Aplikacja TODO w Next.js

Przewodnik tworzenia nowoczesnej aplikacji webowej

Wprowadzenie do Next.js, React, TypeScript i Prisma

13 listopada 2025

Streszczenie

Niniejszy dokument stanowi kompleksowy przewodnik po tworzeniu aplikacji webowej do zarządzania listą zadań (TODO) przy użyciu frameworka Next.js. Materiał przeznaczony jest dla studentów informatyki rozpoczynających naukę programowania aplikacji webowych. Dokument omawia architekturę aplikacji, wykorzystane technologie oraz szczegółową implementację wszystkich komponentów systemu.

Aplikacja została zbudowana w oparciu o najnowsze standardy i najlepsze praktyki programowania webowego, wykorzystując React 19, Next.js 16, TypeScript oraz bazę danych SQLite z ORM Prisma. Przewodnik krok po kroku wyjaśnia budowę aplikacji od podstaw, omawiając zarówno aspekty teoretyczne, jak i praktyczne implementacji.

Spis treści

1 Wprowadzenie

1.1 Cel dokumentu

Niniejszy dokument ma na celu przedstawienie procesu tworzenia nowoczesnej aplikacji webowej od podstaw. Materiał przeznaczony jest dla studentów informatyki, którzy rozpoczynają swoją przygodę z programowaniem aplikacji internetowych. Dokument szczegółowo omawia wszystkie aspekty budowy aplikacji do zarządzania listą zadań (ang. *TODO list*), od konfiguracji projektu, przez implementację logiki biznesowej, aż po stylowanie interfejsu użytkownika.

1.2 Czym jest aplikacja webowa?

Aplikacja webowa to program komputerowy, który działa w przeglądarce internetowej i umożliwia użytkownikowi interakcję poprzez interfejs graficzny dostępny za pośrednictwem protokołu HTTP/HTTPS. W przeciwieństwie do tradycyjnych aplikacji desktopowych, aplikacje webowe nie wymagają instalacji na komputerze użytkownika - wystarczy przeglądarka internetowa.

Współczesne aplikacje webowe można podzielić na kilka kategorii:

- **Aplikacje Single Page Application (SPA)** - aplikacje działające w ramach jednej strony HTML, gdzie zawartość jest dynamicznie aktualizowana bez przeładowania całej strony. Przykłady: Gmail, Facebook, Twitter.
- **Aplikacje z renderowaniem po stronie serwera (SSR - Server-Side Rendering)** - aplikacje, w których HTML jest generowany na serwerze i wysyłany do przeglądarki. Zapewnia to lepsze SEO oraz szybszy pierwszy rendering strony.
- **Aplikacje hybrydowe** - łączą zalety SPA i SSR, umożliwiając renderowanie niektórych stron na serwerze, a innych w przeglądarce. Taki model został zastosowany w prezentowanej aplikacji.

1.3 Architektura full-stack

Aplikacja full-stack składa się z dwóch głównych warstw:

1. **Frontend (warstwa prezentacji)** - część aplikacji działająca w przeglądarce użytkownika, odpowiedzialna za wyświetlanie interfejsu graficznego i obsługę interakcji użytkownika. W prezentowanej aplikacji frontend został zbudowany przy użyciu biblioteki React oraz frameworka Next.js.
2. **Backend (warstwa serwera)** - część aplikacji działająca na serwerze, odpowiedzialna za przetwarzanie logiki biznesowej, dostęp do bazy danych oraz obsługę żądań HTTP. W naszej aplikacji backend został zrealizowany przy użyciu mechanizmu *Server Actions* frameworka Next.js.

1.4 Omówienie aplikacji

Prezentowana aplikacja TODO to pełnofunkcjonalna aplikacja webowa umożliwiająca zarządzanie listą zadań. Użytkownik może:

- Dodawać nowe zadania
- Edytować istniejące zadania
- Oznaczać zadania jako ukończone
- Usuwać zadania z listy
- Przeglądać listę wszystkich zadań posortowanych według daty utworzenia

Aplikacja została zbudowana w oparciu o nowoczesne technologie webowe:

- **Next.js 16** - framework React do budowy aplikacji webowych
- **React 19** - biblioteka JavaScript do budowy interfejsów użytkownika
- **TypeScript 5** - typowany nadzbiór JavaScript zapewniający bezpieczeństwo typów
- **Prisma ORM** - narzędzie do zarządzania bazą danych
- **SQLite** - lekka baza danych SQL
- **CSS Modules** - system stylowania zapewniający izolację styli

1.5 Wymagania wstępne

Aby móc samodzielnie zbudować i uruchomić prezentowaną aplikację, student powinien posiadać:

1. Podstawową znajomość języka JavaScript
2. Podstawową znajomość HTML i CSS
3. Zainstalowane środowisko Node.js (w wersji 18 lub nowszej)
4. Zainstalowany menedżer pakietów npm lub pnpm
5. Edytor kodu (np. Visual Studio Code, WebStorm)
6. Przeglądarkę internetową (np. Chrome, Firefox, Edge)

1.6 Struktura dokumentu

Dokument został podzielony na rozdziały omawiające poszczególne aspekty aplikacji:

- Rozdział 2 omawia wykorzystane technologie i ich rolę w aplikacji
- Rozdział 3 przedstawia strukturę projektu i organizację plików
- Rozdział 4 opisuje pliki konfiguracyjne projektu
- Rozdział 5 omawia model danych i integrację z bazą danych
- Rozdział 6 wyjaśnia system routingu w Next.js

- Rozdziały 7-8 szczegółowo opisują komponenty React
- Rozdział 9 omawia implementację logiki biznesowej
- Rozdział 10 przedstawia stylowanie aplikacji
- Rozdział 11 wyjaśnia proces uruchamiania i developmentu

2 Wykorzystane technologie

Niniejszy rozdział omawia technologie wykorzystane w projekcie oraz ich wzajemne relacje i role w architekturze aplikacji.

2.1 Next.js 16

Next.js to framework React stworzony przez firmę Vercel, umożliwiający budowę aplikacji webowych z renderowaniem po stronie serwera (SSR), generowaniem stron statycznych (SSG) oraz hybrydowym podejściem łączącym obie metody.

2.1.1 App Router

W wersji 13 Next.js wprowadził nowy system routingu o nazwie *App Router*, który zastąpił poprzedni *Pages Router*. Nasza aplikacja wykorzystuje właśnie App Router, który oferuje następujące korzyści:

- **File-system based routing** - routing oparty na strukturze plików i katalogów
- **Server Components** - możliwość renderowania komponentów wyłącznie po stronie serwera
- **Nested Layouts** - zagnieżdżone layouty współdzielone między stronami
- **Server Actions** - funkcje uruchamiane po stronie serwera, wywoływane z komponentów klienckich
- **Streaming** - progresywne renderowanie elementów strony

2.1.2 Kluczowe koncepcje Next.js wykorzystane w projekcie

Renderowanie po stronie serwera (SSR) Strona główna aplikacji (`app/page.tsx`) jest renderowana po stronie serwera. Oznacza to, że HTML jest generowany na serwerze przy każdym żądaniu, a następnie wysyłany do przeglądarki. Zapewnia to:

- Lepsze SEO (wyszukiwarki otrzymują gotowy HTML)
- Szybsze pierwsze wyświetlenie treści (FCP - First Contentful Paint)
- Dostęp do aktualnych danych z bazy przy każdym renderowaniu

Server Actions Server Actions to funkcje uruchamiane po stronie serwera, które mogą być bezpośrednio wywoływane z komponentów klienckich. Eliminuje to konieczność tworzenia osobnych endpointów API. W naszej aplikacji wszystkie operacje CRUD (Create, Read, Update, Delete) są zrealizowane jako Server Actions w pliku `app/actions.ts`.

2.2 React 19

React to biblioteka JavaScript stworzona przez firmę Meta (dawniej Facebook) do budowy interfejsów użytkownika. React wprowadza koncepcję komponentów - samodzielnych, wielokrotnego użytku bloków UI.

2.2.1 Server Components vs Client Components

React 19 wprowadza nową dychotomię komponentów:

- **Server Components** (domyślnie w App Router) - komponenty renderowane wyłącznie na serwerze. Nie zawierają JavaScript w bundle'u klienta, co zmniejsza rozmiar aplikacji. Mogą bezpośrednio komunikować się z bazą danych.
- **Client Components** (oznaczone dyrektywą `'use client'`) - komponenty renderowane zarówno na serwerze (initial render), jak i w przeglądarce. Mogą korzystać z hooków React oraz obsługiwać interakcje użytkownika (kliknięcia, wprowadzanie tekstu itp.).

W naszej aplikacji:

- Server Components: `app/page.tsx`, `app/layout.tsx`, `app/add/page.tsx`, `app/edit/[id]/page.
- Client Components: `app/components/TodoItem.tsx`, `app/components/TodoForm.tsx`

2.2.2 Hooki React wykorzystane w projekcie

useActionState Hook wprowadzony w React 19, umożliwiający obsługę Server Actions w formularzach. Zwraca stan formularza oraz funkcję akcji. Wykorzystany w komponencie `TodoForm.tsx` do obsługi dodawania i edycji zadań.

Przykład użycia:

```
1 const [state, formAction] = useActionState(saveTodo, initialState);
```

Listing 1: Wykorzystanie useActionState w TodoForm

2.3 TypeScript 5

TypeScript to typowany nadzbiór JavaScript, który kompiluje się do czystego JavaScript. Wprowadza statyczne typowanie, co pozwala na wykrywanie błędów już na etapie kompilacji, a nie w runtime.

2.3.1 Korzyści z TypeScript

1. **Bezpieczeństwo typów** - kompilator wykrywa niezgodności typów przed uruchomieniem aplikacji
2. **Lepsze podpowiedzi IDE** - edytor kodu zna typy zmiennych i może oferować dokładne autocomplete
3. **Dokumentacja w kodzie** - typy służą jako dokumentacja interfejsów i funkcji
4. **Łatwiejszy refactoring** - IDE może bezpiecznie zmienić nazwę zmiennej we wszystkich miejscach

2.3.2 Przykłady typowania w projekcie

W projekcie wykorzystano różne konstrukcje TypeScript:

- **Typy interfejsów** - definiowanie kształtu obiektów (np. `TodoItemProps`)
- **Typy generowane przez Prisma** - automatycznie generowane typy modeli bazy danych (np. `Todo`)
- **Typy zwracane przez funkcje** - np. `Promise<FormState>`
- **Typy pomocnicze** - np. `type FormState` w `actions.ts`

2.4 Prisma ORM

Prisma to nowoczesny ORM (Object-Relational Mapping) dla Node.js i TypeScript. ORM to narzędzie umożliwiające pracę z bazą danych przy użyciu obiektów w kodzie, zamiast pisania czystego SQL.

2.4.1 Zalety Prisma

- **Type-safe database access** - pełne typowanie zapytań do bazy danych
- **Migracje** - automatyczne generowanie i zarządzanie migracjami schematu bazy
- **Intuitive API** - prostsze zapytania niż czysty SQL
- **Prisma Studio** - graficzny interfejs do przeglądania i edycji danych

2.4.2 Komponenty Prisma

1. **Prisma Schema** (`prisma/schema.prisma`) - deklaratywna definicja modeli bazy danych
2. **Prisma Client** - automatycznie generowany, typowany klient bazy danych
3. **Prisma Migrate** - system migracji bazy danych

2.5 SQLite

SQLite to lekka, bezserwerowa baza danych SQL, która przechowuje dane w pojedynczym pliku na dysku. Idealnie nadaje się do aplikacji deweloperskich, prototypów oraz małych projektów.

