



UNIWERSYTET BIELSKO-BIAŁSKI

PRACA INŻYNIERSKA

System low-code do budowy interfejsów API

Autor:

Konrad Firlej

Numer Karty Studenckiej: 60043

Promotor:

dr inż. Krzysztof Augustynek

13 czerwca 2025

Spis treści

1 Wprowadzenie	2
1.1 Wstęp	2
1.2 Cel pracy	2
1.3 Przegląd rozdziałów	2
1.4 Przegląd istniejących rozwiązań	3
2 Projekt aplikacji	5
2.1 Analiza dziedziny problemowej	5
2.2 Specyfikacja wymagań	7
2.3 Model przypadków użycia	9
2.4 Specyfikacja przypadków użycia	11
2.5 Architektura logiczna i fizyczna	15
2.6 Model informacyjny	16
2.7 Interfejs użytkownika	18
2.8 Projekt bazy danych	20
2.9 Ideowe przedstawienie interakcji	21
3 Implementacja aplikacji	25
3.1 Wykorzystana technologia	25
3.1.1 Warstwa frontendowa: HTML, CSS, JavaScript/TypeScript	25
3.1.2 Warstwa backendowa: ASP.NET Core	26
3.1.3 Magazyn danych: SQLite	27
3.1.4 Kontrola wersji i monorepo: Git	28
3.1.5 Środowisko programistyczne: Rider i WebStorm	29
3.1.6 Uzasadnienie wyboru stosu jako całości	29
3.2 Implementacja dostępu do danych	30
3.2.1 Zakres odpowiedzialności warstwy danych	30
3.2.2 Model relacyjny i mapowanie encji	31
3.2.3 Migracje i kontrola zmian schematu	34
3.3 Interfejs	35
3.4 Testy jednostkowe (NUnit)	41
4 Zakończenie	42
4.1 Perspektywy rozwojowe aplikacji	42
4.2 Podsumowanie	43

1 Wprowadzenie

1.1 Wstęp

Tematem pracy jest projekt i implementacja Flowforge, aplikacji low-code ułatwiającej tworzenie i udostępnianie przepływów danych jako API. W praktyce zespoły programistyczne często marnują czas na ręczne sklejanie powtarzalnych integracji, walczą z niespójną dokumentacją oraz brakiem podglądu wykonania. Flowforge łączy wizualny edytor bloków z backendem HTTP, dzięki czemu użytkownik układą logikę na canvasie, a serwer udostępnia gotowy endpoint do uruchamiania przepływu.

1.2 Cel pracy

Celem pracy jest zbudowanie lekkiej platformy, która pozwala wizualnie konstruować przepływy i udostępniać je jako wywoływalne API, równocześnie zapewniając planowanie uruchomień, ręczne wywołania, wgląd w historię i wyniki oraz spójny interfejs w jasnym i ciemnym motywie, tak aby zminimalizować konieczność pisania kodu backendowego przy zachowaniu przejrzystości i możliwości audytu.

1.3 Przegląd rozdziałów

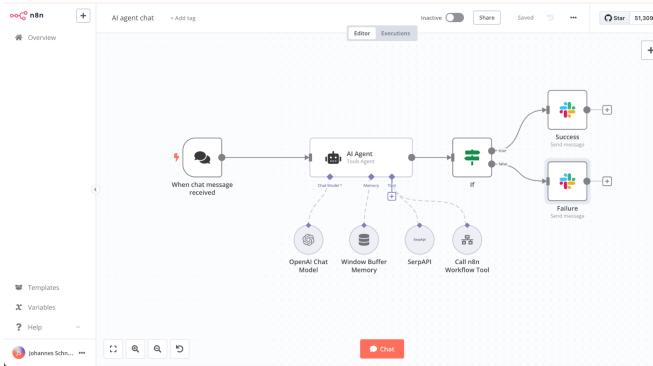
Praca została podzielona na cztery rozdziały ułożone zgodnie z naturalnym przebiegiem realizacji projektu. Rozdział pierwszy zawiera wprowadzenie: określa cel pracy, przedstawia kontekst problemu oraz zestawia aplikację z wybranymi narzędziami obecnymi na rynku automatyzacji. Rozdział drugi opisuje etap projektowy, obejmujący analizę dziedziny, wymagania, przypadki użycia, architekturę systemu, model danych i założenia interfejsu użytkownika.

Rozdział trzeci przedstawia implementację aplikacji. W tej części omówiono zastosowany stos technologiczny, organizację warstwy backendowej i frontendowej, implementację dostępu do danych, migracje schematu, interfejs użytkownika oraz testy jednostkowe. Rozdział czwarty stanowi domknięcie pracy i obejmuje dwie części: perspektywy rozwojowe aplikacji oraz podsumowanie uzyskanych rezultatów.

1.4 Przegląd istniejących rozwiązań

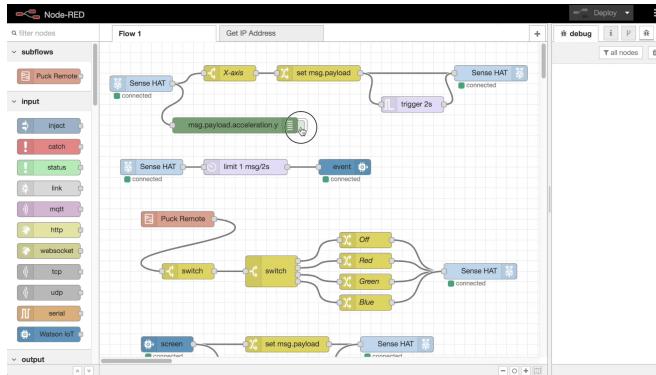
Aby osadzić Flowforge w kontekście rynku, warto zestawić je z najczęściej używanymi platformami automatyzacji. Poniższe punkty będą uzupełnione zrzutami ekranu (widok budowy przepływu lub listy automatyzacji) dla zobrazowania ergonomii interfejsu:

- **n8n[1]** to edytor blokowy z dużym katalogiem konektorów oraz opcją hostowania lokalnego. Przy bardziej złożonych scenariuszach użytkownik często dopisuje własne skrypty JavaScript, a wersjonowanie i testy regresywne trzeba budować samodzielnie.



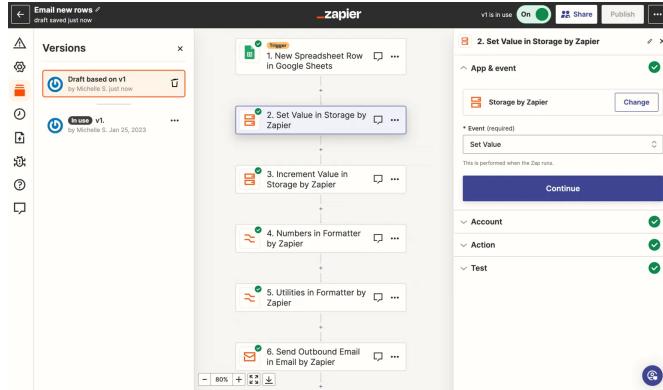
Rysunek 1: Przykładowy widok edytora przepływu w n8n

- **Node-RED[2]** to lekkie narzędzie często używane w IoT, łatwe w instalacji i rozbudowie o własne node'y. Brakuje w nim jednak wygodnej historii wykonania zapytań HTTP oraz kontroli wersji przepływów, co utrudnia audyt i współpracę zespołową.



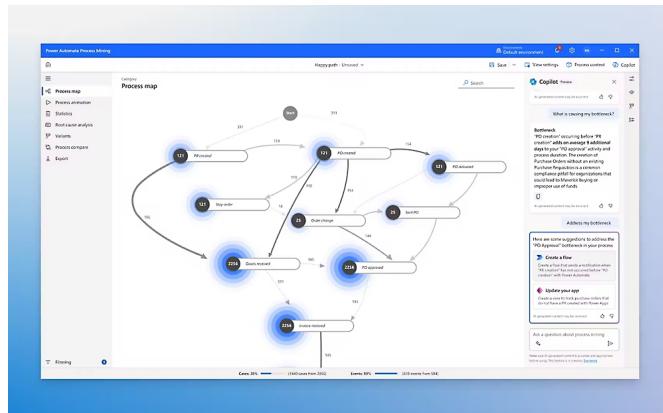
Rysunek 2: Przykładowy widok przepływu w Node-RED

- **Zapier**[3] oferuje bardzo szeroki ekosystem integracji SaaS i szybki start w chmurze, ale dane pozostają poza kontrolą użytkownika, logi są ograniczone czasowo, a brak wariantu do samodzielnego hostowania podnosi koszty dla firm z restrykcyjnymi wymaganiami.



Rysunek 3: Widok listy automatyzacji w Zapier

- **Power Automate**[4] jest mocno zintegrowane z pakietem Microsoft 365 i usługami Azure. Wymaga subskrypcji, a eksport i przenoszenie przepływów poza tenant bywa ograniczone, co utrudnia migracje i testy w izolowanych środowiskach.



Rysunek 4: Widok przepływu w Power Automate

Flowforge ma powstać jako lżejsza alternatywa. System instaluje się lokalnie lub w prywatnej chmurze, a opublikowana wersja workflow automatycznie wystawia endpoint HTTP do wywołania (np. `/api/run/{workflowId}`) z walidacją wejścia. Wbudowana obserwonalność obejmuje historię wykonień, scheduler oraz podgląd ścieżki bloków. Całość działa w jasnym i ciemnym motywie i nie wymaga subskrypcji zewnętrznej chmury.

2 Projekt aplikacji

Ten rozdział opisuje koncepcję budowanego systemu z uwzględnieniem dobrych praktyk inżynierii oprogramowania. Układ podsekcji odzwierciedla naturalny porządek pracy: od zrozumienia domeny, przez wymagania i przypadki użycia, po architekturę, model danych oraz interfejs.

2.1 Analiza dziedziny problemowej

Punktem wyjścia jest zrozumienie, jakie problemy chcemy rozwiązać w obszarze automatyzacji przepływów danych. Obecnie integracje API powstają często jako ad-hoc „spoiwo” pisane w skryptach, co utrudnia ponowne użycie, audyt i utrzymanie. Zespoły potrzebują narzędzi, które pozwoli im szybko składać sekwencje wywołań HTTP, wali-dować i transformować dane, planować uruchomienia oraz śledzić historię wykonaną bez zagłębiania się w infrastrukturę.

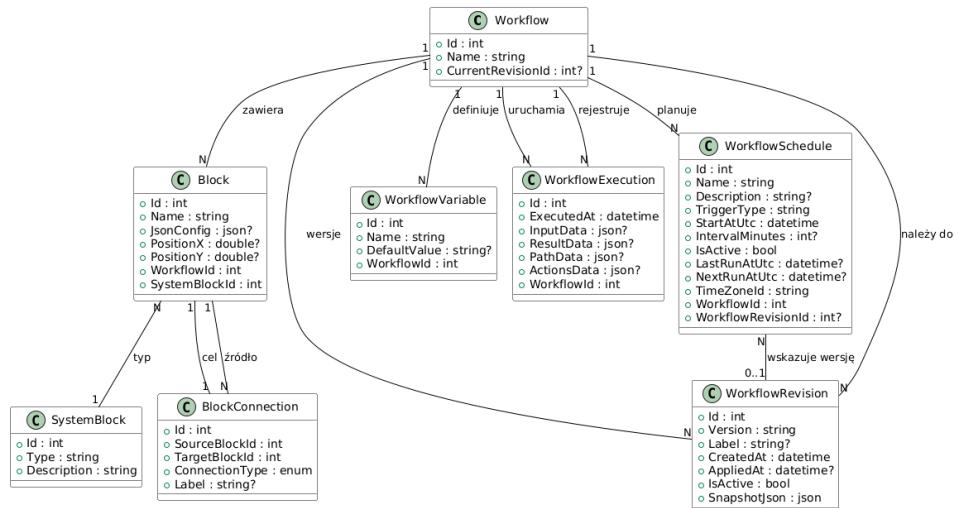
W kontekście Flowforge istotnymi bytami domenowymi są:

- **Workflow.** Główny artefakt użytkownika traktowany jako projekt. Budowany w edytorze blokowym opisuje cały przepływ danych i jest udostępniany jako wywoływalne API. Wspiera wersjonowanie i statusy publikacji.
- **Block.** Atomowa czynność w przepływie. Blok posiada konfigurację oraz porty połączeń (wejścia i wyjścia), które pozwalają definiować kolejność i gałęzie wykonania.
- **Connection.** Połączenie między blokami, które wyznacza kolejność przejścia oraz przebieg gałęzi warunkowych w przepływie.
- **Variable.** Nazwana wartość wejściowa lub wyjściowa workflow. Służy do przenoszenia danych między blokami oraz do parametrów wywołania API.
- **Execution.** Pojedyncze uruchomienie workflow, przechowuje ścieżkę wykonaną przez bloki, payload wejściowy i wyjściowy oraz logi.
- **Version/Snapshot.** Zapis stanu workflow w danym momencie, który umożliwia powrót do stabilnej konfiguracji. Wersje mogą być eksportowane i importowane między instancjami systemu.
- **Schedule.** Definicja planowanego uruchamiania workflow (jednorazowego lub cyklicznego) z parametrami daty i godziny, strefy czasowej pobranej z przeglądarki lub wybranej ręcznie oraz flagą wyłączenia.
- **Block Library.** Katalog systemowych typów bloków dostępny bezpośrednio w aplikacji, z podaniem wyglądu i tym samym panelem konfiguracyjnym co w edytorze. Każdy wpis przechowuje schemat pól, podstawową walidację oraz pokazuje docelowy kształt bloczka (styl, porty, ikona ustawień).

Reguły wynikające z pracy użytkownika w edytorze są spójne z powyższymi bytami. Nazwy workflow, wersji, snapshotów i harmonogramów muszą być unikalne w obrębie projektu, by dało się je łatwo znaleźć w listach i w historii wykonan. Edycja zawsze

odbywa się na wersji roboczej, a publikacja zamraża konfigurację do uruchomień. Execution odwołuje się wyłącznie do opublikowanej wersji i zapisuje ścieżkę bloków oraz logi. Każde połączenie łączy konkretne porty: Start nie ma wejścia, End nie ma wyjścia, a bloki warunkowe czy Switch posiadają wiele wyjść. Pola konfiguracyjne bloków są walidowane (adresy URL, liczby, flagi) przed zapisaniem, a zmienne muszą mieć niepuste nazwy wspólne dla całego workflow, by można je było wstrzykiwać i odczytywać w kolejnych blokach. Snapshot jest niezmienny. Każda zmiana tworzy nową wersję. Harmonogram wymaga daty, godziny i strefy czasowej (domyślnie pobranej z przeglądarki, z możliwością ręcznego wyboru) i nie pozwala ustawić startu w przeszłości względem wybranej strefy. Import/eksport wersji przenosi spójne identyfikatory bloków i zmiennych, aby po wgraniu odtworzyć ten sam przepływ. Biblioteka bloków stanowi źródło prawdy: tylko bloki z katalogu mogą zostać użyte, a każdy ma zdefiniowane porty i schemat pól konfiguracyjnych, które są prezentowane także na karcie „Blocks”.

Tak zdefiniowana dziedzina wyznacza zakres funkcjonalny projektu: wizualny edytor blokowy, zarządzanie wersjami, uruchomienia ręczne i zaplanowane, podgląd historii i wyników, a także kontrolę nad danymi bez uzależnienia od chmury zewnętrznej.



Rysunek 5: Diagram domenowy systemu Flowforge

2.2 Specyfikacja wymagań

Na etapie analizy wymagań oraz ich specyfikacji należy precyzyjnie opisać, czego będzie oczekiwał użytkownik i jakie ramy techniczne ma spełnić system. Poniższe punkty odzwierciedlają planowane możliwości edytora blokowego oraz widoków Workflows, Executions, Scheduler i Blocks, a także zasady wersjonowania i planowania uruchomień, które mają zostać zaimplementowane.

