan mutacji
  const = useMutation(ADD_TODO);

  if (loading) return <p>Ładowanie...</p>;
  if (error) return <p>Błąd: {error.message}</p>;

  const handleAddTodo = () => {
    addTodo({ variables: { type: 'Nowe zadanie' } });
  };

  return (
    <div>
      <button onClick={handleAddTodo} disabled={mutationLoading}>
        Dodaj Zadanie
      </button>
      {data.locations.map(({ id, name, description }) => (
        <div key={id}>
          <h3>{name}</h3>
          <p>{description}</p>
        </div>
      ))}
    </div>
  );
}
```

Uwierzytelnianie Tokenami JWT

Dylemat przechowywania tokenu

Niezależnie od tego, czy używamy REST, czy GraphQL, najpopularniejszą metodą uwierzytelniania w nowoczesnych aplikacjach jest **JWT (JSON Web Token)**. Po pomyślnym zalogowaniu, backend zwraca klientowi token (ciąg znaków), który musi być dołączany do każdego kolejnego żądania.

Powstaje fundamentalne pytanie: gdzie po stronie klienta (w przeglądarce) bezpiecznie przechować ten token? Dwa najczęstsze miejsca to localStorage i httpOnly cookie.

localStorage: Wygoda kosztem bezpieczeństwa

localStorage jest API przeglądarki pozwalającym na przechowywanie danych (klucz-wartość), które są dostępne nawet po zamknięciu przeglądarki.

- **Zaleta:** Bardzo łatwy w implementacji. JavaScript może łatwo odczytać token (localStorage.getItem('token')) i dołączyć go do nagłówka Authorization (co widzieliśmy w przykładzie interceptora axios).
- **Wada (Krytyczna): Podatność na XSS (Cross-Site Scripting)**. Jeśli atakującemu uda się wstrzyknąć na stronę złośliwy skrypt (np. przez lukę w formularzu komentarzy), skrypt ten uzyskuje pełny dostęp do localStorage. Może on natychmiast odczytać token i wysłać go na serwer atakującego, co pozwala na pełne przejęcie sesji użytkownika.

httpOnly cookie: Odporność na XSS, nowe ryzyko

Alternatywą jest przechowywanie tokenu w ciasteczku (cookie) ustawionym przez serwer z flagą httpOnly.

- **Zaleta:** Flaga httpOnly sprawia, że ciasteczko jest *całkowicie niedostępne* dla JavaScriptu po stronie klienta. document.cookie go nie pokaże. To całkowicie eliminuje ryzyko kradzieży tokenu przez atak XSS. Przeglądarka automatycznie dołącza to ciasteczko do każdego żądania wysyłanego do domeny serwera.
- **Wada: Podatność na CSRF (Cross-Site Request Forgery)**. Ponieważ przeglądarka *automatycznie* wysyła ciasteczka z każdym żądaniem (nawet jeśli żądanie pochodzi z innej, złośliwej strony), aplikacja staje się podatna na ataki CSRF. Nowoczesne przeglądarki znacznie ograniczyły to ryzyko dzięki atrybutowi SameSite (szczególnie SameSite=Strict), ale nie wyeliminowały go całkowicie.

Podejście hybrydowe: Najlepsza praktyka 2025

Żadna z powyższych metod nie jest idealna. Dlatego współczesne, bezpieczne aplikacje stosują podejście hybrydowe, które wykorzystuje dwa typy tokenów: **Access Token (Token Dostępowy)** i **Refresh Token (Token Odświeżający)**.

Architektura ta działa następująco:

1. **Access Token (krótkożyjący, np. 15 minut):** Jest przechowywany **w pamięci aplikacji (np. w stanie React lub zmiennej globalnej)**.

- *Bezpieczeństwo XSS:* Wysokie. Skrypt XSS nie może go łatwo odczytać, jeśli nie jest trwale zapisany (choć zaawansowane ataki XSS nadal stanowią ryzyko dla pamięci).