2.5.1 Zalety SQLite w kontekście projektu

- Nie wymaga instalacji osobnego serwera bazy danych
- Łatwa konfiguracja i przenoszenie projektu
- Wystarczająca dla aplikacji o małym i średnim ruchu
- Pełna zgodność ze standardem SQL

W produkcyjnej aplikacji SQLite można łatwo zastąpić inną bazą danych (PostgreSQL, MySQL) zmieniając jedynie konfigurację Prisma.

2.6 Zod

Zod to biblioteka do walidacji danych i tworzenia schematów w TypeScript. Umożliwia deklaratywne definiowanie kształtu danych oraz walidację wartości w runtime.

2.6.1 Wykorzystanie Zod w projekcie

W pliku `app/actions.ts` zdefiniowano schemat walidacji zadania:

```
const todoSchema = z.object({
  text: z.string()
    .min(1, 'Treść zadania jest wymagana')
    .max(200, 'Treść zadania jest za długa (max 200 znaków)')
});
```

Listing 2: Schemat walidacji Zod

Schemat ten zapewnia, że pole `text` jest niepustym ciągiem znaków o długości nie przekraczającej 200 znaków. W przypadku niepowodzenia walidacji, zwracane są odpowiednie komunikaty błędów.

2.7 CSS Modules

CSS Modules to system stylowania, który automatycznie generuje unikalne nazwy klas CSS, zapobiegając konfliktom nazw. Każdy plik `.module.css` jest traktowany jako osobny moduł.

2.7.1 Zalety CSS Modules

- **Scoped styles** - style są automatycznie ograniczone do komponentu
- **Brak konfliktów nazw** - unikalne nazwy klas generowane automatycznie
- **Explicit dependencies** - komponenty jawnie importują swoje style
- **Eliminacja martwego kodu** - nieużywane style można łatwo wykryć

2.7.2 Przykład użycia

```
import styles from './page.module.css';

// Użycie w JSX
<div className={styles.container}>
  <h1>Lista zadań</h1>
</div>
```

Listing 3: Import i użycie CSS Modules

2.8 ESLint

ESLint to narzędzie do statycznej analizy kodu JavaScript/TypeScript, które pomaga wykrywać problemy i egzekwować spójny styl kodowania. Projekt wykorzystuje konfigurację `eslint-config-next`, która zawiera zestaw reguł zalecanych dla projektów Next.js.

3 Struktura projektu

Niniejszy rozdział przedstawia organizację plików i katalogów w projekcie. Zrozumienie struktury projektu jest kluczowe dla efektywnej pracy z aplikacją Next.js.

3.1 Drzewo katalogów

Poniżej przedstawiono kompletną strukturę projektu (z pominięciem katalogów `node_modules` oraz `.next`):

```
nextjs-todo/
|
+-- app/                    # Katalog glowny aplikacji (App Router)
|   |
|   +-- components/         # Komponenty React wielokrotnego uzytku
|   |   +-- TodoForm.tsx
|   |   +-- TodoForm.module.css
|   |   +-- TodoItem.tsx
|   |   +-- TodoItem.module.css
|   |
|   +-- add/                # Strona dodawania zadania
|   |   +-- page.tsx
|   |   +-- add.module.css
|   |
|   +-- edit/               # Strona edycji zadania
|   |   +-- [id]/           # Dynamic route (parametr id)
|   |       +-- page.tsx
|   |       +-- edit.module.css
|   |
|   +-- actions.ts          # Server Actions (CRUD)
|   +-- layout.tsx          # Root Layout aplikacji
|   +-- page.tsx            # Strona glowna (/)
|   +-- page.module.css     # Style strony glownej
|   +-- globals.css         # Globalne style CSS
|   +-- favicon.ico         # Ikona aplikacji
|
+-- lib/                    # Pliki pomocnicze
|   +-- prisma.ts           # Singleton Prisma Client
|   +-- formatDate.ts       # Funkcja formatowania dat
|   +-- formData.ts         # Narzedzia do obslugi FormData
|
+-- prisma/                 # Konfiguracja bazy danych
|   +-- schema.prisma       # Schemat modeli danych
|   +-- dev.db              # Plik bazy SQLite (development)
|   +-- migrations/         # Historia migracji bazy
|
+-- public/                 # Zasoby statyczne
+-- package.json            # Zaleznosci i skrypty npm
+-- package-lock.json       # Lock file (npm)
+-- pnpm-lock.yaml          # Lock file (pnpm)
+-- tsconfig.json           # Konfiguracja TypeScript
+-- next.config.ts          # Konfiguracja Next.js
```

```
+-- .env                    # Zmienne srodowiskowe
+-- .gitignore              # Pliki ignorowane przez Git
+-- README.md               # Dokumentacja projektu
```

Listing 4: Struktura katalogów projektu

3.2 Katalog app/

Katalog `app/` to rdzeń aplikacji Next.js wykorzystującej App Router. Jego struktura bezpośrednio wpływa na routing aplikacji.

3.2.1 Konwencje nazewnictwa

Next.js wykorzystuje specjalne nazwy plików do określania ich roli:

- `page.tsx` - definiuje stronę dostępną pod danym URL
- `layout.tsx` - definiuje layout współdzielony przez strony
- `loading.tsx` - komponent wyświetlany podczas ładowania (nieużywany w projekcie)
- `error.tsx` - komponent obsługi błędów (nieużywany w projekcie)
- `not-found.tsx` - strona 404 (nieużywana w projekcie)

3.2.2 Routing oparty na plikach

Struktura katalogów w `app/` definiuje routing:

Ścieżka pliku	**URL**
`app/page.tsx`	`/`
`app/add/page.tsx`	`/add`
`app/edit/[id]/page.tsx`	`/edit/1`, `/edit/2`, ...

Tabela 1: Mapowanie struktury plików na URL

3.2.3 Dynamic Routes

Katalog `app/edit/[id]/` reprezentuje *dynamic route* - trasę z parametrem dynamicznym. Notacja `[id]` oznacza, że wartość tego segmentu URL jest przekazywana jako parametr do komponentu strony.

Przykład: URL `/edit/5` zostanie obsłużony przez `app/edit/[id]/page.tsx`, a parametr `id` otrzyma wartość `"5"`.

3.3 Katalog app/components/

Katalog `app/components/` zawiera komponenty React wielokrotnego użytku. Każdy komponent składa się z pliku TypeScript (`.tsx`) oraz opcjonalnie pliku styli CSS Modules (`.module.css`).

3.3.1 TodoItem.tsx

Komponent wyświetlający pojedyncze zadanie na liście. Jest to komponent kliencki (`'use client'`), ponieważ obsługuje interakcje użytkownika (zaznaczanie checkbox, klikanie przycisków).

Odwołanie do pliku: `app/components/TodoItem.tsx:1-48`

3.3.2 TodoForm.tsx

Komponent formularza używany zarówno do dodawania, jak i edycji zadań. Działa w dwóch trybach: `'add'` oraz `'edit'`. Również jest komponentem klienckim, ponieważ wykorzystuje hook `useActionState`.

Odwołanie do pliku: `app/components/TodoForm.tsx:1-61`

3.4 Katalog lib/

Katalog `lib/` zawiera pliki pomocnicze i narzędziowe wykorzystywane w różnych częściach aplikacji.

3.4.1 lib/prisma.ts

Plik eksportujący singleton instancji Prisma Client. Singleton pattern zapewnia, że w całej aplikacji istnieje tylko jedna instancja klienta bazy danych, co jest szczególnie istotne w środowisku deweloperskim (Hot Module Replacement w Next.js mógłby tworzyć wiele połączeń).

Odwołanie do pliku: `lib/prisma.ts`

3.4.2 lib/formatDate.ts

Plik zawierający funkcję `formatDate()`, która formatuje obiekt `Date` do czytelnego ciągu znaków w formacie polskim (DD/MM/YYYY HH:mm).

Odwołanie do pliku: `lib/formatDate.ts`

3.4.3 lib/formData.ts

Plik zawierający funkcje pomocnicze do pracy z obiektami `FormData`, np. ekstrakcję i walidację wartości z formularzy.

3.5 Katalog prisma/

Katalog `prisma/` zawiera wszystkie pliki związane z bazą danych i ORM Prisma.

3.5.1 prisma/schema.prisma

Plik definiujący schemat bazy danych - modele, relacje, konfigurację połączenia. Jest to deklaratywny opis struktury bazy danych.

Odwołanie do pliku: `prisma/schema.prisma:1-16`

3.5.2 prisma/dev.db

Plik bazy danych SQLite używany w środowisku deweloperskim. Zawiera wszystkie dane aplikacji.

3.5.3 prisma/migrations/

Katalog zawierający historię migracji bazy danych. Każda migracja jest zapisana w osobnym podkatalogu z plikiem SQL opisującym zmiany w schemacie.

3.6 Pliki konfiguracyjne

3.6.1 package.json

Plik definiujący metadane projektu, zależności (dependencies) oraz skrypty npm.

Odwołanie do pliku: `package.json:1-30`

3.6.2 tsconfig.json

Plik konfiguracyjny kompilatora TypeScript. Definiuje opcje kompilacji, ścieżki aliasów, poziom ścisłości typowania itp.

3.6.3 next.config.ts

Plik konfiguracyjny Next.js. W podstawowej wersji projektu jest pusty (używa domyślnej konfiguracji), ale może zawierać zaawansowane opcje konfiguracyjne frameworka.

3.6.4 .env

Plik zmiennych środowiskowych. Zawiera konfigurację wrażliwą (np. connection string do bazy danych), która nie powinna być commitowana do repozytorium Git.

Przykładowa zawartość:

```
1 DATABASE_URL="file:./dev.db"
```

Listing 5: Przykładowy plik .env

3.7 Katalog public/

Katalog `public/` zawiera zasoby statyczne dostępne publicznie (obrazy, fonty, ikony itp.). Pliki z tego katalogu są serwowane pod ścieżką główną domeny.

4 Pliki konfiguracyjne projektu

Każdy projekt Next.js wymaga odpowiedniej konfiguracji. Niniejszy rozdział omawia kluczowe pliki konfiguracyjne i ich rolę w projekcie.

