AKADEMIA TARNOWSKA
Wydział Nauk Technicznych

Kierunek: ***Informatyka***

2025/2026

Michał Michalski

PRACA INŻYNIERSKA

Projekt i implementacja programu narzędziowego do testowania API aplikacji webowych

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, *2026*

Spis treści

1. Wstęp

Testowanie oprogramowania jest bardzo ważną częścią w procesie wytwarzania oprogramowania. Gwarantuje ono nie tylko zgodność produktu z wymaganiami funkcjonalnymi, ale również zapewnia wysoką jakość kodu oraz stabilność rozwiązania w środowisku produkcyjnym.

W przypadku aplikacji webowych dużą rolę odgrywa interfejs programowania aplikacji (z ang. Application Programming Interface, API), który służy do interakcji z logiką biznesową serwera. Ingeruje on w wiele elementów i części takiej aplikacji - między innymi jest on odpowiedzialny za pobieranie i wysyłanie zasobów od klienta do serwera i bazy danych - dlatego tak ważnym jest napisanie testów integracyjnych.

1.1. Cel pracy

Głównym celem niniejszej pracy jest zaprojektowanie i zbudowanie programu umożliwiającego interakcje do interfejsów programowania API aplikacji webowych oraz analizę zwracanych odpowiedzi w odniesieniu do spodziewanych danych. Narzędzie ma pozwolić użytkownikowi (testerowi) na zdefiniowanie parametrów zapytania, obserwowanie zachowania serwera przy różnych konfiguracjach danych wejściowych oraz wykonywanie testów automatycznych w formie scenariuszy testowych.

1.2. Zakres pracy

Niniejsza praca obejmuje:

- projekt narzędzia,
- implementacja projektu,
- tworzenie testów jednostkowych i integracyjnych dla gwarancji poprawnego działania narzędzia.

2. Analiza biznesowa

Rozdział ten skupia się na przedstawieniu wymagań funkcjonalnych i niefunkcjonalnych projektu. Spełnienie ich wszystkich podczas implementacji jest kluczowe dla zapewnienia jakości pracy.

2.1. Wymagania funkcjonalne

Główną funkcją projektu jest dostarczenie deweloperom oraz testerom stron internetowych i aplikacji webowych rozbudowanego narzędzia do testowania API opartych na protokole HTTP/HTTPS (Hypertext Transfer Protocol Secure). System ma umożliwiać skuteczne sprawdzanie poprawności, wydajności oraz bezpieczeństwa komunikacji pomiędzy klientem a serwerem.

Dla użytkownika kluczowe będą następujące możliwości:

- Tworzenie i zarządzanie testami - użytkownik może tworzyć, edytować, usuwać i zapisywać scenariusze testowe poprzez definiowanie parametrów żądań (np. adres URL, metoda, nagłówki oraz ewentualne dane).
- Testowanie poprawności działania - walidacja kodów odpowiedzi HTTP oraz porównywanie danych zwróconych przez API z oczekiwanymi wynikami, takie jak nagłówki, treść, struktura danych JSON, ciasteczka i tym podobne.
- Obsługa autoryzacji i uwierzytelniania - system umożliwi testowanie zabezpieczonych API z wykorzystaniem powszechnych metod autoryzacji, takich jak Basic Auth, tokeny sesji (Session Tokens) lub Bearer Token, poprzez zapisywanie i wysyłanie zmiennych takich jak klucze API w nagłówkach zapytania lub przez ciasteczka (cookies).
- Raportowanie i analiza wyników - użytkownik będzie mógł generować raporty z wykonanych testów w czytelnym formacie zarówno dla programistów, jak i dla osób nietechnicznych. Raporty będą zawierały statystyki, zestawienia wyników, wykresy oraz wyróżnienie błędów krytycznych.
- Automatyzacja uruchamiania scenariuszy testowych - możliwość ponownego uruchomienia testów po zmianach w kodzie API dla testowania regresji oraz możliwosć ingerencji w współczesnych środowiskach programistycznych za pomocą CI/CD.

2.2. Wymagania niefunkcjonalne

Poza funkcjonalnością kluczowe znaczenie mają również cechy jakościowe systemu, które wpływają na komfort pracy użytkownika i niezawodność narzędzia.

- Wydajność - system musi być w stanie wykonywać dużą liczbę zapytań i odbierać odpowiedzi z API w krótkim czasie podczas testów. Interfejs powinien zapewniać płynność działania oraz minimalizować czas reakcji.
- Użyteczność - interfejs użytkownika powinien cechować się wysoką użytecznością, rozumianą jako intuicyjna obsługa, logiczna struktura funkcji oraz czytelna prezentacja wyników. W tym kontekście istotne jest zachowanie czytelności komunikatów oraz możliwość szybkiego definiowania scenariuszy testowych.
- Niezawodność - system powinien działać w sposób stabilny i przewidywalny niezależnie od przeprowadzanych testów. Nieakceptowalnym byłaby awaria programu podczas ich wykonywania. Program powinien też obsłużyć niespodziewane błędy podczas wysyłania zapytań, na przykład kiedy serwer nie wyśle odpowiedzi (timeout), źle wprowadzony adres lub problemy z połączeniem z internetem.
- Kompatybilność - oprogramowanie powinno być możliwe do uruchomienia na najpopularniejszych systemach desktopowych, takich jak Windows, macOS oraz Linux, co zwiększa dostępność narzędzia dla szerokiej grupy użytkowników. Ponadto pożądane jest, aby system umożliwiał współpracę z popularnymi rozwiązaniami stosowanymi w procesach CI/CD oraz wspierał konteneryzację, co ułatwia jego wdrażanie w środowiskach automatyzacji testów.

3. Stos technologiczny

W niniejszym rozdziale przedstawiony jest stos technologiczny wykorzystywany podczas projektu i implementacji narzędzia. Dobór technologii został podyktowany wymaganiami wymienionymi w rozdziale 2., w którym została przeprowadzona analiza biznesowa.

3.1. C++

C++ jest językiem programowania ogólnego przeznaczenia. Został stworzony przez Bjarne'a Stroustrupa w 1985 roku jako rozszerzenie języka C o elementy programowania obiektowego, a w późniejszych wersjach również funkcjonalnego. Jest jednym z najbardziej popularnych języków programowania oraz ceniony jest za wydajność i efektywność, dzięki czemu dobrze się nadaje do pisania aplikacji desktopowych. [1]

Wybór tego języka był podyktowany znajomością składni oraz środowiska i narzędzi programistycznych wspierających proces tworzenia projektu.

3.2. libcurl

Biblioteka *libcurl* jest odpowiedzialna za wykonywanie zapytań w protokołach HTTP i HTTPS, które są głównie wykorzystywane przez strony WWW oraz aplikacje webowe. Zapewnia stabilny i przetestowany interfejs programistyczny, wykorzystywany również przez znane narzędzie wiersza poleceń *curl*, które jest równocześnie rozwijane z tą biblioteką. [3]

Pomimo tego, że jego API jest głównie stworzone dla języka C, przystosowanie do działania w kodzie C++ jest stosunkowo proste, co pozwala wykorzystać funkcje i idiomy tego języka. Użycie gotowego już rozwiązania jako fundamentu działania programu, którym jest w zasadzie wykonywanie zapytań HTTP, znacząco upraszcza kod źródłowy w kwestii operacji sieciowych, bez potrzeby zagłębiania się w obsługę niższych warstwy modelu OSI (protokół HTTP znajduje się na najwyższej warstwie aplikacji).

3.3. YAML

YAML jest formatem oraz specyfikacją przeznaczoną do reprezentowania danych w ustrukturyzowany sposób w formie pliku tekstowego. Jego rolą w tym projekcie jest możliwość modyfikacji oraz zapisywania scenariuszy testowych, czyli sekwencji testów przeprowadzanych do testowania API aplikacji webowych. Dzięki temu scenariusze testowe są łatwe w edycji, bez potrzeby znajomości składni języka programowania przez testera, a także format tekstowy jest w prosty sposób obsługiwany przez kontrolę wersji, jak np. Git. [4]

3.4. JSON

JSON (JavaScript Object Notation) jest formatem wymiany danych. Posiada prostą gramatykę i jest łatwy w obsłudze i przetwarzaniu przez komputery. Obsługuje struktury danych typu pary "klucz-wartość" oraz tablicy (*array*).