Wymagania funkcjonalne

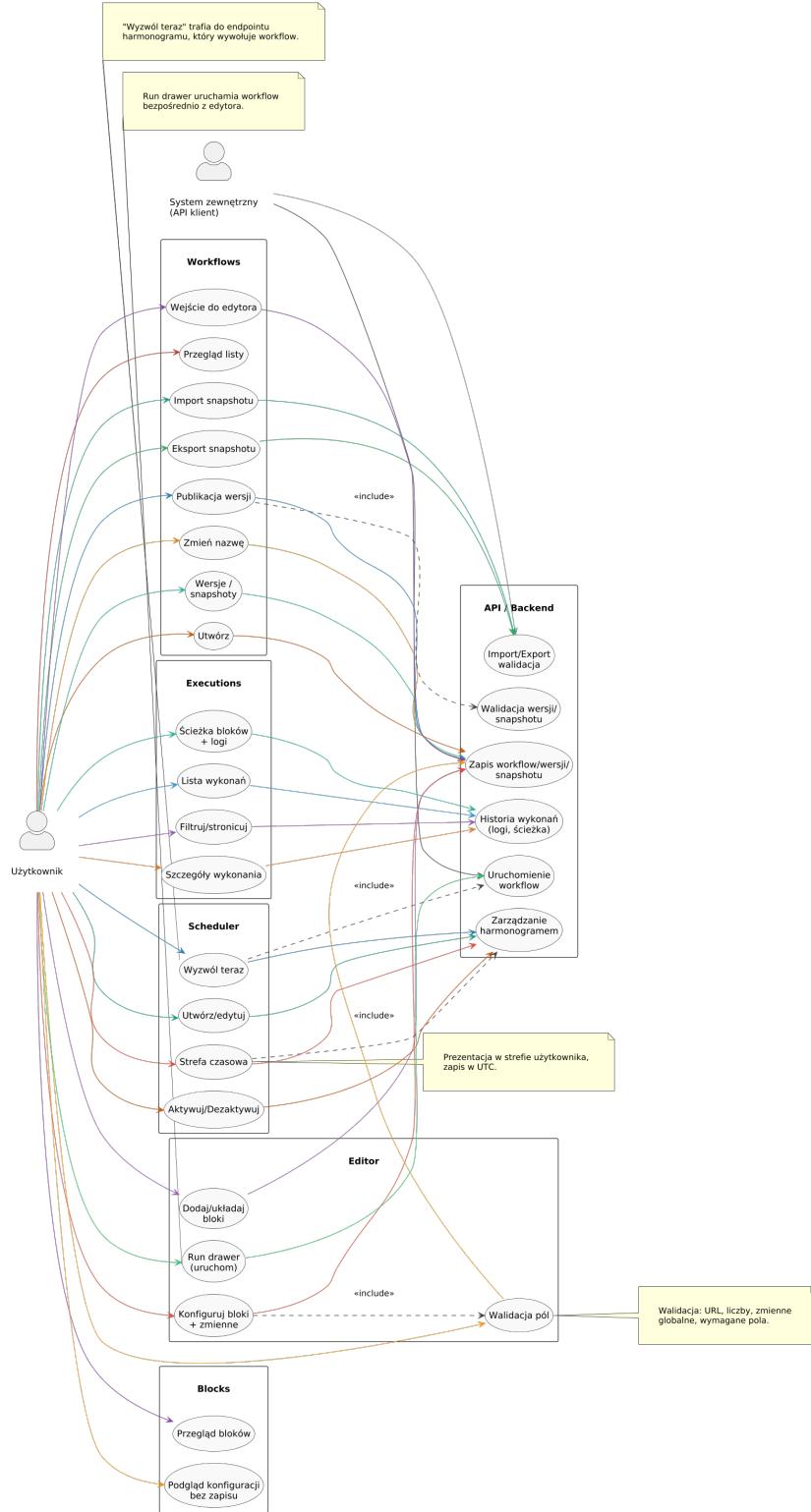
- Użytkownik może zbudować workflow w edytorze blokowym, dodając z palety bloki Start, End, If, Switch, Calculation, Text Transform, Text Replace, HTTP Request, Parser i Wait oraz łącząc ich porty wejść i wyjść.
- Użytkownik może zapisać wersję roboczą, opublikować wersję wykonywalną i utworzyć snapshot do późniejszego przywrócenia lub eksportu; nazwa workflow nie musi być unikalna, rozróżnienie następuje po identyfikatorach wersji i snapshotów.
- Użytkownik może konfigurować każdy blok, a interfejs oraz API sprawdzają poprawność pól. W polach można wprowadzać zmienne globalne zapisane w workflow przez \$nazwa; błędna lub brakująca zmienna jest sygnalizowana.
- Użytkownik może na liście Workflows otworzyć edytor, przejrzeć snapshoty, uruchomić workflow ręcznie z danymi wejściowymi i zobaczyć opublikowane wersje.
- Użytkownik może w widoku Executions sprawdzić historię uruchomień, przefiltrować listę, otworzyć szczegóły pojedynczego przebiegu, obejrzeć metadane, dane wejściowe i wyjściowe, ścieżkę bloków oraz logi.
- Użytkownik może z run drawera uruchomić workflow na żądanie, wprowadzić wartości zmiennych, a po zakończeniu obejrzeć wyniki bez opuszczania panelu.
- Użytkownik może utworzyć harmonogram, edytować go, włączyć lub wyłączyć oraz uruchomić natychmiast. Strefa czasowa jest pobierana z przeglądarki lub wybierana ręcznie, a termin startu nie może leżeć w przeszłości względem tej strefy.
- Użytkownik może importować i eksportować snapshoty, zachowując identyfikatory bloków, połączeń i zmiennych tak, aby po wgraniu odtworzyć ten sam przepływ.
- Użytkownik może w widoku Blocks zobaczyć wszystkie typy bloków, ich wygląd i panel konfiguracji identyczny jak w edytorze, bez zapisywania zmian.
- Użytkownik może przełączyć jasny i ciemny motyw oraz pracować na tych samych widokach Workflows, Executions, Scheduler, Blocks i edytorze.

Wymagania niefunkcjonalne

- UI działa w przeglądarkach Chromium i Firefox w rozdzielczości co najmniej Full HD i zachowuje responsywny układ list oraz edytora.
- Walidacja pól odbywa się po stronie klienta i serwera; komunikaty są powiązane z konkretnym polem i nie blokują innych działań na stronie.
- Edytor z pustym workflow uruchamia się w kilka sekund w środowisku deweloperskim, a dodawanie i przesuwanie bloków jest płynne.
- Harmonogram zapisuje czas w UTC, prezentuje go w strefie użytkownika i obsługuje wszystkie strefy dostępne w przeglądarce.
- Dane o wersjach, snapshotach, harmonogramach i uruchomieniach są trwałe w bazie; eksport snapshotów nie wymaga usług zewnętrznych.
- Motyw jasny i ciemny zachowują spójną typografię i kolorystykę między widokami.
- Środowisko buildów jest odtwarzalne: frontend uruchamiany poleceniami npm w katalogu ‘flowforge.ui’, backend przez ‘dotnet’ na rozwiązaniu.
- Testy jednostkowe backendu (NUnit) [15] można uruchamiać niezależnie od pozostałych komponentów.

Warstwa niefunkcjonalna opisuje tempo i jakość pracy interfejsu, pełną walidację po kliencie i serwerze, prawidłową obsługę stref czasowych, trwałość danych oraz powtarzalne procesy buildów i testów.

2.3 Model przypadków użycia



Rysunek 6: Przypadki użycia Flowforge. Zgrupowane według widoków UI i powiązań z backendem

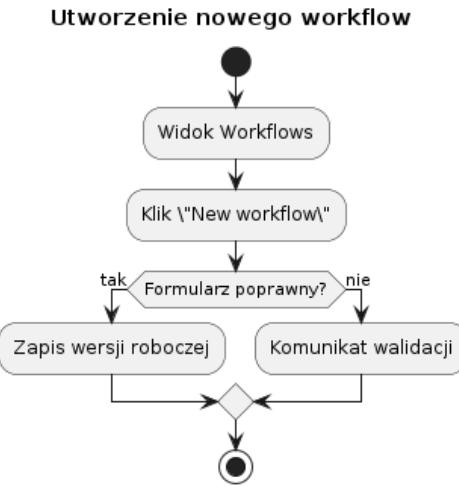
Diagram prowadzi użytkownika przez kolejne grupy akcji:

- **Workflows**: utworzenie nowego projektu, zmiana nazwy, podgląd wersji i snapshotów, import/eksport snapshotu JSON, publikacja wersji oraz wejście do edytora.
- **Editor**: dodawanie i układanie bloków, konfiguracja parametrów i zmiennych, validacja pól (URL, liczby, nazwy zmiennych) oraz uruchomienie z run drawera bez opuszczania edytora.
- **Executions**: przegląd historii, filtrowanie/stronicowanie, szczegóły pojedynczego przebiegu wraz z danymi wejścia/wyjścia, statusem, logami i ścieżką bloków.
- **Scheduler**: tworzenie i edycja harmonogramu, wybór strefy czasowej (prezentacja w strefie użytkownika, zapis w UTC), aktywacja/dezaktywacja oraz „Wyzwól teraz”, które trafia do endpointu harmonogramu i uruchamia workflow.
- **Blocks**: podgląd biblioteki bloków i ich konfiguracji bez zapisywania, aby poznać parametry przed dodaniem do edytora.
- **API/Backend**: każda akcja UI wywołuje serwer (zapis workflow, walidacja snapshotu, uruchomienie workflow, historia wykonień, zarządzanie harmonogramem, walidacja importu/eksportu), a aktor „system zewnętrzny” symbolizuje klienta API korzystającego z uruchomień oraz importu/eksportu poza UI.

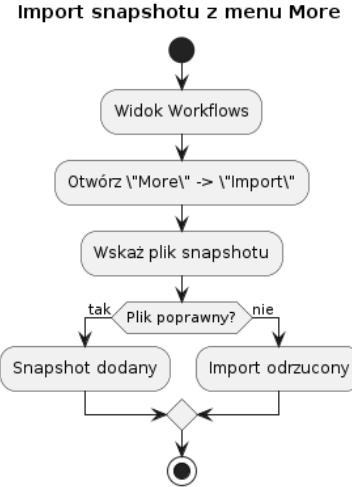
Kolory strzałek rozróżniają ścieżki użytkownika, a relacje include oznaczają czynności wspólne. Jeden diagram pokazuje pełny przepływ od listy workflow przez edytor, wykonania i scheduler aż po warstwę backendu.

2.4 Specyfikacja przypadków użycia

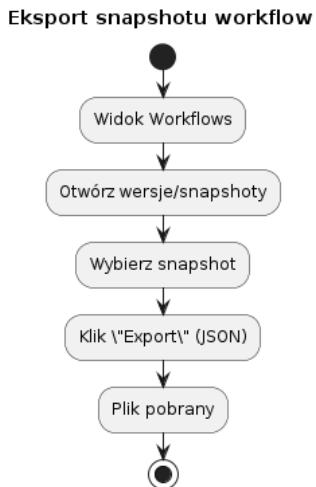
Poniższe diagramy uszczegółowiąją pojedyncze akcje użytkownika w interfejsie Flowforge. Każdy rysunek pokazuje kroki prowadzące do sukcesu oraz główną ścieżkę alternatywną (anulowanie lub błąd validacji).



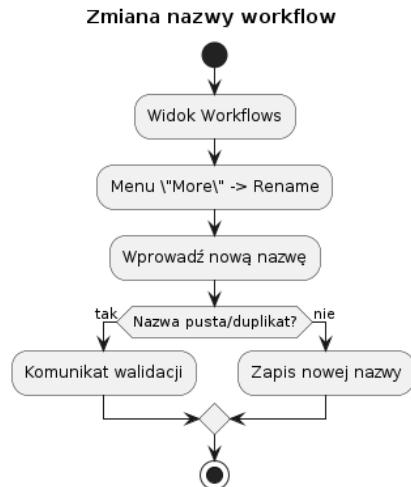
Rysunek 7: Utworzenie nowego workflow z listy Workflows. Użytkownik wypełnia formularz i zapisuje szkic przepływu



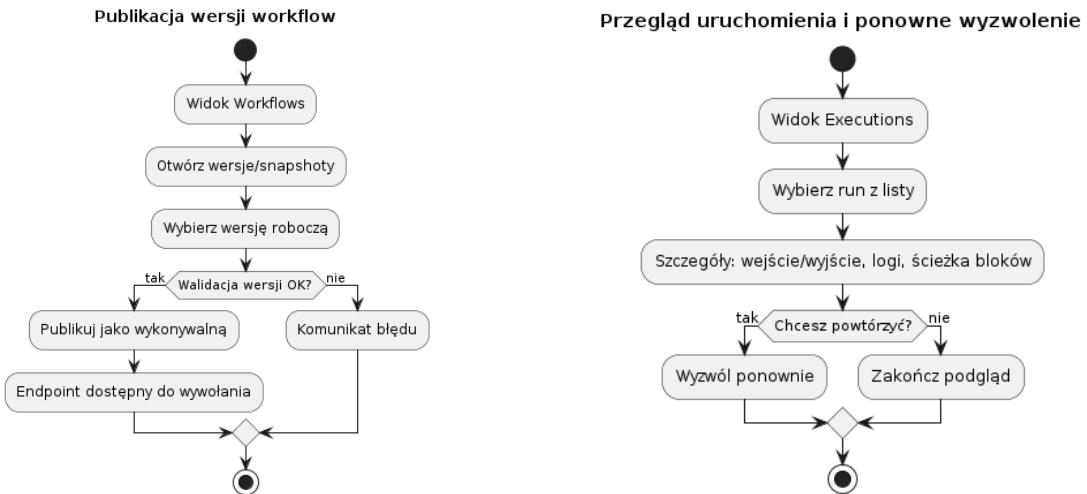
Rysunek 8: Import snapshotu z menu More. Wskazanie pliku JSON i dodanie wersji po pozytywnej walidacji



Rysunek 9: Eksport snapshotu do pliku JSON. Zapis bieżącej konfiguracji przepływu do pobrania

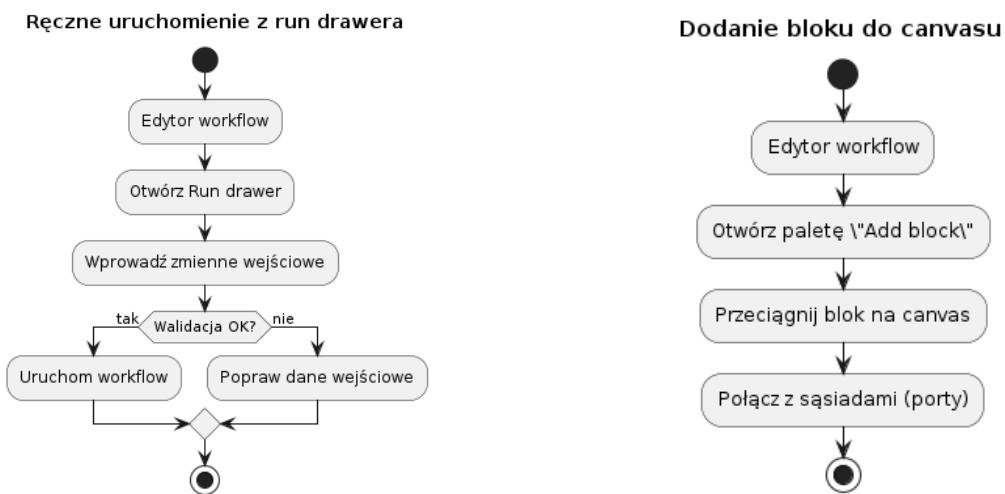


Rysunek 10: Zmiana nazwy workflow w menu More. Wprowadzenie nowej etykiety i zapis na liście



Rysunek 11: Publikacja wersji workflow. Wersja robocza staje się wykonywalna, endpoint HTTP dostępny

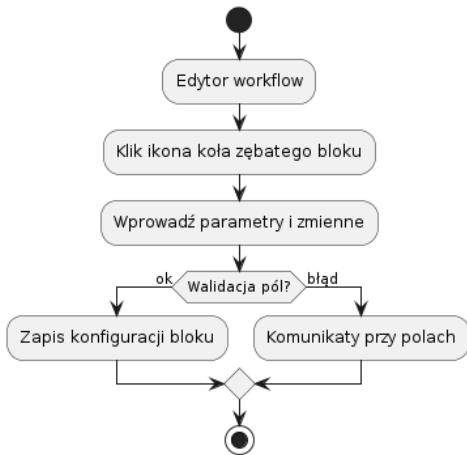
Rysunek 12: Przegląd szczegółów uruchomienia. Wejście, wyjście, logi, ścieżka bloków oraz opcja ponownego uruchomienia



Rysunek 13: Ręczne uruchomienie workflow z run drawera. Wprowadzenie zmiennych i start bez wychodzenia z edytora

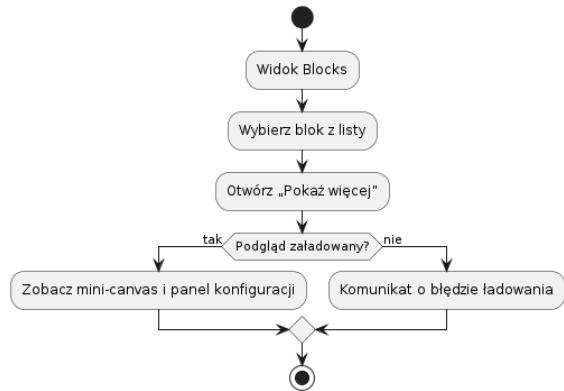
Rysunek 14: Dodanie nowego bloku na canvas. Przeciagnięcie z palety i umieszczenie między portami

Konfiguracja bloku w edytorze



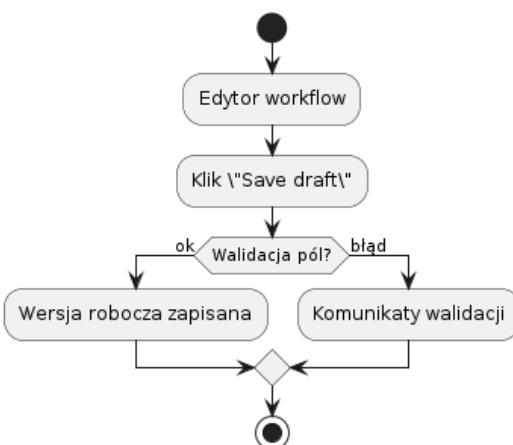
Rysunek 15: Konfiguracja parametrów bloku. Uzupełnienie pól, walidacja i zapis ustawień

Podgląd bloku w sekcji Blocks



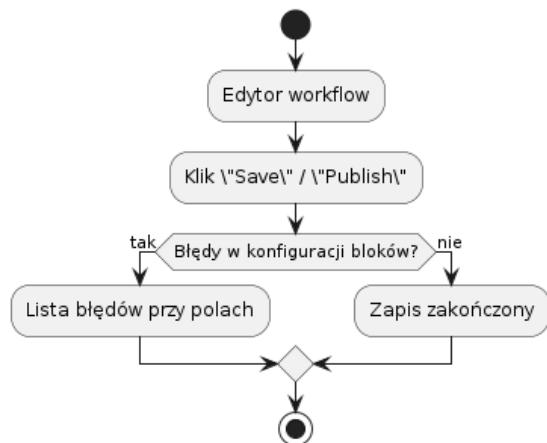
Rysunek 16: Podgląd bloku w sekcji Blocks. Mini-canvas i panel konfiguracji bez zapisywania zmian

