Notatka

Autor: Oliwier Markiewicz

Planowanie biznesowe frontendu

Cel biznesowy

Zadaj sobie pytania:

- **Co aplikacja ma osiągnąć? Czy służy pozyskaniu klientów, sprzedaży, edukacji, czy np. wsparciu obsługi klienta?**
- **Czy jest to:**
  - **MVP (Minimum Viable Product) – szybka walidacja pomysłu?**
  - **Landing page – czysto marketingowa funkcja, bez backendu?**
  - **Aplikacja produkcyjna / SaaS – z wieloma ekranami, kontami użytkowników, integracjami?**

Dlaczego to ważne:

Cel wpływa bezpośrednio na decyzje techniczne – stack, testy, dostępność, CI/CD, performance.

Grupa docelowa

Pytania pomocnicze:

- **Kto konkretnie będzie używał aplikacji? (np. konsumenci B2C, firmy B2B, dzieci, seniorzy, programiści)**
- **Jaki poziom umiejętności technologicznych mają użytkownicy?**
- **Czy korzystają głównie z telefonów, laptopów, czy np. przeglądarek korporacyjnych?**
- **Czy będą potrzebować wsparcia dostępności (np. osoby niedowidzące, poruszające się tylko klawiaturą)?**

Dlaczego to ważne:

Użytkownicy wpływają na UX, dostępność, używane technologie (np. PWA, obsługa IE11, niskie wymagania sprzętowe).

Kluczowe funkcjonalności

Zdefiniuj:

- **Jakie funkcje muszą działać w wersji początkowej (np. rejestracja, logowanie, lista produktów)?**
- **Które funkcje są „miłe do posiadania", ale mogą poczekać?**
- **Co można wdrożyć w kolejnych iteracjach (np. czat, wersja mobilna offline, rekomendacje AI)?**

Wskazówka:

Warto stworzyć roadmapę funkcjonalności z podziałem:

- **Krytyczne (v1)**
- **Przydatne (v1.1 / v2)**
- **Premium / innowacje (v2+)**

Budżet i zasoby

Zastanów się:

- **Ilu ludzi będzie pracować nad projektem? (frontend, backend, design, QA)**
- **Ile mamy czasu na dostarczenie MVP / pełnej wersji?**
- **Czy CI/CD i automatyczne testy są potrzebne od razu, czy później?**
- **Czy projekt będzie utrzymywany długoterminowo czy to jednorazowy release?**
- **Czy budżet pozwala na testy, animacje, refaktoryzację?**

Dlaczego to ważne:

Pozwoli dobrać skalę narzędzi i praktyk. Czasem prosty setup wystarczy, a czasem warto od razu wdrożyć pełną infrastrukturę.

Wymagania techniczne

Zdefiniuj:

- **Czy frontend łączy się z istniejącym API? A może trzeba je zbudować od zera?**
- **Jak wygląda API? Czy mamy dokumentację? Czy będzie GraphQL, REST, czy coś własnego?**
- **Czy aplikacja musi wspierać wiele języków? (i18n, lokalizacja walut, daty, formaty)**
- **Czy musimy dbać o różne strefy czasowe, np. eventy online, daty rezerwacji?**
- **Czy aplikacja będzie działać offline? (PWA, cache)**

Warto dodać:

- **Czy aplikacja będzie używana w przeglądarce, czy też jako WebView / appka mobilna?**
- **Czy potrzebne są integracje z zewnętrznymi systemami (Stripe, Firebase, Google API, itd.)?**

Na koniec – czemu to takie ważne?

Dokładne przemyślenie powyższych kwestii:

- **pozwala dobrać odpowiednią architekturę frontendu (modularność, skalowalność)**
- **ułatwia komunikację z zespołem i interesariuszami**
- **chroni przed kosztownym refaktorem w połowie projektu**
- **tworzy solidny fundament pod przyszły rozwój, marketing i monetyzację**

⟹ TOP 10 praktyk, które powinieneś znać

1. Struktura projektu i organizacja kodu

Dlaczego?
Czytelna struktura pozwala szybko odnaleźć się w projekcie Tobie i każdemu innemu devowi.