4.1 package.json

Plik `package.json` jest sercem każdego projektu Node.js. Zawiera metadane projektu, listę zależności oraz definicje skryptów npm.

```
{
  "name": "nextjs-todo",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "eslint",
    "db:migrate": "prisma migrate dev --create-only --name",
    "db:update": "prisma migrate deploy && prisma generate"
  },
  "dependencies": {
    "@prisma/client": "^6.19.0",
    "dotenv": "^17.2.3",
    "next": "16.0.1",
    "react": "19.2.0",
    "react-dom": "19.2.0",
    "zod": "^4.1.12"
  },
  "devDependencies": {
    "@types/node": "^20",
    "@types/react": "^19",
    "@types/react-dom": "^19",
    "eslint": "^9",
    "eslint-config-next": "16.0.1",
    "prisma": "^6.19.0",
    "typescript": "^5"
  }
}
```

Listing 6: Zawartość pliku package.json

Odwołanie do pliku: `package.json:1-30`

4.1.1 Sekcja scripts

Skrypty npm definiują komendy uruchamiające różne operacje:

- `dev` - uruchamia serwer deweloperski Next.js z hot reload na porcie 3000
- `build` - buduje aplikację do wersji produkcyjnej (optymalizacja, minifikacja)
- `start` - uruchamia serwer produkcyjny (wymaga wcześniejszego `build`)
- `lint` - uruchamia ESLint do analizy kodu

- `db:migrate` - tworzy nową migrację bazy danych (tylko plik SQL, bez wykonania)
- `db:update` - wykonuje migracje i generuje Prisma Client

4.1.2 Dependencies

Zależności produkcyjne (wymagane do uruchomienia aplikacji):

- `@prisma/client` (v6.19.0) - klient bazy danych Prisma
- `dotenv` (v17.2.3) - ładowanie zmiennych środowiskowych z pliku .env
- `next` (v16.0.1) - framework Next.js
- `react` (v19.2.0) - biblioteka React
- `react-dom` (v19.2.0) - renderer React dla przeglądarki
- `zod` (v4.1.12) - biblioteka walidacji danych

4.1.3 DevDependencies

Zależności deweloperskie (wymagane tylko podczas developmentu):

- `@types/*` - definicje typów TypeScript dla różnych pakietów
- `eslint` (v9) - linter JavaScript/TypeScript
- `eslint-config-next` (v16.0.1) - konfiguracja ESLint dla Next.js
- `prisma` (v6.19.0) - CLI Prisma (migracje, Prisma Studio)
- `typescript` (v5) - kompilator TypeScript

4.2 tsconfig.json

Plik `tsconfig.json` konfiguruje kompilator TypeScript. Definiuje opcje kompilacji, które wpływają na sposób tłumaczenia kodu TypeScript na JavaScript.

```
{
  "compilerOptions": {
    "lib": ["dom", "dom.iterable", "esnext"],
    "allowJs": true,
    "skipLibCheck": true,
    "strict": true,
    "noEmit": true,
    "esModuleInterop": true,
    "module": "esnext",
    "moduleResolution": "bundler",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "jsx": "preserve",
    "incremental": true,
    "plugins": [
      {
        "name": "next"
```

```
      }
    ],
    "paths": {
      "@/*": ["./*"]
    }
  },
  "include": ["next-env.d.ts", "**/*.ts", "**/*.tsx", ".next/types/**/*.ts"],
  "exclude": ["node_modules"]
}
```

Listing 7: Zawartość pliku tsconfig.json

4.2.1 Kluczowe opcje compilerOptions

- `strict: true` - włącza wszystkie ścisłe opcje typowania TypeScript. Zapewnia maksymalne bezpieczeństwo typów.
- `lib` - definiuje dostępne biblioteki typów (DOM API, ES2015+ features)
- `jsx: "preserve"` - pozostawia składnię JSX niezmienioną (Next.js sam zajmie się transformacją)
- `moduleResolution: "bundler"` - strategia rozwiązywania modułów zoptymalizowana dla bundlerów (narzędzi pakujących kod, np. Webpack, Vite)
- `paths` - aliasy ścieżek. Notacja `@/*` pozwala importować pliki z katalogu głównego:

```
import { prisma } from '@/lib/prisma';  // zamiast '../../../lib/prisma'

```

- `noEmit: true` - TypeScript nie generuje plików JavaScript (robi to Next.js)

4.3 next.config.ts

Plik `next.config.ts` zawiera konfigurację specyficzną dla Next.js. W podstawowej wersji projektu jest praktycznie pusty:

```
import type { NextConfig } from "next";

const nextConfig: NextConfig = {
  /* config options here */
};

export default nextConfig;
```

Listing 8: Zawartość pliku next.config.ts

4.3.1 Możliwe opcje konfiguracji

Choć w naszym projekcie plik jest pusty, `next.config.ts` może zawierać zaawansowane opcje:

- `reactStrictMode` - włącza tryb ścisły React (wykrywanie problemów)

- `images` - konfiguracja optymalizacji obrazów
- `redirects` - definiowanie przekierowań URL
- `rewrites` - przepisywanie URL
- `env` - zmienne środowiskowe dostępne w przeglądarce
- `webpack` - zaawansowana konfiguracja webpacka

4.4 .env - Zmienne środowiskowe

Plik `.env` zawiera zmienne środowiskowe używane w aplikacji. Nie jest commitowany do repozytorium Git (znajduje się w `.gitignore`), aby chronić wrażliwe dane (hasła, klucze API, connection strings).

Ważne: Do repozytorium commitowany jest natomiast plik `example.env`, który zawiera przykładową konfigurację bez wrażliwych danych. Przed uruchomieniem aplikacji należy:

1. Skopiować plik `example.env` do `.env`:

```
cp example.env .env

```

2. Ewentualnie dostosować wartości zmiennych do lokalnego środowiska

```
# This was inserted by ‘prisma init‘:
# Environment variables declared in this file are automatically made available to Prisma.
# See the documentation for more detail: https://pris.ly/d/prisma-schema#accessing-environment-variables-from-the-schema

# Prisma supports the native connection string format for PostgreSQL, MySQL, SQLite, SQL Server, MongoDB and CockroachDB.
# See the documentation for all the connection string options: https://pris.ly/d/connection-strings

DATABASE_URL=”file:./dev.db”
```

Listing 9: Zawartość pliku .env

4.4.1 DATABASE_URL

Zmienna `DATABASE_URL` definiuje połączenie do bazy danych. W przypadku SQLite jest to ścieżka do pliku:

```
DATABASE_URL=”file:./dev.db”
```

Dla innych baz danych wyglądałoby to inaczej:

```
# PostgreSQL
DATABASE_URL=”postgresql://user:password@localhost:5432/mydb”

# MySQL
DATABASE_URL=”mysql://user:password@localhost:3306/mydb”
```

4.5 .gitignore

Plik `.gitignore` definiuje, które pliki i katalogi nie powinny być śledzone przez Git:

```
# dependencies
/node_modules

# next.js
/.next/
/out/

# production
/build

# misc
.DS_Store
*.pem

# debug
npm-debug.log*
yarn-debug.log*
yarn-error.log*

# local env files
.env
.env.local
.env.development.local
.env.test.local
.env.production.local
```

Listing 10: Fragmenty pliku .gitignore

Kluczowe ignorowane elementy:

- `node_modules` - zależności (mogą być odtworzone przez `npm install`)
- `.next` - pliki wygenerowane przez Next.js
- `.env*` - pliki zmiennych środowiskowych (mogą zawierać sekrety)

5 Baza danych i Prisma ORM

Niniejszy rozdział omawia warstwę persystencji danych w aplikacji. Szczegółowo opisuje schemat bazy danych, model danych oraz mechanizm migracji.

5.1 Wprowadzenie do Prisma

Prisma to nowoczesny ORM (Object-Relational Mapping), który zapewnia warstwę abstrakcji nad bazą danych. Zamiast pisać surowe zapytania SQL, używamy API Prisma w TypeScript, które jest w pełni typowane.

5.1.1 Zalety Prisma

1. **Type Safety** - zapytania są w pełni typowane, kompilator wykryje błędy
2. **Auto-completion** - IDE podpowiada dostępne pola i metody
3. **Migracje** - automatyczne generowanie i zarządzanie migracjami schematu. Migracja to uporządkowana zmiana w strukturze bazy danych (np. dodanie tabeli, kolumny, indeksu). Prisma automatycznie generuje pliki SQL opisujące te zmiany, co pozwala na kontrolowanie ewolucji schematu bazy danych w czasie oraz synchronizację między środowiskami (development, staging, production)
4. **Deklaratywny schemat** - czytelna definicja modeli w języku Prisma Schema

5.2 Schemat Prisma

Schemat Prisma definiuje strukturę bazy danych w pliku `prisma/schema.prisma`. Jest to deklaratywna definicja modeli, relacji oraz konfiguracji połączenia.

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}

model Todo {
  id        Int      @id @default(autoincrement())
  text      String
  completed Boolean  @default(false)
  createdAt DateTime @default(now())
}
```

Listing 11: Pełna zawartość pliku schema.prisma

Odwołanie do pliku: `prisma/schema.prisma:1-16`

5.2.1 Sekcja generator

```
generator client {
  provider = "prisma-client-js"
}
```

Definicja generatora klienta Prisma. Opcja `provider = "prisma-client-js"` oznacza, że zostanie wygenerowany klient JavaScript/TypeScript.

Po każdej zmianie schematu należy uruchomić:

```
npx prisma generate
```

Komenda ta generuje kod TypeScript klienta Prisma w katalogu `node_modules/@prisma/client`.

5.2.2 Sekcja datasource

```
datasource db {
  provider = ”sqlite”
  url      = env(”DATABASE_URL”)
}
```

Definicja źródła danych (bazy danych):

- `provider = "sqlite"` - typ bazy danych (SQLite)
- `url = env("DATABASE_URL")` - connection string pobierany ze zmiennej środowiskowej

Zmienna `DATABASE_URL` jest zdefiniowana w pliku `.env`:

```
DATABASE_URL=”file:./dev.db”
```

5.3 Model Todo

Model Todo reprezentuje pojedyncze zadanie na liście. Składa się z czterech pól:

```
model Todo {
  id        Int      @id @default(autoincrement())
  text      String
  completed Boolean  @default(false)
  createdAt DateTime @default(now())
}
```

Listing 12: Model Todo

Odwołanie do pliku: `prisma/schema.prisma:10-15`

5.3.1 Pola modelu

id: Int

- Typ: Integer (liczba całkowita)
- `@id` - oznacza klucz główny tabeli
- `@default(autoincrement())` - automatycznie inkrementowana wartość dla nowych rekordów
- Unikalna wartość identyfikująca zadanie

text: String

- Typ: String (tekst)
- Przechowuje treść zadania
- Brak domyślnej wartości - pole wymagane przy tworzeniu rekordu

completed: Boolean

- Typ: Boolean (wartość logiczna)
- `@default(false)` - domyślnie zadanie jest nieukończone
- Określa, czy zadanie zostało wykonane

createdAt: DateTime

- Typ: DateTime (data i czas)
- `@default(now())` - automatycznie ustawiana na bieżący czas przy tworzeniu rekordu
- Przechowuje datę i czas utworzenia zadania

5.4 Migracje bazy danych

Migracje to mechanizm wersjonowania schematu bazy danych. Każda zmiana w schemacie Prisma generuje migrację - plik SQL opisujący transformację schematu.

5.4.1 Inicjalna migracja

Pierwsza migracja projektu znajduje się w katalogu:

```
prisma/migrations/20251109083508_init/migration.sql
```

Zawartość pliku migracji:

```
-- CreateTable
CREATE TABLE "Todo" (
    "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    "text" TEXT NOT NULL,
    "completed" BOOLEAN NOT NULL DEFAULT false,
    "createdAt" DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

Listing 13: Inicjalna migracja bazy danych

5.4.2 Analiza SQL

- `CREATE TABLE "Todo"` - tworzy tabelę o nazwie "Todo"
- `INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT` - klucz główny typu całkowitego, automatycznie inkrementowany
- `TEXT NOT NULL` - pole tekstowe, wymagane (nie może być NULL)
- `BOOLEAN NOT NULL DEFAULT false` - pole logiczne z wartością domyślną false
- `DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP` - data/czas z automatyczną wartością bieżącego czasu

5.4.3 Proces migracji

Typowy workflow pracy z migracjami:

1. Modyfikacja pliku `schema.prisma` (np. dodanie nowego pola)
2. Utworzenie migracji: `npx prisma migrate dev --name nazwa_migracji`
3. Prisma automatycznie:
   - Generuje plik SQL z migracją
   - Wykonuje migrację na bazie deweloperskiej
   - Regeneruje Prisma Client

5.5 Prisma Client

Prisma Client to automatycznie generowany klient bazy danych, który zapewnia type-safe API do wykonywania zapytań.