Jest bardzo szeroko wykorzystywany w API aplikacjach webowych, ze względu na prostotę oraz natywną obsługę tego formatu w po stronie klienta (zazwyczaj odnosi się to do przeglądarek internetowych). [5]

3.5. Qt

Qt jest frameworkiem oferującym rozbudowany zestaw narzędzi do tworzenia desktopowych aplikacji. Jego istotną cechą jest wieloplatformowość, umożliwiająca kompilację i uruchamianie programu na popularnych systemach, takich jak Windows, Mac czy Linux. W tym projekcie zostanie wykorzystany głównie do implementacji GUI (graficznego interfejsu użytkownika). [6]

3.6. CMake

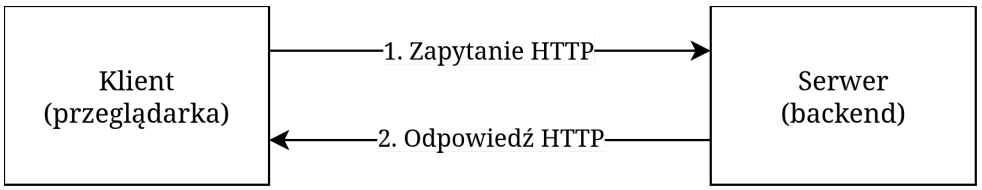
CMake jest narzędziem programistycznym służącym do zarządzania procesem kompilacji projektu. Pozwala na definiowanie konfiguracji budowania w sposób niezależny od konkretnego kompilatora czy środowiska programistycznego, co z kolei umożliwia łatwiejszą kontrolę nad strukturą projektu, integrację z zewnętrznymi bibliotekami oraz automatyzację procesu generowania plików budowania i kompilacji dla różnych systemów operacyjnych. [7]

4. Opis i implementacja systemu

Wszystkie aplikacje webowe cechuje architektura klient-serwer. Oznacza to podział takiej aplikacji na dwie główne części: klient, który inicjuje żądania i prezentuje dane użytkownikowi, oraz serwer, który przetwarza żądania i udostępnia zasoby. W kontekście aplikacji webowych klientem jest najczęściej przeglądarka internetowa lub aplikacja mobilna, a serwerem jest system działający na zdalnej maszynie, obsługujący logikę biznesową, dostęp do baz danych i komunikację sieciową.

Komunikacja klient-serwer odbywa się za pośrednictwem sieci komputerowej. W przypadku publicznych serwisów i stron jest to sieć internet. Protokołem odpowiedzialnym za przesył danych jest HTTP, ponieważ jest to główny protokół wykorzystywany przez przeglądarki internetowe jako klient aplikacji webowej. Jest to proces bardzo podobny w przypadku kiedy użytkownik odwiedza zwykłą stronę internetową, ale aplikacje webowe rozszerzają o zakres wysyłania i odbierania danych, na przykład autoryzacja (logowanie), zapisanie danych formularza, wysłanie zdjęcia i tym podobne.

Kluczową cechą protokołu HTTP jest to, że komunikacja za jego pomocą jest rodzajem typu żądanie-odpowiedź (z ang. request-response), gdzie dla jednego wysłania zapytania w stronę serwera przypada tylko jedna odpowiedź. Ponadto, protokół HTTP jest bezstanowy - oznacza to, że wszystkie dane potrzebne do uzyskania konkretnego zasobu (takie jak rodzaj typu danych, tokeny uwierzytelniające, itp.) muszą zostać zawarte w każdym zapytaniu, a zrealizowane żądania nie wpływają na stan i rezultat następnych odpowiedzi. [8]



Rys 4.1. Zasada działania architektury klient-serwer w kontekście protokołu HTTP.

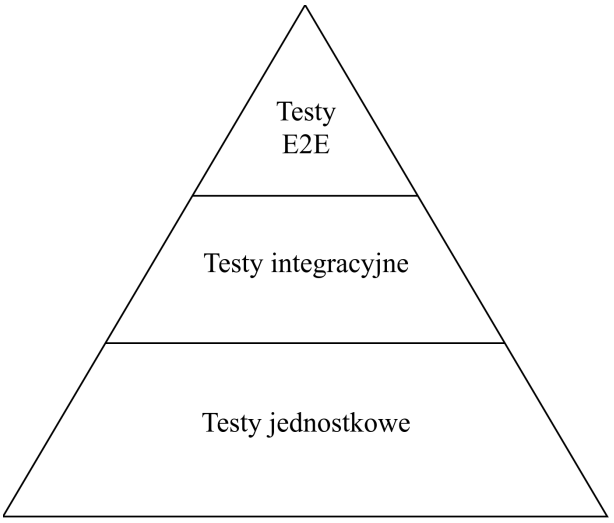
Najczęściej w aplikacjach webowych do przechowywania i przetwarzania danych wykorzystuje się bazy danych. Jednakże w typowej architekturze takiej aplikacji logika związana z dostępem do danych, na przykład wykonywanie kwerend SQL czy zapisywanie rekordów (danych) znajduje się po stronie serwera. Aby klient (front-end) mógł korzystać z tych danych, potrzebny jest interfejs który umożliwia przesyłanie i odbieranie informacji w ustandaryzowany sposób. Taką rolę pełni właśnie API (Application Programming Interface), czyli interfejs programistyczny pozwalający różnym komponentom lub systemom komunikować się ze sobą.

4.1. Charakterystyka testowania API

Testowanie API ma charakter testów integracyjnych, ponieważ koncentruje się na weryfikacji poprawnej komunikacji oraz przepływu danych pomiędzy odrębnymi modułami systemu, a nie jedynie na izolowanej logice pojedynczych funkcji.

W kontekście piramidy testów, testowanie API plasuje się w warstwie pośredniej, łącząc jednostkowe elementy kodu z kompletnym systemem. Uznaje się je za testy integracyjne z następujących powodów:

- Weryfikacja kontraktu: Sprawdzają, czy producent (serwer) i konsument (klient) poprawnie interpretują format danych (np. JSON), nagłówki oraz kody statusów.
- Interakcja z infrastrukturą: Testy te często wymagają uruchomienia wielu różnych komponentów części back-endowej aplikacji webowej, jak np. bazy danych, systemów kolejkowych czy zewnętrznych usług uwierzytelniających, aby potwierdzić, że komponenty te współpracują ze sobą bez błędów.
- Ciągłość procesów biznesowych: Pozwalają na sprawdzenie, czy sekwencja zapytań (np. logowanie, dodanie produktu, zakup) zachowuje spójność stanu aplikacji w różnych jej warstwach.



Rys 4.2. Piramida testów

W odróżnieniu od testów jednostkowych, które operują na poziomie kodu źródłowego i konkretnych funkcji lub klas, testy API operują na poziomie sieciowym, co pozwala wykryć błędy wynikające z konfiguracji środowiska, serializacji obiektów czy problemów z łącznością między usługami. [2]

Ponadto, testy przeprowadzane przez program mają charakter testów czarno-skrzynkowych, (z ang. "black-box testing"), gdzie tester może nie mieć dostępu do

wewnętrznej implementacji testowanego API, a tylko do jej specyfikacji/dokumentacji, która określa jakie dane są wysyłane i jakie dane powinny przyjść.

Rozwiązanie to nie ma na celu zastąpić istniejące metody testowania, takie jak testy jednostkowe kodu lub testy interfejsu użytkownika, a ogólnie rozszerzyć procedurę testowania oprogramowania, skupiając się na zgodności odpowiedzi API z oczekiwanym zachowaniem.

4.2. Architektura programu

Podczas tworzenia programu narzędziowego zostały zaimplementowane osobne komponenty do przeprowadzania testów:

- Parser pliku z zestawem testów do przeprowadzenia - odpowiada za sprawdzenie pliku w poszukiwaniu błędów oraz przetworzenie danych z pliku do struktury danych w pamięci programu w celu możliwości ich użycia w testerze.
- Komponent do uruchamiania testów - odpowiada za zautomatyzowane uruchomienie wszystkich testów z zestawu załadowanych z pliku testów.
- Klient HTTP - odpowiedzialny za wysyłanie i odbieranie danych do/z serwera testowanego API.
- Graficzny interfejs użytkownika - służy do tworzenia, wyświetlania oraz edycji parametrów testów.

Dla wydajnego działania programu, klient HTTP uruchamiany jest na osobnym wątku. Sprawia to, że każde wywołanie zapytania HTTP, które może trwać nawet kilkanaście sekund, nie będzie powodować blokowania aplikacji, które może objawiać się nieresponsywnością interfejsu użytkownika, oraz umożliwia przerwanie zapytania po określonym czasie (kiedy otrzymanie odpowiedzi trwa zbyt długo) lub na żądanie użytkownika.