- *Bezpieczeństwo CSRF:* Pełne. Token nie jest wysyłany automatycznie; musi być ręcznie dołączony do żądania przez klienta (np. przez interceptor axios).

2. **Refresh Token (długożyjący, np. 7 dni):** Jest przechowywany w **httpOnly cookie** z flagami Secure i SameSite=Strict.
   - *Bezpieczeństwo XSS:* Pełne. JavaScript nie ma do niego dostępu.
   - *Bezpieczeństwo CSRF:* Wysokie. SameSite=Strict zapobiega wysyłaniu ciasteczka przy żądaniach między domenami.

Przepływ działania:

- Użytkownik loguje się. Serwer zwraca Access Token (w ciele odpowiedzi JSON) i Refresh Token (w httpOnly cookie).
- Aplikacja przechowuje Access Token w pamięci i używa go do wszystkich żądań API.
- Użytkownik odświeża stronę (F5). Access Token z pamięci znika.
- Aplikacja (np. przy starcie) wysyła żądanie do specjalnego endpointu /refresh_token. Żądanie to *nie zawiera* Access Tokena (bo go nie ma), ale przeglądarka *automatycznie dołącza* do niego httpOnly cookie z Refresh Tokenem.
- Serwer weryfikuje Refresh Token i jeśli jest ważny, odsyła nowy, świeży Access Token (na kolejne 15 minut).
- Aplikacja zapisuje nowy Access Token w pamięci i kontynuuje pracę bez konieczności ponownego logowania użytkownika.

To podejście łączy bezpieczeństwo httpOnly cookie (ochrona przed XSS dla klucza długoterminowego) z odpornością tokenów w pamięci na ataki CSRF.

Tabela: Porównanie strategii przechowywania JWT

Metoda	**localStorage**	**httpOnly cookie**	**Hybrydowa (Pamięć + httpOnly)**
Główne Ryzyko	XSS (Krytyczne)	CSRF (Średnie)	Minimalne (wymaga zaawansowanego XSS)
Ochrona XSS	Brak	Doskonała	Doskonała (dla Refresh Tokena)
Ochrona CSRF	Doskonała (wymaga SameSite)	Ryzyko (łagodzone przez SameSite)	Doskonała (dla Access Tokena)

Trwałość (po F5)	Tak	Tak	Tak (poprzez mechanizm odświeżania)
Złożoność	Niska	Średnia (wymaga koordynacji backendu)	Wysoka (wymaga logiki odświeżania)
Rekomendacja 2025	Nie rekomendowane	Akceptowalne (z SameSite=Strict)	**Rekomendowane**

Konfiguracja Środowisk w Aplikacjach Vite

Problem: Różne konfiguracje dla różnych środowisk

Aplikacja niemal zawsze wymaga innych ustawień w zależności od tego, gdzie jest uruchomiona:

- **Development (Lokalnie):** Łączy się z lokalnym http://localhost:8000/api.
- **Staging (Testowe):** Łączy się z https://api.staging.example.com.
- **Production (Produkcyjne):** Łączy się z https://api.example.com i używa innego klucza API dla usługi analitycznej.

Hardkodowanie tych wartości w kodzie jest niedopuszczalne. Rozwiązaniem są zmienne środowiskowe.

Pliki .env w Vite

Vite ma wbudowane wsparcie dla zmiennych środowiskowych przy użyciu plików .env w głównym katalogu projektu:

- .env: Ładowany zawsze. Zawiera wartości domyślne.
- .env.development: Ładowany tylko w środowisku deweloperskim (npm run dev).
- .env.production: Ładowany tylko w środowisku produkcyjnym (npm run build).
- .env.local: Ładowany zawsze, ale ma wyższy priorytet niż .env. Powinien być w .gitignore i zawierać lokalne sekrety.

Prefiks VITE_: Wbudowana ochrona przed wyciekiem

To kluczowa różnica w stosunku do innych narzędzi. Aby zapobiec przypadkowemu wyciekowi wrażliwych kluczy (np. DB_PASSWORD) do kodu klienta (przeglądarki), Vite eksponuje **tylko** te zmienne, które zaczynają się od prefiksu VITE_.

Przykład:

Plik .env.production:

```
DB_PASSWORD=super_secret_key_123
VITE_API_URL=https://api.example.com
VITE_GOOGLE_ANALYTICS_KEY=GA_PROD_98765
```

W kodzie aplikacji:

```
// Dostęp do zmiennych odbywa się przez specjalny obiekt import.meta.env
console.log(import.meta.env.VITE_API_URL); // "https://api.example.com"
console.log(import.meta.env.VITE_GOOGLE_ANALYTICS_KEY); // "GA_PROD_98765"

// Ta zmienna NIE BĘDZIE dostępna w kodzie klienta!
console.log(import.meta.env.DB_PASSWORD); // undefined
```

Ta funkcja bezpieczeństwa wymusza na deweloperze świadomą decyzję, które zmienne mają być publicznie dostępne w przeglądarce.

Build i Optymalizacja

Polecenie npm run build

Gdy aplikacja jest gotowa do wdrożenia, uruchamiamy polecenie npm run build, które w przypadku Vite wywołuje vite build.

Co dzieje się "pod maską": Rola Rollup

Vite jest znany z błyskawicznego serwera deweloperskiego, który nie wymaga bundlowania (łączenia plików) na bieżąco. Jednak w przypadku budowania na produkcję, takie podejście byłoby nieefektywne (zbyt wiele małych plików do pobrania).

Dlatego vite build używa pod spodem bundlera o nazwie **Rollup**. Rollup wykonuje trzy kluczowe zadania optymalizacyjne :

1. **Bundling (Łączenie):** Łączy wszystkie moduły JavaScript i CSS w mniejszą liczbę plików (tzw. "chunks"), aby zminimalizować liczbę żądań HTTP.
2. **Minification (Minifikacja):** Usuwa wszystkie niepotrzebne znaki z kodu (białe znaki, komentarze) i skraca nazwy zmiennych, aby drastycznie zmniejszyć rozmiar plików.
3. **Tree-shaking (Wstrząsanie drzewem):** To zaawansowana technika "eliminacji martwego kodu" (dead-code elimination). Rollup analizuje ścieżki importu w całej aplikacji i jeśli wykryje, że funkcja (lub cały moduł) została zaimportowana, ale *nigdy nie jest używana*, zostanie ona *usunięta* z końcowego pliku produkcyjnego.

Wynikiem jest folder dist zawierający wysoce zoptymalizowane, statyczne zasoby (HTML, CSS, JS), gotowe do wdrożenia na dowolnym serwerze.

Weryfikacja buildu: vite preview

Po uruchomieniu npm run build, serwer deweloperski (dev) już nie działa. Jak więc przetestować, czy produkcyjny build działa poprawnie?

Vite oferuje polecenie vite preview (uruchamiane przez npm run preview).To polecenie uruchamia prosty serwer, który serwuje pliki ze zbudowanego folderu dist.

Ostrzeżenie: vite preview służy *wyłącznie* do lokalnego podglądu i testowania buildu produkcyjnego. **Absolutnie nie wolno** używać go jako serwera produkcyjnego. Nie jest on ani wydajny, ani bezpieczny, ani skalowalny. Brakuje mu kluczowych funkcji produkcyjnych, takich jak kompresja Gzip/Brotli czy odpowiednie nagłówki bezpieczeństwa.

Deployment: Wdrażanie Aplikacji

Mając folder dist, mamy kilka opcji wdrożenia naszej aplikacji.

Porównanie platform wdrożeniowych

Wybór platformy zależy od złożoności projektu, budżetu i wymaganej kontroli.

- **Vercel / Netlify:** Są to nowoczesne platformy "GitOps" (lub "Jamstack"). Działają poprzez połączenie z repozytorium Git (np. GitHub).
  - *Jak to działa:* Wypychasz kod do repozytorium (np. do gałęzi main), platforma automatycznie wykrywa zmianę, uruchamia npm run build i wdraża wynik na globalnej sieci CDN (Content Delivery Network).
  - *Zalety:* Niesamowita prostota (zero konfiguracji serwera), darmowe plany dla projektów open-source, automatyczne CI/CD, podglądy (preview) dla każdego Pull Requesta, obsługa funkcji serverless.