Zapis wersji roboczej w edytorze



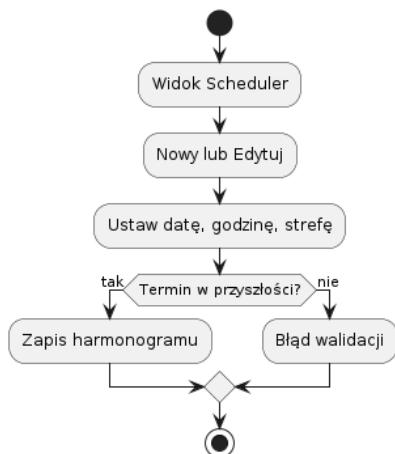
Rysunek 17: Zapis wersji roboczej w edytorze. Aktualizacja szkicu workflow w bazie

Walidacja bloków przy zapisie



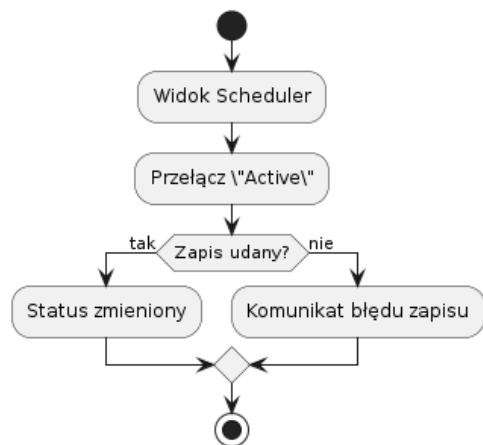
Rysunek 18: Walidacja konfiguracji bloków przy zapisie/publikacji. Błędne pola blokują zapis do wersji

Utworzenie / edycja harmonogramu



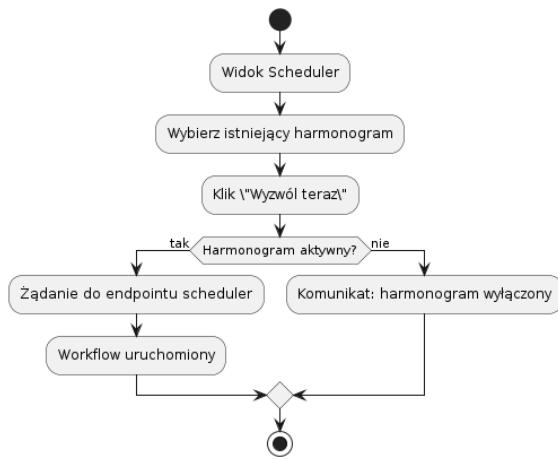
Rysunek 19: Utworzenie lub edycja harmonogramu. Ustawienie daty, godziny i strefy czasowej przed zapisaniem

Aktywacja/dezaktywacja harmonogramu



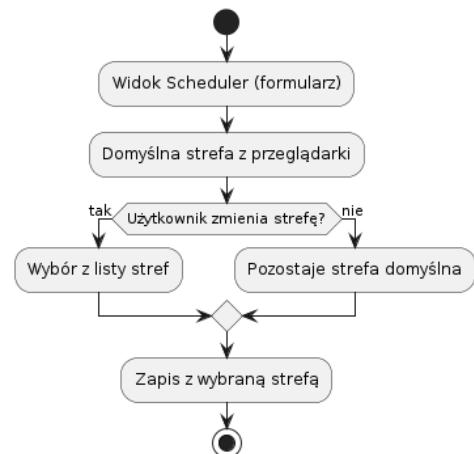
Rysunek 20: Aktywacja lub dezaktywacja harmonogramu. Przełącznik statusu z natychmiastowym zapisem

Wyzwolenie harmonogramu „Wyzwól teraz”



Rysunek 21: Wyzwolenie istniejącego harmonogramu opcją „Wyzwól teraz”. Jednorazowe uruchomienie zdefiniowanego przepływu

Ustawienie strefy czasowej w Scheduler



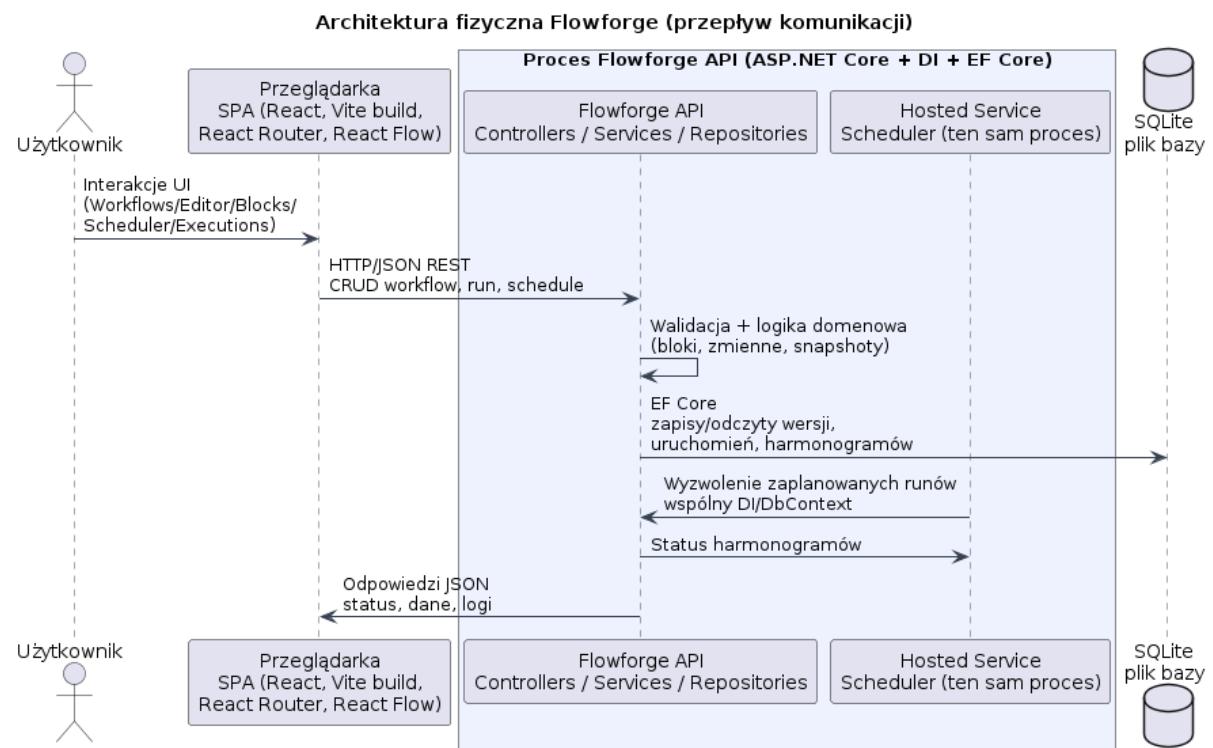
Rysunek 22: Wybór strefy czasowej w formularzu harmonogramu. Automatyczna strefa z przeglądarki lub ręczny wybór

2.5 Architektura logiczna i fizyczna

W tej części opisujemy docelowy podział warstw, który ma zostać wdrożony. Po stronie klienta przewidujemy jedną aplikację SPA [5] w React 19 (budowaną Vite, routowaną React Router, z edytorem opartym o React Flow)[8], komunikującą się wyłącznie po HTTP/JSON z REST API [11].

Warstwa serwerowa pozostanie w ASP.NET Core[11]. Cienkie kontrolery REST przekazują żądania do serwisów domenowych, serwisy korzystają z repozytoriów, a EF Core[12] zapisuje dane w pliku SQLite[13]. W tym samym procesie działa hosted service scheduler, który cyklicznie wyzwala zaplanowane uruchomienia. Serwer HTTP to Kestrel [6], a wstrzykiwanie zależności DI [7] oraz DbContext EF Core utrzymują spójność konfiguracji danych.

W wymiarze fizycznym przewidujemy trzy artefakty: przeglądarka renderująca SPA [5], proces API Kestrel [6] z DI/DbContextem [7] i schedulerem oraz plik bazy na dysku. Rysunek 23 pokazuje, że SPA może być serwowane wspólnie z API albo z osobnego hostingu statycznego; decyzja wdrożenia nie zmienia kontraktu REST.



Rysunek 23: Architektura fizyczna Flowforge. Przepływ komunikacji między SPA, API, schedulerem i bazą SQLite

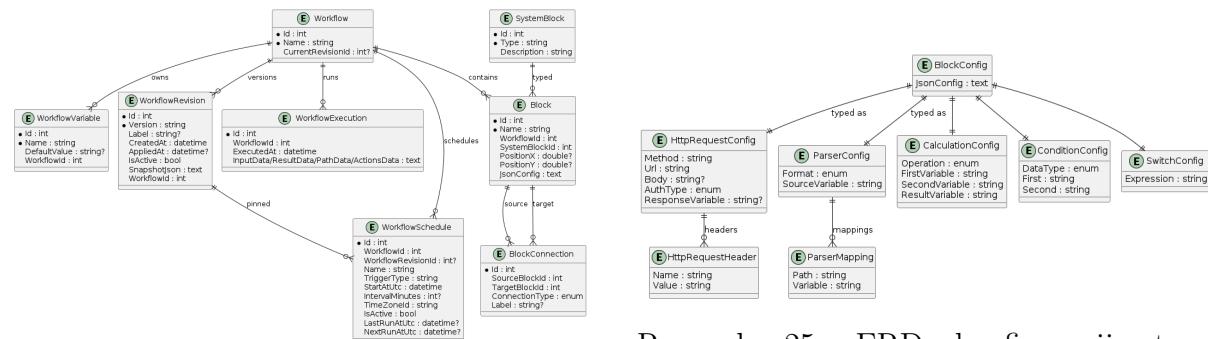
Rysunek 23 obrazuje planowany przepływ: SPA [5] odpytuje API, kontrolery kierują ruch do serwisów, te do repozytoriów i EF Core [12], a zapis trafia do SQLite [13]. Scheduler działa równolegle w tym samym procesie, wywołując serwisy w tle.

Model backendu utrzymujemy jako warstwową kompozycję „Controllers.Services.Repositories” z DI [7]; frontend pozostaje przy klientowskim renderowaniu SPA [5]. Taki podział ma ułatwić testy, wymianę warstw danych i ewentualne rozdzielenie hostingu SPA i API bez zmiany kontraktu.

Podsumowanie decyzji architektonicznej: utrzymujemy SPA [5] w React [8] z dostawą statyczną, backend w ASP.NET Core [11] z REST API, Kestrel [6] i EF Core [12] na SQLite [13], a scheduler jako hosted service w tym samym procesie. Ten układ minimalizuje zależności, pozwala rozwijać UI i API niezależnie, a jednocześnie zachowuje prostą ścieżkę wdrożenia na jednej maszynie lub w rozdzielonych hostach bez modyfikacji protokołów.

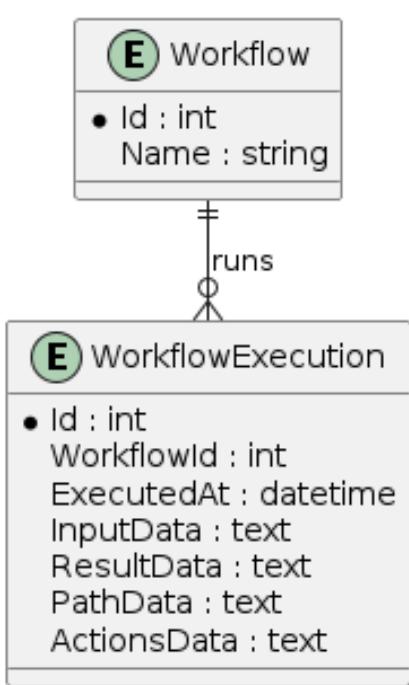
2.6 Model informacyjny

Celem tej części jest przedstawienie pełnego modelu klas w notacji crow's foot na podstawie kodu z `flowforge.api/Models`. Diagramy pokazują kardynalności i główne pola: rdzeń domeny (Rys. 24), konfiguracje bloków (Rys. 25), dane pojedynczego uruchomienia (Rys. 26), harmonogram (Rys. 27) oraz pomocnicze enumy (Rys. 28).

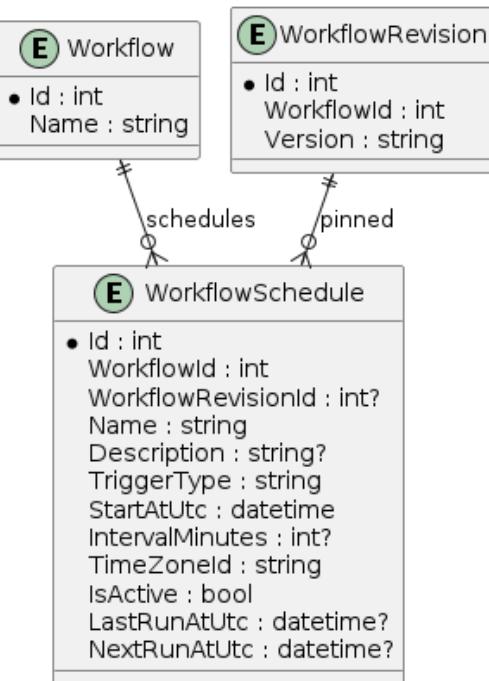


Rysunek 24: ERD rdzenia: Workflow z blokami, połączeniami, zmiennymi, rewiżjami, uruchomieniami i harmonogramami

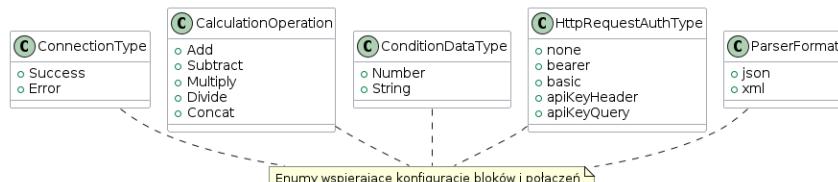
Rysunek 25: ERD konfiguracji: typy JSON bloków (HTTP, parser, kalkulacja, if/switch) i relacje pomocniczych tabel



Rysunek 26: ERD wykonania: WorkflowExecution z referencją do Workflow oraz polami serializującymi wejście, wynik i ścieżki



Rysunek 27: ERD harmonogramu: WorkflowSchedule przypięty do Workflow i opcjonalnej WorkflowRevision, z polami czasu i strefy



Rysunek 28: Enumy wspierające konfiguracje: ConnectionType, CalculationOperation, ConditionDataType, HttpRequestAuthType, ParserFormat

2.7 Interfejs użytkownika

Sekcja 2.7 opisuje wymagane zachowanie interfejsu użytkownika na poziomie funkcjonalnym, bez odwołania do makiet graficznych. Aplikację rozwijano podejściem *Vertical Slice* [22], jednak od początku priorytet nadano logice backendowej: najpierw implementowano model danych, reguły biznesowe i kontrakty API, a dopiero następnie rozwijano warstwę interfejsu. Taka kolejność pozwalała najpierw ustabilizować zachowanie systemu i zweryfikować poprawność przypadków użycia na poziomie usług, co ograniczało ryzyko wielokrotnego przebudowywania UI po zmianach domenowych. Wstępna warstwa UI miała charakter minimalny i służyła głównie do weryfikacji działania przypadków użycia.

Takie podejście zmniejsza ryzyko niespójności między tym, co użytkownik widzi w aplikacji, a tym, co system rzeczywiście wykonuje. Projekt interfejsu nie jest wtedy zbiorem niezależnych ekranów, tylko warstwą prezentacji dla wcześniej zweryfikowanych procesów domenowych. Każda akcja dostępna w UI ma odpowiadać konkretному przypadkowi użycia i jednoznacznej operacji po stronie API, a rezultat działania użytkownika powinien być możliwy do potwierdzenia przez stan bazy. Dzięki temu interfejs staje się narzędziem sterowania procesem, a nie elementem dekoracyjnym o nieustalonym znaczeniu operacyjnym.

Centralnym punktem pracy użytkownika ma być widok **Workflows**. Jego zadaniem jest obsługa cyklu życia przepływu: utworzenie, wybór, publikacja oraz zarządzanie wersjami. Zakłada się, że użytkownik otrzymuje listę przepływów wraz z podstawowymi metadanymi i zestawem działań kontekstowych. Z punktu widzenia użyteczności ważne jest, aby operacje administracyjne były dostępne bez opuszczania listy, natomiast operacje merytoryczne kierowały bezpośrednio do edytora. Widok ten pełni więc funkcję bramy wejściowej do dalszej pracy i porządkuje organizację projektów.

Drugim obszarem jest katalog **Blocks** oraz edytor workflow. Katalog powinien udostępniać zbiór bloków systemowych z krótką charakterystyką funkcji, tak aby użytkownik mógł dobrać element adekwatny do problemu. Edytor ma umożliwiał konstruowanie grafu procesu przez dodawanie bloków, definiowanie połączeń oraz konfigurację parametrów. Istotnym wymaganiem jest zachowanie spójności między reprezentacją graficzną a modelem danych wysyłanym do API. Oznacza to konieczność walidacji podstawowych warunków poprawności jeszcze po stronie klienta, przy jednoczesnym traktowaniu backendu jako warstwy ostatecznej weryfikacji.

Kolejna grupa funkcji dotyczy obserwacji działania systemu. Widok **Executions** ma przedstawiać historię uruchomień w formie umożliwiającej szybkie filtrowanie i identyfikację interesujących przebiegów. Jego rolą nie jest analiza szczegółowa, lecz orientacja operacyjna: kiedy uruchomienie nastąpiło, jaki miało status i którego workflow dotyczyło. Analiza szczegółowa jest realizowana w widoku **Execution Details**, gdzie użytkownik

powinien mieć dostęp do danych wejściowych, wyniku, sekwencji przejścia przez bloki oraz logów. Rozdzielenie tych dwóch poziomów przeglądu zmniejsza obciążenie poznawcze i ułatwia diagnostykę.