5.5.1 Singleton Prisma Client

Plik `lib/prisma.ts` eksportuje singleton instancji Prisma Client:

```
import { PrismaClient } from '@prisma/client';

const globalForPrisma = globalThis as unknown as {
  prisma: PrismaClient | undefined;
};

export const prisma = globalForPrisma.prisma ?? new PrismaClient();

if (process.env.NODE_ENV !== 'production') {
  globalForPrisma.prisma = prisma;
}
```

Listing 14: Singleton Prisma Client

5.5.2 Dlaczego singleton?

W środowisku deweloperskim Next.js używa Hot Module Replacement (HMR), który przeładowuje moduły podczas developmentu. Bez singletona każde przeładowanie tworzyłoby nową instancję `PrismaClient`, co prowadziłoby do wyczerpania połączeń z bazą danych.

Singleton zapewnia, że w całej aplikacji istnieje tylko jedna instancja klienta.

5.6 Przykłady użycia Prisma Client

5.6.1 Pobieranie wszystkich zadań

```
const todos = await prisma.todo.findMany({
  orderBy: {
    createdAt: 'desc',
  },
});
```

Listing 15: Pobieranie wszystkich zadań

5.6.2 Tworzenie nowego zadania

```
await prisma.todo.create({
  data: {
    text: "Kupić mleko"
  },
});
```

Listing 16: Tworzenie zadania

5.6.3 Aktualizacja zadania

```
await prisma.todo.update({
  where: { id: 1 },
  data: { completed: true },
});
```

Listing 17: Aktualizacja zadania

5.6.4 Usunięcie zadania

```
await prisma.todo.delete({
  where: { id: 1 },
});
```

Listing 18: Usunięcie zadania

5.7 Typy generowane przez Prisma

Prisma automatycznie generuje typy TypeScript dla modeli:

```
import type { Todo } from '@prisma/client';

// Typ Todo jest automatycznie wygenerowany:
// {
//   id: number;
//   text: string;
//   completed: boolean;
//   createdAt: Date;
// }
```

Listing 19: Automatycznie generowane typy

Typy te zapewniają bezpieczeństwo typów w całej aplikacji.

6 System routingu Next.js

Routing to mechanizm mapowania URL na odpowiednie komponenty i strony aplikacji. Next.js wprowadza koncepcję *file-system based routing* - struktury URL są definiowane przez strukturę plików i katalogów.

6.1 App Router vs Pages Router

Next.js oferuje dwa systemy routingu:

- **Pages Router** - starszy system oparty na katalogu `pages/`
- **App Router** - nowszy system oparty na katalogu `app/` (używany w projekcie)

Nasza aplikacja wykorzystuje **App Router**, wprowadzony w Next.js 13, który oferuje zaawansowane funkcjonalności takie jak Server Components, nested layouts oraz streaming.

6.2 Konwencje nazewnictwa

App Router wykorzystuje specjalne nazwy plików do definiowania różnych typów zasobów:

Nazwa pliku	Przeznaczenie
`page.tsx`	Komponent strony dostępnej pod danym URL
`layout.tsx`	Layout współdzielony przez strony w danym segmencie
`loading.tsx`	Komponent wyświetlany podczas ładowania
`error.tsx`	Komponent obsługi błędów
`not-found.tsx`	Strona 404 (nie znaleziono)
`route.ts`	API Route Handler

Tabela 2: Specjalne nazwy plików w App Router

W naszym projekcie wykorzystujemy pliki `page.tsx` oraz `layout.tsx`.

6.3 Mapowanie struktury na URL

Struktura katalogów w `app/` bezpośrednio definiuje strukturę URL aplikacji.

6.3.1 Przykłady mapowania

Plik	URL	Opis
`app/page.tsx`	`/`	Strona główna
`app/add/page.tsx`	`/add`	Strona dodawania zadania
`app/edit/[id]/page.tsx`	`/edit/:id`	Strona edycji (dynamiczna)

Tabela 3: Mapowanie plików na URL

6.4 Dynamic Routes - trasy dynamiczne

Trasy dynamiczne umożliwiają tworzenie stron z parametrami w URL. Parametr dynamiczny definiuje się za pomocą nawiasów kwadratowych w nazwie katalogu.

6.4.1 Przykład: app/edit/[id]/page.tsx

Katalog `[id]` oznacza parametr dynamiczny. URL takie jak:

- `/edit/1`
- `/edit/42`
- `/edit/xyz`

będą obsługiwane przez ten sam komponent, a wartość parametru będzie dostępna w komponencie.

6.4.2 Dostęp do parametrów

Parametry dynamiczne są przekazywane do komponentu strony jako props:

```
export default async function EditPage({
  params,
}: {
  params: Promise<{ id: string }>;
}) {
  const { id } = await params;

  // Konwersja string na number
  const todoId = parseInt(id, 10);

  // Pobranie zadania z bazy
  const todo = await getTodoById(todoId);

  // ...
}
```

Listing 20: Dostęp do parametrów dynamicznych

Odwołanie do pliku: `app/edit/[id]/page.tsx`

Uwaga o typach W Next.js 15+, parametry są zwracane jako `Promise`, co wymaga użycia `await` przed dostępem do wartości. Jest to związane z asynchronicznym renderowaniem.

6.5 Layout - wspólny szablon

Pliki `layout.tsx` definiują wspólny szablon HTML otaczający strony. Layout jest współdzielony przez wszystkie strony w danym segmencie i jego podsegmentach.

6.5.1 Root Layout

Plik `app/layout.tsx` to *root layout* - główny szablon całej aplikacji. Musi zawierać tagi `<html>` oraz `<body>`.

```
import type { Metadata } from 'next';
import './globals.css';

export const metadata: Metadata = {
  title: 'TODO List',
  description: 'Aplikacja do zarządzania zadaniami',
};

export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <html lang="pl">
      <body>{children}</body>
    </html>
  );
}
```

Listing 21: Root layout aplikacji

Odwołanie do pliku: `app/layout.tsx`

6.5.2 Analiza root layout

- `metadata` - obiekt definiujący metadane strony (tytuł, opis, używane przez wyszukiwarki i karty social media)
- `lang="pl"` - atrybut języka dokumentu HTML (polski)
- `children` - prop zawierający zawartość renderowanej strony. Layout otacza wszystkie strony swoją strukturą.
- Import `globals.css` - globalne style CSS stosowane do całej aplikacji

6.6 Nawigacja między stronami

Next.js udostępnia komponent `Link` do nawigacji po aplikacji bez przeładowania strony (client-side navigation).

6.6.1 Użycie komponentu Link

```
import Link from 'next/link';

// Proste linki
<Link href="/">Strona główna</Link>
<Link href="/add">Dodaj zadanie</Link>

// Link z dynamicznym parametrem
<Link href={'/edit/${id}'}>Edytuj</Link>
```

Listing 22: Nawigacja z użyciem Link

Przykłady z projektu:

- W `app/page.tsx`: link do `/add`
- W `app/components/TodoItem.tsx`: link do `/edit/{id}`
- W formularzach: link powrotu do `/`

Odwołania do plików:

- `app/page.tsx:24-26` - przycisk "Dodaj nowe zadanie"
- `app/components/TodoItem.tsx:38-40` - przycisk "Edytuj"

6.7 Nawigacja programowa

Oprócz komponentu `Link`, Next.js oferuje funkcje do nawigacji programowej:

6.7.1 redirect()

Funkcja `redirect()` umożliwia przekierowanie użytkownika na inny URL. Używana w Server Actions po zakończeniu operacji.

```
import { redirect } from 'next/navigation';

export async function saveTodo(prevState: FormState, formData: FormData)
    {
  // ... walidacja i zapis do bazy ...

  revalidatePath('/');
  redirect('/');  // Przekierowanie na stronę główną
}
```

Listing 23: Użycie redirect() w Server Action

Odwołanie do pliku: `app/actions.ts:72-73`

6.7.2 useRouter() (client-side)

W komponentach klienckich można użyć hooka `useRouter()`:

```
'use client';
import { useRouter } from 'next/navigation';

function MyComponent() {
  const router = useRouter();

  const handleClick = () => {
    router.push('/add');
  };

  return <button onClick={handleClick}>Dodaj</button>;
}
```

Listing 24: Client-side routing z useRouter

6.8 Cache Revalidation

Next.js automatycznie cache'uje renderowane strony i dane. Funkcja `revalidatePath()` pozwala unieważnić cache dla danej ścieżki.

```
import { revalidatePath } from 'next/cache';

export async function toggleTodo(id: number) {
  // ... operacja na bazie danych ...

  revalidatePath('/');  // Unieważnia cache strony głównej
}
```

Listing 25: Revalidacja cache

Odwołanie do pliku: `app/actions.ts:92`

Po wywołaniu `revalidatePath('/')`, przy następnym odwiedzeniu strony głównej Next.js ponownie wyrenderuje ją po stronie serwera z aktualnymi danymi z bazy.

6.9 Struktura routingu w projekcie

Aplikacja TODO posiada trzy główne trasy:

1. **Strona główna (/)**: wyświetla listę wszystkich zadań
   - Plik: `app/page.tsx`
   - Typ: Server Component
   - Pobiera dane z bazy przy każdym renderowaniu
2. **Dodawanie zadania (/add)**: formularz dodawania nowego zadania
   - Plik: `app/add/page.tsx`
   - Typ: Server Component (kontener dla Client Component)
   - Zawiera komponent TodoForm w trybie 'add'
3. **Edycja zadania (/edit/[id])**: formularz edycji istniejącego zadania
   - Plik: `app/edit/[id]/page.tsx`
   - Typ: Server Component z dynamic route
   - Pobiera dane zadania z bazy na podstawie parametru id
   - Zawiera komponent TodoForm w trybie 'edit'

7 Komponenty serwerowe (Server Components)

React 19 wraz z Next.js 16 wprowadzają nowy paradygmat - *React Server Components*. Komponenty serwerowe są renderowane wyłącznie po stronie serwera i nie trafiają do bundle JavaScript w przeglądarce.

7.1 Server Components vs Client Components

Server Components	Client Components
Renderowane tylko na serwerze	Renderowane na serwerze i kliencie
Nie trafiają do bundle JS	Są częścią bundle JS
Mogą bezpośrednio komunikować się z bazą danych	Nie mogą bezpośrednio łączyć się z bazą
Nie mogą używać hooków React	Mogą używać hooków (useState, useEffect, etc.)
Nie mogą obsługiwać zdarzeń (onClick, onChange)	Obsługują zdarzenia użytkownika
Domyślny typ w App Router	Wymagają dyrektywy 'use client'

Tabela 4: Porównanie Server Components i Client Components

7.2 Kiedy używać Server Components?

Server Components są idealne do:

- Pobierania danych z bazy lub API
- Dostępu do zasobów backendowych
- Przechowywania wrażliwych informacji (API keys, tokeny)
- Zmniejszania bundle size aplikacji
- Renderowania statycznej treści

7.3 Strona główna - app/page.tsx

Strona główna aplikacji jest Server Component. Odpowiada za pobranie listy zadań z bazy i wyświetlenie ich użytkownikowi.

```
import Link from 'next/link';
import { getTodos } from './actions';
import { TodoItem } from './components/TodoItem';
import styles from './page.module.css';

export default async function Home() {
  const todos = await getTodos();

  const todosList = todos.map((todo) => (
      <TodoItem
          key={todo.id}
          id={todo.id}
```

```
            text={todo.text}
            completed={todo.completed}
            createdAt={todo.createdAt}
        />
        )
  );

  return (
    <div className={styles.container}>
      <div className={styles.header}>
        <h1>Lista zadań</h1>
        <Link href="/add" className={styles.addBtn}>
          Dodaj nowe zadanie
        </Link>
      </div>

      {todos.length === 0 ? (
        <p className={styles.emptyMessage}>
          Brak zadań. Dodaj nowe zadanie!
        </p>
      ) : (
        <div className={styles.todoList}>
          {todosList}
        </div>
      )}
    </div>
  );
}
```

Listing 26: Strona główna aplikacji - pełny kod

Odwołanie do pliku: `app/page.tsx:1-38`

7.3.1 Analiza kodu

Funkcja asynchroniczna

```
export default async function Home() {
```

Komponent jest funkcją `async`, co pozwala używać `await` bezpośrednio w ciele funkcji. To możliwe tylko w Server Components.

