



UNIWERSYTET BIELSKO-BIAŁSKI

PRACA INŻYNIERSKA

System low-code do budowy interfejsów API

Autor:

Konrad Firlej

Numer Karty Studenckiej: 60043

Promotor:

dr inż. Krzysztof Augustynek

Streszczenie

Praca opisuje projekt i implementację lekkiego systemu low-code do budowy i uruchamiania przepływów API. Zastosowano stos ASP.NET Core + EF Core + SQLite [1][2][6] oraz frontend Vite + React + React Flow [4][5]. System umożliwia graficzne modelowanie workflowów, automatyczne wersjonowanie, uruchomienia ręczne i harmonogramowane oraz podgląd historii wykonania. Testy jednostkowe w NUnit [7] pokrywają serwisy, kontrolery i egzekutorów bloków. Dokument kończy ocena spełnienia celów, propozycje rozwoju (autoryzacja, skalowanie schedulera) i wnioski praktyczne.

Abstract

The thesis presents the design and implementation of a lightweight low-code platform for orchestrating API workflows. The stack combines ASP.NET Core with EF Core and SQLite on the backend [1][2][6], and Vite + React + React Flow on the frontend [4][5]. The system supports graphical workflow modeling, automatic versioning, manual and scheduled runs, and execution history. NUnit-based unit tests [7] cover services, controllers, and block executors. The document concludes with an evaluation of objectives, planned extensions (auth, distributed scheduler), and practical takeaways.

Oświadczenie

Oświadczam, że niniejszą pracę wykonałem samodzielnie, nie naruszając praw autorskich osób trzecich ani obowiązujących przepisów.

Wykaz skrótów i symboli

- **API** – Application Programming Interface.
- **REST** – Representational State Transfer.
- **CRUD** – Create, Read, Update, Delete.
- **DTO** – Data Transfer Object.
- **HMR** – Hot Module Replacement.
- **UTC** – Coordinated Universal Time.
- **BPMN** – Business Process Model and Notation.
- **CI** – Continuous Integration.

Spis treści

1 Wstęp	1
2 Analiza wymagań	2
2.1 Opis problemu	2
2.2 Grupa docelowa	2
2.3 Analiza konkurencyjnych rozwiązań (edytory grafowe / node-based)	2
2.4 Wymagania funkcjonalne	2
2.5 Wymagania niefunkcjonalne	3
3 Przegląd technologii	4
3.1 Warstwa frontendowa: HTML, CSS, JavaScript/TypeScript	4
3.2 Warstwa backendowa: ASP.NET Core	6
3.3 Magazyn danych: SQLite dziś, ścieżka do MongoDB i relacyjnej produkcji	7
3.4 Kontrola wersji i monorepo: Git	8
3.5 Uzasadnienie wyboru stosu jako całości	9
4 Projekt architektury aplikacji	10
4.1 Opis architektury	10
4.2 Diagramy architektoniczne	10
4.3 Diagram komponentów	11
4.4 Przepływ danych i przypadki użycia	11
4.5 Warstwy i interakcje	12
4.6 Projekt UI/UX	12
4.7 Model bazy danych (szkic ERD)	13
5 Implementacja	14
5.1 Kluczowe moduły i funkcjonalności	14
5.2 Interfejs użytkownika	15
5.3 Obsługa danych i zarządzanie bazą danych	20
5.4 Integracja frontendu i backendu	22
6 Testowanie	23
6.1 Rodzaje testów	23
6.2 Narzędzia testujące	23
6.3 Wyniki testów (28.01.2026, lokalnie)	23
6.4 Błędy i sposoby ich naprawy	23
7 Wyzwania i rozwiązania	24
7.1 Problemy napotkane podczas realizacji	24
7.2 Metody rozwiązania	24
7.3 Optymalizacje i usprawnienia	24
8 Podsumowanie i wnioski	25
8.1 Ocena realizacji celu	25
8.2 Możliwości rozwoju	25
8.3 Refleksje	25
8.4 Wnioski końcowe	25

Spis rysunków

1	Warstwa frontendowa: HTML/CSS oraz stos Vite + React + TypeScript.	4
2	Warstwa serwerowa i stos testów: ASP.NET Core/EF Core oraz NUnit.	6
3	Magazyn danych w trybie deweloperskim: SQLite.	7
4	Kontrola wersji: Git w układzie monorepo.	8
5	Szkic ERD kluczowych encji systemu: workflow jest centralną encją, do której należą rewizje, harmonogramy, wykonania, bloki, połączenia i zmienne.	13
6	Edytor grafu workflowu (React Flow)	15
7	Widok modala wersji	15
8	Formularz harmonogramu	16
9	Historia uruchomień workflowu (widok 1)	16
10	Historia uruchomień workflowu (widok 2)	17
11	Menu kontekstowe oraz toolbar edytora	17
12	Panel konfiguracji bloku	18
13	Edycja zmiennych przepływu	18
14	Stylistika i motyw interfejsu – motyw jasny	19
15	Stylistika i motyw interfejsu – motyw ciemny	19
16	Graficzny schemat relacji w bazie <code>flowforge.db</code> .	21

1 Wstęp

Dynamiczny wzrost usług cyfrowych powoduje, że nawet małe zespoły muszą łączyć pobieranie danych, wywołania API i weryfikację wyników w powtarzalne sekwencje. Brak jednolitego narzędzia skutkuje ręcznym klejeniem skryptów, utratą historii zmian i trudnością w odtworzeniu błędów, szczególnie gdy zadania są wykonywane cyklicznie lub przez różne osoby. Flowforge ma być lekkim systemem low-code, który pozwala budować, uruchamiać i monitorować takie przepływy bez ciężkiej infrastruktury i bez konieczności pisania pełnych aplikacji back-endowych. Tematem pracy jest opis i implementacja tego narzędzia oraz ocena jego przydatności dla małych zespołów, które chcą szybko tworzyć interfejsy API i łączyć je w spójne procesy.

Definicja i zakres projektu obejmuje trzy warstwy: (1) warstwę modelowania przepływu jako grafu bloków i połączeń (React Flow w UI), (2) warstwę usługową udostępniającą REST API do zapisu, wersjonowania i uruchamiania przepływów (ASP.NET Core, EF Core, SQLite w trybie deweloperskim), (3) warstwę wykonawczą z prostym harmonogramem działającym w jednym procesie serwera. Założono działanie na pojedynczej instancji, bez zewnętrznych brokerów, co upraszcza uruchomienie w środowisku lokalnym lub testowym.

Celem projektu jest obniżenie progu wejścia do automatyzacji i integracji API: użytkownik rysuje przepływ jako graf bloków, zapisuje go z automatycznymi wersjami i uruchamia ręcznie lub według prostego harmonogramu. Oczekiwany rezultatem jest szybsze prototypowanie połączeń między usługami, krótszy czas wdrożenia nowych przepływów oraz możliwość bezpiecznego cofania zmian dzięki wersjonowaniu definicji. Motywacją jest luka między prostą listą zadań a platformami klasy BPMN/DAG (Airflow, Prefect) [18][19][17], które są kosztowne, wymagają zaplecza DevOps i niepotrzebnie komplikują małe wdrożenia.

Problem, który aplikacja rozwiązuje, to brak samodzielnego, prostego narzędzia do orkiestracji API w środowiskach o ograniczonych zasobach. W codziennej pracy prowadzi to do pominiętych kroków, niespójnych wersji i braku wglądu w historię wykonan. Flowforge porządkuje proces jako graf, przechowuje historię zmian i wykonan oraz umożliwia audit błędów, co skraca czas realizacji i zmniejsza liczbę pominiętych czynności. Dzięki wersjonowaniu definicji użytkownik może wrócić do stabilnej konfiguracji, a zapis wykonan ułatwia analizę przyczyn błędów.

Podejście technologiczne opiera się na prostocie wdrożenia i minimalnych zależnościach. Backend w ASP.NET Core z EF Core udostępnia REST API do definicji, wersji i uruchomień [1][2]; harmonogram działa w jednym procesie serwera, obliczając najbliższe uruchomienia bez zewnętrznego brokera. Frontend zbudowano w Vite + React + TypeScript, z użyciem React Flow do edycji grafów oraz własnych komponentów do historii i uruchomień, co zapewnia szybki build, HMR i spójny typowany model danych po stronie klienta [4][3][5]. Świadomie pominięto integracje z kolejkami zadań, rozproszonym schedulerem i autoryzacją, aby utrzymać niski narzut instalacji i konfiguracji w małym zespole; te elementy są wskazane jako kierunek dalszych prac.

2 Analiza wymagań

2.1 Opis problemu

Codzienne zarządzanie zadaniami wymaga łączenia wielu kroków: wywołań API, validacji danych, zapisów wyników i powiadomień. Robione ręcznie lub w rozproszonych skryptach prowadzi do pominiętych czynności, niespójnych wersji i trudności w audycie. Potrzebne jest lekkie narzędzie, które pozwala wizualnie zaprojektować sekwencję zadań i uruchamiać ją powtarzalnie.

2.2 Grupa docelowa

Pojedynczy użytkownik lub bardzo mały zespół (1–3 osoby) pracujący lokalnie lub w środowisku testowym, bez dedykowanego działu DevOps. Kluczowe jest szybkie wdrożenie, brak kosztów licencyjnych i możliwość pracy na jednej instancji aplikacji.

2.3 Analiza konkurencyjnych rozwiązań (edytory grafowe / node-based)

- **n8n**: open-source automation z edytorem przepływów opartym na grafie; bogate integracje, ale większy narzut konfiguracyjny niż Flowforge [13].
- **Node-RED**: flow-based programming w przeglądarce; dojrzały ekosystem wtyczek, interfejs mniej nastawiony na wersjonowanie przepływów [14].
- **Flowise**: narzędzie do budowy agentów AI oparte o React Flow; mocno LLM-centric, brak fokus na klasyczne zadania operacyjne [15].
- **LangFlow**: edytor łańcuchów LLM (też React Flow); świetny do prototypów konwersacyjnych, słabszy w zarządzaniu harmonogramem zadań codziennych [16].
- **Camunda Modeler (BPMN)**: graficzny edytor procesów biznesowych; bogata semantyka BPMN, ale cięższe wdrożenie i większa złożoność dla pojedynczego użytkownika [17].