Istotnym elementem koncepcji interfejsu jest także obsługa harmonogramów w widoku **Scheduler**. Wymaga się, aby użytkownik mógł zdefiniować sposób wyzwalania przepływu, zarządzać aktywnością harmonogramu oraz kontrolować parametry czasu, w tym strefę czasową. Ponieważ wykonanie planowe realizuje usługa działająca w tle, interfejs powinien jednoznacznie komunikować, które ustawienia wpływają na kolejne uruchomienia, a które mają charakter informacyjny. Z perspektywy projektowej jest to krytyczne dla ograniczenia błędów operacyjnych wynikających z niejednoznacznej interpretacji czasu.

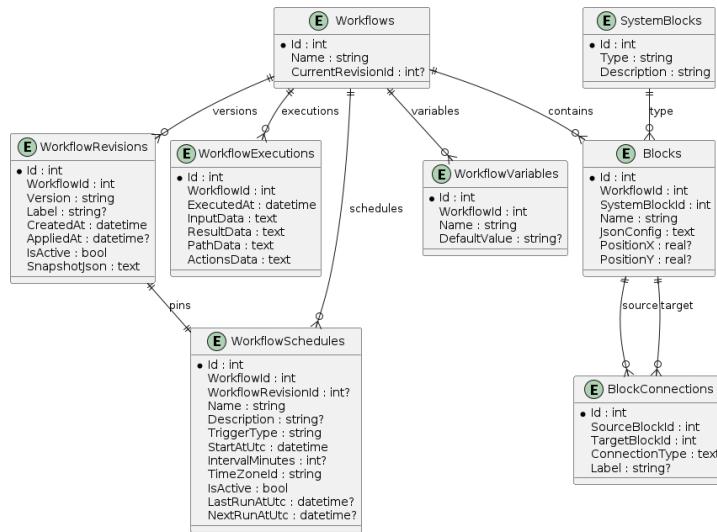
Uzupełnieniem powyższych widoków są formularze kontekstowe, między innymi konfiguracja bloku **Parser**. Formularz ma być osadzony w panelu bocznym, aby użytkownik mógł jednocześnie modyfikować parametry i obserwować strukturę całego przepływu. Wymagane jest zachowanie przewidywalnego mapowania pól formularza na strukturę danych przesyłaną do backendu. Tylko wtedy możliwe jest ograniczenie błędów integracyjnych oraz utrzymanie powtarzalności zachowania bloku przy kolejnych uruchomieniach.

Podsumowując, sekcja 2.7 ma charakter specyfikacji funkcjonalnej interfejsu, a nie prezentacji warstwy wizualnej. Brak makiet nie wynika z pominięcia tego obszaru, lecz z przyjętej kolejności prac: najpierw stabilizacja warstwy backendowej i kontraktów, następnie implementacja frontendu jako reprezentacji istniejących mechanizmów systemu. Takie podejście jest zgodne z celem pracy, w której kluczowe było wykazanie poprawności i spójności działania całego rozwiązania, a nie projektowanie interfejsu jako odrębnego artefaktu niezależnego od logiki wykonawczej.

2.8 Projekt bazy danych

Schemat bazy ma układ relacyjny z encją centralną **Workflows**, do której przez klucze obce powiązano encje zależne: rewizje, bloki, uruchomienia, harmonogramy i zmienne. W literaturze hurtownianej spotyka się pokrewne wzorce organizacji danych, takie jak *hub-and-spoke* [18] oraz *star schema* [19], jednak w tym projekcie zastosowano wariant transakcyjny (OLTP) [20], dostosowany do operacyjnego charakteru systemu i częstych zapisów stanu. Taki układ stanowi bazę dla edytora oraz schedulera. W środowisku deweloperskim zastosowano relacyjną bazę SQLite [13] z mapowaniem przez Entity Framework Core [12]. Poniższe punkty podsumowują przyjęty schemat i decyzje projektowe:

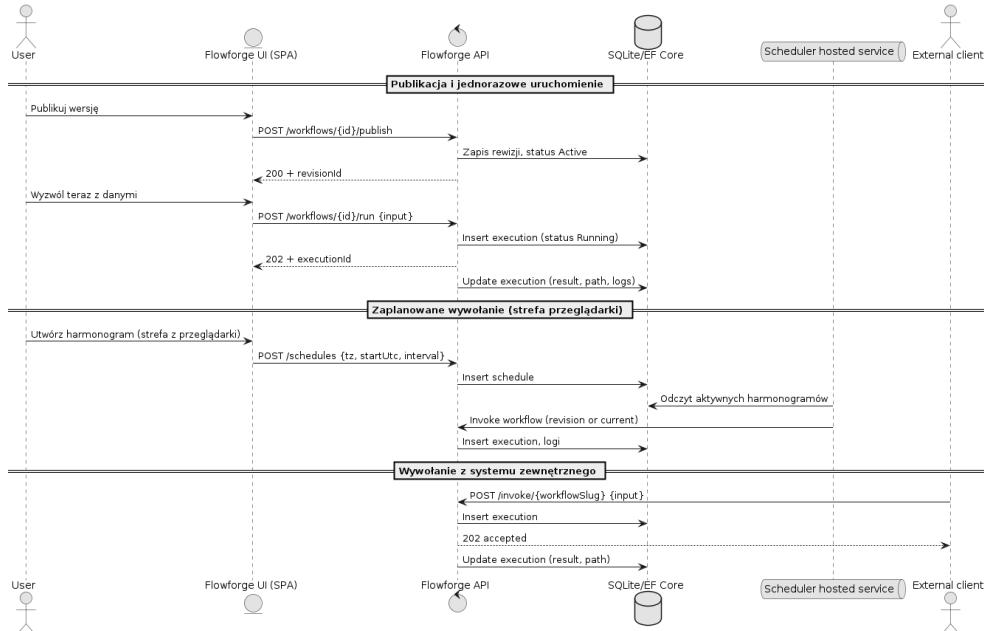
- **Workflows**: nazwa, bieżąca rewizja robocza (SET NULL po skasowaniu rewizji).
- **Revisions**: snapshot JSON, znacznik aktywności, kaskadowe powiązanie z workflow.
- **Executions**: wejście, wynik, ścieżka i logi w JSON; kaskadowo usuwane z workflow.
- **Blocks**: pozycja, konfiguracja JSON, FK do workflow (cascade) i SystemBlocks (restrict).
- **SystemBlocks**: 11 typów zasianych startowo (Start, End, HttpRequest, Parser, Calculation, If, Switch, Loop, Wait, TextTransform, TextReplace).
- **Connections**: źródło, cel, typ i etykieta; oba FK z ON DELETE RESTRICT dla uniknięcia cyklicznych kaskad.
- **Variables**: nazwa, domyślna wartość, FK do workflow (cascade).
- **Schedules**: interwał lub jednorazowy start, strefa czasowa, aktywność, opcjonalna przypięta rewizja (SET NULL po usunięciu), cascade do workflow.
- **Konfiguracje bloków**: trzymane jako JSON w `Blocks.JsonConfig` (HttpRequest, Parser, Calculation, Condition, Switch); dodawanie pól nie wymaga migracji SQL.
- **Enumy i indeksy**: enumy konwertowane do tekstu, migracje tworzą indeksy na wszystkich FK (m.in. `IX_Blocks_WorkflowId`, `IX_BlockConnections_SourceBlockId`, `IX_WorkflowExecutions_WorkflowId`).



Rysunek 29: ERD bazy danych: Workflows, Revisions, Executions, Schedules, Variables, Blocks, SystemBlocks i BlockConnections z kardynalnościami

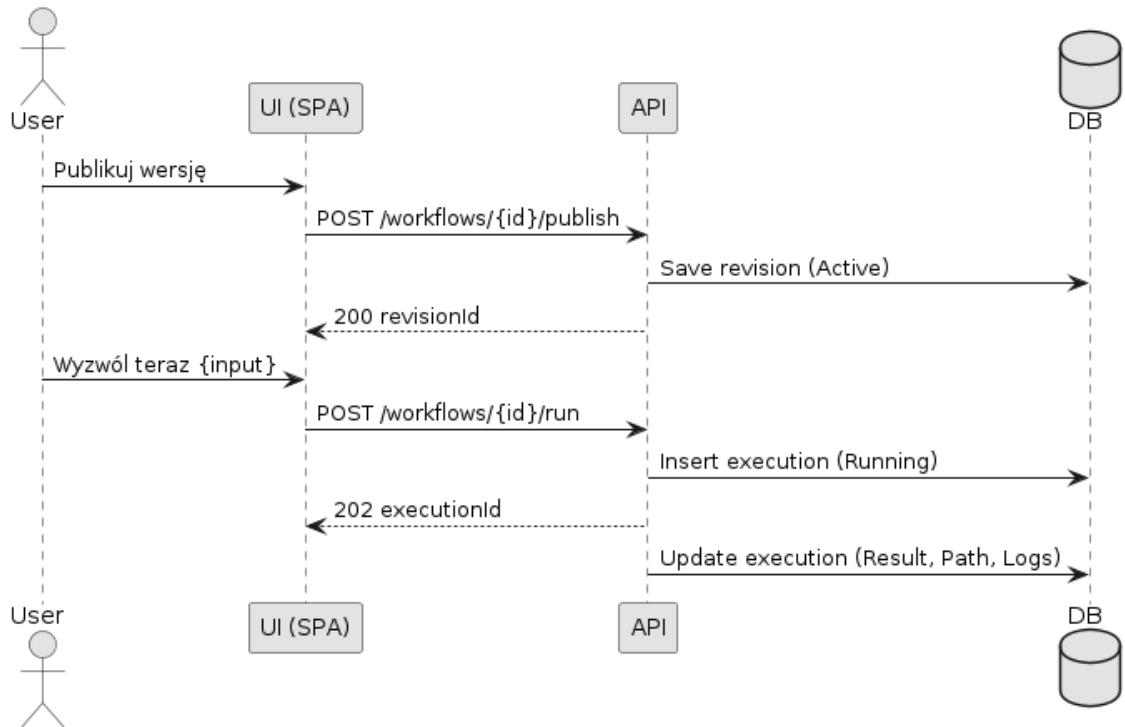
2.9 Ideowe przedstawienie interakcji

W tej sekcji zestawiono diagramy ilustrujące przepływy informacji między warstwami aplikacji. Najpierw pokazano mapę komponentów (SPA [5], API, baza, scheduler, klient zewnętrzny), następnie trzy sekwencje zdarzeń: publikację i jednorazowe uruchomienie, obsługę harmonogramu oraz wywołanie zewnętrzne. Każdy rysunek prezentuje pełną ścieżkę żądania HTTP, zapis stanu w bazie i punkt, w którym powstają logi oraz ścieżka bloków, co ma znaczenie dla obserwonalności i audytu.



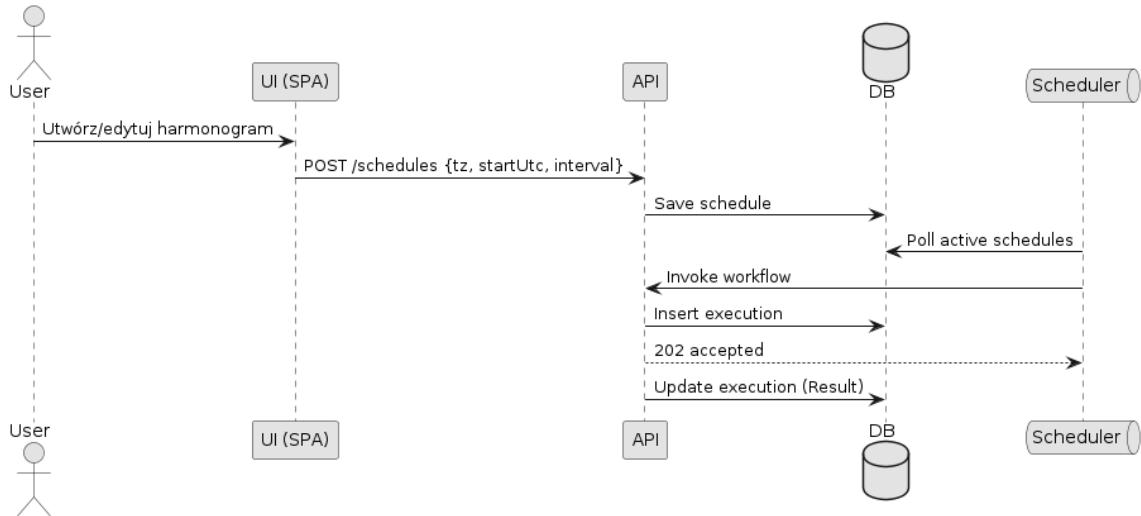
Rysunek 30: Ideowe interakcje komponentów: SPA, API, baza SQLite/EF Core, scheduler i klient zewnętrzny

Rysunek 30 pokazuje główne klocki: SPA [5] i klient zewnętrzny korzystają z jednego kontraktu REST, API pośredniczy we wszystkich operacjach, a scheduler w tle wyzwala workflow na podstawie wpisów zapisanych w bazie SQLite [13] i mapowanych przez EF Core [12]. Jeden punkt trwałości upraszcza audyt i obserwowalność, a brak bezpośrednich połączeń między klientami a schedulerem pozwala kontrolować uprawnienia wyłącznie na poziomie API.



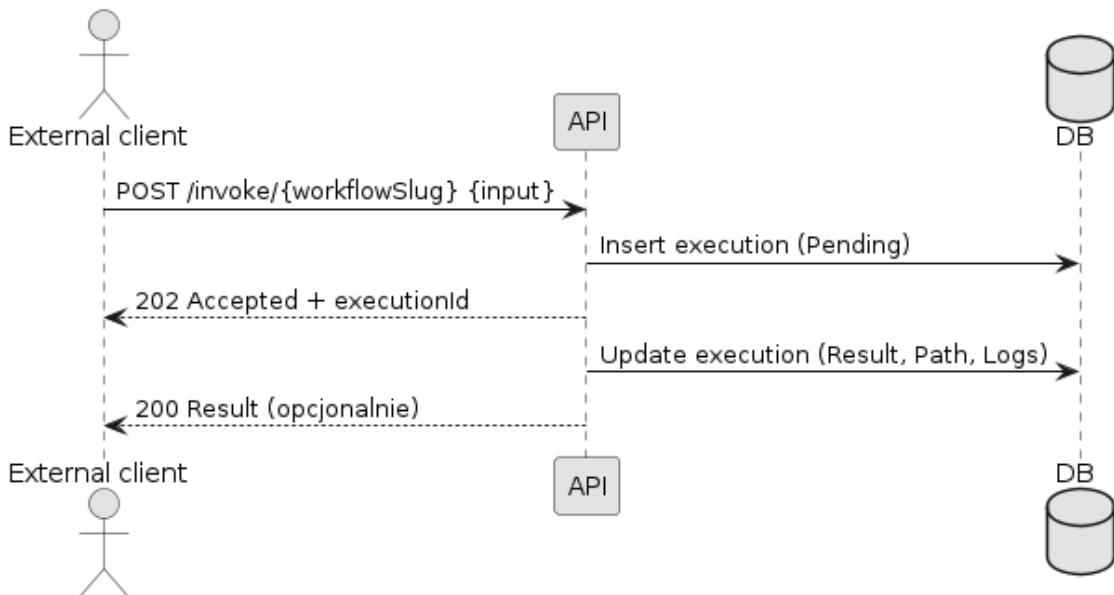
Rysunek 31: Publikacja wersji i jednorazowe wyzwolenie workflow z edytora

Rysunek 31 pokazuje prostą sekwencję: POST /publish tworzy aktywną rewizję, a kolejne POST /run zakłada rekord wykonania, zwraca 202 i po zakończeniu dopisuje wynik, ścieżkę bloków i logi. Edytor przekazuje jedynie dane wejściowe i odbiera identyfikator – cała logika wykonania pozostaje w API, co ułatwia testy i gwarantuje powtarzalność uruchomień.



Rysunek 32: Obsługa harmonogramu przez usługę tła (polling, wyzwolenie, zapis wykonania)

Rysunek 32 ilustruje planowane uruchomienie: UI zapisuje interwał, start i strefę czasową; scheduler cyklicznie pobiera aktywne wpisy, wywołuje workflow przez API, a wynik, ścieżka i logi trafiają do bazy. Parametry czasu przechowywane są w UTC, więc obsługa stref i zmiany czasu nie wymaga dodatkowej logiki.



Rysunek 33: Wywołanie endpointu workflow przez system zewnętrzny

Rysunek 33 pokazuje integrację bez UI: klient zewnętrzny wywołuje endpoint, API tworzy rekord wykonania i zwraca 202; po przetworzeniu dopisuje wynik, ścieżkę i logi widoczne potem w Executions. Ten sam kontrakt HTTP obowiązuje UI i integratora, co ułatwia walidację i diagnozę.

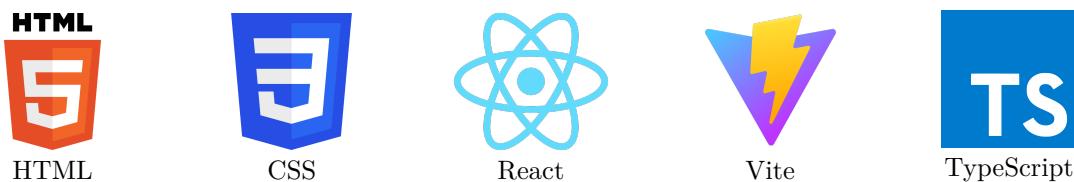
3 Implementacja aplikacji

Rozdział prezentuje, jak projekt został przełożony na działający kod. Zawiera omówienie użytego środowiska programistycznego, zestawu technologii i języków, a także wprowadzenie do kolejnych podsekcji, w których zostaną opisane najważniejsze elementy aplikacji i ich powiązania.

3.1 Wykorzystana technologia

Sekcja ta wprowadza kryteria doboru narzędzi użytych w projekcie oraz porządkuje dalszy opis implementacji. W kolejnych podsekcjach omówiono warstwę interfejsu użytkownika, część serwerową, sposób przechowywania danych oraz organizację pracy z kodem, ze wskazaniem wpływu tych decyzji na realizację wymagań, testowalność i koszt utrzymania pojedynczej instancji systemu.