4.3. Format pliku z scenariuszem testowym

Na potrzeby projektu została zaprojektowana od podstaw struktura pliku definiującego scenariusze testowe. Sam plik jest w czytelnym formacie YAML. Zawarte w nim są instrukcje potrzebne dla programu do przeprowadzenia testu, takie jak zawartość nagłówek oraz głównej treści (body) zapytania HTTP, jak i spodziewane wartości otrzymane w odpowiedzi od serwera. Takie deklaratywne rozwiązanie ma pewne wady i zalety.

Zalety:

- Przejrzysty i łatwo czytelny format, dzięki czemu może mieć on drugorzędową funkcję w postaci dokumentacji testowanego API (informacja o tym co jest wysyłane, a co powinniśmy otrzymać).

Wady:

- Takie rozwiązanie jest bardziej “sztywne” w porównaniu do pisania testów w typowy, imperatywny sposób, co może ograniczyć możliwości takich testów.

4.4. Dokumentacja schematu pliku YAML dla scenariuszy testowych

Struktura opiera się na głównym obiekcie **testSuite**, który definiuje globalne parametry oraz listę testów, takie jak:

- **name** (opcjonalne): Nazwa i krótki opis scenariusza.
- **serverUrl**: Bazowy adres URL testowanego API. Stanowi wspólną część ścieżki dla wszystkich punktów końcowych (endpoints).
- **options** (opcjonalne): Globalne ustawienia wykonawcze:
  - **timeoutSecs**: Maksymalny czas oczekiwania na odpowiedź serwera w sekundach (domyślnie: 120, czyli 2 minuty).
  - **abortOnFail**: Wartość logiczna, przerywa wykonywanie scenariusza po pierwszym nieudanym teście (domyślnie “false”).
  - **abortOnError**: Wartość logiczna, przerywa wykonywanie scenariusza w przypadku błędu testu (domyślnie “false”).
- **variables** (opcjonalne): Lista par `klucz: “wartość”`. Służy do definiowania danych wspólnych dla wszystkich testów, np. tokenów autoryzacyjnych.
- **setUp** (opcjonalne): Lista interakcji z API wykonywanych *przed* każdym testem.
- **tearDown** (opcjonalne): Lista interakcji z API wykonywanych *po* każdym teście.
- **tests**: Lista obiektów definiujących poszczególne przypadki testowe.

Każdy element listy **testSuite.tests** składa się z następujących danych:

- **description** (opcjonalne): Opis celu biznesowego lub technicznego testu.
- **options** (opcjonalne): Ustawienia wykonawcze dla konkretnego testu:
  - **timeoutSecs** (opcjonalne): Maksymalny czas oczekiwania na odpowiedź serwera w sekundach. Nadpisuje globalne ustawienie.
  - **skip**: Flaga oznaczająca, czy dany test należy pominąć podczas wykonywania scenariusza. Domyślnie "*false*".
  - **skipSetUp**: Czy interakcje opisane w "testSuite.setUp" mają zostać pominięte dla tego testu. Domyślnie "*false*".
  - **skipTearDown**: Czy interakcje opisane w "testSuite.tearDown" mają zostać pominięte dla tego testu. Domyślnie "*false*".
- **requests**: Kluczowa sekcja zawierająca listę interakcji z API.

Interakcje z API definiują dane potrzebne do wykonania zapytania oraz oczekiwane dane odpowiedzi z serwera. Ich struktura wygląda następująco:

- **description** (opcjonalne): Opis celu konkretnej interakcji z API.
- **request**: Obiekt zawierający parametry zapytania HTTP (metoda, ścieżka, nagłówki, zawartość body):
  - **path**: Ścieżka do endpointu API wraz z parametrami. Zostanie ona połączona z końcem "testSuite.serverUrl".
  - **method**: Metoda HTTP do wykonania tego zapytania; możliwe wartości to: "GET", "POST", "PUT", "PATCH", "DELETE", "HEAD".
  - **headers** (opcjonalne): Zestaw wartości `klucz: "wartość` definiujący nagłówki dla danego zapytania.
  - **body** (opcjonalne): Zawartość głównej treści zapytania HTTP.
- **response**: Definicja oczekiwanego stanu odpowiedzi (kod statusu, struktura body), służąca do asercji. Wszystkie poniższe opcje są opcjonalne. Jeżeli jest brak takiej opcji, to wartość zwrócona z API nie będzie sprawdzana i zostanie zaakceptowana.
  - **code**: Spodziewany status kodu (np. "200" dla OK lub "404" dla Not Found).
  - **headers**: Spodziewane wartości nagłówek w odpowiedzi.
  - **cookiesSet**: Spodziewane wysłane ciasteczka przez serwer oraz ich dane.
  - **body**: Spodziewane dane zawarte w ciele odpowiedzi z serwera.

W implementacji projektu zostały zaprogramowane trzy metody sprawdzania otrzymanych danych: tekstowo, porównując z plikiem oraz za pomocą schematu JSON.

Porównanie **text** pozwala na badanie zawartości tekstowych odpowiedzi przy użyciu wzorców i operatorów dopasowania.:

- **value**: Spodziewana wartość która będzie porównana z otrzymaną wartością z serwera.
- **matchType** (opcjonalnie): Sposób porównania. Możliwe opcje to:
  - **exact**: Ciąg tekstowy **value** musi być taki sam co otrzymana wartość. Jest to domyślna wartość jeżeli **matchType** nie zostanie zdefiniowane.
  - **contains**: Ciąg tekstowy **value** musi być podciągiem sprawdzanej wartości.
  - **startsWith**: Sprawdzana wartość musi zaczynać się od ciągu zdefiniowanego w **value**.
  - **endsWith**: Sprawdzana wartość musi kończyć się na **value**.
  - **regex**: Traktuje **value** jako wzorzec wyrażenia regularnego i próbuje dopasować do sprawdzanej wartości.
- **save** (opcjonalnie): Zapisz otrzymaną wartość do zmiennej o tej nazwie. Będzie ona zachowana dla przyszłych interakcji z API w obrębie jednego testu.

Porównanie za pomocą pliku wykorzystywane jest do weryfikacji odpowiedzi na podstawie zewnętrznego pliku. Przy definicji należy użyć słowa "**file**". Dostępne opcje to:

- **path**: Ścieżka relatywna do pliku wzorcowego (względem pliku YAML). Odpowiedź serwera musi być identyczna z zawartością wskazanego pliku.

Porównanie do schematu JSON (tryb "**json**" służy do technicznej weryfikacji poprawności struktury JSON. Jest to o tyle przydatne, ponieważ jest to jeden z najbardziej powszechnie wykorzystywanych formatów danych do przesyłania informacji w API aplikacji webowych.

4.5. Przykładowe scenariusze testowe i ich opis

Listing 4.1 zawiera tylko jeden test i w nim tylko jedną interakcję: wysłać zapytanie do *http://example.com/* metodą GET i po otrzymaniu odpowiedzi potwierdzić że:

- kod statusu wynosi 200 (OK);
- nagłówek "Content-Type" to "text/html";
- w zawartości znajduje się ciąg znaków "<h1>Example Domain</h1>".

Listing 4.1. Prosty przykład scenariusza testowego

```
testSuite:
  name: Przykładowy scenariusz testowy
```

```
  serverUrl: http://example.com
  tests:
    "getPage":
      description: "Pobierz stronę główną"
      skip: false
      requests:
        -
          request:
            path: /
            method: GET
          response:
            code: 200
            headers:
              "content-type": "text/html"
            body:
              text:
                matchType: contains
                value: "<h1>Example Domain</h1>"
```

Listing 4.2 przedstawia bardziej skomplikowany przykład który bardziej odzwierciedla rzeczywiste testowanie REST API aplikacji webowej. Nadal znajduje się tylko jeden test, ale za to wykorzystywane już są polecenia do tworzenia i sprawdzania wartości JSON.