2.4 Wymagania funkcjonalne

- Graficzne modelowanie przepływu z bloków (start, akcje API, warunki, zakończenie) z validacją połączeń.
- Ręczne uruchamianie oraz harmonogram (*Once, Interval, Daily*) z wyliczaniem kolejnego startu i zapisem `LastRun/NextRun`.
- Automatyczne wersjonowanie definicji (migawka przy każdym zapisie), podgląd listy wersji, przywracanie i usuwanie.
- Rejestrowanie wykonanów z wejściem, wynikiem, czasem i błędami; filtrowanie i podgląd szczegółów w UI.
- Obsługa zmiennych przepływu (wejściowe/wyjściowe) z validacją typów przy zapisie i uruchomieniu.
- Walidacja cykli i poprawności połączeń na etapie edycji (blokowanie błędnych krawędzi).
- Eksport/import definicji przepływu w formacie JSON w celu backupu lub przeniesienia między środowiskami.
- Edycja i podgląd w przeglądarce (Vite + React + TypeScript, React Flow), podgląd historii uruchomień i wersji na osobnych widokach.

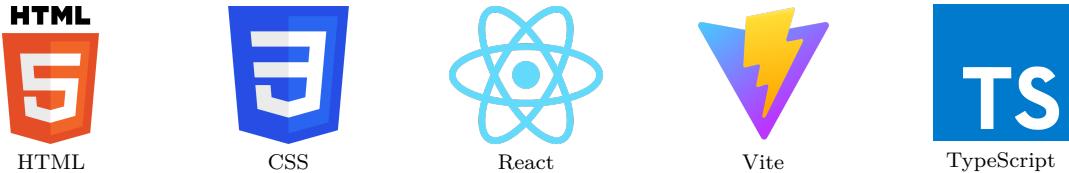
2.5 Wymagania niefunkcjonalne

- Lekka instalacja: pojedyncza instancja ASP.NET Core + SQLite; brak zewnętrznego brokera zadań i dodatkowych usług systemowych.
- Szybka iteracja frontendowa (Vite HMR, typowanie w TS), responsywny UI działający na ekranach desktop i laptop.
- Spójność czasowa: przechowywanie dat w UTC, prezentacja w czasie lokalnym po stronie klienta; jedno źródło czasu dla harmonogramu.
- Niska bariera utrzymania: konfiguracja przez plik `appsettings` lub zmienne środowiskowe, brak konieczności kontenerów ani orkiestracji.
- Testowalność: pokrycie logiki serwisów, repozytoriów i harmonogramu testami jednostkowymi (NUnit); możliwość uruchomienia `dotnet test` bez zewnętrznych zależności.
- Odporność na błędy użytkownika: validacja danych wejściowych na API i w UI, jasne komunikaty o błędach, brak krytycznych awarii przy niepoprawnych danych.
- Przewidywalne zużycie zasobów: praca w pojedynczym procesie, brak długotrwałych workerów; logi ograniczone do niezbędnych zdarzeń.
- Prosty model bezpieczeństwa: tryb zaufanego użytkownika (brak kont i ról), gotowość do późniejszego dodania autoryzacji jako pracy przyszłej.

3 Przegląd technologii

Poniższy przegląd opisuje dobór technologii w projekcie Flowforge w układzie monorepo, uwzględniając warstwę frontendową (HTML, CSS, JavaScript/TypeScript z Vite + React + React Flow), backend (ASP.NET Core zamiast rozważanego Next.js), magazyn danych (SQLite w trybie deweloperskim z możliwością migracji do MongoDB lub relacyjnej bazy produkcyjnej) oraz narzędzia kontroli wersji (Git). Uzasadnienia odnoszą się do wymagań funkcjonalnych i niefunkcjonalnych małego zespołu oraz do kosztu utrzymania pojedynczej instancji.

3.1 Warstwa frontendowa: HTML, CSS, JavaScript/TypeScript



Rysunek 1: Warstwa frontendowa: HTML/CSS oraz stos Vite + React + TypeScript.

Frontend powstał w oparciu o Vite + React + TypeScript [4][3][8], co zapewnia bardzo krótki czas startu deweloperskiego (HMR), typowanie modeli przesyłanych do API oraz czytelną strukturę komponentów. HTML i CSS pozostają fundamentem renderowania – semantyczne znaczniki (nagłówki, listy, formularze) ułatwiają dostępność, a warstwa stylów jest ograniczona do lekkich arkuszów i zmennych kolorystycznych, by nie tworzyć zależności od ciężkich frameworków UI. Styling zachowuje spójność z React Flow [5], który dostarcza kanwę do edycji grafów; kolory i typografia są definiowane we własnych klasach, co upraszcza utrzymanie i eliminację konfliktów między bibliotekami [3].

Wybór TypeScriptu zamiast czystego JavaScriptu wynika z potrzeby jednoznacznego odwzorowania modeli domenowych (Workflow, Block, Connection, Revision) po stronie klienta. Silne typowanie zmniejsza liczbę błędów integracyjnych, zwłaszcza przy serializacji/deskryptacji JSON z API. Alternatywy jak czysty JS przyspieszyłyby start, ale zwiększyłyby ryzyko błędów przy refaktoryzacji. React, w przeciwieństwie do frameworków szablonowych, pozwala budować hermetyczne komponenty edytora, paneli historii i harmonogramu, wykorzystując jeden model stanu. Vue lub Svelte mogłyby przynieść lżejszy runtime, jednak w zespole dostępne były doświadczenia z Reactem oraz gotowe komponenty React Flow, co skraca czas dostarczenia funkcji edycji grafu.

React Flow został wybrany jako biblioteka do wizualnej edycji połączeń, ponieważ zapewnia: (1) interaktywny canvas z obsługą drag & drop, (2) validację połączeń i blokadę cykli na poziomie UI, (3) łatwe mapowanie węzłów na dane domenowe. Podejścia alternatywne (JointJS, Cytoscape) oferują większe możliwości grafowe, ale są cięższe w integracji z prostym przepływem danych i wymagają bardziej rozbudowanej konfiguracji. Dla pojedynczego użytkownika ważniejsza była mała waga bundla i szybkość wdrożenia niż pełne możliwości edycji BPMN.

HTML i CSS wykorzystano zgodnie z minimalnym podejściem: brak frameworka typu Bootstrap zmniejsza rozmiar paczki i eliminuje nadmiar stylisty. Zamiast tego definiowane są zmienne kolorów, spacing i typografia w jednym miejscu, co pozwala utrzymać spójny akcent wizualny (zielone akcenty w UI) i łatwo przełączać motyw jasny/ciemny. W warstwie dostępności zapewniono odpowiedni kontrast i focus styles.

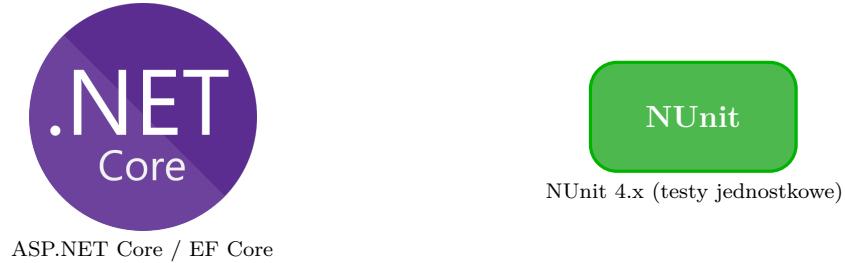
Dodatkowe uzasadnienia wyboru frontendu:

- **Wydajność buildów:** Vite kompliuje tylko zmienione moduły dzięki esbuild i dev-server z ESM; przy małym zespole skraca to pętlę „edytuj–zobacz efekt” do sekund.
- **Testowalność UI:** typowanie i modularność komponentów ułatwia wprowadzenie testów jednostkowych (np. Vitest/RTL) w przyszłości; React Flow dostarcza mockowalny interfejs zdarzeń.
- **Utrzymanie stylów:** CSS Modules lub scoped klasy w Vite pozwalają unikać kolizji nazw i utrzymać mały surface area. Brak design systemu typu Material eliminuje zależność od wersji i możliwe breaking changes.
- **Dostępność i i18n:** semantyczny HTML + kontrola nad drzewem DOM ułatwiają dodanie ARIA i lokalizacji, co byłoby trudniejsze przy zamkniętych komponentach frameworków UI.
- **Wielkość bundla:** bazowy build Vite z React i React Flow mieści się w kilkuset kB gz, co jest akceptowalne dla pojedynczego użytkownika; brak ciężkich frameworków CSS trzyma rozmiar w ryzach.
- **Linting i formatowanie:** ESLint + Prettier włączone w projekcie (Vite) utrzymują spójność stylu kodu; TypeScript wymusza jawne typy w interfejsach API.
- **Konfiguracja środowisk:** pliki .env dla URL API i trybu dark/light, bez rozbudowanych build-time flag; prostota ważna przy braku CI/CD.
- **Przybliżone metryki:** cold build (dev) trwa ok. 1–2 s na typowym laptopie deweloperskim, HMR < 300 ms dla pojedynczej zmiany komponentu; bundel produkcyjny po minifikacji i gzip to rzad 300–500 kB (w zależności od ilości ikon/grafiki), co przekłada się na ładowanie rzędu kilkuset ms przy połączeniu 10–20 Mbps.

Potencjalne alternatywy odrzucone:

- **Next.js + App Router:** zaoferowałby SSR/SSG, lecz narzut konfiguracji i dłuższe buildy nie były uzasadnione dla aplikacji single-tenant.
- **Angular:** pełny framework dałby DI i szablony, ale wiązałby z bardziej rozbudowaną strukturą projektu i cięższym bundlem.
- **Electron/Tauri:** mogłyby dać desktop, lecz celem była aplikacja webowa z minimalnym footprintem.

3.2 Warstwa backendowa: ASP.NET Core



Rysunek 2: Warstwa serwerowa i stos testów: ASP.NET Core/EF Core oraz NUnit.

Wybrano ASP.NET Core [1] z EF Core [2] jako warstwę serwerową, bo zapewnia spójność modeli domenowych, stabilny hosting w jednym procesie (wraz z `BackgroundService` dla harmonogramu) oraz szybkie testowanie z NUnit [7] + Moq + InMemory EF Core [2] bez zależności od zewnętrznych usług. UI pozostaje w Vite/React [4][3], co zmniejsza sprzężenie i pozwala niezależnie rozwijać obie warstwy. Middlewares ograniczono do minimum (CORS, routing, Swagger w trybie dev), a logika harmonogramu działa w jednym procesie, zgodnie z założeniem pracy na pojedynczej instancji [1].