3.1.1 Warstwa frontendowa: HTML, CSS, JavaScript/TypeScript



Rysunek 34: Warstwa frontendowa: HTML/CSS oraz stos Vite + React + TypeScript.

Frontend powstał w oparciu o Vite + React + TypeScript [10][8][9], co zapewnia bardzo krótki czas startu deweloperskiego (HMR), typowanie modeli przesyłanych do API oraz czytelną strukturę komponentów. HTML i CSS pozostają fundamentem renderowania, a warstwa stylów jest ograniczona do lekkich arkuszów i zmiennych kolorystycznych, by nie tworzyć zależności od ciężkich frameworków UI. Stylowanie zachowuje spójność z React Flow [14], który dostarcza kanwę do edycji grafów; kolory i typografia są definiowane we własnych klasach, co upraszcza utrzymanie i eliminację konfliktów między bibliotekami [8].

Wybór TypeScriptu [9] zamiast czystego JavaScriptu wynika z potrzeby jednoznacznego odwzorowania modeli domenowych po stronie klienta. Silne typowanie zmniejsza liczbę błędów integracyjnych, zwłaszcza przy serializacji/deskrypcji JSON z API. Alternatywy jak czysty JS przyspieszyłyby start, ale zwiększyłyby ryzyko błędów przy refaktoryzacji. React [8], w przeciwieństwie do frameworków szablonowych, pozwala budować hermetyczne komponenty edytora, paneli historii i harmonogramu, wykorzystując jeden model stanu.

React Flow [14] został wybrany jako biblioteka do wizualnej edycji połączeń, ponieważ zapewnia interaktywny canvas z obsługą drag & drop, walidację połączeń i blokadę cykli

na poziomie UI, łatwe mapowanie węzłów na dane domenowe.

HTML i CSS wykorzystano zgodnie z minimalnym podejściem: brak frameworka typu Bootstrap zmniejsza rozmiar paczki i eliminuje nadmiar stylów. Zamiast tego definiowane są zmienne kolorów, spacing i typografia w jednym miejscu, co pozwala utrzymać spójny akcent wizualny i łatwo przełączać motyw jasny/ciemny.

3.1.2 Warstwa backendowa: ASP.NET Core



Rysunek 35: Stos serwerowy: ASP.NET Core/EF Core oraz testy NUnit.

Backend działa jako ASP.NET Core Web API z podziałem na kontrolery, serwisy i repozytoria [11][7]. Kontrolery obsługują kontrakt HTTP i mapowanie DTO, serwisy realizują reguły biznesowe, a repozytoria operują na `DbContext`. Taki podział ułatwia testowanie i utrzymanie oraz ogranicza mieszanie logiki domenowej z warstwą transportową.

Konfiguracja startowa obejmuje rejestrację zależności DI [7] dla modułów workflowów, bloków, połączeń, zmiennych, rewizji i harmonogramu. Zastosowano też mechanizm executorów (`IBlockExecutor`), który oddziela logikę pojedynczych bloków od głównego algorytmu wykonania.

Centralnym elementem logiki jest serwis wykonania workflowu. Wczytuje graf bloków i połączeń, uruchamia przebieg od bloku `Start` i deleguje kolejne kroki do właściwych executorów. Odpowiada również za sterowanie przebiegiem, kontrolę cykli i zapis historii działań.

W ramach tej samej aplikacji planuje się uruchomienie schedulera jako usługi działającej w tle. Usługa ma cyklicznie sprawdzać harmonogramy gotowe do wykonania, uruchamiać workflow oraz aktualizować pola `LastRunAtUtc` i `NextRunAtUtc`. Cały mechanizm ma pracować na czasie UTC, z przeliczeniem na strefy czasowe przy wyznaczaniu kolejnych terminów uruchomienia.

W warstwie trwałości przewidziano EF Core z jawnie opisanymi relacjami i regułami usuwania w `FlowforgeDbContext` [12]. Po uruchomieniu system ma wykonywać migracje oraz weryfikować dostępność wymaganych bloków systemowych, aby zapewnić spójny stan danych potrzebnych do działania edytora i endpointów wykonawczych.

3.1.3 Magazyn danych: SQLite



Rysunek 36: Magazyn danych w trybie deweloperskim: SQLite.

W planowanej architekturze magazyn danych ma opierać się na SQLite [13], aby uprościć uruchomienie systemu i ograniczyć narzut administracyjny na etapie rozwoju projektu. Baza plikowa ma pozwolić na szybkie odtworzenie środowiska lokalnego bez instalacji osobnego serwera SQL, co jest istotne przy częstych zmianach modeli i testach funkcjonalnych.

Model danych ma być utrzymywany przez EF Core [12] z jawnym mapowaniem relacji między workflowami, blokami, połączeniami, harmonogramami i historią wykonań. Zakłada się stosowanie kluczy głównych i obcych, reguł usuwania kaskadowego tam, gdzie usunięcie obiektu nadzawanego powinno czyścić dane zależne, oraz ograniczeń `restrict/set null` w miejscach wymagających zachowania spójności referencyjnej.

Ewolucja schematu ma odbywać się przez migracje EF Core [12], tak aby każda zmiana modelu była wersjonowana i możliwa do odtworzenia w innych środowiskach. Plan zakłada uruchamianie migracji przy starcie aplikacji, co zmniejsza ryzyko niespójności między kodem domenowym a strukturą tabel oraz skracą czas przygotowania środowiska demonstracyjnego.

W warstwie operacyjnej przewidziano zapisywanie znaczników czasu w UTC oraz utrzymanie lekkiego profilu danych, bez dużych załączników binarnych w bazie. Dla danych o zmiennej strukturze (np. konfiguracje bloków i ślady wykonania) zakłada się przechowywanie treści serializowanych, co upraszcza rozwój funkcji edytora. W przypadku wzrostu obciążenia docelowy model ma umożliwiać przejście na serwerową bazę SQL bez zmiany kontraktów API i logiki aplikacyjnej.

3.1.4 Kontrola wersji i monorepo: Git



Rysunek 37: Kontrola wersji: Git w układzie monorepo.

W ramach projektu przyjęto organizację kodu w jednym repozytorium obejmującym część serwerową, interfejs użytkownika i testy [17]. Takie podejście pozwala wersjonować całą zmianę funkcjonalną jako spójną całość: razem z modyfikacją logiki backendu można zapisać odpowiadające jej zmiany w UI i testach. Dzięki temu historia projektu jest czytelniejsza, a ryzyko niespójności między warstwami systemu jest mniejsze.

Monorepo porządkuje również codzienną pracę. Przegląd zmian odbywa się w jednym miejscu, a odtworzenie środowiska jest prostsze, ponieważ struktura projektu i zasady pracy są wspólne dla wszystkich modułów. Jednocześnie każdy komponent zachowuje własny cykl zależności i narzędzi (`dotnet restore`, `npm install`), co pozwala utrzymać techniczną niezależność poszczególnych części aplikacji.

W zakresie praktyki wersjonowania zakłada się krótkie gałęzie robocze, częste scalanie oraz oznaczanie istotnych wersji tagami. Dla utrzymania porządku konieczne jest także konsekwentne pomijanie artefaktów lokalnych i plików builda, takich jak `bin/`, `obj/` i `node_modules/`. Z punktu widzenia pracy inżynierskiej taki model ułatwia odtworzenie kolejnych etapów rozwoju systemu i powiązanie decyzji projektowych z konkretnymi zmianami w kodzie.

3.1.5 Środowisko programistyczne: Rider i WebStorm



Rysunek 38: Użyte środowiska IDE: JetBrains Rider oraz JetBrains WebStorm

W opisie zaplecza technicznego uwzględniono dwa środowiska IDE: JetBrains Rider [23] oraz JetBrains WebStorm [24]. W praktyce Rider był używany do rozwoju warstwy backendowej (.NET), natomiast WebStorm do implementacji i utrzymania warstwy frontendowej (React/TypeScript). Nie są to elementy docelowego stosu uruchomieniowego aplikacji, ale narzędzia wspierające proces implementacji, debugowania i uruchamiania testów.

3.1.6 Uzasadnienie wyboru stosu jako całości

Dobór stosu technologicznego oparto na trzech kryteriach: spójności modelu danych między frontendem i backendem, prostocie wdrożenia oraz możliwości dalszego rozwoju systemu bez przebudowy podstaw architektury. Połączenie TypeScriptu [9] w warstwie klienckiej i kontraktów JSON po stronie API ogranicza ryzyko rozbieżności semantycznych oraz ułatwia utrzymanie jednolitej validacji danych wejściowych i wyników wykonania workflow.

Istotnym argumentem był również koszt utrzymania środowiska. Zastosowanie SQLite [13] i jednego procesu aplikacyjnego dla API oraz harmonogramu pozwala uruchomić system bez dodatkowej infrastruktury bazodanowej i bez skomplikowanej konfiguracji usług pomocniczych. Taka organizacja jest wystarczająca dla zakładanego obciążenia projektu i upraszcza przygotowanie środowiska demonstracyjnego.

3.2 Implementacja dostępu do danych

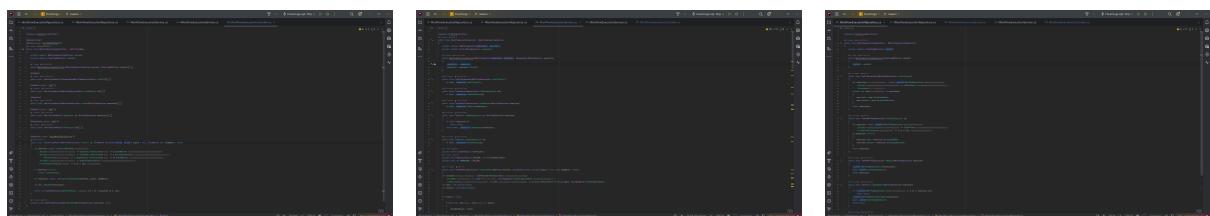
Warstwa dostępu do danych jest elementem łączącym logikę backendu z rzeczywistym stanem aplikacji. Jej rola nie ogranicza się do prostego zapisu i odczytu rekordów, ale obejmuje także utrzymanie spójności między modułami odpowiedzialnymi za edycję workflow, wykonywanie bloków oraz planowanie uruchomień. W projekcie przyjęto założenie, że model domenowy powinien być czytelny na poziomie kodu, a struktura relacyjna ma wynikać z tego modelu, a nie odwrotnie. Dzięki temu łatwiej zachować spójny język opisu systemu zarówno w dokumentacji, jak i w implementacji.

3.2.1 Zakres odpowiedzialności warstwy danych

W implementacji dostępu do danych rozdzielono trzy poziomy odpowiedzialności. Kontrolery API odpowiadają za przyjęcie żądania i mapowanie danych wejściowych, serwisy za reguły biznesowe, natomiast repozytoria za komunikację z bazą danych. Takie rozdzielenie upraszcza rozwój systemu, ponieważ zmiany w sposobie przechowywania danych nie muszą od razu wpływać na logikę endpointów. Jednocześnie logika biznesowa nie jest rozpraszana po wielu klasach infrastrukturalnych, co ułatwia testowanie oraz analizę błędów.



Rysunek 39: Podział odpowiedzialności w warstwie dostępu do danych

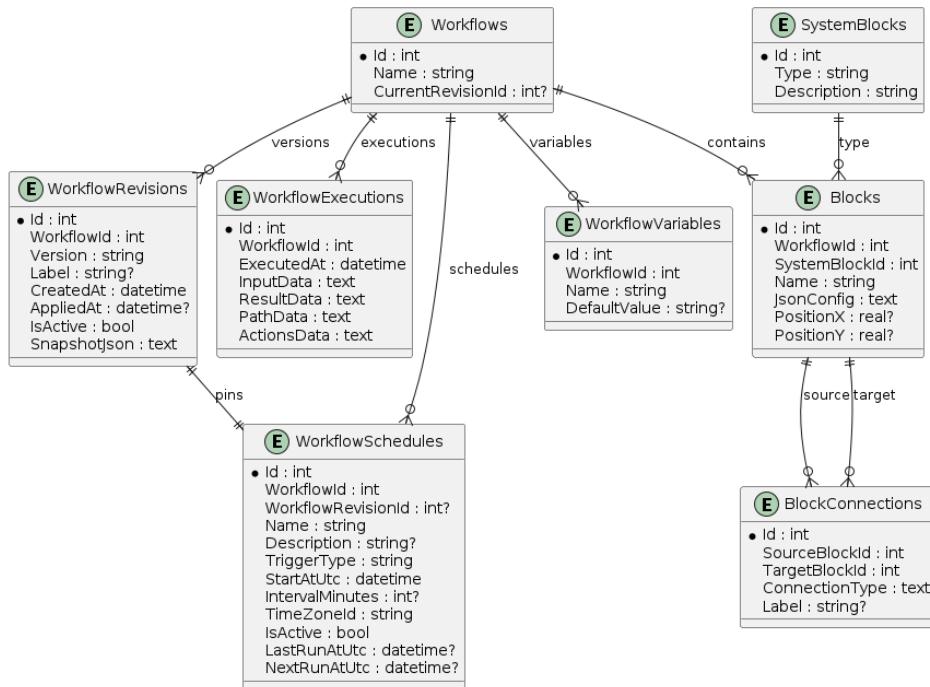


Rysunek 40: Implementacja przepływu end-to-end: `WorkflowExecutionController`, `WorkflowExecutionService` i `WorkflowExecutionRepository`

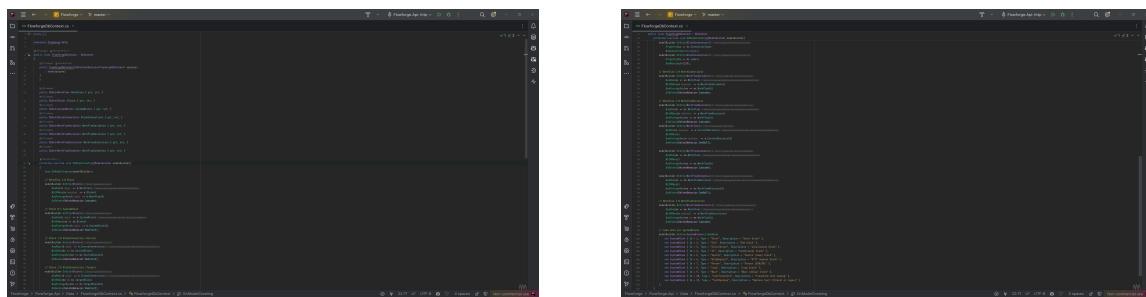
W praktyce taki układ pozwala precyzyjnie kontrolować zapytania wykonywane do bazy. Operacje odczytu i zapisu są skupione w repozytoriach dedykowanych konkretnym obszarom domeny, takim jak workflow, wykonania czy harmonogramy. Dzięki temu możliwe jest tworzenie metod odzwierciedlających rzeczywiste przypadki użycia, zamiast operowania na ogólnych zapytaniach o niejasnym przeznaczeniu. Ma to duże znaczenie przy utrzymaniu aplikacji, gdzie czytelność kodu przekłada się bezpośrednio na tempo wprowadzania zmian.

3.2.2 Model relacyjny i mapowanie encji

Model danych oparto na encjach odpowiadających głównym pojęciom domenowym. Workflow pełni rolę obiektu nadziedznego dla bloków, zmiennych, rewizji i historii uruchomień. Bloki są połączone relacjami reprezentującymi przepływ wykonania, a harmonogramy wskazują, kiedy dana definicja ma zostać uruchomiona. Taki model umożliwia odwzorowanie zarówno statycznej struktury przepływu, jak i zdarzeń dynamicznych, które pojawiają się podczas działania systemu. W tej sekcji diagram pełni funkcję referencyjną, a nacisk położono na mapowanie w kodzie.



Rysunek 41: ERD całości modelu danych Flowforge



Rysunek 42: Implementacja kontekstu danych FlowforgeDbContext: encje, relacje i reguły integralności

```

// WorkflowExecution.cs
using System;
using Flowforge.Models;
using Flowforge.Api.Models;

public class WorkflowExecution
{
    [JsonIgnore]
    public string Id { get; set; }

    [JsonIgnore]
    public string ExecutionId { get; set; }

    [JsonIgnore]
    public string? InputData { get; set; }

    [JsonIgnore]
    public string? ResultData { get; set; }

    [JsonIgnore]
    public string? PathData { get; set; }

    [JsonIgnore]
    public string? ActionsData { get; set; }

    [JsonIgnore]
    public string? Metadata { get; set; }

    [NotMapped]
    [JsonPropertyName("Input")]
    public string? Input { get; set; }

    [NotMapped]
    [JsonPropertyName("Result")]
    public string? Result { get; set; }

    [NotMapped]
    [JsonPropertyName("Path")]
    public string? Path { get; set; }

    [NotMapped]
    [JsonPropertyName("Actions")]
    public string? Actions { get; set; }

    [NotMapped]
    [JsonPropertyName("Metadata")]
    public string? Metadata { get; set; }

    public int WorkflowId { get; set; }

    public Workflow Workflow { get; set; } = null;

    [NotMapped]
    [JsonPropertyName("path")]
    public string? SerializedPath { get; set; }

    [NotMapped]
    [JsonPropertyName("actions")]
    public string? SerializedActions { get; set; }

    [NotMapped]
    [JsonPropertyName("metadata")]
    public string? SerializedMetadata { get; set; }

    [NotMapped]
    [JsonPropertyName("result")]
    public string? SerializedResult { get; set; }

    [NotMapped]
    [JsonPropertyName("input")]
    public string? SerializedInput { get; set; }
}

```

Rysunek 43: Model **WorkflowExecution**: mapowanie pól trwałych i pól transportowych (`JsonIgnore`, `NotMapped`)

Model **WorkflowExecution** opisuje pojedyncze wykonanie procesu i stanowi centralny punkt zapisu historii uruchomień. Definiuje identyfikator wykonania, powiązanie z rekordem **Workflow**, znaczniki czasu oraz metadane potrzebne do ustalenia, kiedy i w jakim kontekście uruchomiono dany przepływ. W praktyce to właśnie ten model buduje podstawę ścieżki audytowej: pozwala nie tylko stwierdzić, że wykonanie miało miejsce, ale także odtworzyć jego przebieg i rezultat.