Listing 4.2. Przykład testowania REST API z wykorzystaniem JSON

```
testSuite:
  name: Test API forum społecznościowego
  serverUrl: https://social.example.net/api
  variables:
    apiKey: "api_key_secret_"
  tests:
    threadTest:
      description: "Utwórz wątek"
      requests:
        -
          description: "Zapostuj nowy wątek"
          request:
            path: "/thread"
            method: POST
            headers:
              x-api-key: "${apiKey}"
            body:
```

```
      json:
        title: "Nowy wątek"
        content: "Cześć. To jest test. Lorem ipsum."
    response:
      code: 201
      body:
        jsonSchema:
          type: object
          properties:
            id:
              type: string
              save: threadId
  -
    description: "Otrzymaj nowo utworzony wątek"
    request:
      path: "/thread/${threadId}"
      method: GET
      headers:
        x-api-key: "${apiKey}"
    response:
      code: 200
      body:
        jsonSchema:
          type: object
          properties:
            id:
              type: string
              value: "${threadId}"
            title:
              type: string
              value: "Nowy wątek"
            content:
              type: string
              value: "Cześć. To jest test. Lorem ipsum."
```

Pierwszą interakcją jest wykonanie zapytania do endpointu “/thread” methodą POST w celu utworzenia nowego wątku. W celu autoryzacji wykorzystywany jest tutaj klucz API który jest przekazywany za pomocą nagłówka “x-api-key”. W zawartości (body) zapytania znajduje się dokument JSON o określonej wartości “title” i “content”. Oczekiwaną odpowiedzią jest kod statusu “201” oraz dokument JSON zawierający w nadrzędnym obiekcie właściwość “id” typu “string”, którego wartość jest zapisywana do zmiennej “threadId”, potrzebnego do następnej interakcji z API w tym teście.

Drugą i ostatnią interakcją jest wykonanie zapytania do "/thread/${threadId}", gdzie część "${threadId}" zostaje zamieniona wartością wyciągniętą z poprzedniego zapytania tego samego testu. Ma to na celu uzyskanie z serwera zasobu dopiero co utworzonego wątku i sprawdzeniu wartości czy są takie same jak przy jego tworzeniu.

4.6. Analiza wybranych fragmentów kodu źródłowego

W niniejszym podrozdziale dokonano analizy wybranych fragmentów kodu źródłowego, które ilustrują kluczowe aspekty funkcjonalności omawianego systemu.

4.6.1. Ładowanie pliku scenariusza testowego do pamięci programu

Aby móc wykonywać testy zdefiniowane w pliku YAML scenariusza testowego, najpierw trzeba odczytać ten plik i załadować wszystkie jego dane do pamięci programu w celu możliwości operowania na nich. Odpowiedzialna za to jest klasa TestSuiteLoader, której funkcja "load_from_file" przedstawiona w listingu 4.3 odczytuje ze ścieżki plik i próbuje odczytać jej strukturę.

Listing 4.3. Definicja funkcji TestSuiteLoader

```
class TestSuiteLoader {
public:
    static TestSuite load_from_file(const std::string& file_path);

private:
    static TestSuite parse_test_suite(const YAML::Node &node);
    static Test parse_test(const YAML::Node &node);
    static Test::HttpRequest parse_http_request(const YAML::Node &node);
    static Test::RequestInstructions parse_request_instructions(const
YAML::Node &node);
    static Test::ResponseInstructions parse_response_instructions(const
YAML::Node &node);
    static Test::CookieSetCheck parse_cookie_set_check(const YAML::Node
&node);
    static TextValueCheck parse_text_value_check(const YAML::Node &node);
    static FileValueCheck parse_file_value_check(const YAML::Node &node);
    static JsonValueCheck parse_json_value_check(const YAML::Node &node);
    static std::unordered_map<std::string, std::string>
parse_variables(const YAML::Node &node);

    template<typename T>
    static T get_optional(const YAML::Node& node, const std::string& key,
T default_value);
};
```

Odczytywanie poszczególnych struktur scenariusza testowego z pliku, takich jak na przykład poszczególne testy, dane do wysłania, spodziewane dane otrzymane z serwera, itp. zostało podzielone na poszczególne funkcje.

Listing 4.4. Logika parsowania wartości z pliku scenariusza testowego

```
TestSuite TestSuiteLoader::load_from_file(const std::string& file_path) {
    YAML::Node root = YAML::LoadFile(file_path);
    if (!root["testSuite"] || !root["testSuite"].IsMap()) {
        std::cout << root.Mark().line << ":" << root.Mark().column <<
"\n";

        throw std::runtime_error("Parsing error: `testSuite` map element
has not been found in the file.");
    }

    return parse_test_suite(root["testSuite"]);
}

TestSuite TestSuiteLoader::parse_test_suite(const YAML::Node &node) {
    TestSuite test_suite;

    test_suite.name = get_optional<std::string>(node, "name", "");

    auto server_url = node["serverUrl"].as<std::string>();
    // remove any trailing slashes from server url
    server_url.erase(server_url.find_last_not_of('/') + 1);
    test_suite.server_url = std::move(server_url);

    test_suite.timeout_secs = get_optional<long>(node, "timeoutSecs",
120L);
    test_suite.abort_on_fail = get_optional<bool>(node, "abortOnFail",
false);
    test_suite.abort_on_error = get_optional<bool>(node, "abortOnError",
false);

    if (node["variables"] && node["variables"].IsMap()) {
        test_suite.global_variables = parse_variables(node["variables"]);
    }

    if (node["setUp"] && node["setUp"].IsMap()) {
        test_suite.set_up = parse_test(node["setUp"]);
    }
    if (node["tearDown"] && node["tearDown"].IsMap()) {
        test_suite.tear_down = parse_test(node["tearDown"]);
    }

    if (!node["tests"] || !node["tests"].IsMap()) {
        throw std::runtime_error("Parsing error: could not find `tests`
```

```
list element in test suite.");
    }
    for (auto it : node["tests"]) {
        auto testName = it.first.as<std::string>();
        test_suite.tests.emplace(testName, parse_test(it.second));
    }

    return test_suite;
}

Test TestSuiteLoader::parse_test(const YAML::Node &node) {
    if (!node || !node.IsMap()) {
        throw std::runtime_error("Parsing error: one of the `test`s is of
incorrect type");
    }
    Test t{};

    t.description = get_optional<std::string>(node, "description", "");
    t.skip = get_optional<bool>(node, "skip", false);
    t.skip_set_up = get_optional<bool>(node, "skipSetUp", false);
    t.skip_tear_down = get_optional<bool>(node, "skipTearDown", false);

    if (!node["requests"] || !node["requests"].IsSequence()) {
        throw std::runtime_error("TestSuiteLoader: could not find list of
`requests` in a test");
    }
    for (const auto& r : node["requests"]) {
        t.requests.push_back(parse_http_request(r));
    }

    return t;
}
// ...
```

Funkcja “load_from_file” jest publicznym interfejsem klasy TestSuiteLoader i jest wykorzystywana do odczytu pliku scenariusza testowego. Warto zwrócić uwagę na warunki które sprawdzają czy dana wartość w pliku istnieje. W przypadku opcjonalnych zmiennych, jeżeli brak, nic się nie dzieje - konstruktor struktury Test już zawiera domyślne wartości. W przypadku wymaganych zmiennych, funkcja zostaje przerwana i wyrzuca wyjątek.

4.6.2. Przeprowadzanie scenariusza testowego

Listing 4.5. przedstawia logikę odpowiedzialną za uruchamianie i przeprowadzanie scenariusza testowego i znajdujących się w nich testów.

Listing 4.5. Logika przeprowadzania scenariusza testowego i jego testów

```
TestResults TestSuite::run_test(Test& test, HttpClient &client_session,
std::unordered_map<std::string, std::string> vars_store) const {
    TestResults test_result;

    for (auto &r : test.requests) {
        HttpClient::HttpResponse request_result;

        // zastąp wartości zdefiniowanymi zmiennymi
        auto url = var_substitution(this->server_url + "/" +
r.request.path, vars_store);
        for (auto &header_value: r.request.headers | std::views::values)
{
            header_value = var_substitution(header_value, vars_store);
        }
        if (r.request.body) {
            auto rq =
var_substitution(std::string(r.request.body.value().begin(),
r.request.body.value().end()), vars_store);
            r.request.body = std::vector<char>(rq.begin(), rq.end());
        }

        try {
            test_result.number_of_requests_made += 1;
            std::chrono::steady_clock::time_point request_start_time
                = std::chrono::steady_clock::now();

            request_result = client_session.sendRequest(
                r.request.method,
                url,
                r.request.headers,
                r.request.body,
                timeout_secs
                );

            test_result.requests_runtime_ms +=
            std::chrono::duration_cast<std::chrono::milliseconds
            (std::chrono::steady_clock::now() -
request_start_time).count();
        } catch (const std::runtime_error &e) {
            // błąd wykonywania zapytania.
            test_result.result = TEST_ERROR;
            test_result.failure_reasons.emplace_back(e.what());

            // Nie rób kolejnych interakcji jeżeli wykryto błąd
            return test_result;
        }
```

```
        // sprawdzanie statusu kodu odpowiedzi
        if (r.response.status_code != HTTP_RESPONSE_CODE_UNSET &&
request_result.httpCode != r.response.status_code) {
            test_result.failure_reasons.push_back("Bad status code: got "
+ std::to_string(request_result.httpCode) + ", but expected " +
std::to_string(r.response.status_code));
        }

        // sprawdzanie nagłówków
        for (const auto &[h, value_check] : r.response.headers) {
            // ...
            if (header == request_result.headers.cend()) {
                test_result.failure_reasons.push_back(
                    "Header name \"" + header_name + "\" specified in the
test suite, but not found in the response");
            } else {
                value_check.check_value(header->second,
test_result.failure_reasons, vars_store);
            }
        }

        // ...

        // Sprawdź zawartość body odpowiedzi
        if (r.response.body) {
            r.response.body->check_value(request_result.body,
test_result.failure_reasons, vars_store);
        }

        if (!test_result.failure_reasons.empty()) {
            test_result.result = TEST_FAILED;
            // Nie rób kolejnych interakcji jeżeli znaleziono błędy
            break;
        } else {
            test_result.result = TEST_PASSED;
        }
    }

    return test_result;
}
```