Dodatkowe aspekty backendu:

- **Wydajność i profil zasobów:** ASP.NET Core w konfiguracji Kestrel + pojedynczy wątek harmonogramu zużywa mało pamięci; brak zewnętrznych workerów oznacza przewidywalny CPU footprint.
- **Migracje schematu:** EF Core migracje wersjonują zmiany modeli i pozwalają na powtarzalne wdrożenia; dla środowisk docelowych można wygenerować skrypt SQL lub zastosować migracje runtime.
- **Bezpieczeństwo:** choć aplikacja działa w trybie zaufanego użytkownika, ASP.NET Core zapewnia sanitizację nagłówków, mechanizmy limitowania rozmiaru żądań i łatwe dołączenie autoryzacji w przyszłości.
- **Observability:** wbudowane logowanie i metryki zdrowia (health checks) można włączyć bez zmiany architektury, co ułatwia późniejsze monitorowanie.
- **Konfiguracja:** `appsettings.json` oraz zmienne środowiskowe pozwalają rozdzielić konfigurację dev/prod; brak twardych wpisów sekretów w repo.
- **Serializacja:** System.Text.Json stosuje camelCase kontrakty spójne z frontendem; unika się nadmiarowych atrybutów, co upraszcza modele.
- **Testy:** NUnit pokrywa serwisy, repozytoria i kontrolery z in-memory SQLite; to umożliwia powtarzalne testy bez zewnętrznej bazy.
- **Przybliżone metryki:** zimny start API (dotnet run) to kilka sekund; pojedyncze zapytanie GET/POST w warunkach lokalnych < 10 ms; harmonogram wywołuje cykl co ok. 30 s i operuje na niewielkiej liczbie rekordów, więc obciążenie CPU jest marginalne.

3.3 Magazyn danych: SQLite dziś, ścieżka do MongoDB i relacyjnej produkcji



Rysunek 3: Magazyn danych w trybie deweloperskim: SQLite.

W trybie deweloperskim wykorzystano SQLite [6], bo:

- nie wymaga instalacji osobnego serwera – istotne przy pojedynczym użytkowniku,
- współdziała z EF Core i migracjami,
- wystarcza do historii wersji i wykonania w małym obciążeniu.

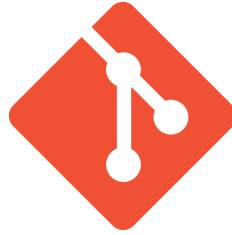
Jednocześnie zaprojektowano model w sposób neutralny: encje mają jednoznaczne klucze, a dane strukturalne (definicja grafu) przechowywane są w polach JSON. Pozwala to na potencjalną migrację do MongoDB, jeśli pojawi się potrzeba przechowywania dużych, schematowo elastycznych definicji lub logów, lub do pełnoprawnej relacyjnej bazy (PostgreSQL) dla większej liczby użytkowników. Dzięki warstwie repozytoriów wymiana dostawcy bazy wymaga głównie konfiguracji i ewentualnych dostosowań migracji.

Istotnym wymogiem niefunkcjonalnym była spójność czasowa – wszystkie znaczniki czasowe przechowywane są w UTC, co ułatwia migracje między silnikami baz danych i minimalizuje błędy strefowe. Dane operacyjne są kompaktowe: definicje workflowów, zmienne, harmonogramy, historia wykonania. Brak ciężkich binarnych załączników upraszcza backupy i przenoszenie między środowiskami.

Dodatkowe uzasadnienia i scenariusze migracji:

- **MongoDB jako wariant NoSQL:** elastyczne przechowywanie definicji grafów (JSON) i logów wykonania; dobra opcja, jeśli wzrośnie liczba zapisów lub pojawi się potrzeba zapytań ad-hoc po strukturach JSON.
- **PostgreSQL dla środowiska wielużytkownikowego:** transakcyjność i blokady na poziomie wiersza przydatne przy równoczesnej edycji wielu przepływów; wsparcie dla kolumn JSONB pozwala zachować model bez dużych zmian.
- **Strategia backupu:** w SQLite plik bazy może być wersjonowany (poza repo) i kopowany jako snapshot; przy migracji do serwera relacyjnego można stosować pg_dump lub repliki read-only do raportowania.
- **Indeksowanie:** kluczowe indeksy na identyfikatorach workflow, datach uruchomień i wersjach zapewniają szybkie listowanie historii; w NoSQL należałyby obrać klucze partycjonujące (np. workflowId + data).
- **Limity i rotacja:** historia wykonania może być rotowana (np. ostatnie N wpisów per workflow) w SQLite; przy migracji do bazy serwerowej można dodać mechanizm archiwizacji do cold storage.
- **Spójność:** transakcje EF Core zabezpieczają zapis definicji i powiązanych bloków; w scenariuszu NoSQL trzeba rozważyć dwufazowe zapisy lub wzorce outbox, co jest świadomie odłożone.
- **Przybliżone metryki:** mała baza SQLite (kilkadziesiąt workflowów, setki wykonania) mieści się w kilku MB; zapytania historii są praktycznie natychmiastowe lokalnie. Przy wzroście do tysięcy wykonania na workflow może być potrzebne dodanie indeksu po dacie wykonania i paginacji po stronie API.

3.4 Kontrola wersji i monorepo: Git



Rysunek 4: Kontrola wersji: Git w układzie monorepo.

Repozytorium jest prowadzone jako monorepo z folderami `flowforge.api`, `flowforge.ui`, `flowforge.nunit`. Wybór Git wynika z [12]:

- **Spójność zmian:** API, UI i testy wersjonowane razem pozwalają na atomowe commity obejmujące całą funkcjonalność (np. nowy endpoint + widok + test).
- **Prosta historia:** pojedyncza gałąź główna z krótkimi feature branchami ułatwia śledzenie regresji i code review.
- **Integracja z narzędziami:** Git współpracuje z CI/CD, choć w projekcie lokalnym kluczowe jest łatwe cofanie zmian przy pomocy commitów i tagów.

W monorepo utrzymywane są osobne pliki konfiguracyjne (solution .NET, package.json dla UI), co ułatwia budowę niezależnych artefaktów. Brak wielu pakietów NPM publikowanych do rejestru obniża złożoność; komponenty współdzielone są po prostu importowane względnie wewnątrz projektu.

Git wspiera także pracę w trybie „zaufanego użytkownika”: nie ma złożonego modelu ról w aplikacji, więc kontrola dostępu do kodu i historii zmian pozostaje w systemie wersjonowania. Dla pojedynczego użytkownika szczególnie wartościowa jest możliwość tworzenia tagów lub branchy eksperymentalnych przy zmianach przepływu.

Dodatkowe praktyki w monorepo:

- **Konwencje commitów:** krótkie, rozkazujące komunikaty ułatwiają śledzenie zmian i automatyczne generowanie changelogów.
- **Izolacja zależności:** `flowforge.ui` posiada własne `node_modules`, a `flowforge.api` korzysta z `dotnet restore`; brak globalnych instalacji zmniejsza ryzyko konfliktów wersji.
- **Testy w gałęziach:** `dotnet test` można uruchomić lokalnie przed scaleniem; w przyszłości łatwo dodać prosty pipeline CI (GitHub Actions) bez zmiany struktury repo.
- **Tagowanie releasów:** tagi mogą odpowiadać migawkom bazy lub stabilnym wersjom API/UI, co ułatwia rollback.
- **.gitignore i artefakty:** ignorowane są `bin/`, `obj/`, `node_modules/` oraz pliki tymczasowe LaTeX; utrzymuje to repo czyste i zmniejsza szanse na konflikty.
- **Strategia branchy:** krótkie feature branche, merge bez squash dla zachowania historii; przy większym zespole można wprowadzić trunk-based z krótkimi FF merges.
- **Przybliżone metryki:** repo z kodem źródłowym (bez `node_modules`) to rzad setek MB; clone na lokalnym SSD trwa sekundy. `dotnet restore` i `npm install` jednorazowo zajmują kilkaset MB, co jest akceptowalne dla pojedynczego dewelopera.

3.5 Uzasadnienie wyboru stosu jako całości

Zestaw (Vite + React + TS) + (ASP.NET Core + EF Core + SQLite) + Git został dobrany pod kątem:

- **Szybkiego startu deweloperskiego:** HMR w Vite, brak instalacji bazy serwerowej, szablon `dotnet new` i automatyczne migracje.
- **Małego narzutu operacyjnego:** pojedynczy proces API z wbudowanym schedulerem, frontend statyczny serwowany z tego samego serwera lub przez Vite dev server w trybie deweloperskim.
- **Łatwej refaktoryzacji:** typowanie w TS oraz w C#, wspólny kontrakt JSON redukuje błędy integracji.
- **Rozszerzalności:** możliwość podmiany bazy, dodania autoryzacji lub zewnętrznego schedulera bez zmiany podstawowego modelu danych i kontraktów API.
- **Kosztu:** brak opłat licencyjnych, brak wymogu hostingu usług dodatkowych (broker, kolejki, zewnętrzna baza) w środowisku małego zespołu.

Takie podejście wspiera wymagania funkcjonalne (graficzne modelowanie, wersjonowanie, harmonogram, historia uruchomień) oraz niefunkcjonalne (lekkość, przewidywalność, niski nakład utrzymania).

4 Projekt architektury aplikacji

4.1 Opis architektury

Architektura ma prosty podział klient-serwer. Klient (Vite + React + React Flow) renderuje interfejs w przeglądarce i komunikuje się z API wyłącznie przez HTTP/JSON. Serwer to pojedyncza aplikacja ASP.NET Core (Kestrel), w której:

- kontrolery REST wystawiają operacje CRUD na workflowach, wersjach, harmonogramach i wykonaniach,
- serwisy domenowe kapsulkują logikę biznesową (walidacja grafu, tworzenie migawek, obliczanie `NextRunAtUtc`, wykonywanie bloków),
- repozytoria EF Core mapują encje na SQLite i prowadzą migracje schematu,
- `BackgroundService` pełni rolę lekkiego schedulera, działając w tym samym procesie co API, bez zewnętrznego brokera czy kolejki.