Struktura klasy została podzielona na dwie warstwy reprezentacji danych. Pierwsza warstwa to pola trwałe (`InputData`, `ResultData`, `PathData`, `ActionsData`), przechowywane w bazie w postaci tekstowej i traktowane jako źródło prawdy po stronie persystencji. Druga warstwa to pola transportowe (`Input`, `Result`, `Path`, `Actions`), używane przez lo-

gikę aplikacyjną i odpowiedzi API. Dzięki temu model nie miesza wymagań schematu relacyjnego z wygodą pracy na typach domenowych po stronie kodu.

Adnotacje `JsonIgnore` i `NotMapped` wyznaczają granicę odpowiedzialności między ORM i serializacją HTTP. `JsonIgnore` ukrywa pola *Data przed bezpośrednią ekspozycją w kontrakcie API, co ogranicza ryzyko udostępniania surowego formatu technicznego. Z kolei `NotMapped` informuje EF Core, że pola transportowe nie mają być mapowane jako oddzielne kolumny. W efekcie jedna klasa może jednocześnie obsługiwać zapis w bazie i czytelny kontrakt danych zwracanych klientowi.

Mechanizm getterów i setterów realizuje konwersję między obiema reprezentacjami. Odczyt deserializuje dane JSON do obiektów używanych przez silnik wykonawczy i UI, natomiast zapis serializuje struktury domenowe z powrotem do postaci trwałej. Taki układ upraszcza ewolucję modelu, ponieważ zmiany po stronie API lub logiki wykonania można w wielu przypadkach wprowadzać bez przebudowy relacyjnego schematu tabel. W kontekście implementacji dostępu do danych model `WorkflowExecution` definiuje więc zarówno format utrwalania wyników, jak i sposób ich bezpiecznej prezentacji na zewnątrz systemu.

```

1  <?xml version="1.0" encoding="utf-8" ?>
2  <assembly>
3      <!-- Namespaces -->
4      <namespaces>
5          <namespace>FlowForge.Repositories;</namespace>
6      </namespaces>
7  </assembly>
8  <!-- Interfaces & Contracts -->
9  <interface>IWorkflowExecutionRepository</interface>
10 <!-- Implementation -->
11 <class>WorkflowExecutionRepository : IWorkflowExecutionRepository</class>
12     <!-- Fields -->
13     private readonly FlowForgeDbContext _context;
14
15     <!-- Constructors -->
16     public WorkflowExecutionRepository(FlowForgeDbContext context)
17     {
18         _context = context;
19     }
20
21     <!-- Methods -->
22     #region IWorkflowExecutionRepository<-->
23     public async Task<IList<WorkflowExecution>> GetAllAsync()
24     {
25         var executions = await _context.WorkflowExecutions
26             .Where(e => e.Status != Status.Canceled || e.Status != Status.Failed)
27             .ToListAsync();
28
29         foreach (var exec in executions)
30         {
31             exec.Path = exec.SerializedPath;
32             exec.Actions = exec.SerializedActions;
33         }
34
35         return executions;
36     }
37
38     #endregion
39
40     #region IWorkflowExecutionRepository<-->
41     public async Task<WorkflowExecution> GetByIdAsync(int id)
42     {
43         var execution = await _context.WorkflowExecutions
44             .Include(e => e.Actions)
45             .Where(e => e.Id == id)
46             .FirstOrDefaultAsync();
47
48         if (execution != null)
49         {
50             execution.Path = execution.SerializedPath;
51             execution.Actions = execution.SerializedActions;
52         }
53
54         return execution;
55     }
56
57     #endregion
58
59     #region ITask<WorkflowExecution>
60     public async Task<WorkflowExecution> AddAsync(WorkflowExecution execution)
61     {
62         _context.Add(execution);
63         await _context.SaveChangesAsync();
64     }
65
66     #endregion
67
68     #region ITask<WorkflowExecution>
69     public async Task<bool> UpdateAsync(WorkflowExecution execution)
70     {
71         _context.Update(execution);
72         await _context.SaveChangesAsync();
73     }
74
75     #endregion
76
77     #region ITask<int>
78     public async Task<int> DeleteAsync(int id)
79     {
80         _context.Remove(_context.WorkflowExecutions
81             .Where(e => e.Id == id));
82         await _context.SaveChangesAsync();
83     }
84
85     #endregion
86
87     #region ITask<void>
88     public void Dispose()
89     {
90         _context.Dispose();
91     }
92
93     #endregion
94
95     #region IDisposable<-->
96     public void Dispose()
97     {
98         _context.Dispose();
99     }
100 #endregion
101 </class>
102 </assembly>

```

Rysunek 44: Odczyt danych wykonania i mapowanie pól `Path/Actions` w `WorkflowExecutionRepository`

Konfiguracja relacji została zaprojektowana tak, aby odzwierciedlać konsekwencje biznesowe usuwania i aktualizacji danych. W relacjach silnie zależnych stosowane są reguły kaskadowe, natomiast tam, gdzie utrata danych mogłyby utrudnić analizę historii, preferowane są ograniczenia typu `restrict` lub `set null`. Pozwala to uniknąć przypadkowego usunięcia kluczowych informacji, np. powiązań potrzebnych do odtworzenia przebiegu wykonania. Z perspektywy inżynierskiej jest to istotne, ponieważ integralność danych staje

się częścią kontraktu systemu, a nie jedynie efektem ubocznym działania ORM.

Dla pól o zmiennej strukturze, takich jak konfiguracje wybranych bloków, przyjęto zapis serializowany. Rozwiążanie to ułatwia rozwój aplikacji w sytuacji, gdy lista parametrów może się zmieniać wraz z dodawaniem nowych typów bloków. Jednocześnie kluczowe atrybuty identyfikujące i łączące obiekty pozostają relacyjne, co zapewnia wydajne filtrowanie i sortowanie danych. Takie połączenie elastyczności oraz rygoru strukturalnego dobrze odpowiada charakterowi narzędzia low-code.

3.2.3 Migracje i kontrola zmian schematu

Zmiany modelu danych są wersjonowane z użyciem migracji EF Core [12]. Każda modyfikacja encji, relacji lub ograniczeń jest zapisywana jako osobny krok, który można odtworzyć w innym środowisku. Takie podejście porządkuje proces rozwoju i ogranicza ryzyko ręcznych, niespójnych zmian w bazie. Ma to również znaczenie metodyczne: historia migracji pokazuje, jak ewoluował model danych wraz z kolejnymi wymaganiami.

The screenshot shows two side-by-side code editors in a Visual Studio-like interface. Both editors have the same file open: Program.cs from a FlowForge project.

Top Editor (Migration Configuration):

```

1  using System.Text.Json.Serialization;
2  using FlowForge.Data;
3  using FlowForge.Models;
4  using FlowForge.Repositories;
5  using FlowForge.Services;
6  using Microsoft.EntityFrameworkCore;
7
8  var builder = WebApplication.CreateApplicationBuilder(args);
9
10 // Dodawanie serwisiów do kontenera DI
11 builder.Services.AddControllers();
12
13 builder.Services.AddDbContext<FlowForgeDbContext>(options =>
14     options.UseSqlite(builder.Configuration.GetConnectionString("DefaultConnectionString")));
15
16 builder.Services.AddIdentity()
17     .AddDefaultIdentityOptions(<-- reference here)
18     .AddJsonFormatter(<-- reference here);
19
20 builder.Services.AddControllers()
21     .AddJsonFormatter(<-- reference here);
22
23 builder.Services.AddRepositories()
24     .AddConnections(<-- reference here)
25     .AddSerializers(<-- reference here);
26
27 builder.Services.AddRepositories<WorkflowRepository, WorkflowRepository>();
28 builder.Services.AddRepositories<WorkflowTaskRepository, WorkflowTaskRepository>();
29 builder.Services.AddRepositories<BlockRepository, BlockRepository>();
30 builder.Services.AddRepositories<BlockService, BlockService>();
31
32 builder.Services.AddRepositories<WorkflowLabelRepository, WorkflowLabelRepository>();
33 builder.Services.AddRepositories<WorkflowLabelService, WorkflowLabelService>();
34
35 builder.Services.AddRepositories<BlockConnectionRepository, BlockConnectionRepository>();
36 builder.Services.AddRepositories<BlockConnectionService, BlockConnectionService>();
37
38 builder.Services.AddRepositories<SystemLocalRepository, SystemLocalRepository>();
39 builder.Services.AddRepositories<SystemLocalService, SystemLocalService>();
40
41
42 
```

Bottom Editor (Migration Execution):

```

1  using Microsoft.EntityFrameworkCore;
2
3  var app = builder.Build();
4
5  using (var scope = app.Services.CreateScope())
6  {
7      var context = scope.ServiceProvider.GetRequiredService<FlowForgeDbContext>();
8
9      context.Database.Migrate();
10
11      var requiredBlocks = new List<SystemBlock>()
12      {
13          new SystemBlock { Type = "start", Description = "Start block" },
14          new SystemBlock { Type = "end", Description = "End block" },
15          new SystemBlock { Type = "calculation", Description = "Calculation block" },
16          new SystemBlock { Type = "text", Description = "Text block" },
17          new SystemBlock { Type = "textblock", Description = "Text block" },
18          new SystemBlock { Type = "httprequest", Description = "HTTP request block" },
19          new SystemBlock { Type = "httpresponse", Description = "HTTP response block" },
20          new SystemBlock { Type = "httperror", Description = "HTTP error block" },
21          new SystemBlock { Type = "loop", Description = "Loop block" },
22          new SystemBlock { Type = "wait", Description = "Wait (delay) block" },
23          new SystemBlock { Type = "textreplace", Description = "Text replace block" },
24          new SystemBlock { Type = "textreplace", Description = "Replace last literal or regex" }
25      };
26
27      foreach (var block in requiredBlocks)
28      {
29          if (!context.SystemBlocks.Any(b => b.Type == block.Type))
30          {
31              context.SystemBlocks.Add(block);
32          }
33          else
34          {
35              var existingSystemBlock = context.SystemBlocks.First(b => b.Type == block.Type);
36              existingSystemBlock.Description = block.Description;
37          }
38      }
39
40      context.SaveChanges();
41
42      // Konfiguracja pipeline
43 
```

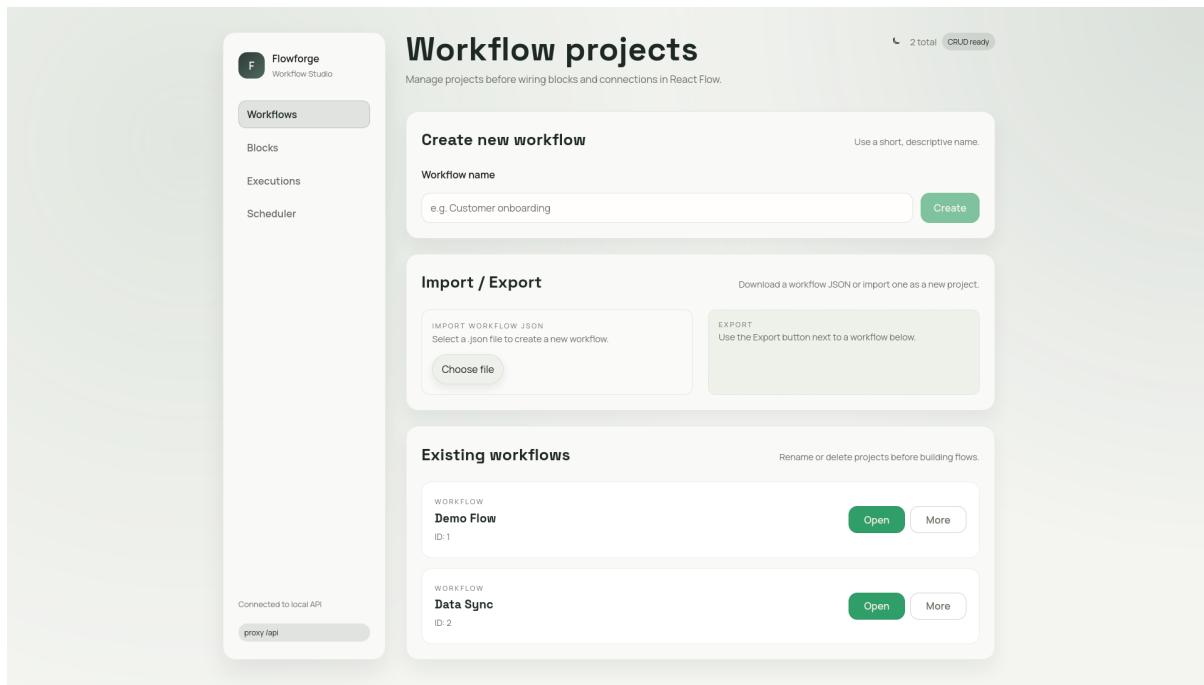
Rysunek 45: Konfiguracja `UseSqlite(...)` oraz wywołanie `Database.Migrate()` w `Program.cs`

Istotnym elementem jest automatyczne dopasowanie schematu przy uruchomieniu aplikacji. Dzięki temu nową instancję systemu można odtworzyć bez dodatkowych kroków administracyjnych. Po stronie implementacji oznacza to mniejszą liczbę błędów wynikających z różnic wersji bazy oraz bardziej przewidywalne uruchomienie aplikacji na nowych maszynach.

3.3 Interfejs

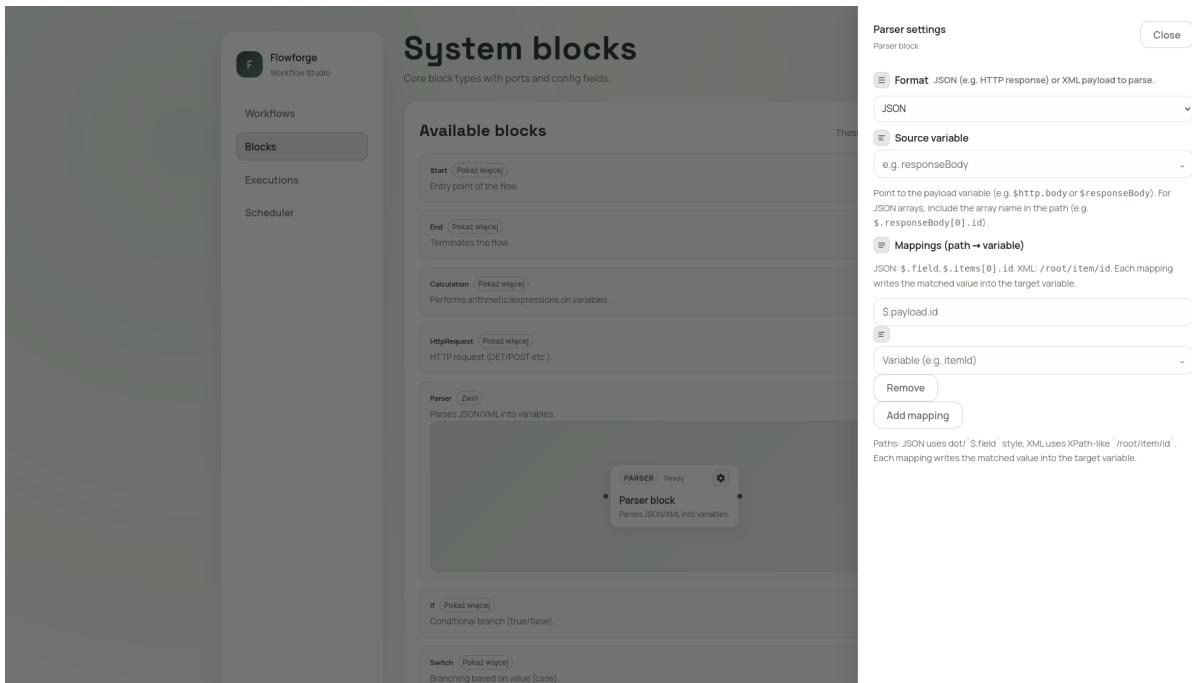
W tej sekcji przedstawiono działający interfejs aplikacji w jasnym motywie oraz rzeczywiste widoki systemu z dostępnymi operacjami użytkownika.

- **Workflows:** tworzenie przepływu, wejście do edytora, akcje kontekstowe i zarządzanie wersjami.
- **Blocks:** przegląd bloków systemowych i przejście do konfiguracji.
- **Editor:** budowanie grafu procesu, łączenie bloków i edycja parametrów.
- **Executions:** historia uruchomień z szybkim podglądem statusu i czasu wykonania.
- **Execution Details:** analiza wejścia, wyniku, ścieżki bloków i logów.
- **Scheduler:** planowanie cyklicznych uruchomień i ręczne wyzwalanie.
- **Parser config:** konfiguracja bloku Parser w bocznym panelu.



Rysunek 46: Interfejs: widok Workflows

Na rysunku 46, znajduje się widok **Workflows** prezentujący listę przepływów wraz z metadanymi, statusem publikacji i zestawem operacji administracyjnych dostępnych bezpośrednio z tabeli.



Rysunek 47: Interfejs: widok Blocks

Na rysunku 47, znajduje się widok **Blocks** pokazujący katalog bloków systemowych z podziałem na typy funkcjonalne, co ułatwia wybór komponentów przed budową lub modyfikacją przepływu.