Pobieranie danych

```
const todos = await getTodos();
```

Bezpośrednie wywołanie Server Action `getTodos()`, która komunikuje się z bazą danych. Wykonywane po stronie serwera przy każdym renderowaniu strony.

Mapowanie danych na komponenty

```
const todosList = todos.map((todo) => (
  <TodoItem
    key={todo.id}
    id={todo.id}
    text={todo.text}
    completed={todo.completed}
    createdAt={todo.createdAt}
  />
));
```

Transformacja tablicy zadań na tablicę komponentów React. Każdy element otrzymuje unikalny `key` (id zadania), wymagany przez React do efektywnego renderowania list.

Warunkowe renderowanie

```
{todos.length === 0 ? (
  <p>Brak zadań. Dodaj nowe zadanie!</p>
) : (
  <div>{todosList}</div>
)}
```

Jeśli lista jest pusta, wyświetla komunikat. W przeciwnym razie renderuje listę zadań.

Nawigacja

```
<Link href="/add" className={styles.addBtn}>
  Dodaj nowe zadanie
</Link>
```

Komponent `Link` do nawigacji client-side na stronę `/add`.

7.4 Strona dodawania - app/add/page.tsx

Strona dodawania zadania jest prostym Server Component, który renderuje formularz.

```
import { TodoForm } from '../components/TodoForm';
import Link from 'next/link';
import styles from './add.module.css';

export default function AddPage() {
  return (
    <div className={styles.container}>
      <div className={styles.formHeader}>
        <h1>Dodaj nowe zadanie</h1>
        <Link href="/" className={styles.backLink}>
              Powrót do listy
        </Link>
      </div>
      <TodoForm mode="add" />
    </div>
  );
}
```

Listing 27: Strona dodawania zadania

Odwołanie do pliku: `app/add/page.tsx`

7.4.1 Analiza

Komponent jest prosty:

- Renderuje nagłówek strony
- Renderuje link powrotu do listy głównej
- Renderuje komponent `TodoForm` w trybie `'add'`

Zauważ, że sam komponent `AddPage` jest Server Component, ale renderuje Client Component (`TodoForm`). To dozwolone - Server Components mogą renderować Client Components.

7.5 Strona edycji - app/edit/[id]/page.tsx

Strona edycji jest bardziej złożonym Server Component z dynamicznym parametrem.

```tsx
import { notFound } from 'next/navigation';
import { getTodoById } from '@/app/actions';
import { TodoForm } from '@/app/components/TodoForm';
import Link from 'next/link';
import styles from './edit.module.css';

export default async function EditPage({
  params,
}: {
  params: Promise<{ id: string }>;
}) {
  const { id } = await params;
  const todo = await getTodoById(parseInt(id, 10));

  if (!todo) {
    notFound();
  }

  return (
    <div className={styles.container}>
      <div className={styles.formHeader}>
        <h1>Edytuj zadanie</h1>
        <Link href="/" className={styles.backLink}>
              Powrót do listy
        </Link>
      </div>
      <TodoForm mode="edit" todo={todo} />
    </div>
  );
}
```

Listing 28: Strona edycji zadania - pełny kod

Odwołanie do pliku: `app/edit/[id]/page.tsx`

7.5.1 Analiza kodu

Parametry dynamiczne

```tsx
export default async function EditPage({
  params,
}: {
  params: Promise<{ id: string }>;
}) {
  const { id } = await params;
```

Parametr `id` jest przekazywany przez routing Next.js. W Next.js 15+ parametry są zwracane jako Promise, więc wymagają `await`.

Pobieranie danych

```tsx
const todo = await getTodoById(parseInt(id, 10));
```

Pobranie zadania z bazy na podstawie ID. Parametr URL jest stringiem, więc konwertujemy na liczbę.

Obsługa błędów

```
if (!todo) {
  notFound();
}
```

Jeśli zadanie o danym ID nie istnieje, wywołujemy funkcję `notFound()`, która wyświetli stronę 404.

Renderowanie formularza

```
<TodoForm mode="edit" todo={todo} />
```

Renderowanie formularza w trybie edycji, przekazując istniejące dane zadania.

7.6 Root Layout - app/layout.tsx

Root layout definiuje globalną strukturę HTML aplikacji.

```
import type { Metadata } from 'next';
import './globals.css';

export const metadata: Metadata = {
  title: 'TODO List',
  description: 'Aplikacja do zarządzania zadaniami',
};

export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <html lang="pl">
      <body>{children}</body>
    </html>
  );
}
```

Listing 29: Root layout - pełny kod

Odwołanie do pliku: `app/layout.tsx`

7.6.1 Metadata

```
export const metadata: Metadata = {
  title: 'TODO List',
  description: 'Aplikacja do zarządzania zadaniami',
};
```

Metadane aplikacji eksportowane jako stała. Next.js automatycznie wstawia je do `<head>` dokumentu HTML:

```
<head>
  <title>TODO List</title>
  <meta name="description" content="Aplikacja do zarządzania zadaniami"
   />
</head>
```

7.6.2 Struktura HTML

Layout musi zwracać kompletny dokument HTML z `<html>` i `<body>`:

```
return (
  <html lang="pl">
    <body>{children}</body>
  </html>
);
```

Prop `children` zawiera zawartość renderowanej strony. Layout otacza wszystkie strony aplikacji swoją strukturą.

7.7 Zalety Server Components w projekcie

7.7.1 Mniejszy bundle JavaScript

Server Components nie są wysyłane do przeglądarki, co znacząco zmniejsza rozmiar bundle. W naszej aplikacji:

- `app/page.tsx` - nie trafia do bundle (tylko TodoItem jako Client Component)
- `app/add/page.tsx` - nie trafia do bundle
- `app/edit/[id]/page.tsx` - nie trafia do bundle

7.7.2 Bezpieczeństwo

Server Components mogą bezpośrednio komunikować się z bazą danych bez eksponowania danych uwierzytelniających do klienta:

```
// Bezpieczne - wykonywane tylko na serwerze
const todos = await getTodos();
```

Connection string do bazy, klucze API, sekrety - wszystko pozostaje na serwerze.

7.7.3 Lepsze SEO

Server Components są renderowane do HTML na serwerze, co oznacza, że:

- Wyszukiwarki otrzymują pełny HTML z danymi
- Brak problemu z indeksowaniem treści generowanych JavaScript
- Szybsze First Contentful Paint (FCP)

7.7.4 Dostęp do zasobów backendowych

Server Components mogą bezpośrednio używać:

- Systemów plików (Node.js `fs` module)
- Zmiennych środowiskowych serwera
- Baz danych bez warstwy API

8 Komponenty klienckie (Client Components)

Client Components to komponenty React, które wykonują się zarówno na serwerze (initial render), jak i w przeglądarce (hydration oraz dalsze interakcje). Są niezbędne do obsługi interaktywności użytkownika.

8.1 Dyrektywa 'use client'

Aby oznaczyć komponent jako Client Component, należy dodać dyrektywę `'use client'` na początku pliku:

```
'use client';

import { useState } from 'react';

export function MyComponent() {
  const [count, setCount] = useState(0);
  // ...
}
```

Listing 30: Dyrektywa use client

8.2 Kiedy używać Client Components?

Client Components są wymagane gdy komponent:

- Używa hooków React (`useState`, `useEffect`, `useActionState`, etc.)
- Obsługuje zdarzenia użytkownika (`onClick`, `onChange`, etc.)
- Korzysta z API przeglądarki (`window`, `localStorage`, etc.)
- Używa bibliotek zależnych od przeglądarki

8.3 TodoItem - komponent pojedynczego zadania

Komponent `TodoItem` wyświetla pojedyncze zadanie na liście i obsługuje interakcje użytkownika (zaznaczanie, usuwanie).

```
'use client';

import Link from 'next/link';
import { deleteTodo, toggleTodo } from '../actions';
import styles from '../page.module.css';
import {formatDate} from '@/lib/formatDate';

type TodoItemProps = {
  id: number;
  text: string;
  completed: boolean;
  createdAt: Date;
};

export function TodoItem({ id, text, completed, createdAt }:
   TodoItemProps) {
```

```
  const handleToggle = async () => {
    await toggleTodo(id);
  };

  const handleDelete = async () => {
    await deleteTodo(id);
  };

  const formattedDate = formatDate(createdAt);

  return (
    <div className={styles.todoItem}>
      <label className={styles.todoCheckbox}>
        <input
          type="checkbox"
          checked={completed}
          onChange={handleToggle}
        />
        <span className={completed ? styles.completed : ''}>{text}</span
    >
      </label>
      <div className={styles.todoMeta}>
        <span className={styles.todoDate}>{formattedDate}</span>
        <Link href={'/edit/${id}'} className={styles.editBtn}>
          Edytuj
        </Link>
        <button onClick={handleDelete} className={styles.deleteBtn}>
          Usuń
        </button>
      </div>
    </div>
  );
}
```

Listing 31: Komponent TodoItem - pełny kod

Odwołanie do pliku: `app/components/TodoItem.tsx:1-48`

8.3.1 Analiza kodu

Dyrektywa use client Kod: `'use client';`

Oznacza komponent jako Client Component. Wymagane, ponieważ komponent używa event handlerów.

Definicja typów props TypeScript type definiujący kształt props przekazywanych do komponentu:

```
type TodoItemProps = {
  id: number;
  text: string;
  completed: boolean;
  createdAt: Date;
};
```

Listing 32: Definicja typu props TodoItem

Zapewnia type safety - kompilator wykryje niezgodności typów.

Event handlery Odpowiedzialne za obsługę zdarzeń (akcja po kliknięciu w element)

```
const handleToggle = async () => {
  await toggleTodo(id);
};

const handleDelete = async () => {
  await deleteTodo(id);
};
```

Listing 33: Funkcje obsługi zdarzeń

Funkcje obsługujące zdarzenia użytkownika. Są asynchroniczne, ponieważ wywołują Server Actions.

- **handleToggle**: wywoływana przy zmianie stanu checkboxa. Wywołuje Server Action `toggleTodo(id)`, która zmienia stan `completed` zadania w bazie danych.
- **handleDelete**: wywoływana przy kliknięciu przycisku "Usuń". Wywołuje Server Action `deleteTodo(id)`, która usuwa zadanie z bazy.