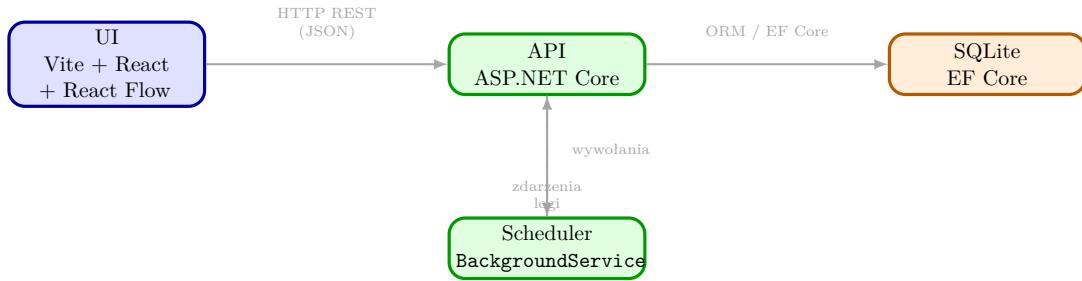
Świadomie zrezygnowano z warstw pośrednich (reverse proxy, message broker, worker pool), by zmniejszyć złożoność i zależności. W trybie deweloperskim frontend może być serwowany przez Vite dev-server (HMR), a w produkcji – jako statyczne pliki z tego samego procesu ASP.NET Core. Całość mieści się na jednej instancji, co upraszcza wdrożenie w środowisku pojedynczego użytkownika lub małego zespołu.

4.2 Diagramy architektoniczne

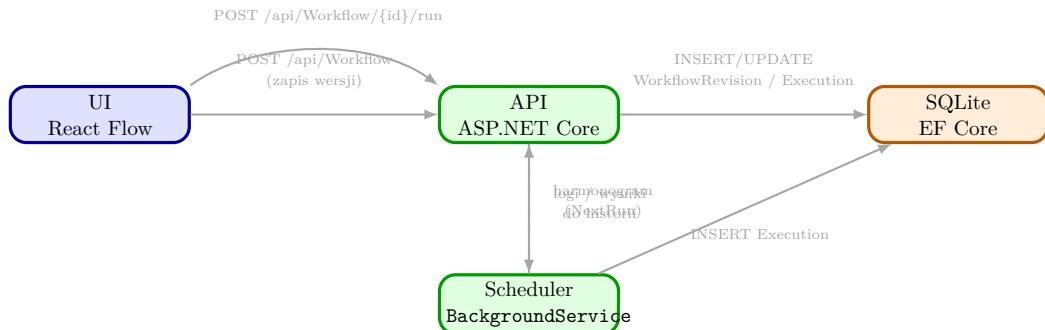
Na potrzeby dokumentacji wyróżniono dwa widoki graficzne oraz listę przypadków użycia:

- **Diagram komponentów** (następny podrozdział) – ukazuje podział na UI, API + scheduler oraz SQLite, wraz z interfejsami komunikacji (REST, ORM) i zwrotną wymianą logów.
- **Diagram przepływu danych** – pokazuje główne ścieżki żądań i zapisów: tworzenie wersji, uruchomienie ręczne, harmonogram oraz zapis historii. Etykiety strzałek odpowiadają scenariuszom opisany w sekcji „Przepływ danych i przypadki użycia”.
- **Przypadki użycia** – cztery podstawowe scenariusze: edycja workflowu, uruchomienie ręczne, wykonanie cykliczne oraz przegląd/przywracanie historii. Ze względu na pojedynczego użytkownika i brak ról, scenariusze są opisane tekstowo bez dodatkowego diagramu UML.

4.3 Diagram komponentów



4.4 Przepływ danych i przypadki użycia



Przepływ danych: edycja i zapis wersji, uruchomienia ręczne, harmonogram oraz zapis historii w SQLite.

Działanie poszczególnych scenariuszy (odpowiadają strzałkom na głównym diagramie):

- **Edycja przepływu:** UI zapisuje definicję przez POST /api/Workflow; API tworzy nową WorkflowRevision w SQLite.
- **Uruchomienie ręczne:** UI wywołuje POST /api/Workflow/{id}/run; API zapisuje WorkflowExecution i wynik w bazie.
- **Harmonogram:** BackgroundService oblicza NextRunAtUtc, wywołuje API, a zapis wykonania trafia do SQLite.
- **Historia i wersje:** UI pobiera WorkflowRevision i WorkflowExecution; przywracanie przez POST /api/WorkflowRevision/{id}/restore.

4.5 Warstwy i interakcje

Warstwy są rozdzielone, ale współpracują w prostych, powtarzalnych kanałach:

- UI ↔ API: żądania HTTP/JSON (zapis definicji, uruchomienia, harmonogramy, historia); odpowiadają stany domenowe mapowane 1:1 na strukturę frontu.
- API ↔ SQLite: EF Core tłumaczy encje na zapytania; migracje utrzymują zgodność schematu. Zapis/odczyt jest transakcyjny w obrębie pojedynczych operacji.
- Scheduler ↔ API/DB: `BackgroundService` wywołuje serwisy domenowe tak samo jak klient, dzięki czemu logika nie jest dublowana; po wykonaniu zapisuje `WorkflowExecution`.
- UI → statyczne zasoby: w produkcji pliki Vite są serwowane z tego samego procesu ASP.NET Core, w deweloperskim z Vite dev-server (HMR), bez wpływu na API.
- **Prezentacja:** komponenty React tworzą edytor grafu (React Flow), listę wersji, historię uruchomień i scheduler. Komunikacja z API odbywa się przez wywołania fetch/axios z JSON. Stan lokalny przechowuje graf w tej samej strukturze co backend (węzły, krawędzie, zmienne), co ogranicza mapowanie i ryzyko niespójności.
- **API:** kontrolery ASP.NET Core (m.in. workflow, workflow revision, schedule, execution) wystawiają kontrakty REST. Serwisy domenowe walidują graf, tworzą migawki, obliczają `NextRunAtUtc` i uruchamiają przepływy. Repozytoria EF Core mapują encje na SQLite, a DI ASP.NET Core zapewnia wstrzykiwanie kontekstów i serwisów.
- **Scheduler:** pojedynczy `BackgroundService` działa w tym samym procesie co API, cyklicznie (ok. 30 s) pobiera aktywne harmonogramy, oblicza najbliższe uruchomienia, wywołuje serwis uruchomień i aktualizuje `LastRunAtUtc/NextRunAtUtc`. Brak rozproszonego lockingu oznacza, że uruchomienie wielu instancji wymagałoby dodatkowej synchronizacji.
- **Dane:** EF Core utrzymuje migracje oraz integralność referencyjną. Encje odwzorowują workflow, bloki, połączenia, wersje, harmonogramy, wykonania i zmienne; JSON definicji przepływu przechowywany jest w kolumnach tekstowych, a daty w UTC. Plik bazy (`flowforge.api/flowforge.db`) umożliwia przenoszenie środowiska jednym plikiem.

4.6 Projekt UI/UX

Interfejs ma wspierać szybkie modelowanie przepływów bez przeładowania ekranów. Przyjęto lekkie własne style (CSS + zmienne kolorów) zamiast ciężkiego framework'a, co obniża narzut i ułatwia utrzymanie spójności jasnego oraz ciemnego motywów.

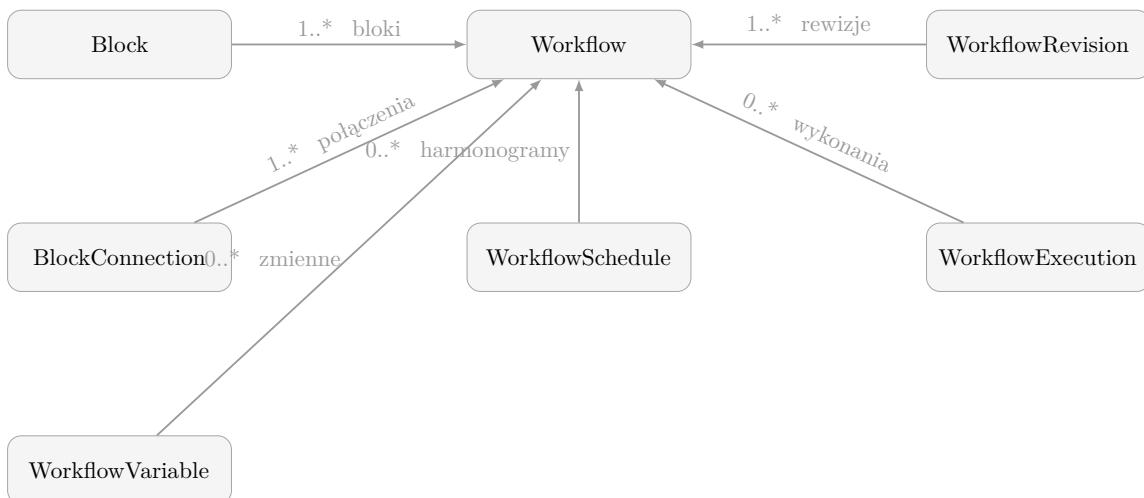
- **Edytor grafu:** canvas React Flow z walidacją połączeń, blokadą cykli i pilnowaniem unikalnych identyfikatorów bloków. UI odwzorowuje typy portów, dzięki czemu błędne krawędzie są blokowane na etapie rysowania, a zapis nie wymaga dodatkowej korekty.
- **Wersje:** modal listuje `WorkflowRevision` wraz z datą i etykietą; każda operacja zapisu tworzy migawkę automatycznie. Użytkownik może przywrócić lub usunąć wersję bez ręcznego eksportu pliku.
- **Scheduler:** formularz z trybami *Once*, *Interval*, *Daily*; pola formularza włączają się kontekstowo. `NextRun/LastRun` są pokazywane w czasie lokalnym przeglądarki, a przechowywane w UTC, co ogranicza błędy stref czasowych.
- **Historia uruchomień:** tabela prezentuje datę, status, wynik i ścieżkę bloków; szczegóły wykonania (wejścia/wyjścia) są uporządkowane chronologicznie, co ułatwia analizę regresji i porównanie z wersją przepływu.
- **Nawigacja i stan:** panel główny prezentuje listę workflowów; wejście w edytor jest jednoekranowe (brak zakładek), co skraca czas przełączania kontekstu. Stany ladowania i błędów są sygnalizowane lekkimi toasts/alertami bez blokowania ekranu.