  - *Specjalizacje:* Vercel jest stworzony przez twórców Next.js i jest dla niego optymalny. Netlify ma bogatszy ekosystem wbudowanych usług, takich jak Netlify Forms czy Identity.
- **GitHub Pages:** Darmowa usługa hostingu statycznego od GitHub.
  - *Zalety:* Całkowicie darmowa, zintegrowana z repozytorium.
  - *Wady:* Obsługuje *tylko* strony statyczne (brak funkcji serverless, brak backendu). Idealna dla dokumentacji, portfolio lub prostych stron.
- **Własny VPS (np. DigitalOcean, AWS) + Nginx:**
  - *Jak to działa:* Wynajmujesz wirtualny serwer (VPS), instalujesz system operacyjny (np. Ubuntu), serwer WWW (np. Nginx) i *samodzielnie* konfigurujesz wszystko: przesyłanie plików dist (np. przez scp), konfigurację Nginx do serwowania plików, certyfikaty SSL, logi itp..
  - *Zalety:* Pełna kontrola nad środowiskiem, możliwość hostowania dowolnego backendu (Node, Python, Go) na tej samej maszynie.
  - *Wady:* Pełna odpowiedzialność. Wymaga zaawansowanej wiedzy z zakresu administracji systemami, bezpieczeństwa i sieci. Jest to najbardziej złożona i czasochłonna opcja.

Tabela: Porównanie opcji wdrożenia

Cecha	**Vercel / Netlify**	**GitHub Pages**	**Własny VPS (Nginx)**
Łatwość użycia	Bardzo łatwe (Git-Ops)	Łatwe	Bardzo trudne (pełna konfiguracja)
Wsparcie Backendu	Funkcje Serverless	Brak	Pełne (dowolny backend)
CI/CD	Wbudowane, automatyczne	Wymaga GitHub Actions	Wymaga ręcznej konfiguracji (np. GitHub Actions + SSH)
Globalny CDN	Tak (wbudowany)	Tak (wbudowany)	Nie (wymaga ręcznej konfiguracji np. Cloudflare)

Koszt (startowy)	Darmowy plan	Darmowy	Płatny (od ~$5/mc)
Idealne dla	Aplikacje Jamstack, Next.js, projekty komercyjne	Portfolio, dokumentacja, strony statyczne	Złożone aplikacje full-stack, niestandardowe wymagania

Przykład praktyczny: Wdrożenie na Vercel z GitHub

Proces wdrożenia aplikacji Vite na Vercel jest niezwykle prosty :

1. Wypchnij kod aplikacji Vite do repozytorium na GitHub.
2. Zaloguj się na Vercel (używając konta GitHub).
3. Kliknij "Add New... Project".
4. Wybierz repozytorium GitHub, które chcesz wdrożyć.
5. Vercel automatycznie wykryje, że jest to projekt Vite i ustawi odpowiednie polecenie budowania (vite build) oraz katalog wyjściowy (dist).
6. Kliknij "Deploy".
7. Po kilku minutach aplikacja jest wdrożona i dostępna globalnie pod adresem *.vercel.app. Od tego momentu każdy git push do gałęzi main automatycznie wywoła nowe wdrożenie produkcyjne.

CI/CD: Wprowadzenie do GitHub Actions

Vercel i Netlify ukrywają przed nami złożoność CI/CD. Jeśli jednak korzystamy z GitHub Pages lub własnego VPS, musimy skonfigurować ten proces samodzielnie. Najpopularniejszym narzędziem do tego celu jest **GitHub Actions**.

CI/CD to skrót od:

- **CI (Continuous Integration - Ciągła Integracja):** Praktyka automatycznego uruchamiania testów i budowania aplikacji *za każdym razem*, gdy ktoś wypycha kod do repozytorium. Zapobiega to "psuciu" głównej gałęzi kodu.
- **CD (Continuous Deployment - Ciągłe Wdrażanie):** Automatyczne wdrażanie aplikacji na produkcję *po tym*, jak pomyślnie przejdzie ona etap CI (testy i budowanie).

Przykład praktyczny: Workflow walidacyjny (CI)

Tworzymy plik w naszym repozytorium: .github/workflows/ci.yml. Ten plik definiuje "workflow", czyli zestaw kroków do wykonania przez GitHub.

Poniższy workflow walidacyjny uruchomi się przy każdym push i pull_request, aby sprawdzić, czy kod jest poprawny (przechodzi testy i buduje się).