4.6.3. Klient HTTP

Za wysyłanie i odbieranie zapytań HTTP odpowiada klasa **HttpClient**. Jest to interfejs do obsługi zapytań HTTP, które oddziela logikę biznesową od technicznych szczegółów przesyłania danych.

Kluczową zmienną tego obiektu jest instancja klasy **HttpTransport**, która obsługuje protokół HTTP. To znaczy, że podczas normalnego działania aplikacji, klasa ta jest zaimplementowana z wykorzystaniem biblioteki *libcurl*, aby wykonywać zapytania przez sieć, ale podczas testów jest używany *mock*, czyli obiekt, którego używa się zamiast rzeczywistej implementacji w trakcie testów jednostkowych - nie wykonuje on prawdziwych zapytań, a jedynie na sztywno ustalone odpowiedzi, aby test nie musiał zależeć od dostępności zewnętrznego serwera.

Listing 4.6. Definicja klasy HttpClient

```
#ifndef APITESTER_HTTPCLIENT_H
#define APITESTER_HTTPCLIENT_H

#include "HttpTransport.h"
#include <memory>

class HttpClient {
public:
    using HttpCookie = HttpTransport::HttpCookie;
    using HttpResponse = HttpTransport::HttpResponse;

    explicit HttpClient(std::unique_ptr<HttpTransport> transport =
nullptr);
    ~HttpClient();

    HttpResponse sendRequest(
        const std::string& method,
        const std::string& url,
        const std::vector<std::pair<std::string, std::string>>& headers,
        std::optional<std::reference_wrapper<const std::vector<char>>>
body = std::nullopt,
        long timeout_secs = 120
    );

    void abort();

private:
    std::unique_ptr<HttpTransport> transport_;
};

#endif //APITESTER_HTTPCLIENT_H
```

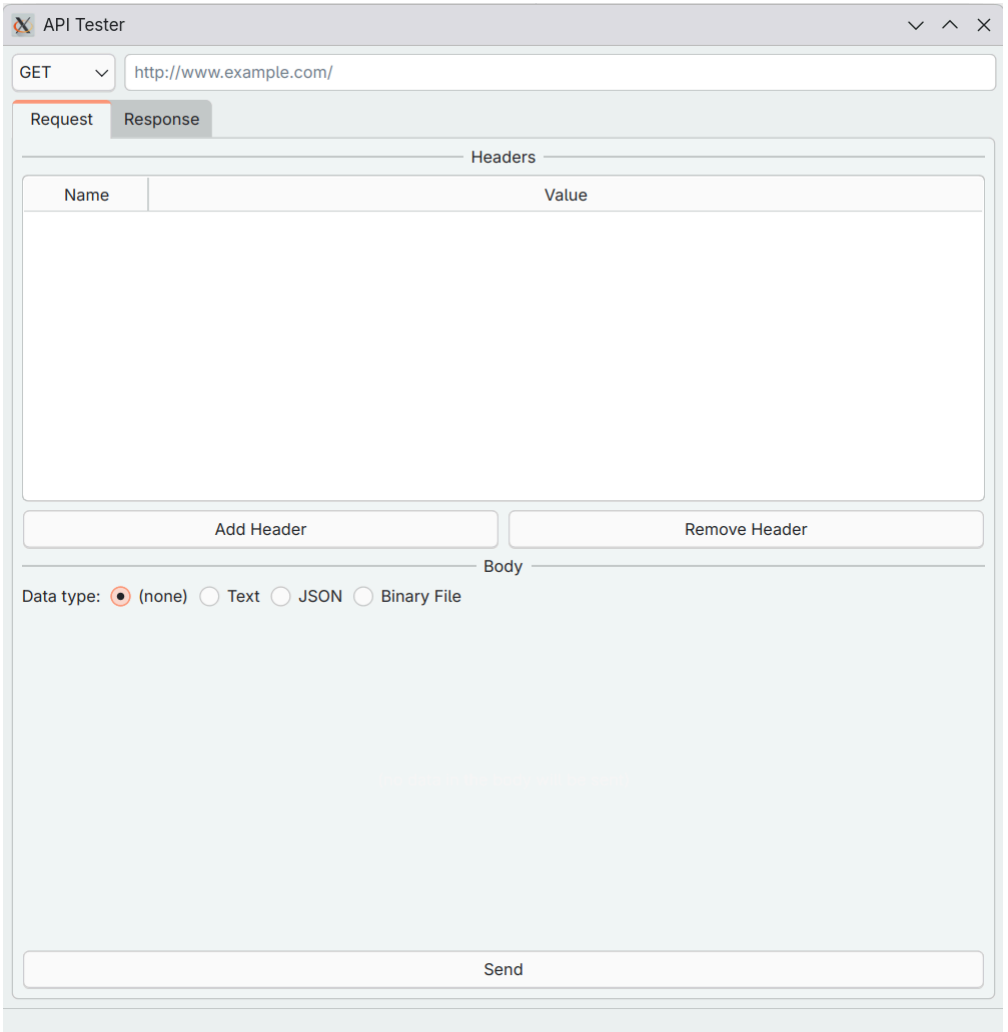
5. Interfejsy użytkownika

Aplikacja składa się z dwóch trybów: jeden przedstawia interfejs graficzny, w którym można wykonywać interakcje z testowanym API aplikacji webowej, dzięki czemu można uzupełniać scenariusz testu o potrzebne dane. Drugi tryb jest uruchamiany w linii poleceń, podając ścieżkę do pliku scenariusza testowego, zostaje przeprowadzony zautomatyzowany proces testowania na podstawie instrukcji z pliku.

5.1. Interfejs graficzny do wykonywania interakcji z API

Ze względu na to, że jest to aplikacja wieloplatformowa, mogą występować drobne różnice w wyglądzie interfejsu, jednakże układ kontrolek pozostaje taki sam. Poniższe zrzuty ekranu zostały wykonane w systemie Linux.