- **Formularze i walidacja:** pola liczbowe (np. interwał) ograniczone zakresem, walidacja klienta od razu blokuje zapis, a komunikaty błędów są zwięzłe (po polsku) i wskazują konkretny parametr.
- **Dostępność:** czytelny kontrast dla tekstu i krawędzi węzłów, wielkość fontu nie mniejsza niż 14 px; klawisz Esc zamknie modal, Tab obsługuje kolejność pól formularza.
- **Typografia i kolory:** wykorzystano czystą rodzinę Lato/Noto bez szeryfów, rozmiar bazowy 16 px, zmienne kolorów (akcent zielony) dla zachowania spójności w obu motywach.
- **Responsywność:** layout zoptymalizowany pod laptopy i monitory biurkowe; kolumny tabel zwijają mniej istotne kolumny na mniejszych szerokościach. Brak wsparcia mobilnego jest świadomym ograniczeniem (docelowo środowisko desktop).

4.7 Model bazy danych (szkic ERD)

Najważniejsze decyzje projektowe dla warstwy danych:

- **SQLite jako magazyn:** pojedynczy plik `flowforge.db` pozwala przenieść środowisko kopiując plik; brak zależności na serwer bazy. Typy kolumn dobrane do SQLite (INTEGER, TEXT, DATETIME w UTC).
- **Daty w UTC:** pola `NextRunAtUtc`, `LastRunAtUtc`, `AppliedAt` przechowywane w UTC, a UI prezentuje czas lokalny, co upraszcza harmonogram.
- **Definicje w JSON:** struktura workflowu (bloki, krawędzie, zmienne) serializowana do JSON w `WorkflowRevision`; pozwala na ewolucję schematu bez migracji kolumn dla każdej właściwości węzła.
- **Migracje EF Core:** każda zmiana modelu jest wersjonowana w migracjach, co umożliwia powtarzalne odtwarzanie bazy w dewelopmencie.
- **Spójność:** relacje $1..*$ do workflowu; brak cascadowych usunięć dla rewizji i wykonania (chronią historię), usuwanie wymaga jawniej operacji w API.
- **Indeksy:** klucz główny każdej encji (INTEGER AUTOINCREMENT); indeksy na `WorkflowId` w tabelach zależnych oraz na `IsActive` w `WorkflowRevision` i `WorkflowSchedule` (szybki wybór bieżącej wersji i aktywnych harmonogramów).



Rysunek 5: Szkic ERD kluczowych encji systemu: workflow jest centralną encją, do której należą rewizje, harmonogramy, wykonania, bloki, połączenia i zmienne.

5 Implementacja

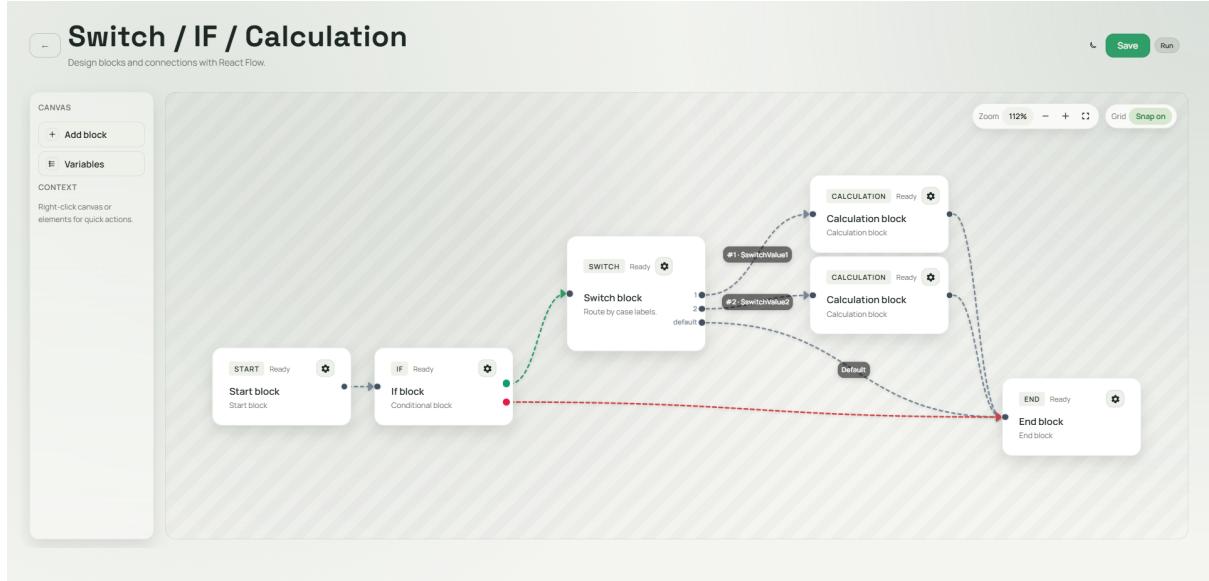
5.1 Kluczowe moduły i funkcjonalności

- **WorkflowController / WorkflowService:** tworzenie, edycja i uruchamianie workflowów; walidacja grafu (brak cykli, poprawne połączenia); migawka `WorkflowRevision` przy każdym zapisie; mapowanie DTO ↔ encje.
- **WorkflowRevisionController:** listowanie, przywracanie i usuwanie rewizji; jedna aktywna wersja na workflow (`IsActive=true`).
- **WorkflowScheduleController + WorkflowSchedulerHostedService / SchedulerService:** CRUD harmonogramów, `NextRunAtUtc` dla trybów *Once/Interval/Daily*, „Run now”, zapis `LastRunAtUtc`, stan aktywności.
- **WorkflowExecutionController:** odczyt historii wykonań, wynik, status, ścieżka bloków; dane do tabeli Executions (JSON).
- **BlockController / BlockService:** CRUD bloków, walidacja typu i pozycji na kanwie; format zgodny z React Flow.
- **BlockConnectionController / BlockConnectionService:** tworzenie/usuwanie krawędzi, kontrola typów portów, zapobieganie cyklom.
- **WorkflowVariableController / WorkflowVariableService:** zmienne wejściowe/środowiskowe przypisane do workflowu, serializowane do definicji.
- **SystemBlockController / SystemBlockService:** katalog bloków systemowych (szablony akcji) dostępnych w edytorze.
- **BackgroundService (worker):** cykliczny worker (30 s) w procesie API; pobiera aktywne harmonogramy, wywołuje `WorkflowService.RunScheduledAsync`, zapisuje `WorkflowExecution`; brak brokera/kolejki.

5.2 Interfejs użytkownika

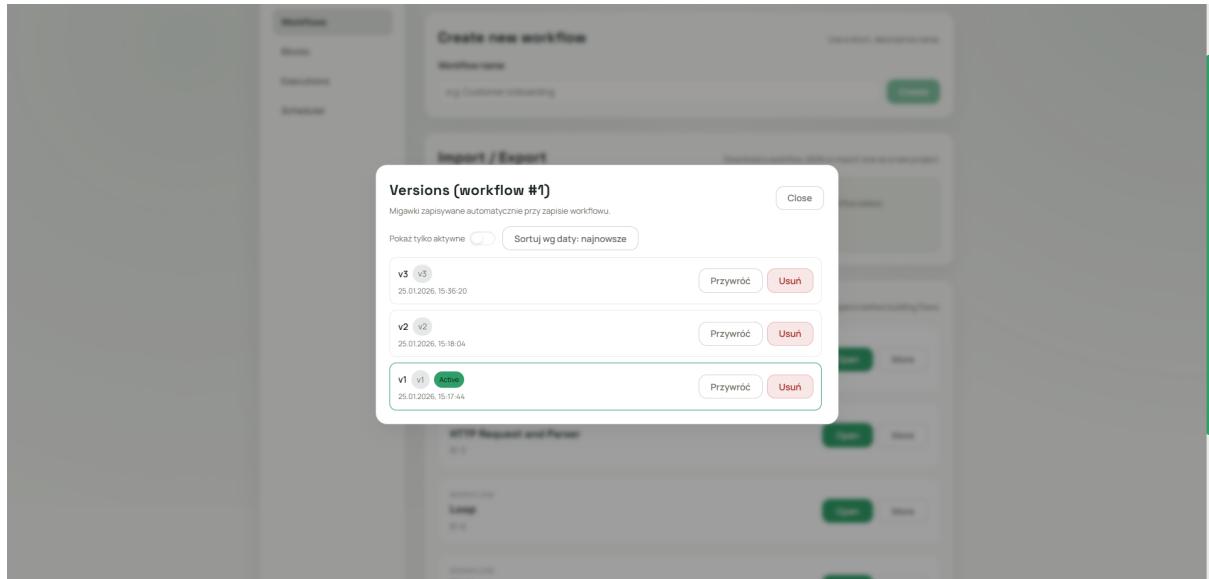
UI w `flowforge.ui` (Vite + React + TypeScript + React Flow) zostało zaprojektowane jako lekka, szybka aplikacja przeglądarkowa, której struktura danych pozostaje zgodna z modelem domenowym API. Każdy kluczowy widok opisano poniżej wraz z poglądową ilustracją (placeholder PNG); docelowo można tu wstawić zrzuty ekranu z działającej aplikacji.

Edytor grafu Płotno React Flow odwzorowuje węzły i krawędzie workflowu, blokuje cykle i niepoprawne połączenia typów. Struktura grafu w pamięci jest identyczna ze schematem zwracanym przez API.



Rysunek 6: Edytor grafu workflowu (React Flow)

Modal wersji Modal *Versions* prezentuje listę rewizji, umożliwia przywracanie i usuwanie oraz pokazuje aktywną wersję. Migawka tworzona jest automatycznie przy każdym zapisie definicji.



Rysunek 7: Widok modala wersji

Scheduler Formularz trybów *Once/Interval/Daily* pokazuje tylko relevantne pola; wartości **NextRun/LastRun** są prezentowane w czasie lokalnym, a logika czasu pozostaje w UTC.

Rysunek 8: Formularz harmonogramu

Historia uruchomień Tabela *Executions* pokazuje status, datę lokalną, wynik i ścieżkę bloków; szczegóły można rozwinąć inline.