Formatowanie daty

Wywołanie funkcji pomocniczej formatującej datę do czytelnej formy (DD/MM/YYYY HH:mm).

```
const formattedDate = formatDate(createdAt);
```

Listing 34: Formatowanie daty utworzenia

Checkbox i obsługa onChange

Checkbox kontrolowany przez prop `completed`. Event `onChange` wywołuje `handleToggle`.

```
<input
  type="checkbox"
  checked={completed}
  onChange={handleToggle}
/>
```

Listing 35: Checkbox zmiany stanu zadania

Warunkowe stylowanie

Jeśli zadanie jest ukończone (`completed === true`), dodawana jest klasa CSS `styles.completed`, która przekreśla tekst.

```
<span className={completed ? styles.completed : ''}>
  {text}
</span>
```

Listing 36: Warunkowe dodawanie klasy CSS

Przycisk usuwania

Przycisk obsługujący event `onClick`. Wywołuje `handleDelete`.

```
<button onClick={handleDelete} className={styles.deleteBtn}>
  Usuń
</button>
```

Listing 37: Przycisk usuwania zadania

8.4 TodoForm - formularz dodawania/edycji

Komponent `TodoForm` to uniwersalny formularz działający w dwóch trybach: `'add'` (dodawanie) oraz `'edit'` (edycja).

```
'use client';

import Link from 'next/link';
import { useActionState } from 'react';
import { type FormState, saveTodo } from '@/app/actions';
import type { Todo } from '@prisma/client';
import styles from './TodoForm.module.css';

interface TodoFormProps {
  mode: 'add' | 'edit';
  todo?: Todo;
}

const initialState: FormState = {};

export function TodoForm({ mode, todo }: TodoFormProps) {
  const [state, formAction] = useActionState(saveTodo, initialState);

  return (
    <form action={formAction} className={styles.todoForm}>
      {todo && <input type="hidden" name="id" value={todo.id} />}

      <div className={styles.formGroup}>
        <label htmlFor="text">Treść zadania:</label>
        <input
          type="text"
          id="text"
          name="text"
          defaultValue={todo?.text || ''}
          placeholder="Wpisz treść zadania..."
          maxLength={200}
          autoFocus
          aria-describedby="text-error"
        />
        {state.errors?.text && (
          <div id="text-error" className={styles.error}>
            {state.errors.text.map((error, index) => (
              <p key={index}>{error}</p>
            ))}
          </div>
        )}
      </div>

      {state.message && (
```

```
45          <div className={styles.errorMessage}>
46            <p>{state.message}</p>
47          </div>
48        )}
49
50        <div className={styles.formActions}>
51          <button type="submit" className={styles.submitBtn}>
52            {mode === 'add' ? 'Dodaj zadanie' : 'Zapisz zmiany'}
53          </button>
54          <Link href="/" className={styles.cancelBtn}>
55            Anuluj
56          </Link>
57        </div>
58      </form>
59    );
60  }
```

Listing 38: Komponent TodoForm - pełny kod

Odwołanie do pliku: `app/components/TodoForm.tsx:1-61`

8.4.1 Analiza kodu

Props interface

```
1  interface TodoFormProps {
2    mode: 'add' | 'edit';
3    todo?: Todo;
4  }
```

Listing 39: Interface props TodoForm

Interface definiujący props:

- `mode`: typ literalny - tylko 'add' lub 'edit'
- `todo`: opcjonalny obiekt typu Todo (wymagany tylko w trybie edit)

Hook useActionState

```
1  const [state, formAction] = useActionState(saveTodo, initialState);
```

Listing 40: Użycie hooka useActionState

Hook React 19 przeznaczony do pracy z Server Actions w formularzach. Przyjmuje:

- `saveTodo` - Server Action do wywołania przy submit
- `initialState` - początkowy stan formularza (pusty obiekt)

Zwraca:

- `state` - aktualny stan formularza (zawiera błędy walidacji lub komunikaty)
- `formAction` - funkcja do użycia jako `action` w formularzu

Formularz z action

```
<form action={formAction} className={styles.todoForm}>
```

Listing 41: Formularz z Server Action

Atrybut `action` wskazuje na funkcję Server Action. Po submit formularza Next.js automatycznie wywołuje `formAction` z danymi formularza.

Ukryte pole ID

```
{todo && <input type="hidden" name="id" value={todo.id} />}
```

Listing 42: Warunkowe ukryte pole ID

W trybie edycji dodawane jest ukryte pole z ID zadania. Server Action używa tego ID do rozróżnienia operacji dodawania od edycji.

Pole tekstowe z defaultValue

```
<input
  type="text"
  id="text"
  name="text"
  defaultValue={todo?.text || ''}
  placeholder="Wpisz treść zadania..."
  maxLength={200}
  autoFocus
/>
```

Listing 43: Input tekstowy formularza

Atrybuty:

- `name="text"` - nazwa pola używana w FormData
- `defaultValue` - w trybie edit zawiera tekst zadania, w trybie add pusty string
- `maxLength={200}` - walidacja HTML ograniczająca długość do 200 znaków
- `autoFocus` - automatyczne ustawienie focus na polu po załadowaniu strony

Wyświetlanie błędów walidacji

```
{state.errors?.text && (
  <div id="text-error" className={styles.error}>
    {state.errors.text.map((error, index) => (
      <p key={index}>{error}</p>
    ))}
  </div>
)}
```

Listing 44: Wyświetlanie błędów walidacji pola

Jeśli Server Action zwróci błędy walidacji, są one wyświetlane pod polem. Każdy błąd jest osobnym paragrafem.

Komunikat ogólny błędu

```
{state.message && (
  <div className={styles.errorMessage}>
    <p>{state.message}</p>
  </div>
)}
```

Listing 45: Wyświetlanie komunikatu błędu

Wyświetlanie ogólnego komunikatu błędu (np. błąd bazy danych), jeśli istnieje w `state`.

Warunkowy tekst przycisku

```
<button type="submit">
  {mode === 'add' ? 'Dodaj zadanie' : 'Zapisz zmiany'}
</button>
```

Listing 46: Przycisk submit z warunkowym tekstem

Tekst przycisku zmienia się w zależności od trybu formularza.

8.5 Interakcja Server Actions z Client Components

Client Components mogą wywoływać Server Actions na dwa sposoby:

8.5.1 1. Poprzez event handlery

```
'use client';

import { deleteTodo } from '../actions';

export function TodoItem({ id }: { id: number }) {
  const handleDelete = async () => {
    await deleteTodo(id);  // Wywołanie Server Action
  };

  return <button onClick={handleDelete}>Usuń</button>;
}
```

Listing 47: Wywołanie Server Action z event handlera

Przykład: TodoItem wywołuje `toggleTodo` i `deleteTodo`

8.5.2 2. Poprzez atrybut action formularza

```
'use client';

import { useActionState } from 'react';
import { saveTodo } from '../actions';

export function TodoForm() {
  const [state, formAction] = useActionState(saveTodo, {});

  return (
    <form action={formAction}>
```

```
      {/* pola formularza */}
    </form>
  );
}
```

Listing 48: Server Action jako action formularza

Przykład: TodoForm używa `saveTodo` jako action

8.6 Progressive Enhancement

Formularze z Server Actions działają również bez JavaScript (progressive enhancement). Jeśli użytkownik wyłączy JavaScript:

- Formularz nadal można wysłać (zwykły POST request)
- Server Action zostanie wykonana po stronie serwera
- Strona zostanie przeładowana z nowymi danymi

To zwiększa dostępność i niezawodność aplikacji.

9 Server Actions - logika biznesowa

Server Actions to funkcje uruchamiane po stronie serwera, które mogą być wywoływane z komponentów klienckich. Eliminują potrzebę tworzenia osobnych endpointów API (REST, GraphQL) dla prostych operacji CRUD.

9.1 Wprowadzenie do Server Actions

Server Actions zostały wprowadzone w React 18 i Next.js 13. Umożliwiają bezpośrednie wywoływanie funkcji serwerowych z komponentów klienckich.

9.1.1 Zalety Server Actions

- **Prostsza architektura** - brak potrzeby tworzenia API layer
- **Type safety** - pełne typowanie między klientem a serwerem
- **Automatyczna serializacja** - Next.js obsługuje przekazywanie danych
- **Progressive enhancement** - działają nawet bez JavaScript
- **Bezpieczeństwo** - kod wykonywany tylko na serwerze

9.2 Dyrektywa 'use server'

Aby oznaczyć funkcję jako Server Action, należy dodać dyrektywę `'use server'` na początku pliku lub wewnątrz funkcji:

```
'use server';

export async function myServerAction() {
  // Kod wykonywany tylko na serwerze
}
```

Listing 49: Dyrektywa use server

9.3 Plik app/actions.ts

Wszystkie Server Actions w projekcie znajdują się w pliku `app/actions.ts`. Poniżej przedstawiono pełny kod pliku z szczegółowymi komentarzami.

```
'use server';

import {revalidatePath} from 'next/cache';
import {redirect} from 'next/navigation';
import {z} from 'zod';
import {prisma} from '@/lib/prisma';

/**
 * Schemat walidacyjny
 */
const todoSchema = z.object({
  text: z.string()
    .min(1, 'Treść zadania jest wymagana')
    .max(200, 'Treść zadania jest za długa (max 200 znaków)'),
```

```
});

/**
 * Typ stanu formularza zwracanego przez server actions
 */
export type FormState = {
  errors?: {
    text?: string[];
  };
  message?: string;
};
```

Listing 50: Plik app/actions.ts - część 1: imports i schemat

Odwołanie do pliku: `app/actions.ts:1-24`

9.3.1 Analiza - część 1

Dyrektywa use server

```
'use server';
```

Oznacza, że wszystkie eksportowane funkcje w pliku są Server Actions.

Schemat walidacyjny Zod

```
const todoSchema = z.object({
  text: z.string()
    .min(1, 'Treść zadania jest wymagana')
    .max(200, 'Treść zadania jest za długa (max 200 znaków)'),
});
```

Definicja schematu walidacji:

- Pole `text` musi być stringiem
- Minimum 1 znak (niepuste)
- Maximum 200 znaków
- Każde naruszenie zwraca odpowiedni komunikat błędu

Typ FormState

```
export type FormState = {
  errors?: {
    text?: string[];
  };
  message?: string;
};
```

Typ zwracany przez `saveTodo`. Zawiera:

- `errors` - opcjonalny obiekt z błędami walidacji (każde pole ma tablicę komunikatów)
- `message` - opcjonalny komunikat ogólny (np. błąd bazy danych)

9.4 Operacje CRUD

9.4.1 getTodos - pobieranie wszystkich zadań

```
export async function getTodos() {
    return prisma.todo.findMany({
        orderBy: {
            createdAt: 'desc',
        },
    });
}
```

Listing 51: Funkcja getTodos

Odwołanie do pliku: `app/actions.ts:25-31`

Analiza

- Zwraca wszystkie zadania z bazy danych
- Sortuje według daty utworzenia malejąco (newest first)
- Zwraca `Promise<Todo[]>`
- Używana na stronie głównej (`app/page.tsx`)

9.4.2 getTodoById - pobieranie jednego zadania

```
export async function getTodoById(id: number) {
    return prisma.todo.findUnique({
        where: {id},
    });
}
```

Listing 52: Funkcja getTodoById

Odwołanie do pliku: `app/actions.ts:103-107`

Analiza

- Przyjmuje parametr `id` typu number
- Zwraca jedno zadanie lub `null` jeśli nie znaleziono
- Używana na stronie edycji (`app/edit/[id]/page.tsx`)

9.4.3 saveTodo - dodawanie lub edycja zadania

Najważniejsza i najbardziej złożona Server Action w projekcie.

```
export async function saveTodo(
  prevState: FormState,
  formData: FormData
): Promise<FormState> {
  const validatedFields = todoSchema.safeParse({
    text: formData.get('text'),
  });
```

```
8
9   if (!validatedFields.success) {
10    return {
11      errors: validatedFields.error.flatten().fieldErrors,
12    };
13  }
14
15  const id = formData.get('id');
16  const isEdit = id && id !== '';
17
18  try {
19    if (isEdit) {
20      await updateTodoInternal(
21        parseInt(id as string, 10),
22        validatedFields.data.text
23      );
24    } else {
25      await addTodoInternal(validatedFields.data.text);
26    }
27  } catch {
28    return {
29      message: 'Błąd bazy danych: Nie udało się ${isEdit ? 'zaktualizowa
    ć' : 'dodać'} zadania.',
30    };
31  }
32
33  revalidatePath('/');
34  redirect('/');
35 }
```

Listing 53: Funkcja saveTodo - pełny kod

Odwołanie do pliku: `app/actions.ts:46-74`

Parametry funkcji

- `prevState: FormState` - poprzedni stan formularza (używany przez useActionState)
- `formData: FormData` - obiekt zawierający dane z formularza

Walidacja danych

```
1 const validatedFields = todoSchema.safeParse({
2   text: formData.get('text'),
3 });
4
5 if (!validatedFields.success) {
6   return {
7     errors: validatedFields.error.flatten().fieldErrors,
8   };
9 }
```

Proces walidacji:

1. Pobierz pole `text` z FormData
2. Zwaliduj używając schematu Zod

3. Jeśli walidacja się nie powiodła, zwróć błędy
4. Błędy są w formacie `{text: ['komunikat1', 'komunikat2']}`

Rozróżnienie dodawania od edycji

```
const id = formData.get(‘id‘);
const isEdit = id && id !== ‘‘;
```

Jeśli FormData zawiera pole `id`, to jest to operacja edycji. W przeciwnym razie - dodawanie.