Po uruchomieniu programu, użytkownikowi zostaje przedstawione to okno:

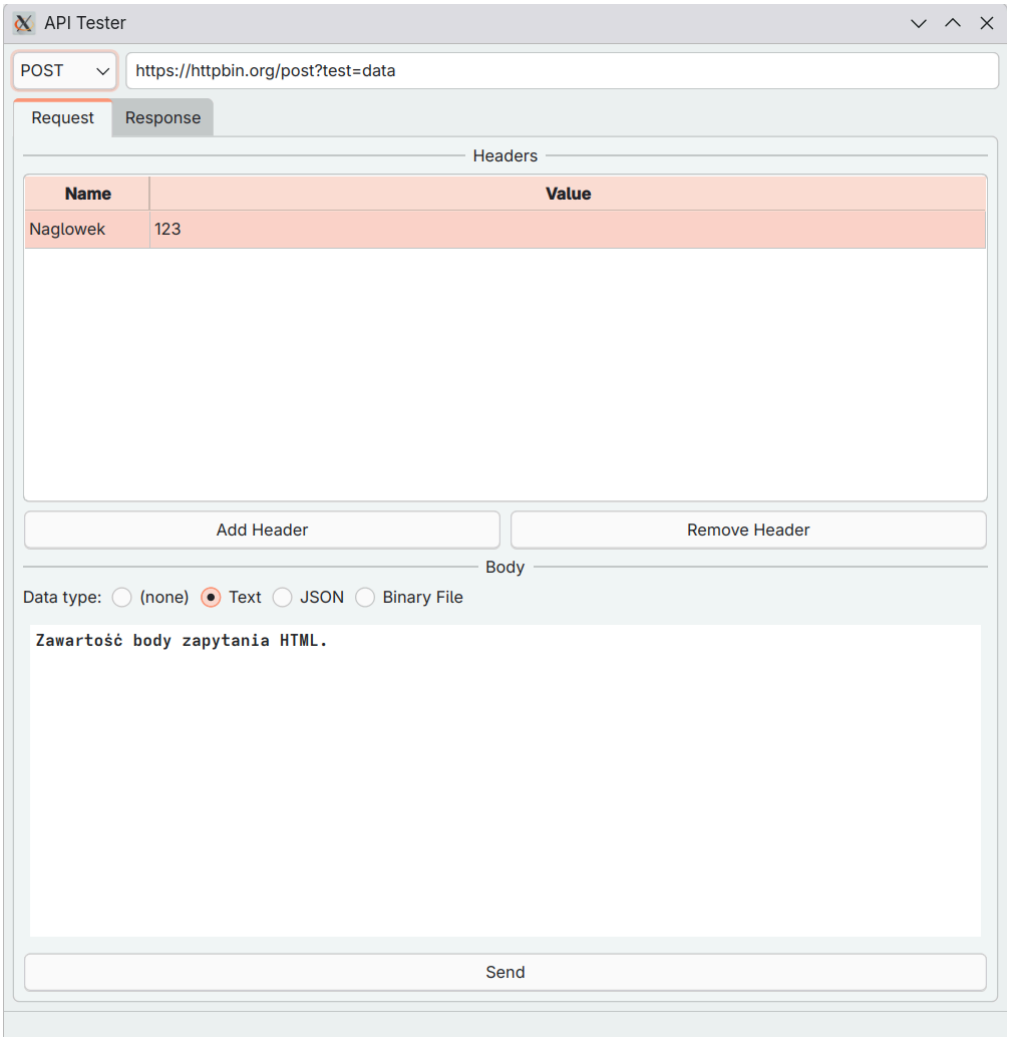


Rys 5.1. Graficzny interfejs użytkownika aplikacji

W górnej części użytkownik może z listy wybieranej dobrać metodę zapytania HTTP. Możliwe wartości to: “GET”, “POST”, “PATCH”, “PUT”, “DELETE” oraz “HEAD”. Obok znajduje się pole tekstowe do wprowadzenia adresu URL do punktu końcowego testowanego API.

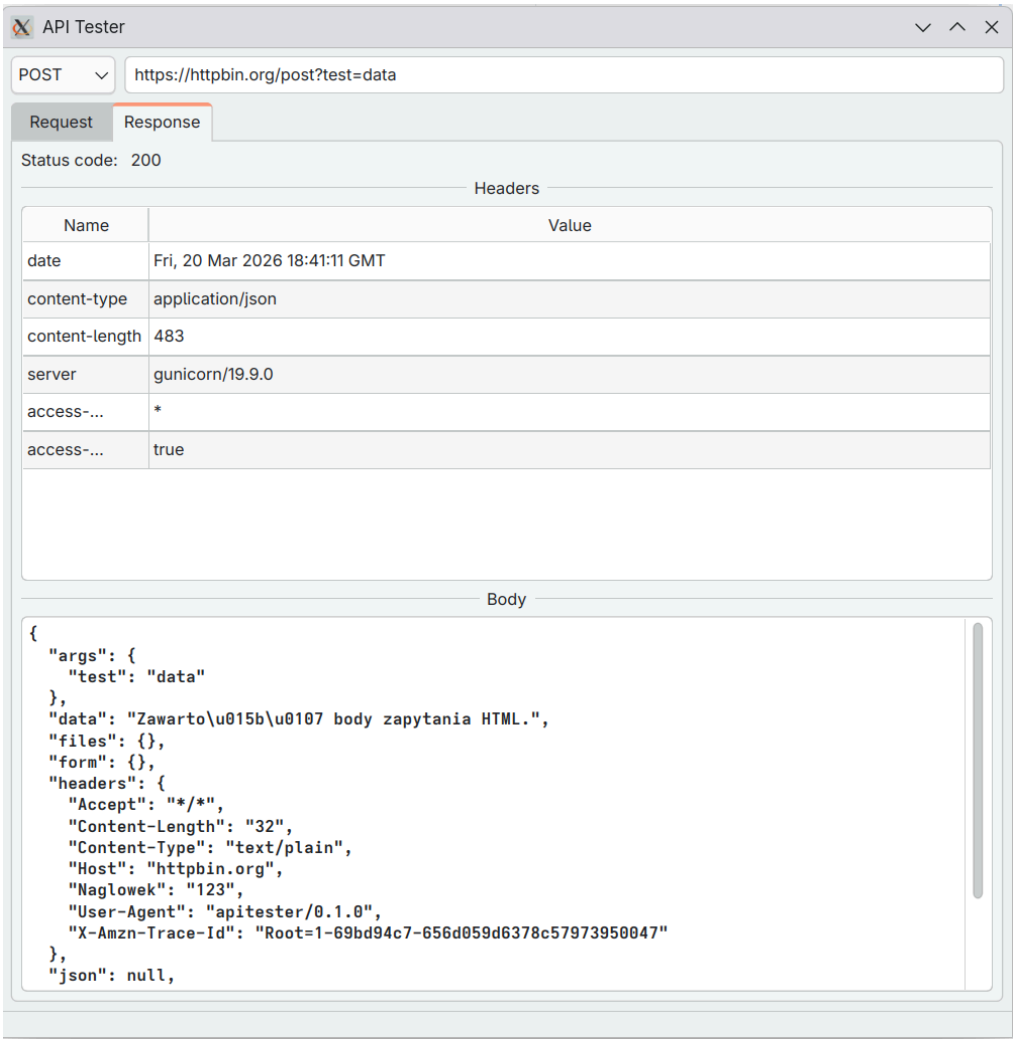
Zaraz poniżej znajduje się tabela do ustawiania nagłówka zapytania. Po kliknięciu przycisku “Add Header” zostanie utworzone nowe pole, w którym tester może uzupełnić nazwę oraz wartość nagłówka.

Pod nagłówkami znajduje się miejsce do ustawienia treści zapytania (body). Dostępne są do wyboru rodzaje formatu takiej treści. W zależności od wybranej opcji, poniżej znajdzie się miejsce do uzupełnienia treści (np. pole tekstowe lub przycisk do wybrania pliku).



Rys 5.2. Dane do wykonania zapytania HTTP.