Rysunek 9: Historia uruchomień workflowu (widok 1)

Execution #83

Workflow: TextReplace Executed at: 28.01.2026, 17:35:16 Path length: 3

Inputs

```
[{"text": "Hello", "result": ""}]
```

Results

```
[{"text": "Hello", "result": "Hello me take a look mate"}]
```

Path

```
Start --> Text Replace --> End
```

Actions

- Start block
- TextReplace → result (1 rule(s))
- End block

Rysunek 10: Historia uruchomień workflowu (widok 2)

Menu kontekstowe i toolbar Prawy klik na węźle lub krawędzi pozwala na usunięcie/duplikację. Toolbar zawiera akcje zapisu, cofnięcia oraz otwarcia modalni Versions i Scheduler.

Switch / IF / Calculation

Design blocks and connections with React Flow.

CANVAS

- + Close palette
- E Variables

CONTEXT

Right-click canvas or elements for quick actions.

Blocks

Add blocks anywhere on the canvas.

Add block

Search blocks... All Flow Logic Action

Flow

- Start Entry point for the workflow.
- End Finish the workflow.
- Wait Pause execution for a duration.

Logic

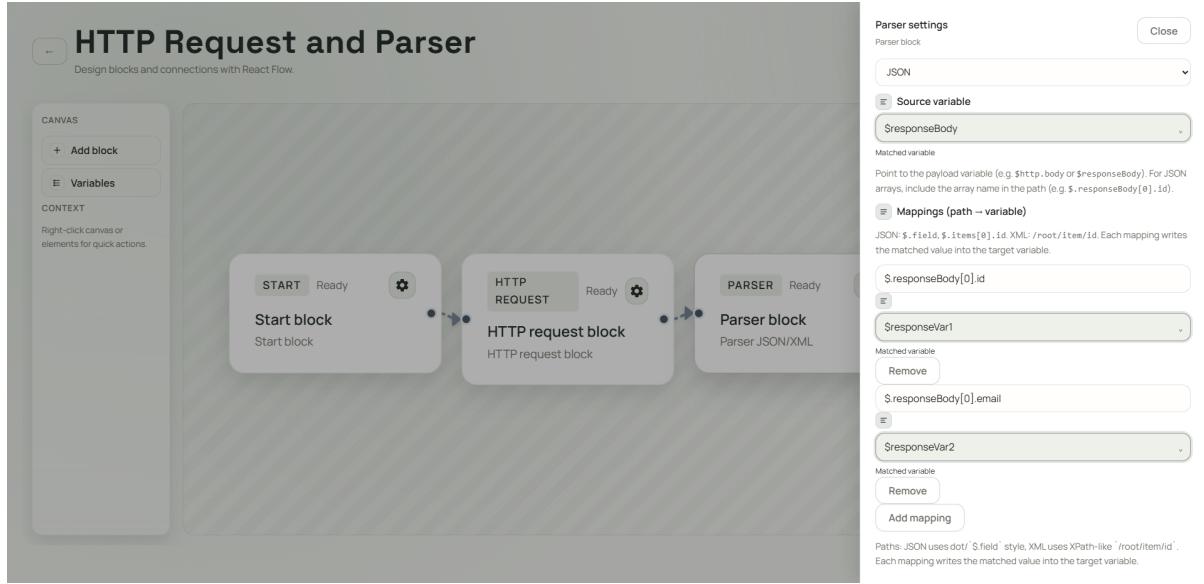
- If Route based on a condition.
- Switch Multi-branch on cases.
- Loop Repeat a branch multiple times.
- Text Transform Trim / lower / upper.
- Calculation Compute variables.

Action

- HTTP request Call external HTTP APIs.
- Parser Extract values from JSON or XML.

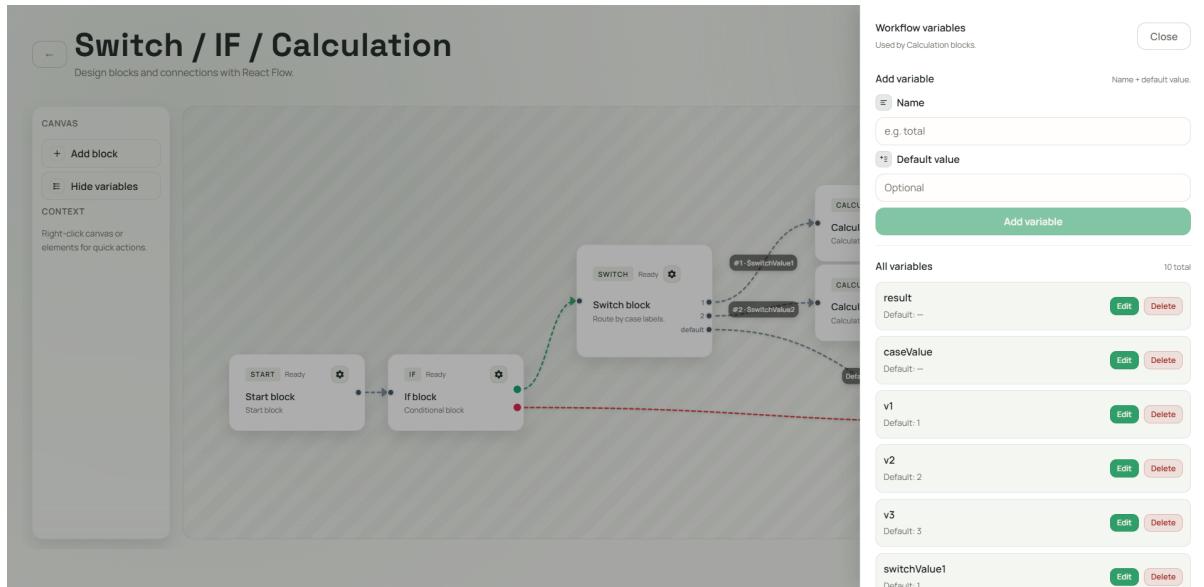
Rysunek 11: Menu kontekstowe oraz toolbar edytora

Konfiguracja bloków Panel boczny umożliwia edycję etykiety, parametrów wejścia/wyjścia i pozycji węzła. Zmiany od razu aktualizują stan grafu i definicję zapisywana w API.



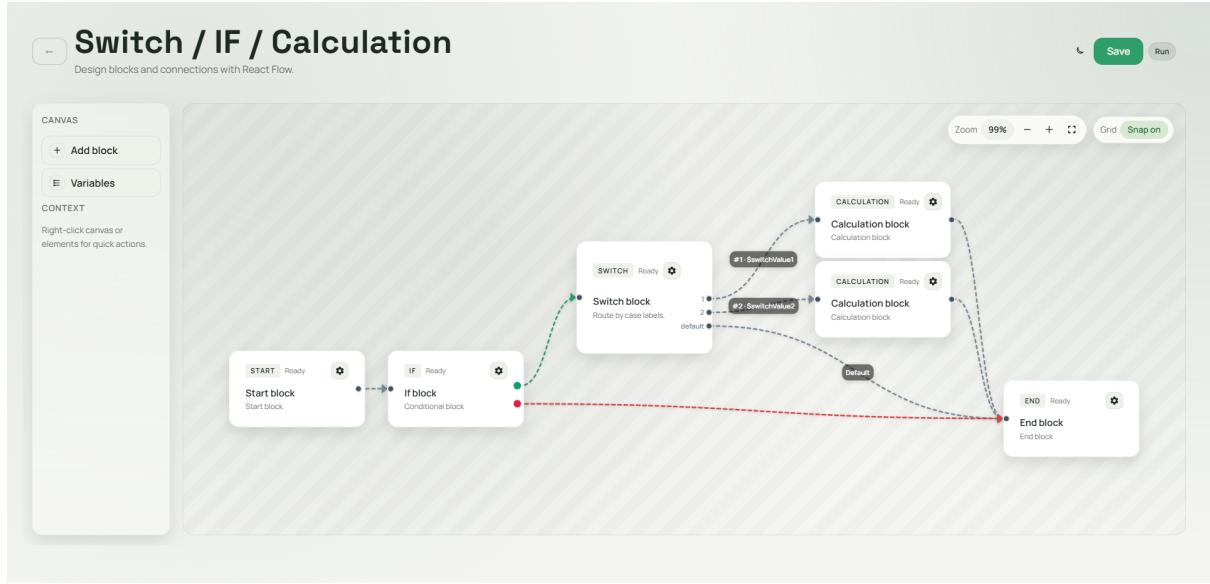
Rysunek 12: Panel konfiguracji bloku

Zarządzanie zmiennymi Widok *Variables* pozwala definiować zmienne wejściowe/środowiskowe z validacją unikalności i typu; dane są serializowane do `WorkflowRevision`.

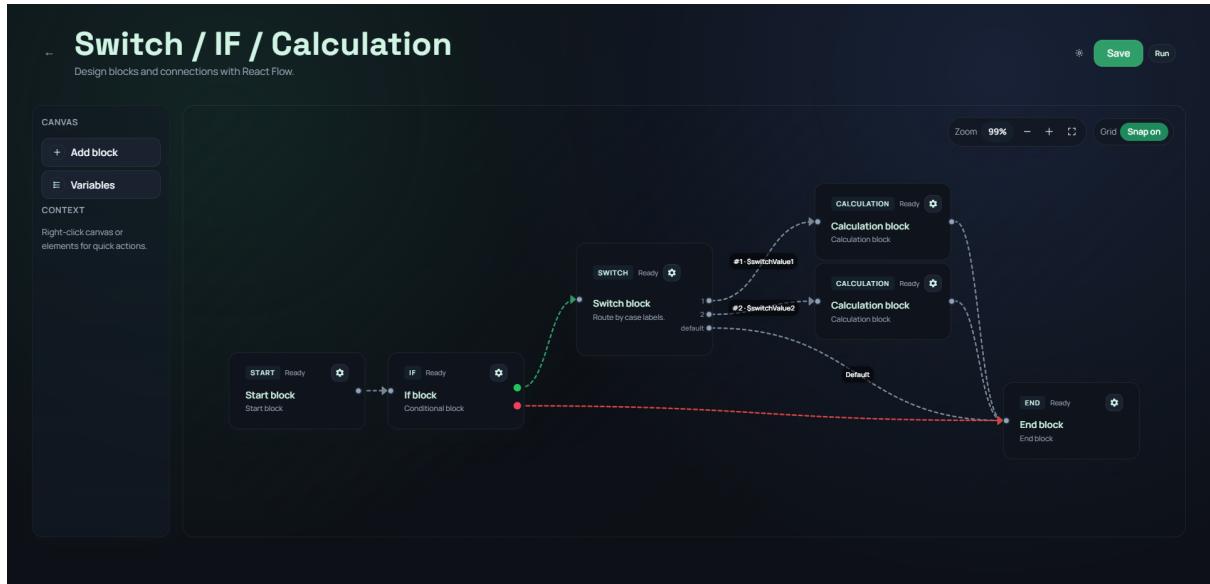


Rysunek 13: Edycja zmiennych przepływu

Styl i motyw Lekki CSS oparty na zmiennych kolorów (akcent zielony), bez Bootstrap/Material; układ responsywny na laptopy/desktop.