Wykonanie operacji

```
try {
  if (isEdit) {
    await updateTodoInternal(
      parseInt(id as string, 10),
      validatedFields.data.text
    );
  } else {
    await addTodoInternal(validatedFields.data.text);
  }
} catch {
  return {
    message: ‘Błąd bazy danych: ...‘,
  };
}
```

Wywołanie odpowiedniej funkcji wewnętrznej. W przypadku błędu bazy danych, zwraca komunikat.

Revalidation i redirect

```
revalidatePath(‘/‘);
redirect(‘/‘);
```

- `revalidatePath('/')` - unieważnia cache strony głównej, wymuszając ponowne pobranie danych
- `redirect('/')` - przekierowuje użytkownika na stronę główną

9.4.4 Funkcje wewnętrzne (helper functions)

```
async function addTodoInternal(text: string): Promise<void> {
  await prisma.todo.create({
    data: { text },
  });
}

async function updateTodoInternal(id: number, text: string): Promise<
    void> {
  await prisma.todo.update({
    where: { id },
    data: { text },
```

```
  });
}
```

Listing 54: Funkcje pomocnicze addTodoInternal i updateTodoInternal

Odwołanie do pliku: `app/actions.ts:33-44`
Te funkcje są wewnętrzne (nie eksportowane), używane tylko przez `saveTodo`.

9.4.5 toggleTodo - zmiana statusu zadania

```
export async function toggleTodo(id: number) {
  const todo = await prisma.todo.findUnique({
    where: { id },
  });

  if (!todo) {
    return { error: 'Todo not found' };
  }

  await prisma.todo.update({
    where: { id },
    data: {
      completed: !todo.completed,
    },
  });

  revalidatePath('/');
}
```

Listing 55: Funkcja toggleTodo

Odwołanie do pliku: `app/actions.ts:76-93`

Analiza

1. Pobiera zadanie z bazy po ID
2. Sprawdza, czy zadanie istnieje
3. Aktualizuje pole `completed` na przeciwną wartość
4. Rewaliduje cache strony głównej

Używana przez komponent TodoItem przy zmianie stanu checkboxa.

9.4.6 deleteTodo - usuwanie zadania

```
export async function deleteTodo(id: number) {
  await prisma.todo.delete({
    where: { id },
  });

  revalidatePath('/');
}
```

Listing 56: Funkcja deleteTodo

Odwołanie do pliku: `app/actions.ts:95-101`

Analiza

1. Usuwa zadanie z bazy danych po ID
2. Rewaliduje cache strony głównej

Używana przez komponent TodoItem przy kliknięciu przycisku "Usuń".

9.5 Mechanizm revalidatePath

Funkcja `revalidatePath()` z `next/cache` unieważnia cache dla danej ścieżki.

9.5.1 Dlaczego potrzebna revalidation?

Next.js automatycznie cache'uje renderowane strony dla wydajności. Po zmianie danych w bazie, cache może zawierać nieaktualne dane. `revalidatePath('/')` informuje Next.js, że cache strony głównej jest nieważny i przy następnym żądaniu należy ponownie wyrenderować stronę.

9.5.2 Rodzaje revalidation

- **revalidatePath(path)** - unieważnia cache dla konkretnej ścieżki
- **revalidateTag(tag)** - unieważnia cache dla zasobów oznaczonych tagiem

W projekcie używamy `revalidatePath('/')`, ponieważ wszystkie operacje CRUD wpływają na listę zadań wyświetlaną na stronie głównej.

9.6 Mechanizm redirect

Funkcja `redirect()` z `next/navigation` przekierowuje użytkownika na inny URL.

```
import {redirect} from `next/navigation`;

export async function saveTodo(...) {
  // ... operacja na bazie ...

  revalidatePath(`/`);
  redirect(`/`);  // Przekierowanie na stronę główną
}
```

Listing 57: Użycie redirect

Uwaga: `redirect()` rzuca wyjątek (throw), co przerywa wykonanie funkcji. Dlatego musi być ostatnią instrukcją w Server Action.

9.7 Type Safety między klientem a serwerem

Server Actions zapewniają pełne typowanie:

```
// Server Action zwraca Promise<FormState>
export async function saveTodo(...): Promise<FormState> { ... }

// Client Component używa tego typu
import { type FormState, saveTodo } from `@/app/actions`;
```

```
6
7 const [state, formAction] = useActionState<FormState>(saveTodo, {});
8 //     ^^^^^ - TypeScript zna typ state
```

Listing 58: Type safety w Server Actions

Jeśli zmienimy typ zwracany przez Server Action, TypeScript natychmiast zgłosi błędy we wszystkich miejscach użycia.

9.8 Obsługa błędów

Server Actions mogą zwracać błędy na dwa sposoby:

9.8.1 1. Błędy walidacji

```
1 if (!validatedFields.success) {
2   return {
3     errors: validatedFields.error.flatten().fieldErrors,
4   };
5 }
```

Zwracany obiekt z polem `errors`. Client Component wyświetla błędy pod odpowiednimi polami formularza.

9.8.2 2. Błędy systemowe

```
1 try {
2   // operacja na bazie
3 } catch {
4   return {
5     message: ‘Błąd bazy danych: ...‘,
6   };
7 }
```

Zwracany obiekt z polem `message`. Client Component wyświetla ogólny komunikat błędu.

10 Stylowanie aplikacji

Aplikacja wykorzystuje CSS Modules do stylowania komponentów. Niniejszy rozdział omawia system stylowania oraz konkretne style użyte w projekcie.

10.1 CSS Modules

CSS Modules to podejście do stylowania, które automatycznie izoluje style komponentów, zapobiegając konfliktom nazw klas CSS.

10.1.1 Jak działają CSS Modules?

1. Konwencja nazewnictwa Pliki CSS Modules mają rozszerzenie `.module.css`:

```
page.module.css
TodoForm.module.css
TodoItem.module.css
```

2. Import i użycie

```
import styles from `./page.module.css`;

export default function Page() {
  return <div className={styles.container}>...</div>;
}
```

Listing 59: Import CSS Modules

3. Generowanie unikalnych nazw Next.js automatycznie przekształca nazwy klas na unikalne:

```
.container {
  max-width: 600px;
}
```

Listing 60: Przed (w pliku CSS)

```
<div class="page_container__a1b2c3">
```

Listing 61: Po (w HTML)

Dzięki temu klasa `.container` w jednym module nie koliduje z `.container` w innym.

10.1.2 Zalety CSS Modules

- **Scoped styles** - style ograniczone do komponentu
- **Brak konfliktów** - unikalne nazwy klas
- **Explicit dependencies** - komponenty jawnie importują swoje style
- **Dead code elimination** - nieużywane style można wykryć
- **Composition** - możliwość kompozycji klas z różnych modułów

10.2 Globalne style - globals.css

Plik `app/globals.css` zawiera globalne style stosowane do całej aplikacji. Importowany jest w root layout.

```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI',
    'Roboto', 'Oxygen', 'Ubuntu', 'Cantarell',
    'Fira Sans', 'Droid Sans', 'Helvetica Neue',
    sans-serif;
  background-color: #f5f5f5;
  color: #333;
  line-height: 1.6;
}

a {
  color: #0070f3;
  text-decoration: none;
}

a:hover {
  text-decoration: underline;
}
```

Listing 62: Fragmenty pliku globals.css

10.2.1 Analiza globals.css

CSS Reset

```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}
```

Resetuje domyślne marginesy i paddingi przeglądarki. `box-sizing: border-box` sprawia, że padding i border są wliczane w szerokość elementu.

Globalne style body

- `font-family` - system fonts (natywne fonty systemu operacyjnego)
- `background-color: #f5f5f5` - jasne tło
- `color: #333` - ciemnoszary tekst
- `line-height: 1.6` - większa czytelność tekstu

Style linków

```
a {
  color: #0070f3;  /* niebieski */
  text-decoration: none;
}

a:hover {
  text-decoration: underline;
}
```

Linki są niebieskie bez podkreślenia. Po najechaniu myszką pojawia się podkreślenie.

10.3 Style strony głównej - page.module.css

Plik `app/page.module.css` zawiera style dla strony głównej.

```
.container {
  max-width: 600px;
  margin: 0 auto;
  padding: 20px;
}

.header {
  display: flex;
  justify-content: space-between;
  align-items: center;
  margin-bottom: 30px;
  padding-bottom: 20px;
  border-bottom: 2px solid #ddd;
}

.addBtn {
  background-color: #0070f3;
  color: white;
  padding: 10px 20px;
  border-radius: 5px;
  border: none;
  cursor: pointer;
  text-decoration: none;
}

.addBtn:hover {
  background-color: #0051cc;
}

.todoList {
  display: flex;
  flex-direction: column;
  gap: 10px;
}

.todoItem {
  background-color: white;
  padding: 15px;
  border-radius: 8px;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
  display: flex;
```

```
  justify-content: space-between;
  align-items: center;
}

.todoCheckbox {
  display: flex;
  align-items: center;
  gap: 10px;
  flex: 1;
  cursor: pointer;
}

.completed {
  text-decoration: line-through;
  color: #999;
}

.editBtn {
  background-color: #f39c12;
  color: white;
  padding: 5px 10px;
  border-radius: 4px;
  margin-right: 5px;
}

.deleteBtn {
  background-color: #e74c3c;
  color: white;
  padding: 5px 10px;
  border-radius: 4px;
  border: none;
  cursor: pointer;
}
```

Listing 63: Fragmenty page.module.css

10.3.1 Analiza kluczowych klas

.container - główny kontener

```
.container {
  max-width: 600px;
  margin: 0 auto;
  padding: 20px;
}
```

- `max-width: 600px` - ogranicza szerokość na dużych ekranach
- `margin: 0 auto` - wyśrodkowanie poziome
- `padding: 20px` - wewnętrzne odstępy

.header - nagłówek z przyciskiem

```
.header {
  display: flex;
  justify-content: space-between;
```

```
  align-items: center;
}
```

Flexbox layout:

- `space-between` - tytuł po lewej, przycisk po prawej
- `align-items: center` - pionowe wyśrodkowanie elementów

.todoList - lista zadań

```
.todoList {
  display: flex;
  flex-direction: column;
  gap: 10px;
}
```

Flexbox column:

- `flex-direction: column` - elementy jeden pod drugim
- `gap: 10px` - odstęp 10px między zadaniami

.todoItem - pojedyncze zadanie

```
.todoItem {
  background-color: white;
  padding: 15px;
  border-radius: 8px;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
  display: flex;
  justify-content: space-between;
}
```

Styl karty (card):

- Białe tło na szarym tle strony
- Zaokrąglone rogi (`border-radius`)
- Subtelny cień (`box-shadow`)
- Flexbox: checkbox po lewej, przyciski po prawej

.completed - ukończone zadanie

```
.completed {
  text-decoration: line-through;
  color: #999;
}
```

Przekreślony tekst i jasnoszary kolor dla ukończonych zadań.