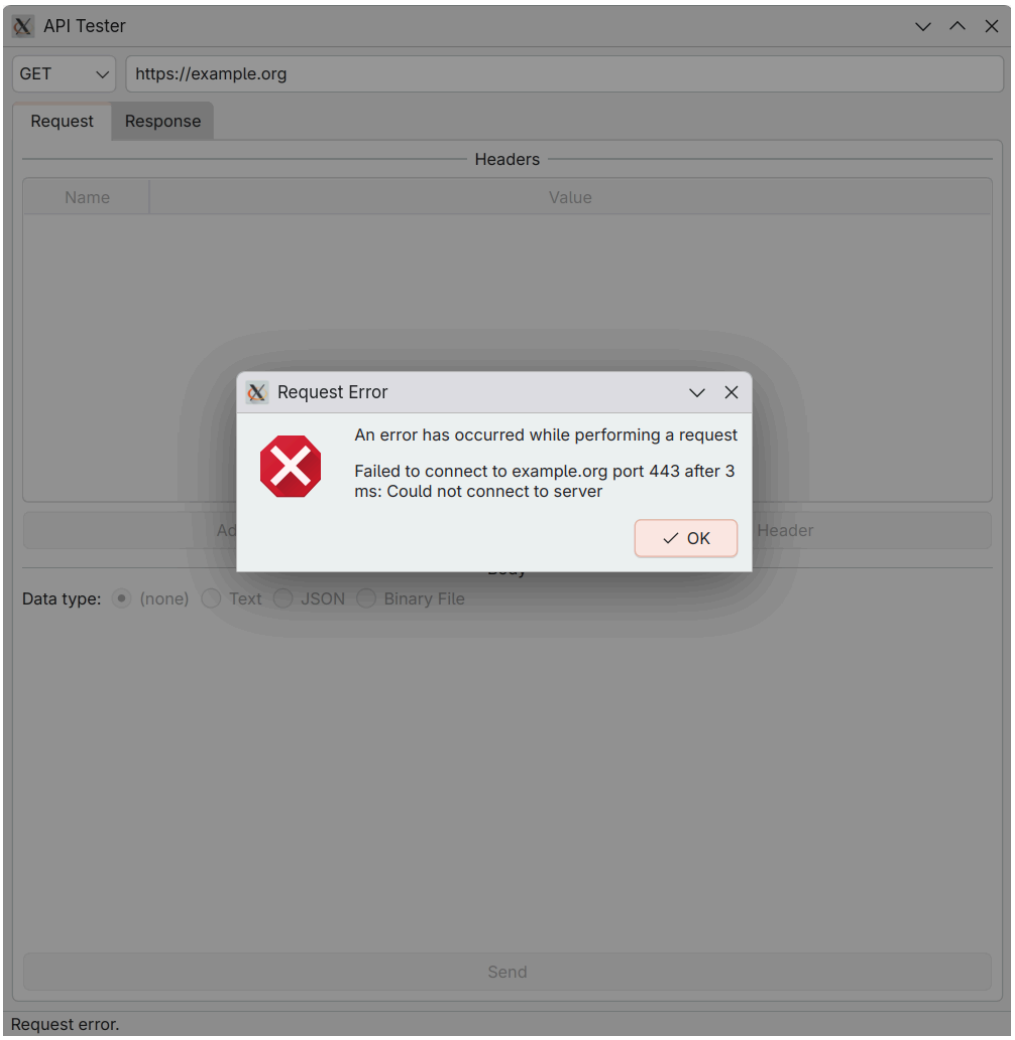
Po kliknięciu przycisku “Send”, aplikacja wysyła zapytanie do serwera pod którym jest ustawiony adres URL. Jeżeli otrzymamy odpowiedź od serwera. to aplikacja przejdzie do zakładki “Response” i wyświetli otrzymane dane.



Rys 5.3. Dane otrzymane po wykonaniu zapytania HTTP.

Po kolei znajduje się: status odpowiedzi (w tym przypadku 200), otrzymane nagłówki oraz treść (body) z serwera.

Jednakże, gdyby z jakiegoś powodu wysłanie zapytania zakończyłoby się niepowodzeniem (na przykład z powodu problemów z połączeniem z siecią), to zostanie wyświetlony komunikat z błędem:



Rys 5.4. Błąd wykonywania zapytania.

5.2. Uruchamianie scenariuszy testowych w linii poleceń

Uruchamianie scenariusza testów oraz wyświetlanie wyników odbywa się w interfejsie linii poleceń (z ang. command-line interface). Dzięki temu interakcja może zostać uruchomiona za pomocą skryptów powłokowych a także w zautomatyzowanych systemach CI/CD, gdzie interakcja z interfejsem graficznym nie byłaby możliwa.

Do uruchomienia programu w tym trybie należy wprowadzić w wierszu polecenia następującą komendę: **ścieżka/do/programu/apitester ścieżka/do/pliku/testowego.yaml**. Na przykład: **./apitester /home/user/testsuite.yaml** uruchomi scenariusz testowy który znajduje się w pliku **/home/user/testsuite.yaml**.

Po wykonaniu wszystkich testów z scenariusza, program wydrukuje w linii poleceń raport z wynikami, oraz zakończy działanie z kodem **0** jeżeli nie znaleziono żadnych nieprawidłowości, bądź **1** jeżeli wystąpiły jakiekolwiek błędy.

```
~ $ apitester /tmp/apitester/testsuite.yml

================================================================================
No.    Test name                      Status     Req. runtime Req. sent
--------------------------------------------------------------------------------
[   1] createPost                     PASSED          826ms          2
[   2] getRecentPosts                 PASSED           35ms          1
[   3] getUserInfo                    PASSED           33ms          1
[   4] likePost                       PASSED          356ms          2
================================================================================
SUMMARY: 4 of 4 tests passed.
```

Rys 5.5. Raport wykonania scenariusza testowego.

- **No.** jest to numer kolejnościowy testu, zaczynając od 1.
- **Test name** jest to nazwa przeprowadzonego testu.
- **Status** oznacza rezultat testu. Możliwe są:
    - **PASSED**: To oznacza, że test przebiegł pomyślnie.
    - **FAILED**: Występuje kiedy test zakończy się niepowodzeniem z powodu otrzymania nieprawidłowych danych, innych od spodziewanych.
    - **ERROR**: Występuje, kiedy podczas testowania program napotka błąd w wykonaniu interakcji z API (np. brak połączenia z serwerem).
    - **SKIPPED**: Występuje wtedy, kiedy test zostanie pominięty, ponieważ wartość **skipped** jest ustawiona na **true**.
    - **N/A**: Oznacza to, że do tego testu nigdy nie doszło. Może to się stać na przykład wtedy, kiedy wartość **abortOnError** lub **abortOnFailure** scenariusza testowego jest ustawiona na **true** i wcześniejszy test zakończył się niepowodzeniem.
- Wartość pod **Req. runtime** oznacza czas w milisekundach oznaczający czas poświęcony na wykonanie zapytania i oczekiwania na odpowiedź. Czas sprawdzania wartości w ramach testu nie jest wliczony.
- **Req. sent** jest to ilość wykonanych interakcji w ramach jednego testu. Przydatne, kiedy w ramach jednego testu jest więcej niż jedno zapytanie, więc można sprawdzić w którym zapytaniu wystąpiło niepowodzenie.

W przypadku niepomyślnego testu lub wystąpienia błędu, pod problematycznym testem w raporcie zostanie wyświetlony komunikat co poszło nie tak, przedstawiony na rysunkach 5.6 i 5.7.

```
~ $ apitester /tmp/apitester/testsuite2.yml

================================================================================
No.    Test name                     Status     Req. runtime Req. sent
--------------------------------------------------------------------------------
[   1] createPost                    FAILED          547ms          2
       └─ JSON validation error in /cookies/post: Expected string value "defdefdef", but got "abcabcabc"
================================================================================
SUMMARY: 0 of 1 tests passed.
```

Rys 5.6. Raport testu zakończonego niepowodzeniem.

```
~ $ apitester /tmp/apitester/testsuite2.yml

================================================================================
No.    Test name                     Status     Req. runtime Req. sent
--------------------------------------------------------------------------------
[   1] createPost                    ERROR              0ms          1
       └─ Failed to connect to example.org port 80 after 4 ms: Could not connect to server
================================================================================
SUMMARY: 0 of 1 tests passed.
```

Rys 5.7. Raport testu zakończonego błędem połączenia.

6. Testy

Napisanie oraz wykonanie testów na kodzie źródłowym projektu którego zadaniem jest testowanie aplikacji, jest koniecznym procesem do poprawnego działania oraz gwarancji poprawnych wyników testowania. Niedopuszczalnym byłoby doprowadzić do poważnego błędu w programie, który mógłby zwracać nieprawidłowe wyniki testów.

Jako że kod źródłowy programu został napisany w języku programowania C++, zostanie użyty tutaj framework GoogleTest, który jest de facto standardem w tego typu projektach.

6.1. Testy jednostkowe klas sprawdzających wartości JSON

Te testy mają za zadanie sprawdzić poprawność działania mechanizmu sprawdzania struktury i wartości danych w formacie JSON.

Listing 6.1 przedstawia wybrane testy, w rzeczywistości jest ich kilkadziesiąt, każdy testujący inną funkcjonalność tego systemu (np. porównanie różnego rodzaju typów, lub "ograniczeń").