📁 Rekomendowana struktura:

```
src/
├── app/          # konfiguracja globalna, router
├── pages/        # strony / route-level components
├── features/     # konkretne funkcjonalności (np. login, cart)
├── entities/     # elementy wielokrotnego użycia (np. UserCard)
├── shared/       # utils, hooki, komponenty wspólne
└── widgets/      # UI elementy złożone, ale współdzielone
```

2. TypeScript i typowanie

Po co?
TypeScript wykrywa błędy **zanim klikniesz „Start dev server”**.

Dobre praktyki:

- Typuj propsy: type ButtonProps = { onClick: () => void }
- Typuj API response: interface ProductResponse { id: string; price: number }
- Zamiast any, używaj: unknown, Record<string, unknown>, Partial<T>, Pick<T, K>

3. Custom hooki – separacja logiki od widoku

Dlaczego?

Trzymanie fetchy, logiki i eventów w komponentach to droga do chaosu.

```
// useProducts.ts

export const useProducts = () => {

 const { data, error, isLoading } = useSWR("/api/products", fetcher)

 return { data, error, isLoading }

}
```

A w komponencie:

```
const ProductList = () => {

 const { data, isLoading } = useProducts()

 // ...

}
```

4. Design Patterns w React

Po co?

Wzorce projektowe to **sprawdzone sposoby budowy skalowalnych aplikacji**.

Warto znać:

- **Compound Components** – np. Tabs, Accordion, Modal
- **Container/Presentational** – logika vs. UI
- **Render Props** – elastyczne przekazywanie logiki
- **Custom Hooks + Context** – współdzielona logika
- **State Machines (XState)** – pełna kontrola flowów

5. Testowanie aplikacji

Dlaczego?

Testy ratują przed bugami.

Rodzaje testów:

- **Unit tests (Vitest, Jest)** – funkcje, hooki
- **Component tests (React Testing Library)** – zachowania UI
- **E2E tests (Playwright, Cypress)** – cały user flow

6. Styl i jakość kodu

Po co?
Ujednolicenie stylu kodu to mniej błędów, lepszy teamwork i czystsze PR-y.

Jak to wdrożyć:

- ESLint + Prettier
- Husky (pre-commit hooki)
- Reguły nazewnictwa, zakaz magicznych stringów/liczb

7. Hosting

Gdzie hostować?

- **Vercel** – szybki deploy, preview PR
- **Netlify** – bardzo dobry dla SPA
- **Firebase Hosting** – lekki, prosty

🪄 **Dodatki:**

- Auto deploy z GitHub
- SSL za darmo
- Obsługa 404 / SPA fallback

8. CI/CD i Docker – automatyczne testowanie i wdrażanie

Po co?
Automatyzacja = brak ręcznego deployowania i błędów przez zapomnienie. Docker - tworzy kontener.

Pipeline przykład (GitHub Actions):

- **lint**
- **test**
- **build**
- **Deploy do Vercel/Netlify/AWS**
- **Publikacja obrazu Dockera**

9. UX i dostępność (a11y)

Dlaczego warto?
Użytkownicy mają różne potrzeby – zadbaj o każdego.

Co warto wdrożyć:

- **Skeletony i fallbacki (nie biały ekran)**
- **aria-labels, role, focus states**
- **Kontrast kolorów, duże klikane obszary**

10. Bezpieczeństwo frontendu

Co trzeba wiedzieć?