```
#.github/workflows/ci.yml
name: CI Workflow - Test and Build

# Uruchom ten workflow przy każdym push i pull request
on: [push, pull_request]

jobs:
  test_and_build:
    runs-on: ubuntu-latest # Użyj maszyny wirtualnej z Ubuntu

    steps:
      # Krok 1: Pobranie kodu z repozytorium
      - name: Checkout code
        uses: actions/checkout@v4 #

      # Krok 2: Konfiguracja środowiska Node.js
      - name: Setup Node.js 20.x
        uses: actions/setup-node@v4 #
        with:
          node-version: '20.x'
          cache: 'npm' # *Kluczowa optymalizacja*: buforuje node_modules

      # Krok 3: Instalacja zależności
      - name: Install dependencies
        # 'npm ci' jest szybsze i bardziej deterministyczne niż 'npm install' w CI
        run: npm ci #

      # Krok 4: Uruchomienie testów
      - name: Run tests
        run: npm test #  - Jeśli to się nie uda, workflow zatrzyma się tutaj

      # Krok 5: Weryfikacja budowania produkcyjnego
      - name: Run build
        run: npm run build #  - Sprawdza, czy aplikacja buduje się bez błędów
```

Ten workflow *niczego nie wdraża*. Jego jedynym celem jest walidacja i ochrona gałęzi main. Jeśli testy lub budowanie nie powiedzie się, GitHub Actions oznaczy Pull Request jako "zepsuty" i uniemożliwi jego zmergowanie.

Oddzielny plik (np. deploy.yml) byłby skonfigurowany tak, aby uruchamiał się *tylko* po udanym CI na gałęzi main (np. on: push: branches: [main]) i zawierał kroki do faktycznego wdrożenia (np. kopiowanie plików dist na VPS przez SSH lub użycie akcji do wdrożenia na GitHub Pages).

Monitorowanie Błędów i Obserwowalność

Aplikacja jest wdrożona. Skąd mamy wiedzieć, czy działa poprawnie u użytkowników? Słynny problem "u mnie działa" jest prawdziwy – błędy często zależą od przeglądarki, sieci, rozszerzeń lub specyficznych działań użytkownika. Poleganie na zgłoszeniach użytkowników jest nieefektywne.

Śledzenie Błędów vs. Powtórki Sesji

Istnieją dwa główne podejścia do monitorowania aplikacji frontendowych

1. **Sentry (Śledzenie Błędów):** Odpowiada na pytanie: "*Co* poszło nie tak?". Sentry jest narzędziem stworzonym dla deweloperów. Jego zadaniem jest przechwytywanie każdego nieobsłużonego wyjątku w kodzie (np. TypeError: cannot read property 'name' of undefined), grupowanie błędów i dostarczanie deweloperowi pełnego *stack trace* (śladu stosu), "okruszków" (breadcrumbs – co użytkownik klikał przed błędem) oraz informacji o środowisku (przeglądarka, OS).
2. **LogRocket (Powtórki Sesji):** Odpowiada na pytanie: "*Co widział* użytkownik, gdy coś poszło nie tak?". LogRocket działa jak "kamera bezpieczeństwa" (lub "czarna skrzynka") dla aplikacji. Nagrywa sesję użytkownika – zmiany w DOM, ruchy myszy, logi konsoli i żądania sieciowe – pozwalając deweloperowi obejrzeć wideo z dokładną reprodukcją problemu. Jest to nieocenione przy debugowaniu trudnych do odtworzenia błędów UX, a także identyfikuje sygnały frustracji użytkownika, takie jak "rage clicks" (wielokrotne, szybkie klikanie) czy "dead clicks" (klikanie na nieaktywne elementy).