Rysunek 14: Stylistyka i motyw interfejsu – motyw jasny



Rysunek 15: Stylistyka i motyw interfejsu – motyw ciemny

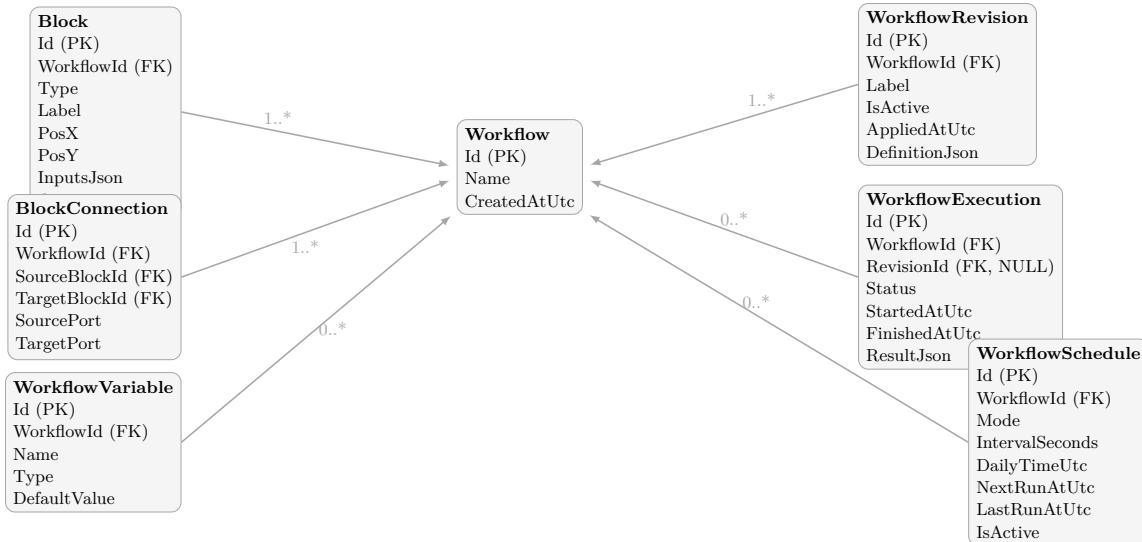
5.3 Obsługa danych i zarządzanie bazą danych

Warstwa danych korzysta z Entity Framework Core i SQLite, aby uprościć uruchomienie lokalne i zachować możliwość migracji do docelowej bazy bez zmiany logiki biznesowej.

- **Magazyn:** w dewelopmencie pojedynczy plik `flowforge.db` (SQLite) w katalogu `flowforge.api/`. W produkcji możliwa podmiana connection stringa na serwerową bazę relacyjną [6].
- **Migracje:** folder `Migrations/` zawiera zmiany schematu; polecenie `dotnet ef database update` odtwarza bazę. Migracje są wersjonowane w repozytorium [2].
- **Model encji:** `Workflow` z powiązaniemi do `WorkflowRevision`, `WorkflowSchedule`, `WorkflowExecution`, `Block`, `BlockConnection`, `WorkflowVariable`. Definicje workflowu są serializowane do JSON w tabeli `WorkflowRevision`, co pozwala ewoluować strukturę węzłów bez każdorazowych migracji kolumn [2].
- **Czas:** wszystkie znaczniki (`AppliedAt`, `NextRunAtUtc`, `LastRunAtUtc`) przechowywane w UTC; konwersja do strefy użytkownika wykonywana w UI.
- **Integralność:** klucze obce zabezpieczają relacje do `Workflow`; brak kaskadowego usuwania rewizji i wykonania chroni historię. `IsActive` na `WorkflowRevision` i `WorkflowSchedule` utrzymuje status bieżących rekordów.
- **Indeksy:** na `WorkflowId` w tabelach zależnych oraz na `IsActive`; poprawiają odczyt aktywnych wersji i harmonogramów. Unikalność identyfikatorów bloków i krawędzi pilnowana w logice serwisu.
- **Backup i odtwarzanie:** kopia bazy to skopiowanie pliku `flowforge.db`. W CI/preview baza jest tworzona od zera z migracji, by uniknąć zależności od binariów.
- **Walidacja danych:** atrybuty modelu + walidacja serwisów (cykle, typy portów, spójność grafu) przed zapisaniem JSON; `ApiController` zwraca błędy `ModelState` do UI.
- **Seed/fixtures w CI:** pipeline testowy uruchamia migracje na czystej bazie SQLite i opcjonalnie wgrywa minimalne dane startowe (1 workflow, 1 rewizja, brak wykonania), co zapewnia powtarzalność testów bez zależności od pliku `flowforge.db`.

Mapowanie modeli w warstwie backend

- **Workflow:** agregat główny; serwis waliduje graf (cykle, typy portów) i zapisuje nową `WorkflowRevision` przy każdej zmianie. Zawiera kolekcje bloków, połączeń, rewizji, harmonogramów, wykonień i zmiennych.
- **WorkflowRevision:** migawka definicji w `DefinitionJson`. Tylko jedna rewizja może być aktywna (`IsActive=true`); przywracanie wersji uaktualnia ten znacznik.
- **WorkflowSchedule:** harmonogram z trybem Once/Interval/Daily, polami czasu w UTC i flagą `IsActive`. `WorkflowSchedulerHostedService` czyta aktywne rekordy i aktualizuje `NextRunAtUtc`/`LastRunAtUtc`.
- **WorkflowExecution:** zapisuje wynik uruchomienia (status, czas start/koniec, rezultat) z referencją do workflowu i opcjonalnie rewizji, co umożliwia audyt względem wersji definicji.
- **Block / BlockConnection:** reprezentują węzły i krawędzie; pozycje i porty są serializowane do JSON w rewizji, ale encje pozwalają na walidację i CRUD w API. Krawędzie wskazują węzły źródło/cel przez FK.
- **WorkflowVariable:** zmienne powiązane z workflow; typ i wartość domyślna trafiają do definicji i są odtwarzane w UI.
- **SystemBlock:** katalog bloków systemowych (szablony akcji) wystawiany przez `SystemBlockController`; nie ma FK do workflow.



Rysunek 16: Graficzny schemat relacji w bazie `flowforge.db`.

5.4 Integracja frontendu i backendu

Aktualna integracja jest w całości REST/JSON; frontend (Vite + React + TS) wywołuje kontrolery ASP.NET Core wprost, bez warstwy pośredniej.

- **Kontrakt:** DTO w UI są 1:1 z modelami API (workflow, revision, schedule, execution, block, connection, variable); jedyna konwersja to daty UTC ↔ czas lokalny w UI [1].
- **Transport:** fetch/axios, Content-Type: application/json. Brak autoryzacji (tryb zaufanego użytkownika). Kody 4xx/5xx mapowane na toastify/alerty; błędy ModelState wyświetlane inline w formularzach.
- **Spójność wersji:** zapis workflowu zawsze tworzy nową WorkflowRevision; UI po zapisie odczytuje aktywną rewizję, minimalizując rozjazd między stanem grafu w pamięci a danymi w bazie.
- **Obsługa czasu:** backend przechowuje daty w UTC; UI pokazuje je w czasie lokalnym (scheduler, historia). Wymiana dat w ISO 8601.
- **Błędy i retry:** brak automatycznego retry; UI prezentuje komunikat i pozwala powtórzyć akcję. Dłuższe operacje (run workflow) mają optymistyczny stan „in progress” do chwili otrzymania statusu WorkflowExecution.
- **Konfiguracja adresu API:** endpoint jest w zmiennej konfiguracyjnej frontu; w CI testy end-to-end korzystają z lokalnie uruchomionego API.
- **Ścieżki i paginacja:** listy wersji i wykonań są pobierane stronicowane (skip/take); identyfikator workflowu przekazywany jest w ścieżce (/api/Workflow/{id}/...) i przechowywany w stanie routera UI.
- **Walidacja i kody:** 400 z wypełnionym ModelState (pokazywane per pole), 404 dla zasobów, 409 dla konfliktu wersji (potencjalnie przy wielu instancjach). Błędy domenowe (np. cykl w grafie) są zwracane jako zwięzły komunikat.
- **Swagger/OpenAPI:** w trybie deweloperskim włączony jest Swagger UI, co pozwala zsynchronizować kontrakty z typami TS i ręcznie wywołać endpointy podczas debugowania.
- **CORS i proxy dev:** w produkcji statyczne pliki i API mogą być serwowane z jednego hosta (brak CORS); w deweloperskim Vite dev-server działa z proxy na API, więc nie ma potrzeby dodatkowych nagłówków.
- **Idempotencja zapisu:** każda operacja zapisu tworzy rewizję, ale aktywna wersja jest nadpisywana zgodnie z IsActive; UI po zapisie odczytuje zwróconą rewizję, co ogranicza wyścigi.
- **Limit rozmiaru i typy:** definicje workflowów to JSON rzędu kilkudziesięciu kB; domyślny limit ASP.NET Core (30 MB) jest zapasem. Typy logiczne i daty są serializowane według domyślnych reguł System.Text.Json (camelCase).
- **Brak kanałów push:** komunikacja pozostaje żądanie–odpowiedź; długie operacje są odpytywane przez UI (polling wykonania), co upraszcza wdrożenie bez WebSocketów.

