SYSTEM AUTOMATYCZNEGO WYKONYWNIA ZADAŃ Z INTEGRACJĄ ZEW. API

Adam Szada-Borzyszkowski 235985

1. **Wstęp**

Celem projektu jest stworzenie aplikacji Todo List, która umożliwia użytkownikom zarządzanie listą zadań. Główne założenia projektu to:

* Organizacja zadań: Aplikacja ma umożliwiać użytkownikom dodawanie nowych zadań, nadawanie im nazw, określanie czasu wykonania oraz oznaczanie ich jako zwykłe zadania lub zadania związane z zewnętrznym API.
* Zarządzanie zadaniami związanych z API: Aplikacja powinna umożliwiać użytkownikom dodawanie zadań, które wymagają wykonania zapytania do zewnętrznego API. Użytkownicy mogą podać adres URL API oraz parametry zapytania. Aplikacja powinna wykonywać te zadania zgodnie z harmonogramem i zapisywać odpowiedzi z API
* Interfejs graficzny: Aplikacja powinna posiadać prosty, intuicyjny interfejs graficzny, który umożliwia łatwe dodawanie, usuwanie i aktualizowanie zadań. Interfejs powinien również wyświetlać listę zadań wraz z ich szczegółami, takimi jak nazwa, czas wykonania, status i informacje dotyczące zewnętrznego API.
* Powiadomienia e-mail: Aplikacja powinna obsługiwać wysyłanie powiadomień e-mail w przypadku zakończenia wykonania zadania związanych z API. Po wykonaniu zadania, aplikacja powinna wysłać wiadomość e-mail z informacjami dotyczącymi zadania i jego statusu.
* Odporność na błędy: Aplikacja powinna być odporna na błędy i obsługiwać sytuacje takie jak błędne żądania, problemy z zewnętrznym API czy błędy połączenia z bazą danych. W przypadku wystąpienia błędów, aplikacja powinna odpowiednio reagować i dostarczać odpowiednie komunikaty błędów.

1. **Wykorzystane technologie**

Projekt wykorzystuje różne technologie i narzędzia do jego implementacji. Oto krótki opis wykorzystanych technologii:

Docker: Projekt korzysta z technologii konteneryzacji Docker, co umożliwia łatwe uruchomienie i zarządzanie różnymi składnikami aplikacji. Istniejące pliki Dockerfile, definiują konfigurację i zależności dla kontenerów.

MongoDB: Baza danych MongoDB jest wykorzystywana w projekcie do przechowywania informacji o zadaniach i odpowiedziach z zewnętrznego API.

Express.js: Aplikacja serwerowa jest oparta na frameworku Express.js, który zapewnia obsługę trasowania, obsługę żądań HTTP i budowę interfejsu API.

Frontend: W folderze app/public znajdują się pliki HTML i CSS, które tworzą prosty interfejs użytkownika. Plik app/public/index.html zawiera strukturę strony, a plik app/public/styles/styles.css definiuje style CSS.

Axios: Do wykonywania zapytań HTTP do zewnętrznego API wykorzystywana jest biblioteka Axios, która umożliwia łatwe tworzenie i obsługę żądań HTTP.

Mongoose: Do komunikacji z bazą danych MongoDB wykorzystywany jest ODM (Object Data Modeling) o nazwie Mongoose. Mongoose pozwala na definiowanie schematów danych i zapewnia łatwe operacje CRUD na kolekcjach MongoDB.

Node.js: Projekt jest oparty na środowisku uruchomieniowym Node.js, które umożliwia wykonywanie kodu JavaScript po stronie serwera.

Python: W folderze python znajduje się skrypt send\_email.py, który jest wywoływany przez aplikację w celu wysyłania powiadomień e-mail. Wykorzystuje on bibliotekę Pythona do wysyłania wiadomości e-mail.

docker-compose: Plik docker-compose.yml definiuje konfigurację i zależności dla kontenerów Docker, co umożliwia łatwe uruchomienie całej aplikacji i jej zależności.

Projekt wykorzystuje te technologie i narzędzia w celu stworzenia kompletnego systemu do zarządzania listą zadań, obsługi zapytań do zewnętrznego API, przechowywania danych w bazie MongoDB oraz wysyłania powiadomień e-mail.

1. **Struktura projektu**

Projekt wykorzystuje 4 kontenery, które wykorzystuje do uruchamiania różnych składników aplikacji, takich jak serwer aplikacyjny oparty na Express.js, baza danych MongoDB, narzędzie do zarządzania bazą danych Mongo Express, oraz środowisko Python do wysyłania powiadomień e-mail.