Rysunek 48: Interfejs: widok Executions

Na rysunku 48, znajduje się widok **Executions** z historią uruchomień, statusem i czasem wykonania, a także możliwością filtrowania rekordów i przejścia do szczegółów konkretnego wykonania.

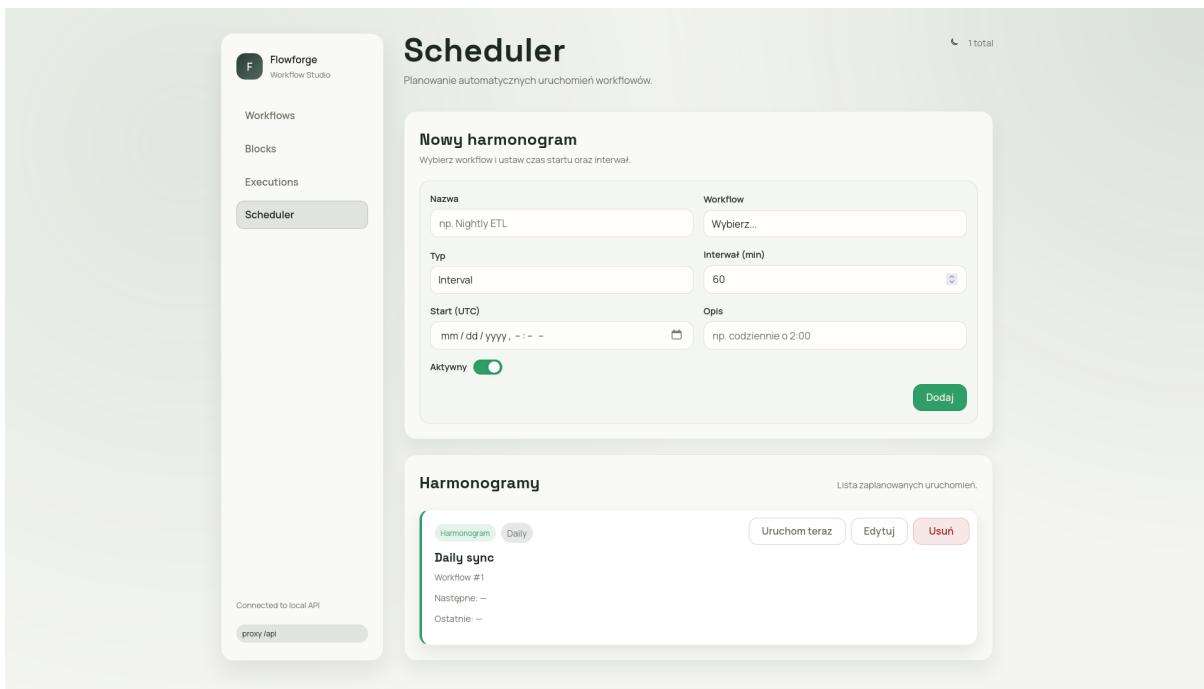
The screenshot shows the Flowforge Workflow Studio interface, specifically the 'Execution Details' page for run #36. The top navigation bar includes 'Workflows', 'Blocks', 'Executions' (which is selected), and 'Scheduler'. The main content area is titled 'Execution #36' and displays the following sections:

- Overview**: Shows workflow and run metadata: Workflow (TextTransform), Executed at (2/5/2026, 7:09:48 PM), Path length (3).
- Inputs**: Variables passed into the run, showing a JSON object: { "id": "2", "result": "" }.
- Results**: Output variables produced by the workflow, showing \$ID (3) and RESULT (test test testt).
- Path**: Order of block execution with a quick flow map, showing three blocks: 1 Start (Executed), 2 Text Transform (Executed), and 3 End (Executed). The flow map shows arrows from Start to Text Transform and from Text Transform to End.
- Actions**: Detailed execution logs, listing Start block, TextTransform Lower → result, and End block.

At the bottom left, it says 'Connected to local API' and 'proxy /api'. The bottom right corner shows the date and time: 2/5/2026, 7:09:48 PM.

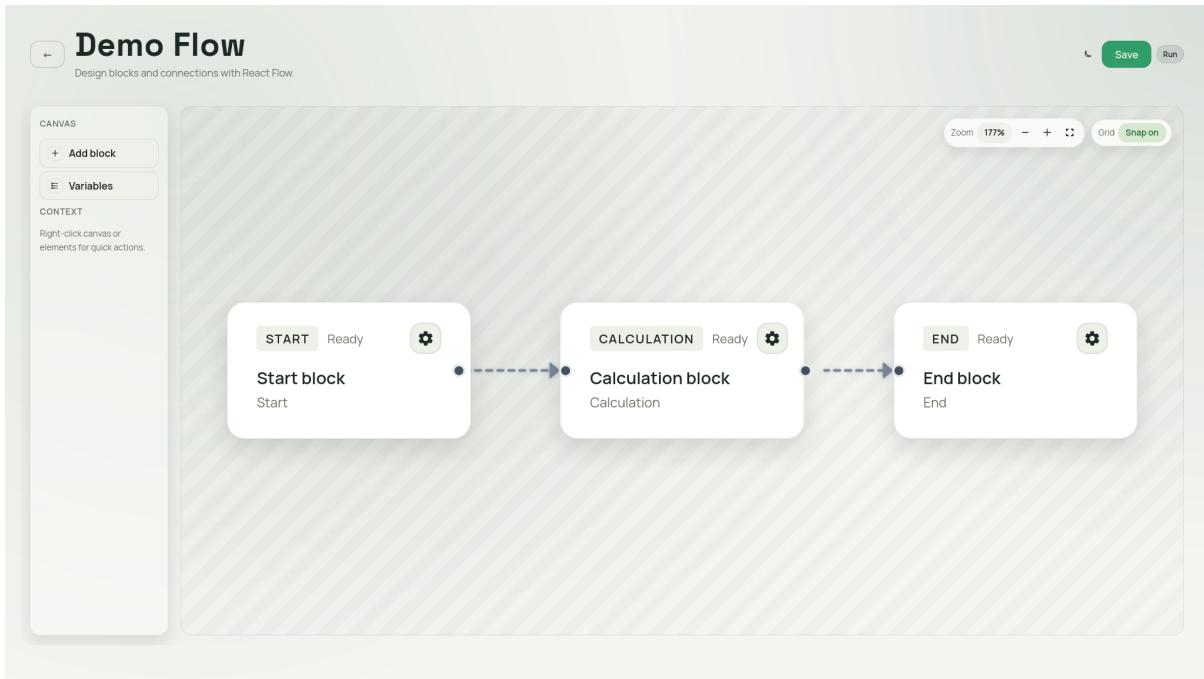
Rysunek 49: Interfejs: widok Execution Details

Na rysunku 49, znajduje się widok **Execution Details** zawierający dane wejściowe, wynik, ścieżkę przejścia między blokami oraz logi diagnostyczne potrzebne do analizy przebiegu.



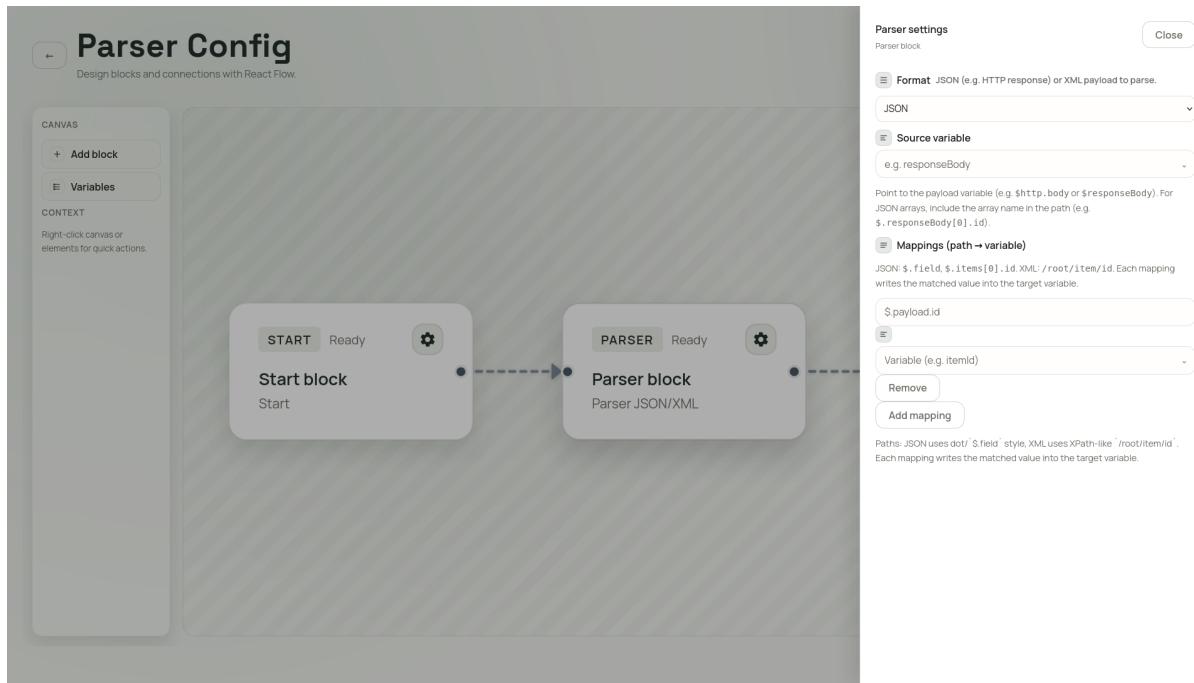
Rysunek 50: Interfejs: widok Scheduler

Na rysunku 50, znajduje się widok **Scheduler** przeznaczony do konfigurowania uruchomień planowych, aktywacji harmonogramów, wyboru strefy czasowej i bieżącej kontroli parametrów cyklu.



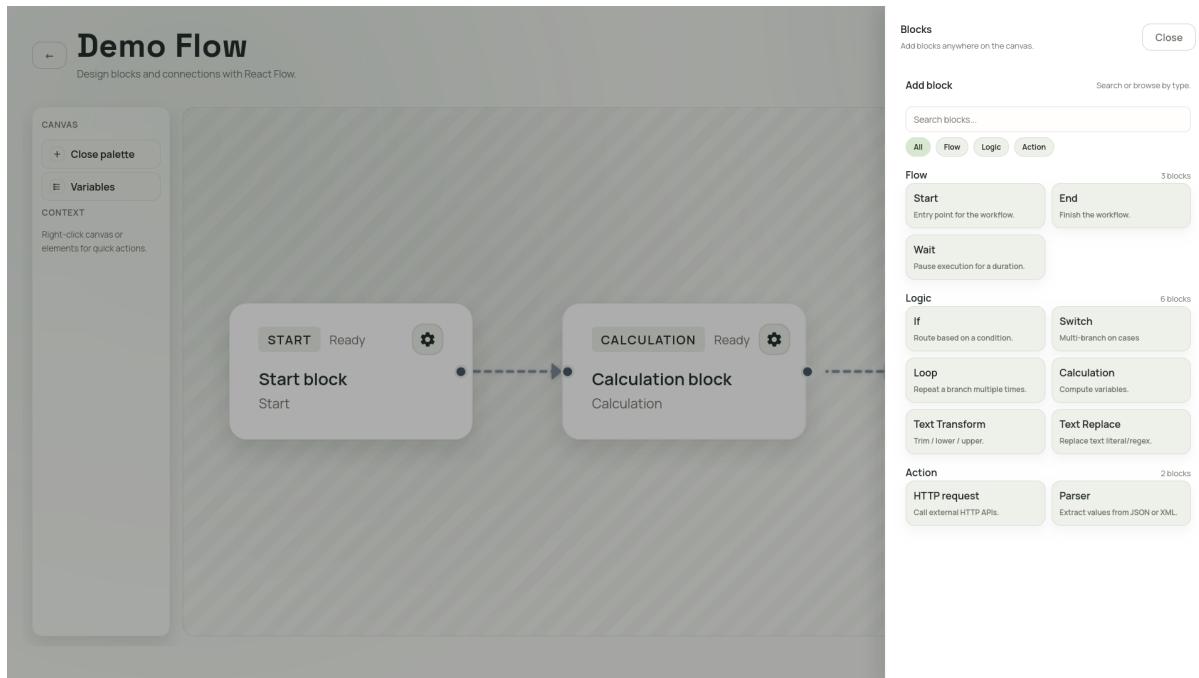
Rysunek 51: Interfejs: widok edytora workflow

Na rysunku 51, znajduje się główny edytor workflow, w którym użytkownik buduje graf procesu przez dodawanie bloków, łączenie ich krawędziami oraz weryfikację struktury przepływu przed publikacją.



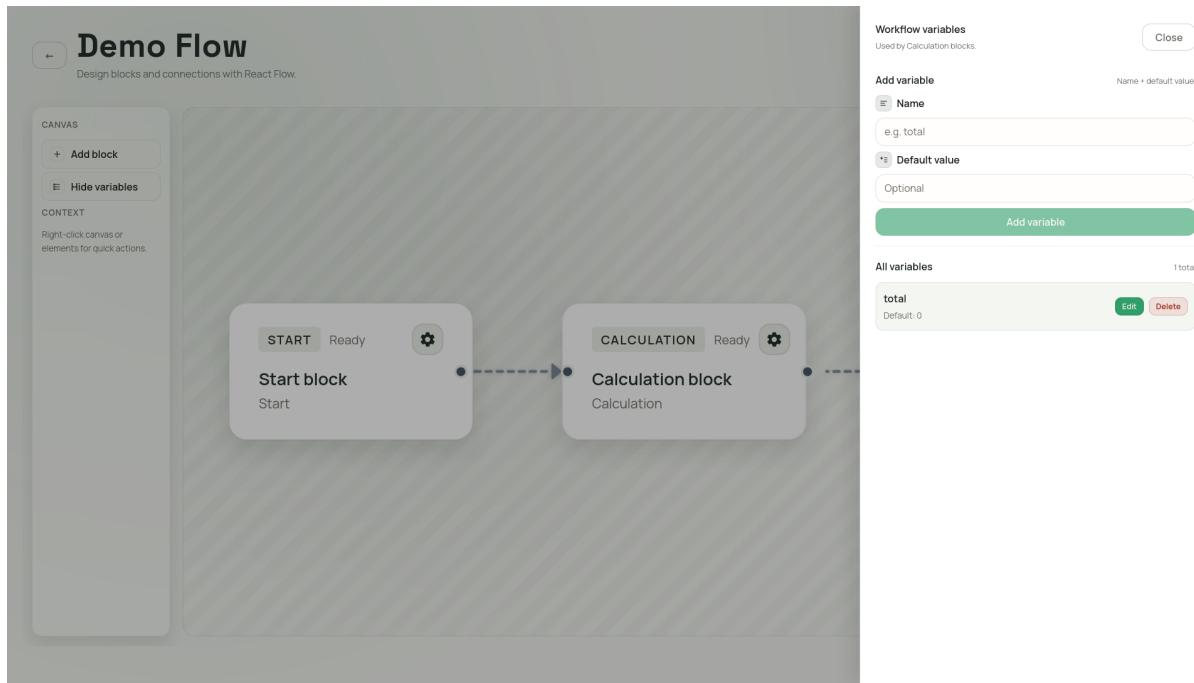
Rysunek 52: Interfejs: konfiguracja bloku Parser

Na rysunku 52, znajduje się formularz konfiguracji bloku **Parser** z polami formatu danych, źródła wejścia i mapowania wyników, co pozwala precyzyjnie zdefiniować transformację treści.



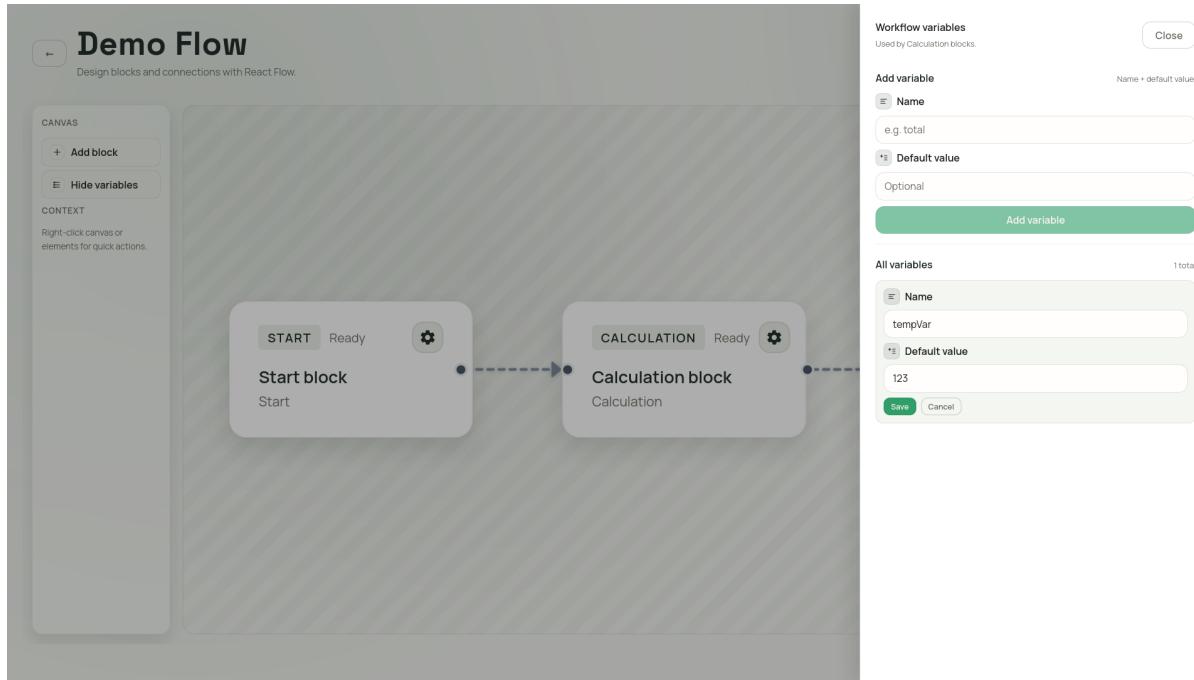
Rysunek 53: Interfejs: context menu dodawania bloków

Na rysunku 53, znajduje się menu kontekstowe dodawania bloków dostępne na canvasie edytora, które skraca ścieżkę pracy i umożliwia szybkie rozszerzanie grafu bez zmiany widoku.