- **Nie trzymaj secretów w kodzie (.env!)**
- **Waliduj wszystkie dane wejściowe**
- **Zabezpiecz przed XSS, CSRF, CORS**
- **Używaj narzędzi jak helmet, Content-Security-Policy**

⟹ Co robić krok po kroku (plan wdrożenia)

1. **Ustal strukturę projektu**
   ➤ Podziel projekt wg. features/, shared/, entities/
2. **Wprowadź TypeScript (jeśli jeszcze nie masz)**
   ➤ Zainstaluj TS, przekonwertuj .js → .tsx, dodaj typy
3. **Stwórz swój pierwszy custom hook**
   ➤ Wyodrębnij logikę fetchowania lub formularza
4. **Dodaj ESLint + Prettier + Husky**
   ➤ Automatyczne sprawdzanie i formatowanie kodu
5. **Wdróż jeden pattern designu**
   ➤ Np. zamień duży komponent w Compound Component
6. **Dodaj testy**
   ➤ Zacznij od jednego testu do komponentu i jednego do hooka
7. **Skonfiguruj Docker i DockerHub**
   ➤ Utwórz konto na docker hub oraz obraz dockera
8. **Skonfiguruj CI/CD (np. GitHub Actions)**
   ➤ Testy + build + deploy na push/PR
9. **Zdeplouj projekt (np. na Vercel)**
   ➤ Podłącz repo, ustaw preview builds
10. **Popraw UX – loadingi, dostępność, aria**
    ➤ Dodaj Skeleton, popraw kontrast, przetestuj klawiaturą
11. **Zadbaj o bezpieczeństwo**
    ➤ Ukryj API klucze, waliduj formularze

⟹ 🎁 BONUSY

Checklista: najlepsze praktyki w tworzeniu nowoczesnego frontendu

Struktura projektu

- Projekt podzielony na features, shared, entities, pages, widgets
- Komponenty zorganizowane według funkcjonalności, nie typu pliku
- Brak „wszystkiego w jednym folderze”

TypeScript

- Każdy komponent i funkcja ma typy
- API response’y opisane przez interfejsy
- Używane Partial<>, Pick<>, Record<> zamiast any

Hooki i logika aplikacji

- Logika wyciągnięta z komponentów do hooków
- Hooki odpowiedzialne za API, formularze, stan lokalny
- Nie trzymasz fetchów bezpośrednio w JSX

Design Patterns

- Stosowane wzorce: Compound Components, Container/Presentational
- Zastosowanie Context + Custom Hooks
- Myślenie o flow aplikacji w kategorii state machine'ów

Testowanie

- Są testy jednostkowe dla funkcji i hooków
- Są testy komponentów (interakcje, renderowanie)
- Są testy E2E (np. logowanie, dodanie do koszyka)

Styl i jakość kodu

- Projekt skonfigurowany z ESLint i Prettier
- Formatowanie i lintowanie automatyczne (pre-commit)
- Unikasz magicznych wartości i duplikacji

Hosting

- Projekt zdeployowany na Vercel, Netlify lub Firebase
- Obsługa wersji preview (PR = osobny link)
- Działa fallback na 404 i SPA routing

CI/CD

- Używasz GitHub Actions / GitLab CI / CircleCI
- Pipeline zawiera lint, testy i build
- Oddzielne środowiska: staging / production

UX i dostępność

- Dodane aria-label, role, focus states
- Stany ładowania (skeletony, spinnery)
- Odpowiedni kontrast i dostępność klawiaturowa

Bezpieczeństwo

- Nie ma kluczy ani secretów w kodzie
- Walidacja danych wejściowych (formy, API)
- Zabezpieczenia przed XSS, CSRF i CORS

Rekomendowane narzędzia i zasoby

- **Playground do testowania hooków:** usehooks.com
- **Design Patterns w React:** 🌐 5 tricków w React, o których mogłeś nie słyszeć !
- **GitHub Actions Starter:** github.com/actions/starter-workflows
- **Testing cheatsheet:** testing-library.com/docs/cheatsheet
- **Tailwind UI Patterns:** tailwindcomponents.com