Oba rynki się przenikają – Sentry dodało podstawowe powtórki sesji, a LogRocket oferuje śledzenie błędów. Jednak ich fundamentalne przeznaczenie jest inne. Sentry to narzędzie diagnostyki technicznej (stack trace), podczas gdy LogRocket to narzędzie kontekstu wizualnego (wideo).

Tabela: Porównanie Sentry vs. LogRocket 2025

Cecha	**Sentry**	**LogRocket**
Główne Przeznaczenie	Śledzenie błędów i wydajności	Powtórki sesji i analityka UX
Kluczowa Funkcjonalność	Grupowanie błędów, stack trace, alerty	Odtwarzanie wideo sesji, mapy ciepła
Idealne do...	Debugowania problemów	Debugowania problemów

	technicznych, monitorowania backendu	wizualnych/UX, analizy ścieżek użytkownika
Model Cenowy (głównie)	Płatność za liczbę zdarzeń (błędów)	Płatność za liczbę sesji (nagrań)
Wsparcie dla Backendu	Szerokie (Node, Python, Java, etc.)	Ograniczone (skupienie na frontendzie)

Rekomendacja: Zawsze zaczynaj od wdrożenia Sentry. Jest to fundamentalne narzędzie do utrzymania "zdrowia" aplikacji. LogRocket dodaje się później, gdy potrzebny jest głębszy wgląd w zachowanie użytkownika lub gdy błędy zgłaszane przez Sentry są trudne do odtworzenia.

Przykład praktyczny: Integracja Sentry z React

Dodanie Sentry do React jest proste i składa się z dwóch kluczowych kroków:

Krok 1: Instalacja i inicjalizacja
npm install @sentry/react
W main.jsx (lub index.js), *przed* wywołaniem ReactDOM.createRoot:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import * as Sentry from "@sentry/react";

// 1. Inicjalizacja Sentry
Sentry.init({
  dsn: "TWÓJ_KLUCZ_DSN_Z_PANELU_SENTRY",
  integrations:,
  // Ustawienie próbkowania (sampling)
  tracesSampleRate: 1.0, // 100% transakcji do monitorowania wydajności
  replaysSessionSampleRate: 0.1, // 10% sesji będzie nagrywane
});

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

Krok 2: Konfiguracja Error Boundary (Kluczowe!)
Inicjalizacja Sentry przechwytuje globalne błędy, ale nie przechwyci błędów renderowania wewnątrz komponentów React (ponieważ React sam je łapie). Aby Sentry mogło je zobaczyć, musimy owinąć naszą aplikację w dostarczony przez Sentry komponent ErrorBoundary.
W App.jsx (lub tam, gdzie znajduje się główny router):

```
import * as Sentry from "@sentry/react";
import MyRoutes from './MyRoutes'; // Przykładowy komponent z routingiem

function App() {
 return (
  // 2. Owiń całą aplikację (lub jej kluczowe części)
  <Sentry.ErrorBoundary
   fallback={<p>Wystąpił nieoczekiwany błąd. Nasz zespół został powiadomiony.</p>}
  >
   <MyRoutes />
  </Sentry.ErrorBoundary>
 );
}