Rysunek 54: Interfejs: widok zmiennych

Na rysunku 54, znajduje się widok zmiennych workflow wykorzystywanych przez bloki podczas wykonania, z listą nazw i wartości wspólnych dla całego przebiegu procesu.



Rysunek 55: Interfejs: edycja zmiennych

Na rysunku 55, znajduje się formularz edycji zmiennych umożliwiający zmianę wcześniejszej zdefiniowanych parametrów przepływu oraz kontrolę poprawności danych przekazywanych między blokami.

3.4 Testy jednostkowe (NUnit)

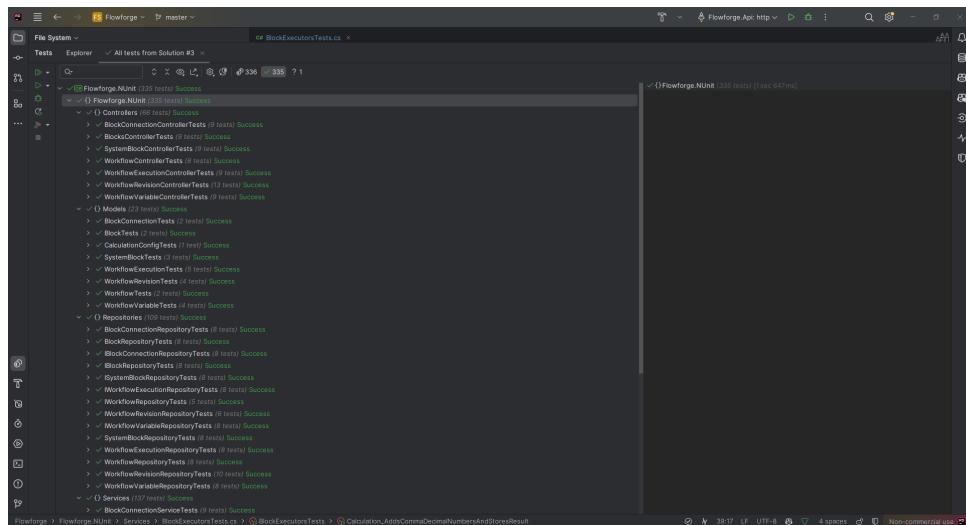
Weryfikację poprawności logiki aplikacji przeprowadzono z użyciem testów jednostkowych w framework'u NUnit [15], uruchamianych w projekcie `flowforge.nunit`. Zakres testów obejmuje kontrolery, serwisy, repozytoria oraz modele domenowe. Każdy przypadek testowy koncentruje się na jednej odpowiedzialności i jest wykonywany w izolacji od pozostałych warstw, co ogranicza ryzyko błędnej interpretacji wyniku i ułatwia wykrywanie regresji.

Podstawowy sposób uruchamiania testów zapewnia interfejs linii poleceń [16]:

```
1 dotnet test flowforge.nunit/Flowforge.NUnit.csproj
```

Listing 1: Uruchamianie testów jednostkowych NUnit z poziomu CLI

Mimo dostępności trybu CLI, w praktyce testy wykonywano głównie w środowisku JetBrains Rider [23], korzystając z zakładki **Tests**. Takie podejście przyspiesza analizę wyników i skraca czas przejścia od błędu do poprawki.



Rysunek 56: Uruchomienie testów jednostkowych projektu `Flowforge.NUnit`

Na rysunku 56, znajduje się widok uruchomienia testów jednostkowych wraz z informacją o przebiegu wykonania i podsumowaniem wyników.

4 Zakończenie

Zrealizowany system potwierdza, że podejście backend-first [21] pozwala zbudować stabilny fundament dla aplikacji low-code, a następnie rozwijać interfejs użytkownika bez utraty spójności modelu domenowego. W trakcie prac wdrożono mechanizm definiowania i wykonywania workflow, obsługę harmonogramów, historię uruchomień oraz podstawowy zestaw testów jednostkowych. Uzyskany rezultat spełnia założenia funkcjonalne i tworzy punkt wyjścia do dalszego rozwoju produktu.

4.1 Perspektywy rozwojowe aplikacji

Pierwszym naturalnym kierunkiem rozwoju jest konteneryzacja środowiska uruchomieniowego z użyciem Dockera [25]. Rozdzielenie komponentów na obrazy (API, front-end, baza danych lub wolumen danych) uprościłoby proces wdrożenia, a także zwiększyło powtarzalność środowisk między komputerem deweloperskim, serwerem testowym i środowiskiem produkcyjnym. W praktyce oznacza to mniejszą liczbę problemów konfiguracyjnych i szybsze uruchamianie nowych instancji systemu.

Drugim obszarem jest rozbudowa warstwy testów o testy end-to-end z użyciem Playwright [26]. Obecne testy jednostkowe dobrze zabezpieczają logikę backendu, jednak nie obejmują pełnych scenariuszy użytkownika przechodzących przez UI, API i warstwę danych jednocześnie. Dodanie testów E2E dla kluczowych ścieżek (utworzenie workflow, publikacja, uruchomienie, analiza wykonania, konfiguracja harmonogramu) podniosłoby wiarygodność kolejnych wydań i ograniczyło ryzyko regresji po zmianach interfejsu.

Z punktu widzenia procesu wytwarzania zasadne jest także wdrożenie pipeline'ów CI/CD w repozytorium Git, np. z wykorzystaniem GitHub Actions [27]. Pipeline powinien wykonywać automatycznie: komplikację rozwiązania, testy jednostkowe, testy E2E, kontrolę jakości kodu oraz budowę artefaktów. Taki model skraca czas informacji zwrotnej, a jednocześnie wprowadza obiektywną bramkę jakości przed scaleniem zmian do gałęzi głównej.

W warstwie funkcjonalnej aplikacja może być rozwijana przez dodawanie nowych typów bloków systemowych, np. konektorów do baz danych oraz bloków odpytujących dane bezpośrednio z relacyjnych źródeł. Rozszerzanie katalogu bloków zwiększa zakres przypadków użycia, które użytkownik może zrealizować bez modyfikacji kodu źródłowego.

Kolejnym krokiem może być wprowadzenie mechanizmu tworzenia własnych bloków przez GUI. W wariantie podstawowym użytkownik definiowałby parametry wejściowe i wyjściowe, reguły validacji oraz prostą logikę transformacji w bezpiecznym sandboxie wykonawczym. W wariantie zaawansowanym możliwe byłoby udostępnienie kreatora bloków z wersjonowaniem, testem podglądu i publikacją do lokalnego katalogu organizacji. Takie rozwiązanie zwiększa elastyczność platformy, ale wymaga równoległego zaprojektowania kontroli bezpieczeństwa, limitów zasobów i polityki uprawnień.

Istotnym usprawnieniem pozostaje również moduł kont użytkowników oraz uprawnień. W aktualnym stanie aplikacja zakłada pojedynczy kontekst pracy, natomiast wdrożenie modelu wieloużytkownikaowego pozwoliłoby przypisywać role (np. administrator, projektant workflow, obserwator), ograniczać dostęp do wybranych procesów oraz prowadzić audyt zmian na poziomie użytkownika. Rozszerzenie o uwierzytelnianie i autoryzację umożliwiłoby bezpieczne użycie systemu w zespołach oraz organizacjach o bardziej formalnych wymaganiach dostępowych.

4.2 Podsumowanie

Aplikacja została wykonana zgodnie z założeniami projektowymi przyjętymi na początku pracy, jako rozwiązanie umożliwiające tworzenie, uruchamianie i monitorowanie workflow w architekturze low-code. Realizację oparto na połączeniu backendu ASP.NET Core [11], modelu danych EF Core [12] oraz interfejsu SPA [5] opartego na React [8]. W efekcie powstało rozwiązanie obejmujące pełny cykl życia przepływu: od definicji bloków i połączeń, przez publikację i uruchomienie, po analizę wyników i historii wykonań.

Istotnym rezultatem projektu jest zachowanie spójności między modelem domenowym, API i interfejsem użytkownika. Rozdzielenie odpowiedzialności na kontrolery, serwisy i repozytoria uporządkowało kod oraz ograniczyło mieszanie logiki biznesowej z warstwą infrastrukturalną, a migracje EF Core [12] i automatyczne dopasowanie schematu bazy zwiększyły powtarzalność uruchomień. Warstwę jakościową oparto na testach jednostkowych backendu [15], które stanowią stabilną bazę do dalszego rozwoju.

Do głównych zalet aplikacji należą: szybkie uruchomienie bez rozbudowanej infrastruktury, czytelny model tworzenia przepływów blokowych oraz możliwość śledzenia przebiegu wykonania na podstawie historii i danych wejścia/wyjścia. Z perspektywy rynkowej jest to kierunek uzasadniony, ponieważ organizacje potrzebują narzędzi automatyzujących procesy wewnętrzne bez konieczności każdorazowego programowania dedykowanych integracji. Oznacza to, że aplikacja odpowiada na realną potrzebę skrócenia czasu wdrażania automatyzacji i obniżenia kosztu utrzymania rozwiązań procesowych.

Literatura

- [1] Dokumentacja n8n. Dostęp: 9 lutego 2026. <https://n8n.io>
- [2] Projekt Node-RED. Dostęp: 9 lutego 2026. <https://nodered.org>
- [3] Zapier Platform Overview. Dostęp: 9 lutego 2026. <https://zapier.com>
- [4] Microsoft Power Automate. Dostęp: 9 lutego 2026. <https://powerautomate.microsoft.com>
- [5] MDN Web Docs: Single-page application. Dostęp: 10 lutego 2026. <https://developer.mozilla.org/en-US/docs/Glossary/SPA>
- [6] Microsoft Docs: Kestrel web server implementation in ASP.NET Core. Dostęp: 10 lutego 2026. <https://learn.microsoft.com/aspnet/core/fundamentals/servers/kestrel>
- [7] Microsoft Docs: Dependency injection in ASP.NET Core. Dostęp: 10 lutego 2026. <https://learn.microsoft.com/aspnet/core/fundamentals/dependency-injection>
- [8] React Documentation. Dostęp: 10 lutego 2026. <https://react.dev>
- [9] TypeScript Handbook. Dostęp: 10 lutego 2026. <https://www.typescriptlang.org/docs/>
- [10] Vite Documentation. Dostęp: 10 lutego 2026. <https://vitejs.dev/guide/>
- [11] ASP.NET Core documentation. Dostęp: 10 lutego 2026. <https://learn.microsoft.com/aspnet/core>
- [12] Entity Framework Core documentation. Dostęp: 10 lutego 2026. <https://learn.microsoft.com/ef/core>
- [13] SQLite Documentation. Dostęp: 10 lutego 2026. <https://www.sqlite.org>
- [14] React Flow documentation. Dostęp: 10 lutego 2026. <https://reactflow.dev>
- [15] NUnit Documentation. Dostęp: 10 lutego 2026. <https://nunit.org/>
- [16] Microsoft Docs: dotnet test. Dostęp: 19 lutego 2026. <https://learn.microsoft.com/dotnet/core/tools/dotnet-test>
- [17] Pro Git Book. Dostęp: 10 lutego 2026. <https://git-scm.com/book>
- [18] W. H. Inmon, *Building the Data Warehouse*, 4th ed., Wiley, 2005.

- [19] R. Kimball, M. Ross, *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd ed., Wiley, 2013.
- [20] A. Silberschatz, H. F. Korth, S. Sudarshan, *Database System Concepts*, 7th ed., McGraw-Hill, 2019.
- [21] S. Newman, *Building Microservices*, 2nd ed., O'Reilly Media, 2021.
- [22] J. Bogard, “Vertical Slice Architecture”, 2018. Dostęp: 19 lutego 2026. <https://www.jimmybogard.com/vertical-slice-architecture/>
- [23] JetBrains Rider Documentation. Dostęp: 19 lutego 2026. <https://www.jetbrains.com/rider/documentation/>
- [24] JetBrains WebStorm Documentation. Dostęp: 19 lutego 2026. <https://www.jetbrains.com/webstorm/documentation/>
- [25] Docker Documentation. Dostęp: 19 lutego 2026. <https://docs.docker.com>
- [26] Playwright Documentation. Dostęp: 19 lutego 2026. <https://playwright.dev/docs/intro>
- [27] GitHub Actions Documentation. Dostęp: 19 lutego 2026. <https://docs.github.com/actions>

Spis rysunków

1	Przykładowy widok edytora przepływu w n8n	3
2	Przykładowy widok przepływu w Node-RED	3
3	Widok listy automatyzacji w Zapier	4
4	Widok przepływu w Power Automate	4
5	Diagram domenowy systemu Flowforge	6
6	Przypadki użycia Flowforge. Zgrupowane według widoków UI i powiązań z backendem	9
7	Utworzenie nowego workflow z listy Workflows. Użytkownik wypełnia formularz i zapisuje szkic przepływu	11
8	Import snapshotu z menu More. Wskazanie pliku JSON i dodanie wersji po pozytywnej walidacji	11
9	Eksport snapshotu do pliku JSON. Zapis bieżącej konfiguracji przepływu do pobrania	11
10	Zmiana nazwy workflow w menu More. Wprowadzenie nowej etykiety i zapis na liście	11

11	Publikacja wersji workflow. Wersja robocza staje się wykonywalna, endpoint HTTP dostępny	12
12	Przegląd szczegółów uruchomienia. Wejście, wyjście, logi, ścieżka bloków oraz opcja ponownego uruchomienia	12
13	Ręczne uruchomienie workflow z run drawera. Wprowadzenie zmiennych i start bez wychodzenia z edytora	12
14	Dodanie nowego bloku na canvas. Przeciągnięcie z palety i umieszczenie między portami	12
15	Konfiguracja parametrów bloku. Uzupełnienie pól, walidacja i zapis ustawień	13
16	Podgląd bloku w sekcji Blocks. Mini-canvas i panel konfiguracji bez zapisywania zmian	13
17	Zapis wersji roboczej w edytorze. Aktualizacja szkicu workflow w bazie .	13
18	Walidacja konfiguracji bloków przy zapisie/publikacji. Błędne pola blokują zapis do wersji	13
19	Utworzenie lub edycja harmonogramu. Ustawienie daty, godziny i strefy czasowej przed zapisaniem	14
20	Aktywacja lub dezaktywacja harmonogramu. Przełącznik statusu z natychmiastowym zapisem	14
21	Wyzwolenie istniejącego harmonogramu opcją „Wyzwól teraz”. Jednorazowe uruchomienie zdefiniowanego przepływu	14
22	Wybór strefy czasowej w formularzu harmonogramu. Automatyczna strefa z przeglądarki lub ręczny wybór	14
23	Architektura fizyczna Flowforge. Przepływ komunikacji między SPA, API, schedulerem i bazą SQLite	15
24	ERD rdzenia: Workflow z blokami, połączeniami, zmiennymi, rewizjami, uruchomieniami i harmonogramami	16
25	ERD konfiguracji: typy JSON bloków (HTTP, parser, kalkulacja, if/switch) i relacje pomocniczych tabel	16
26	ERD wykonania: WorkflowExecution z referencją do Workflow oraz polami serializującymi wejście, wynik i ścieżki	17
27	ERD harmonogramu: WorkflowSchedule przypięty do Workflow i opcjonalnej WorkflowRevision, z polami czasu i strefy	17
28	Enumy wspierające konfiguracje: ConnectionType, CalculationOperation, ConditionDataType, HttpRequestAuthType, ParserFormat	17
29	ERD bazy danych: Workflows, Revisions, Executions, Schedules, Variables, Blocks, SystemBlocks i BlockConnections z kardynalnościami	21
30	Ideowe interakcje komponentów: SPA, API, baza SQLite/EF Core, scheduler i klient zewnętrzny	21
31	Publikacja wersji i jednorazowe wyzwolenie workflow z edytora	23

32	Obsługa harmonogramu przez usługę tła (polling, wyzwolenie, zapis wykonania)	24
33	Wywołanie endpointu workflow przez system zewnętrzny	24
34	Warstwa frontendowa: HTML/CSS oraz stos Vite + React + TypeScript.	25
35	Stos serwerowy: ASP.NET Core/EF Core oraz testy NUnit.	26
36	Magazyn danych w trybie deweloperskim: SQLite.	27
37	Kontrola wersji: Git w układzie monorepo.	28
38	Użyte środowiska IDE: JetBrains Rider oraz JetBrains WebStorm	29
39	Podział odpowiedzialności w warstwie dostępu do danych	30
40	Implementacja przepływu end-to-end: kontroler, serwis i repozytorium	30
41	ERD całości modelu danych Flowforge	31
42	Implementacja kontekstu danych FlowforgeDbContext	31
43	Model WorkflowExecution: mapowanie pól trwałych i transportowych	32
44	Odczyt danych wykonania i mapowanie pól Path/Actions	33
45	Konfiguracja UseSqlite(...) i wywołanie Database.Migrate()	34
46	Interfejs: widok Workflows	35
47	Interfejs: widok Blocks	36
48	Interfejs: widok Executions	36
49	Interfejs: widok Execution Details	37
50	Interfejs: widok Scheduler	38
51	Interfejs: widok edytora workflow	38
52	Interfejs: konfiguracja bloku Parser	39
53	Interfejs: context menu dodawania bloków	39
54	Interfejs: widok zmiennych	40
55	Interfejs: edycja zmiennych	40
56	Uruchomienie testów jednostkowych projektu Flowforge.NUnit	41

Listings

1	Uruchamianie testów jednostkowych NUnit z poziomu CLI	41
---	---	----