Listing 6.1. Testy jednostkowe JsonSchema

```
TEST_F(JsonSchemaTest, StringSchemaBasicValidation) {
    StringSchema schema;
    std::vector<ValidationError> errors;
    ValidationPath path;

    // typu string, prawidłowo
    auto valid_json = R"("hello")"_json;
    NlohmannJsonValue value(valid_json);

    schema.validate(value, errors, path, empty_vars);
    EXPECT_TRUE(errors.empty());
}

TEST_F(JsonSchemaTest, StringSchemaRejectsNumber) {
    StringSchema schema;
    std::vector<ValidationError> errors;
    ValidationPath path;

    // spodziewano się string, a otrzymano number
    auto invalid_json = R"(42)"_json;
    NlohmannJsonValue value(invalid_json);

    schema.validate(value, errors, path, empty_vars);
```

```
    // więc powinny być zapisane błędy walidacji
    EXPECT_FALSE(errors.empty());
}

TEST_F(JsonSchemaTest, ArrayOfObjectsSchema) {
    // utwórz schemat dla listy obiektów użytkowników:
    // [ { name: string, age: number } ]
    auto user_schema = std::make_unique<ObjectSchema>();
    user_schema->addProperty("name", std::make_unique<StringSchema>());
    user_schema->addProperty("age", std::make_unique<NumberSchema>());

    auto array_schema =
std::make_unique<ArraySchema>(std::move(user_schema));

    std::vector<ValidationError> errors;
    ValidationPath path;

    auto json = R"([{"name": "Alice", "age":  25}, {"name": "Bob", "age":
23}])"_json;
    NlohmannJsonValue value(json);

    array_schema->validate(value, errors, path, empty_vars);
    EXPECT_TRUE(errors.empty());
}
```

6.2. Testy klienta HTTP

Testowanie takich części systemu jak klient HTTP lub każdej innej części korzystającej z niej rodzi pewne wyzwanie, ponieważ zamiast wysyłać zapytania do zdalnego serwera, lub tworzyć nowy serwer tylko na potrzeby testu, można zamiast tego napisać tzw. “mocka”, który zamiast wykonywać prawdziwe połączenia HTTP, zwraca ustalone uprzednio przez testera dane.

Listing 6.2. Testy HttpClient z wykorzystaniem mocka HttpTransport

```
class MockHttpTransport : public HttpTransport {
public:
    HttpResponse sendRequest(
        const std::string& method,
        const std::string& url,
        const std::vector<std::pair<std::string, std::string>>& headers,
        std::optional<std::reference_wrapper<const std::vector<char>>>
body = std::nullopt,
        long timeout_secs = 120
    ) override {
        return sendRequest_impl();
    }
```

```
    MOCK_METHOD(HttpResponse, sendRequest_impl, ());
    MOCK_METHOD(void, abort, (), (override));

private:
    HttpResponse default_response_;
};

HttpTransport::HttpResponse create_ok_response() {
    HttpTransport::HttpResponse response;
    response.httpCode = 200;
    response.headers.emplace_back("Content-Type", "application/json");
    response.body = {'o', 'k'};
    return response;
}

class HttpClientWithMockTest : public ::testing::Test {
protected:
    void SetUp() override {
        mock_transport_ = std::make_unique<MockHttpTransport>();
        mock_transport_ptr_ = mock_transport_.get();
    }

    std::unique_ptr<MockHttpTransport> mock_transport_;
    MockHttpTransport* mock_transport_ptr_;
};


TEST_F(HttpClientWithMockTest, SendRequestDelegates) {
    HttpTransport::HttpResponse mock_response = create_ok_response();

    EXPECT_CALL(*mock_transport_ptr_, sendRequest_impl)
        .Times(1)
        .WillOnce(::testing::Return(mock_response));

    HttpClient client(std::move(mock_transport_));
    auto response = client.sendRequest(
        "GET",
        "http://api.example.com/users",
        {}
    );

    EXPECT_EQ(response.httpCode, 200);
    EXPECT_EQ(response.headers[0].first, "Content-Type");
    EXPECT_EQ(response.headers[0].second, "application/json");
    EXPECT_EQ(response.body.size(), 2);
}
TEST_F(HttpClientWithMockTest, MultipleRequestsWithDifferentResponses) {
```

```
    HttpTransport::HttpResponse response1;
    response1.httpCode = 201;
    std::string s1 = "created";
    response1.body = std::vector<char>(s1.begin(), s1.end());

    HttpTransport::HttpResponse response2;
    response2.httpCode = 404;
    std::string s2 = "not found";
    response2.body = std::vector<char>(s2.begin(), s2.end());

    EXPECT_CALL(*mock_transport_ptr_, sendRequest_impl)
        .WillOnce(::testing::Return(response1))
        .WillOnce(::testing::Return(response2));

    HttpClient client(std::move(mock_transport_));

    auto resp1 = client.sendRequest("POST",
"http://api.example.com/users", {});
    EXPECT_EQ(resp1.httpCode, 201);

    auto resp2 = client.sendRequest("GET",
"http://api.example.com/users/999", {});
    EXPECT_EQ(resp2.httpCode, 404);
}
```

6.3. Weryfikacja poprawności testów

Wszystkie wykonane testy przeszły bez błędów.

```
[ RUN      ] JsonSchemaTest.ObjectSchemaRejectsArray
[       OK ] JsonSchemaTest.ObjectSchemaRejectsArray (0 ms)
[ RUN      ] JsonSchemaTest.ArraySchemaOfStrings
[       OK ] JsonSchemaTest.ArraySchemaOfStrings (0 ms)
[ RUN      ] JsonSchemaTest.ArraySchemaOfNumbers
[       OK ] JsonSchemaTest.ArraySchemaOfNumbers (0 ms)
[ RUN      ] JsonSchemaTest.ArraySchemaRejectsObject
[       OK ] JsonSchemaTest.ArraySchemaRejectsObject (0 ms)
[ RUN      ] JsonSchemaTest.NestedObjectSchema
[       OK ] JsonSchemaTest.NestedObjectSchema (0 ms)
[ RUN      ] JsonSchemaTest.ArrayOfObjectsSchema
[       OK ] JsonSchemaTest.ArrayOfObjectsSchema (0 ms)
[ RUN      ] JsonSchemaTest.NumberConstraintRange
[       OK ] JsonSchemaTest.NumberConstraintRange (0 ms)
[ RUN      ] JsonSchemaTest.ErrorContainsPath
[       OK ] JsonSchemaTest.ErrorContainsPath (0 ms)
[----------] 24 tests from JsonSchemaTest (3 ms total)

[----------] Global test environment tear-down
[==========] 81 tests from 5 test suites ran. (75 ms total)
[  PASSED  ] 81 tests.

~/CLionProjects/apitester/cmake-build-debug/tests $
```

Rys 6.1. Raport z wynikami testów projektu.

7. Podsumowanie i wnioski

W ramach niniejszej pracy inżynierskiej udało się zgodnie z założeniami zaimplementować projekt. Opracowane narzędzie pozwala na automatyzację testowania poprawności działania interfejsów API aplikacji webowych.

Do implementacji wykorzystano bibliotekę *libcurl* do wysyłania zapytań HTTP, framework Qt do stworzenia interfejsu graficznego ułatwiającego wykonywanie interakcji z API, wszystko napisane w języku C++ w standardzie C++20. Scenariusze testowe są tworzone i przechowywane w czytelnym formacie YAML.

W ramach dalszego rozwoju programu, można zaimplementować możliwości sprawdzania innych typów danych często stosowanych w aplikacjach internetowych, jak na przykład XML lub dane wbudowane w sam dokument HTML. Innym pomysłem jest umożliwienie interakcji z bazą danych testowanej aplikacji webowej, gdzie po wykonaniu interakcji z interfejsem, móc sprawdzać czy wysłana zawartość została zapisana w bazie danych.

Podsumowując, zrealizowany program spełnia wszystkie założenia oraz wymagania i jest gotowa do wdrożenia na produkcję. Realizacja tego projektu znacząco pogłębiła moją wiedzę w zakresie tworzenia narzędzia mogącego stanowić krytyczną część w procesie wytwarzania i testowania oprogramowania.

Bibliografia

[1] Bjarne Stroustrup, *Język C++. Kompendium wiedzy. Wydanie IV*, tłum. Łukasz Piwko, Helion, 2014

[2] Gayathri Mohan, *Testowanie full stack. Praktyczny przewodnik dostarczania oprogramowania wysokiej jakości*, tłum. Radosław Meryk, Helion, 2023

[3] *libcurl* [online] - dostęp 19.01.2026 https://curl.se/libcurl/

[4] *About - YAML Ain't Markup Language* [online] - dostęp 19.01.2026 https://yaml.org/about/

[5] *JSON* [online] - dostęp 19.01.2026 https://www.json.org/json-pl.html

[6] *About Qt* [online] - dostęp 19.01.2026 https://wiki.qt.io/About_Qt

[7] *About CMake* [online] - dostęp 19.01.2026 https://cmake.org/about/

[8] HTTP: Hypertext Transfer Protocol | MDN [online] - dostęp 11.03.2026 https://developer.mozilla.org/en-US/docs/Web/HTTP

Spis rysunków

Spis listingów