export default App;
```

Teraz, jeśli jakikolwiek komponent potomny MyRoutes wyrzuci błąd podczas renderowania, Sentry.ErrorBoundary przechwyci go, wyświetli użytkownikowi interfejs awaryjny (fallback) i automatycznie wyśle pełny raport z komponentowym stack trace do Sentry.

Wprowadzenie do Progressive Web App (PWA)

Definicja i korzyści

Progressive Web App (PWA) to nie jest konkretna technologia, ale zestaw standardów i praktyk, które pozwalają aplikacji internetowej naśladować zachowanie aplikacji natywnej (mobilnej lub desktopowej).

Kluczowe korzyści to:

1. **Instalowalność:** Użytkownik może "dodać do ekranu głównego". Aplikacja uruchamia się wtedy we własnym oknie, bez paska adresu przeglądarki, dając wrażenie aplikacji natywnej.

2. **Wsparcie Offline (Offline-first):** Dzięki technologii zwanej **Service Worker**, aplikacja może buforować swoje zasoby (HTML, CSS, JS, obrazy, a nawet dane API). Pozwala to na uruchomienie aplikacji i przeglądanie treści nawet bez połączenia z internetem.

Przykład praktyczny: Konfiguracja vite-plugin-pwa

Ręczne tworzenie Service Workera i manifestu aplikacji jest złożone. W ekosystemie Vite mamy do dyspozycji wtyczkę vite-plugin-pwa, która automatyzuje 90% tego procesu.

Krok 1: Instalacja
npm install vite-plugin-pwa -D
Krok 2: Konfiguracja vite.config.js
Plugin dodajemy do pliku konfiguracyjnego Vite, definiując w nim manifest aplikacji oraz strategię aktualizacji.

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'
import { VitePWA } from 'vite-plugin-pwa'

export default defineConfig({
  plugins:

    // Definicja Web App Manifest
    // To ten plik mówi przeglądarce, jak aplikacja ma wyglądać po "instalacji"
    manifest: { //
      name: 'Moja Aplikacja PWA',
      short_name: 'MojaPWA',
      description: 'Opis mojej progresywnej aplikacji webowej.',
      theme_color: '#ffffff', // Kolor paska narzędzi aplikacji
      icons: [ // Ikony używane na ekranie głównym
        {
          src: 'pwa-192x192.png', // Plik musi znajdować się w folderze /public
          sizes: '192x192',
          type: 'image/png'
        },
        {
          src: 'pwa-512x512.png', // Plik musi znajdować się w folderze /public
          sizes: '512x512',
          type: 'image/png'
        }
      ]
    },
```

```
    // Opcjonalnie: konfiguracja Service Workera (Workbox)
    // Domyślnie Workbox buforuje wszystkie statyczne zasoby
    workbox: {
      globPatterns: ['**/*.{js,css,html,ico,png,svg}'] // Buforuj te pliki
    }
   })
 ]
})
```

Krok 3: Dodanie ikon
Należy utworzyć wymagane ikony (np. pwa-192x192.png) i umieścić je w folderze /public.
Po wykonaniu npm run build, wtyczka vite-plugin-pwa automatycznie :

1. Wygeneruje plik manifest.webmanifest na podstawie obiektu manifest.
2. Wygeneruje plik Service Workera (sw.js) używając biblioteki Google Workbox, który będzie buforował zasoby zdefiniowane w workbox.globPatterns.
3. Wstrzyknie odpowiednie tagi <link rel="manifest"> oraz skrypt rejestrujący Service Worker do pliku index.html.

Podsumowanie i Audyt: Analiza Raportu Lighthouse

Audyt wdrożonej aplikacji

Po wdrożeniu aplikacji na Vercel (lub inną platformę) musimy zamknąć pętlę i zmierzyć jakość naszej pracy. Podstawowym narzędziem do tego celu jest **Lighthouse**, zintegrowany z narzędziami deweloperskimi Chrome (DevTools).

Aby przeprowadzić audyt:

1. Otwórz wdrożoną aplikację w przeglądarce Chrome (najlepiej w trybie Incognito, aby rozszerzenia nie zakłócały wyników).
2. Otwórz DevTools (F12).
3. Przejdź do zakładki "Lighthouse".
4. Wybierz kategorie (np. Performance, Accessibility, SEO) i tryb (Mobile).
5. Kliknij "Analyze page load".