6 Testowanie

6.1 Rodzaje testów

- **Jednostkowe (NUnit)** [7]:
 - harmonogram i serwisy: `CalculateNextRun` (Once/Interval/Daily), walidacja cykli, brak bloku Start, ścieżki Switch/Loop/Wait w `WorkflowExecutionService`;
 - repozytoria (CRUD) na EF Core InMemory [2];
 - kontrolery: kody 200/400/404, zgodność `id` w ścieżce i body;
 - egzekutorzy bloków: `Calculation` (także divide-by-zero), `If`, `TextReplace`, `TextTransform`, `Parser` (JSON/XML), `Switch`, `Default`, `HttpRequest` na stubie HTTP.
- **Integracyjne**: brak – planowane przez `WebApplicationFactory` + SQLite/InMemory do weryfikacji całego pipeline'u HTTP, migracji i modeli DTO.
- **Manualne**: UI (`npm run dev`) + Swagger; klikane scenariusze: zapis/rewizje, harmonogram *Once/Interval/Daily*, uruchomienie ręczne, przywrócenie wersji, historia uruchomień, komunikaty walidacji formularzy.

6.2 Narzędzia testujące

- **NUnit 4.x + Microsoft.NET.Test.Sdk + NUnit3TestAdapter**: uruchamianie testów w `flowforge.nunit` [7].
- **Moq.EntityFrameworkCore + EF Core InMemory**: izolowane testy serwisów i repozytoriów bez realnej bazy [9][2].
- **coverlet.collector**: opcjonalne zbieranie pokrycia `-collect:plat Code Coverage` [10].
- **Swagger UI** i przeglądarka: szybka weryfikacja ręczna kontraktów API [11].

6.3 Wyniki testów (28.01.2026, lokalnie)

- `dotnet test flowforge.nunit/Flowforge.NUnit.csproj` – 333/333 testów przeszło (lokalnie, 28.01.2026); NU1900 to ostrzeżenie o braku dostępu do indeksu NuGet, nie wpływa na wynik.
- Pokrycie jednostkowe (skrócone podsumowanie): harmonogram (Once/Interval/Daily), walidacja grafu i braków bloków, ścieżki Switch/Loop/Wait, CRUD kontrolerów, repozytoria, egzekutorzy bloków (Calculation/If/TextReplace/TextTransform/Parser/Switch/Default/HttpRequest). Scenariusze negatywne: brak Start, brak zmiennych, błędne etykiety Switch, divide-by-zero, brak zmiennej źródłowej w Parserze.
- Pokrycia procentowego nie liczono w tej sesji; dla pełnej metryki można uruchomić `-collect:plat Code Coverage` i przeanalizować raport coverlet.

6.4 Błędy i sposoby ich naprawy

- **Rozjazdy czasu w schedulerze**: ujednolicono UTC w bazie i konwersję w UI; testy `CalculateNextRun` na tryby Once/Interval/Daily wykrywają regresje.
- **Cykle w grafie**: dodano walidację w serwisie i testy, które blokują zapisy z pętlą nieskończoną.
- **Brak bloków startowych**: testy wykonania sprawdzają, że brak Start nie wywołuje bloków; UI sygnalizuje błąd zapisując definicję.
- **Krawędzie typu Switch**: testy sprawdzają dopasowanie etykiet po normalizacji; naprawiono przypadki, gdy etykieta zaczynała się od # lub zmiennej.
- **Walidacja wejścia w kontrolerach**: atrybuty `[ApiController]` + testy kontrolerów pilnują zwracania 400/404; naprawiono sytuacje, gdy brakowało sprawdzenia zgodności `id` w ścieżce z body.

7 Wyzwania i rozwiązania

7.1 Problemy napotkane podczas realizacji

- Rozbieżności stref czasowych między UI a API (harmonogram, historia).
- Cykle w grafie i błędne etykiety krawędzi `Switch`, powodujące złe ścieżki wykonania.
- Brak bloków startowych lub brak zmiennych wejściowych przy uruchomieniu workflowu.
- Brak pełnych testów dla egzekutorów bloków (HTTP, Parser, TextTransform) i schedulera.
- Potencjalne konflikty schematu przy przywracaniu rewizji (usuwanie bloków/krawędzi).

7.2 Metody rozwiązańia

- Ujednolicono przechowywanie czasu w UTC w bazie (`NextRunAtUtc`, `LastRunAtUtc`, `ExecutedAt`) i konwersję w UI [1]; testy `CalculateNextRun` w NUnit potwierdzają poprawność.
- Dodano validację cykli i typów portów w serwisie oraz w UI (React Flow) [5]; testy pokrywają ścieżki `Switch/Loop/Wait` i etykiety z prefiksem #.
- Wykonanie bez bloku `Start` zwraca pustą ścieżkę; UI blokuje zapis, a testy jednostkowe sprawdzają scenariusz braku startu.
- Dopuszczono testy egzekutorów: `HttpRequest` (stub HTTP), `Parser` (JSON/XML), `TextTransform`, `Calculation` (divide-by-zero), co ogranicza regresje [7].
- W przywracaniu rewizji usuwane są krawędzie przed blokami, co eliminuje błędy referencyjne w EF Core [2].

7.3 Optymalizacje i usprawnienia

- Minimalny zestaw middleware (CORS, routing, Swagger dev) zmniejsza opóźnienia i złożoność serwera [1].
- Prosty model danych w SQLite z JSON dla definicji workflowu ułatwia migracje i utrzymanie [6].
- Brak zewnętrznego brokera – harmonogram w `BackgroundService` redukuje narzut operacyjny; ewentualną skalowalność można dodać przez rozproszony lock w przyszłości.
- UI bez ciężkich frameworków CSS (własne zmienne + React Flow) utrzymuje mały bundle i szybkie HMR [5][4].
- Monorepo Git pozwala na atomowe zmiany API/UI/testów; testy jednostkowe (NUnit) odpalone lokalnie są szybkie (1 s) i nie wymagają zewnętrznych usług [12][7].

8 Podsumowanie i wnioski

8.1 Ocena realizacji celu

Zaprojektowano i zaimplementowano lekki system low-code do budowy i uruchamiania przepływów API. Cel uproszczenia orkiestracji dla pojedynczego użytkownika został osiągnięty: można modelować workflow jako graf (React Flow [5]), wersjonować definicje (EF Core [2]), uruchamiać ręcznie lub według prostego harmonogramu (BackgroundService w ASP.NET Core [1]) i przeglądać historię wykonania. Prosty stos (ASP.NET Core + Vite/React + SQLite) obniża koszt wdrożenia i utrzymania.

8.2 Możliwości rozwoju

- Dodanie autoryzacji i rôle (JWT/OAuth) oraz logowania audytowego.
- Skalowanie harmonogramu w środowisku wieloinstancyjnym (rozproszone locki, kolejka zadań).
- Integracje bloków zewnętrznych (HTTP z retry, kolejki, e-mail/SMS) i testy integracyjne z WebApplicationFactory.
- Eksport/import przepływów, wersjonowanie schematów zmiennych oraz szersze metryki (prometheus).

8.3 Refleksje

Praca potwierdziła, że nawet w małym zespole można uzyskać pełny cykl: modelowanie, wersjonowanie, uruchamianie oraz monitoring, opierając się na bibliotekach open-source (React, React Flow, ASP.NET Core, EF Core, NUnit) [3][5][1][2][7]. Największym wyzwaniem była spójność czasu (UTC vs lokalny) i walidacja grafu; iteracyjne testy jednostkowe pozwoliły szybko eliminować regresje.

8.4 Wnioski końcowe

Flowforge w obecnym kształcie spełnia wymagania funkcjonalne i niefunkcjonalne zdefiniowane dla pojedynczego użytkownika: szybkie uruchomienie, graficzna edycja przepływów, wersjonowanie, prosty harmonogram, historia wykonania. Ograniczenia (brak autoryzacji, brak rozproszonego schedulera, brak metryk) są świadomymi kompromisami i wyznaczają kierunek dalszego rozwoju. W praktyce projekt pokazuje, że stack .NET + React + SQLite może efektywnie wspierać narzędzia low-code bez ciężkiej infrastruktury, pod warunkiem konsekwentnego testowania (NUnit) [7] i trzymania danych czasowych w UTC.

Bibliografia

Literatura

- [1] Microsoft, *ASP.NET Core documentation*, <https://learn.microsoft.com/aspnet/core>, dostęp 2026-01-28.
- [2] Microsoft, *Entity Framework Core documentation*, <https://learn.microsoft.com/ef/core>, dostęp 2026-01-28.
- [3] Meta, *React Documentation*, <https://react.dev>, dostęp 2026-01-28.
- [4] E. You, *Vite Guide*, <https://vitejs.dev/guide>, dostęp 2026-01-28.
- [5] React Flow Team, *React Flow Docs*, <https://reactflow.dev/docs>, dostęp 2026-01-28.
- [6] SQLite Consortium, *SQLite Documentation*, <https://www.sqlite.org/docs.html>, dostęp 2026-01-28.
- [7] NUnit Project, *NUnit Documentation*, <https://docs.nunit.org>, dostęp 2026-01-28.
- [8] Microsoft, *TypeScript Handbook*, <https://www.typescriptlang.org/docs/>, dostęp 2026-01-28.
- [9] Moq Project, *Moq Documentation*, <https://github.com/moq/moq>, dostęp 2026-01-28.
- [10] Coverlet Project, *Coverlet Docs*, <https://github.com/coverlet-coverage/coverlet>, dostęp 2026-01-28.
- [11] SmartBear, *Swagger/OpenAPI Documentation*, <https://swagger.io/specification/>, dostęp 2026-01-28.
- [12] S. Chacon, B. Straub, *Pro Git*, Apress 2014, dostęp on-line: <https://git-scm.com/book/en/v2>, dostęp 2026-01-28.
- [13] n8n GmbH, *n8n Documentation*, <https://docs.n8n.io>, dostęp 2026-01-28.
- [14] OpenJS Foundation, *Node-RED Documentation*, <https://nodered.org/docs/>, dostęp 2026-01-28.
- [15] FlowiseAI, *Flowise Docs*, <https://docs.flowiseai.com>, dostęp 2026-01-28.
- [16] LangFlow Project, *LangFlow Documentation*, <https://docs.langflow.org>, dostęp 2026-01-28.
- [17] Camunda, *Camunda Modeler Documentation*, <https://docs.camunda.io/docs/components/modeler/desktop-modeler/>, dostęp 2026-01-28.
- [18] Apache Software Foundation, *Apache Airflow Documentation*, <https://airflow.apache.org/docs/>, dostęp 2026-01-28.
- [19] Prefect Technologies, *Prefect Documentation*, <https://docs.prefect.io>, dostęp 2026-01-28.

Załączniki

- Kod źródłowy: repozytorium <https://github.com/kofifi/Flowforge> (monorepo: API, UI, testy).