Przyciski - kolorystyka

- `.addBtn` - niebieski (#0070f3)
- `.editBtn` - pomarańczowy (#f39c12)
- `.deleteBtn` - czerwony (#e74c3c)

Kolorystyka zgodna z konwencją:

- Niebieski - akcja podstawowa (dodawanie)
- Pomarańczowy - akcja zmiany (edycja)
- Czerwony - akcja destrukcyjna (usuwanie)

10.4 Style formularza - TodoForm.module.css

Plik `app/components/TodoForm.module.css` styluje formularz dodawania/edycji.

```
.todoForm {
  background-color: white;
  padding: 30px;
  border-radius: 8px;
  box-shadow: 0 2px 8px rgba(0, 0, 0, 0.1);
}

.formGroup {
  margin-bottom: 20px;
}

.formGroup label {
  display: block;
  margin-bottom: 8px;
  font-weight: 600;
  color: #333;
}

.formGroup input {
  width: 100%;
  padding: 12px;
  border: 1px solid #ddd;
  border-radius: 4px;
  font-size: 16px;
}

.formGroup input:focus {
  outline: none;
  border-color: #0070f3;
  box-shadow: 0 0 0 3px rgba(0, 112, 243, 0.1);
}

.error {
  color: #dc2626;
  font-size: 14px;
  margin-top: 5px;
}

```

```
.errorMessage {
  background-color: #fef2f2;
  border: 1px solid #fecaca;
  color: #dc2626;
  padding: 12px;
  border-radius: 4px;
  margin-bottom: 20px;
}

.submitBtn {
  background-color: #0070f3;
  color: white;
  padding: 12px 24px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  font-size: 16px;
  font-weight: 600;
}

.submitBtn:hover {
  background-color: #0051cc;
}
```

Listing 64: Fragmenty TodoForm.module.css

10.4.1 Analiza

.formGroup - kontener pola Grupuje label i input razem z marginami.

input:focus - stan focus

```
.formGroup input:focus {
  outline: none;
  border-color: #0070f3;
  box-shadow: 0 0 0 3px rgba(0, 112, 243, 0.1);
}
```

Po kliknięciu w pole:

- Usuwa domyślny outline
- Zmienia kolor obramowania na niebieski
- Dodaje subtelną niebieską poświatę (box-shadow)

.error - błędy walidacji

```
.error {
  color: #dc2626;  /* czerwony */
  font-size: 14px;
  margin-top: 5px;
}
```

Czerwone komunikaty błędów pod polami formularza.

.errorMessage - komunikat ogólny

```
.errorMessage {
  background-color: #fef2f2;  /* jasny czerwony */
  border: 1px solid #fecaca;
  color: #dc2626;
  padding: 12px;
  border-radius: 4px;
}
```

Wyróżniona ramka z komunikatem błędu (np. błąd bazy danych).

10.5 Responsive Design

Aplikacja jest responsywna dzięki:

- **Flexbox** - elastyczny layout dostosowujący się do rozmiaru ekranu
- **max-width** - ograniczenie szerokości na dużych ekranach
- **Relative units** - użycie `em`, `rem`, `%` zamiast stałych pikseli
- **Mobile-first approach** - bazowe style działają na małych ekranach
- **Device Breakpoints** - aplikacja nie stosuje Media Queries CSS, ale ich wprowadzenie nie stanowi problemu.

11 Uruchamianie i rozwój aplikacji

Rozdział omawia proces uruchomienia i rozwoju - od pierwszego sklonowania projektu, przez uruchomienie w trybie deweloperskim, aż po budowanie wersji produkcyjnej.

11.1 Wymagania systemowe

Przed rozpoczęciem pracy z projektem należy zainstalować:

1. **Node.js** (wersja 24.0.0 lub nowsza)
   - Sprawdzenie wersji: `node --version`
   - Pobieranie: https://nodejs.org
2. **npm** (instalowany automatycznie z Node.js)
   - Sprawdzenie wersji npm: `npm --version`
3. **Git** (do klonowania repozytorium)
   - Sprawdzenie wersji: `git --version`
   - Pobieranie: https://git-scm.com

11.2 Pierwsze uruchomienie projektu

11.2.1 Krok 1: Instalacja zależności

Po sklonowaniu lub pobraniu projektu, należy zainstalować wszystkie zależności:

```
npm install
```

Listing 65: Instalacja zależności

Komenda ta:

- Odczytuje plik `package.json`
- Pobiera wszystkie zależności z npm registry
- Instaluje je w katalogu `node_modules`
- Generuje lub aktualizuje lock file (`package-lock.json` lub `pnpm-lock.yaml`)

11.2.2 Krok 2: Konfiguracja zmiennych środowiskowych

Skopiuj plik przykładowy `example.env` do `.env`:

```
cp example.env .env
```

Listing 66: Utworzenie pliku .env

Lub ręcznie utwórz plik `.env` w katalogu głównym projektu:

```
DATABASE_URL="file:./dev.db"
```

Listing 67: Zawartość pliku .env

Dla SQLite wystarczy powyższa konfiguracja. Dla innych baz danych (PostgreSQL, MySQL) należy podać odpowiedni connection string.

11.2.3 Krok 3: Inicjalizacja bazy danych

Uruchom migracje Prisma aby utworzyć bazę danych i tabele:

```
npm run db:update
```

Listing 68: Wykonanie migracji

Komenda ta wykonuje:

```
prisma migrate deploy && prisma generate
```

- `prisma migrate deploy` - wykonuje wszystkie pending migrations
- `prisma generate` - generuje Prisma Client

Po wykonaniu tej komendy:

- Zostanie utworzony plik `prisma/dev.db` (baza SQLite)
- Zostanie utworzona tabela `Todo`
- Prisma Client zostanie wygenerowany w `node_modules/@prisma/client`

11.2.4 Krok 4: Uruchomienie serwera deweloperskiego

```
npm run dev
```

Listing 69: Uruchomienie dev server

Komenda ta:

- Uruchamia Next.js w trybie deweloperskim
- Startuje serwer na porcie 3000
- Włącza Hot Module Replacement (HMR)
- Włącza Fast Refresh (natychmiastowe odświeżanie po zmianach w kodzie)

Output w terminalu:

```
> nextjs-todo@0.1.0 dev
> next dev

    Next.js 16.0.1
  - Local:        http://localhost:3000
  - Environments: .env

   Starting...
   Ready in 2.3s
```

Aplikacja jest dostępna pod adresem http://localhost:3000.

11.3 Skrypty npm

Wszystkie dostępne skrypty zdefiniowane w `package.json`:

11.3.1 npm run dev - serwer deweloperski

```
npm run dev
```

- Uruchamia Next.js w trybie development
- Port: 3000
- Hot Module Replacement włączony
- Source maps włączone
- Nie optymalizuje kodu (szybszy build)

11.3.2 npm run build - budowanie produkcji

```
npm run build
```

- Kompiluje aplikację do wersji produkcyjnej
- Optymalizuje kod (minifikacja, tree-shaking)
- Generuje statyczne HTML dla stron
- Tworzy skompresowane pliki JavaScript

11.3.3 npm run start - serwer produkcyjny

```
npm run start
```

- Uruchamia zbudowaną aplikację (wymaga wcześniejszego `npm run build`)
- Port: 3000
- Optymalizowany kod
- Brak HMR
- Gotowe do wdrożenia

11.3.4 npm run lint - analiza kodu

```
npm run lint
```

- Uruchamia ESLint
- Sprawdza kod pod kątem błędów i niezgodności ze stylem
- Używa konfiguracji `eslint-config-next`

11.3.5 npm run db:migrate - tworzenie migracji

```
npm run db:migrate nazwa_migracji
```

- Tworzy nową migrację bazy danych
- Generuje plik SQL w `prisma/migrations`
- Nie wykonuje migracji (tylko tworzy plik)

Przykład:

```
npm run db:migrate add_priority_field
```

11.3.6 npm run db:update - wykonanie migracji

```
npm run db:update
```

- Wykonuje wszystkie pending migrations
- Generuje Prisma Client
- Aktualizuje bazę danych do najnowszej wersji schematu

11.4 Workflow deweloperski

Typowy cykl pracy nad projektem:

11.4.1 Uruchomienie środowiska

```
# Terminal 1: serwer deweloperski
npm run dev
```

Serwer uruchamia się i nasłuchuje na zmiany w plikach. Aby zakończyć użyj skrótu CTRL+C.

11.4.2 Edycja kodu

- Otwórz plik w edytorze (np. `app/page.tsx`)
- Wprowadź zmiany
- Zapisz plik
- Next.js automatycznie wykryje zmiany i odświeży przeglądarkę (Fast Refresh)

11.4.3 Modyfikacja schematu bazy

Jeśli zmieniasz schemat Prisma:

```
# 1. Edytuj prisma/schema.prisma
# 2. Utwórz migrację
npm run db:migrate nazwa_zmiany

# 3. Wykonaj migrację
npm run db:update
```

11.4.4 Sprawdzenie jakości kodu

```
npm run lint
```

Napraw ewentualne błędy zgłoszone przez ESLint.

11.4.5 Testowanie

- Otwórz http://localhost:3000 w przeglądarce
- Przetestuj funkcjonalności aplikacji
- Sprawdź konsolę przeglądarki (F12) pod kątem błędów
- Aby zakończyć użyj skrótu CTRL+C

11.5 Prisma Studio - GUI bazy danych

Prisma oferuje graficzny interfejs do przeglądania i edycji danych:

```
npx prisma studio
```

Output:

```
Environment variables loaded from .env
Prisma Studio is running on http://localhost:5555
```

W przeglądarce pod adresem http://localhost:5555 dostępny jest graficzny interfejs do:

- Przeglądania tabel
- Dodawania rekordów
- Edycji rekordów
- Usuwania rekordów

11.6 Debugowanie

11.6.1 Console.log w Server Components

W Server Components `console.log()` wypisuje do terminala (nie do konsoli przeglądarki):

```
export default async function Home() {
  const todos = await getTodos();
  console.log(‘Liczba zadań:‘, todos.length);  // Output w terminalu
  // ...
}
```

Listing 70: Debugowanie Server Component

11.6.2 Console.log w Client Components

W Client Components `console.log()` wypisuje do konsoli przeglądarki:

```
‘use client‘;

export function TodoItem({ id }) {
  const handleClick = () => {
    console.log(‘Kliknięto zadanie:‘, id);  // Output w konsoli przeglą
   darki
  };
  // ...
}
```

Listing 71: Debugowanie Client Component

11.6.3 React DevTools

Zainstaluj rozszerzenie React DevTools do przeglądarki:

- Chrome: https://chrome.google.com/webstore
- Firefox: https://addons.mozilla.org

Umożliwia:

- Inspekcję drzewa komponentów
- Podgląd props i state
- Śledzenie renderowania komponentów

11.7 Rozwiązywanie problemów

11.7.1 Błąd migracji

Jeśli migracje są w niespójnym stanie:

```
# Reset bazy (UWAGA: usuwa wszystkie dane!)
npx prisma migrate reset
```