Interpretacja wyników Lighthouse

Lighthouse ocenia stronę w pięciu kluczowych kategoriach, wystawiając notę od 0 do 100 (gdzie 90+ to wynik doskonały).

1. **Performance (Wydajność):** Najważniejsza i najbardziej złożona kategoria. Mierzy, jak szybko strona jest *postrzegana* przez użytkownika. Skupia się na trzech **Core Web Vitals** (Kluczowych Wskaźnikach Internetowych), które są również czynnikiem rankingowym Google :
   - **LCP (Largest Contentful Paint):** *Wydajność ładowania.* Jak szybko na ekranie pojawia się największy element (np. główny obrazek, blok tekstu)?.
   - **TBT (Total Blocking Time):** *Interaktywność.* Jak długo główny wątek przeglądarki jest "zablokowany" przez JavaScript, uniemożliwiając reakcję na kliknięcie użytkownika?.
   - **CLS (Cumulative Layout Shift):** *Stabilność wizualna.* Czy elementy strony "skaczą" podczas ładowania (np. reklama ładuje się i przesuwa tekst, na który użytkownik chciał kliknąć)?.
2. **Accessibility (Dostępność):** Sprawdza, czy strona jest użyteczna dla osób z niepełnosprawnościami (np. czy obrazki mają atrybuty alt, czy kontrast kolorów jest wystarczający).
3. **Best Practices (Najlepsze Praktyki):** Audyt techniczny (np. czy strona używa HTTPS, czy nie ma błędów w konsoli).
4. **SEO (Optymalizacja dla Wyszukiwarek):** Podstawowe sprawdzenie, czy strona ma elementy kluczowe dla SEO (np. tag <title>, <meta description>).
5. **PWA (Progressive Web App):** (Pojawi się, jeśli wykryje Service Worker). Sprawdza, czy manifest jest poprawny, czy aplikacja działa offline i czy jest instalowalna.

Wynik Lighthouse (np. "Performance: 60") nie jest wyrokiem, ale początkiem analizy. Prawdziwa wartość tego narzędzia leży poniżej wyników, w sekcjach "Opportunities" (Szanse) i "Diagnostics" (Diagnostyka). Lighthouse to zaawansowane narzędzie diagnostyczne. Mówi nam *dokładnie*, co spowalnia naszą stronę (np. "Zoptymalizuj obrazy", "Usuń zasoby blokujące renderowanie") i dostarcza listę konkretnych zadań do wykonania, aby poprawić doświadczenie użytkownika.

Literatura

1. https://www.apollographql.com/docs/react/get-started (Data dostępu: 1.10.2025) – Oficjalny podręcznik konfiguracji Apollo Client, szczegółowo wyjaśniający mechanizmy buforowania (InMemoryCache) i zarządzania zapytaniami GraphQL.
2. https://auth0.com/blog/refresh-tokens-what-are-they-and-when-to-use-them/ (Data dostępu: 1.10.2025) – Specjalistyczna analiza mechanizmów odświeżania tokenów (Refresh Tokens), stanowiąca fundament dla omawianego na wykładzie hybrydowego modelu uwierzytelniania.
3. https://docs.github.com/en/actions/use-cases-and-examples/building-and-testing/building-and-testing-nodejs (Data dostępu: 1.10.2025) – Dokumentacja techniczna GitHub Actions, opisująca procesy CI/CD, automatyzację testów oraz budowanie artefaktów produkcyjnych dla środowisk Node.js.
4. https://docs.sentry.io/platforms/javascript/guides/react/ (Data dostępu: 1.10.2025) – Przewodnik integracji Sentry z aplikacjami React, omawiający zaawansowaną diagnostykę błędów, konfigurację Error Boundary oraz monitorowanie wydajności.
5. https://web.dev/articles/vitals (Data dostępu: 1.10.2025) – Kompendium wiedzy na temat Core Web Vitals (LCP, FID, CLS), niezbędne do interpretacji wyników audytu Lighthouse i optymalizacji doświadczenia użytkownika.