├── app

│   ├── Dockerfile

│   ├── index.html

│   ├── package-lock.json

│   ├── package.json

│   ├── public

│   │   ├── index.html

│   │   └── styles

│   │   └── styles.css

│   └── todo.js

├── docker-compose.yml

├── mongo

│   ├── Dockerfile

│   └── mongodb.conf

├── mongo-express

│   └── Dockerfile

└── python

├── Dockerfile

├── requirements.txt

└── send\_email.py

**Kontener "app":**

Dockerfile: Plik konfiguracyjny Dockerfile, który definiuje konfigurację kontenera.

index.html: Plik HTML, który reprezentuje strukturę strony internetowej dla interfejsu użytkownika.

package-lock.json: Plik zależności dla aplikacji.

package.json: Plik konfiguracyjny dla aplikacji Node.js.

public: Katalog zawierający pliki publiczne, takie jak pliki CSS, obrazy, etc.

index.html: Plik HTML dla strony głównej.

styles: Katalog zawierający pliki CSS dla stylizacji strony.

styles.css: Plik CSS dla aplikacji.

todo.js: Plik JavaScript zawierający kod aplikacji serwerowej opartej na Express.js.

**Kontener "mongo":**

Dockerfile: Plik konfiguracyjny Dockerfile, który definiuje konfigurację kontenera.

mongodb.conf: Plik konfiguracyjny MongoDB.

**Kontener "mongo-express":**

Dockerfile: Plik konfiguracyjny Dockerfile, który definiuje konfigurację kontenera.

**Kontener "python":**

Dockerfile: Plik konfiguracyjny Dockerfile, który definiuje konfigurację kontenera.

requirements.txt: Plik zawierający zależności Pythona dla aplikacji.

send\_email.py: Skrypt Pythona odpowiedzialny za wysyłanie powiadomień e-mail.

**Plik "docker-compose.yml": Plik konfiguracyjny docker-compose.yml, który definiuje konfigurację wszystkich kontenerów i ich wzajemne zależności.**

Aplikacja serwerowa todo.js udostępnia następującą funkcjonalność:  
  
1) Konfiguracja i Inicjalizacja:

Kod zaczyna się od importowania potrzebnych modułów i tworzenia instancji aplikacji Express.

2) Middleware i Statyczne Pliki:

Używa express.json() jako middleware do analizy danych JSON w żądaniach. Ustawia ścieżkę do folderu publicznego i dostarcza pliki statyczne z tego folderu. Tworzy endpoint /mongo-express, który jest dostępny dzięki modułowi mongo-express, co pozwala na przeglądanie zawartości bazy danych MongoDB.

3) Strona Główna:

Gdy użytkownik odwiedzi główny adres (GET /), serwer wysyła plik HTML strony głównej.

4) Pobieranie Zawartości Folderu:

Gdy użytkownik odwiedzi /folder-content (GET), serwer odczytuje zawartość swojego folderu, w którym jest uruchomiony. Lista plików jest wysyłana w odpowiedzi JSON.

5) Dodawanie Zadań:

Wysyłając żądanie POST do /tasks, można dodać nowe zadanie. Serwer sprawdza, czy wprowadzona data jest przyszła, a następnie zapisuje zadanie w bazie danych. Odpowiednio reaguje na błędy.

6) Pobieranie Zadań:

Gdy użytkownik odwiedzi /tasks (GET), serwer pobiera listę wszystkich zadań z bazy danych. Formatuje je i wysyła w odpowiedzi JSON.

7) Usuwanie Zadania:

Gdy zostanie wysłane żądanie DELETE do /tasks/:id, serwer usuwa zadanie o podanym ID z bazy danych.

8) Aktualizacja Statusu Zadania:

Wysyłając żądanie PATCH do /tasks/:id, można zaktualizować status zadania (np. na "Completed"). Odpowiednio reaguje na błędy.

9) Pobieranie Odpowiedzi z API:

Gdy użytkownik odwiedzi /api-responses/:id (GET), serwer pobiera odpowiedź na żądanie zewnętrznego API, zapisaną w bazie danych.

10) Wykonywanie Zadań API:

Serwer używa biblioteki node-cron do cyklicznego wykonywania funkcji executePendingAPIs() co minutę. Funkcja ta sprawdza, czy są zadania API, które powinny zostać wykonane w danej chwili, i je wykonuje.

11) Wykonywanie Zadania API:

Funkcja executeAPI(url, params) wykonuje żądanie POST do danego URL z określonymi parametrami. Odpowiedź zapisywana jest w bazie danych, a ID zapisanej odpowiedzi jest zwracane.

12) Wysyłanie E-maili:

Funkcja sendEmail(task, status, email) wysyła e-maila z danym zadaniem i statusem. Wywołuje skrypt Pythona (send\_email.py) za pomocą spawn.

13) Połączenie z MongoDB:

Serwer łączy się z bazą danych MongoDB za pomocą mongoose.

14) Schematy i Modele:

Definiowane są schematy i modele dla kolekcji "Todo" oraz "APIResponse" w bazie danych.

15) Uruchamianie Serwera:

Aplikacja nasłuchuje na porcie 3000. Po uruchomieniu serwera, cron uruchamia funkcję executePendingAPIs() cyklicznie.

1. **Wnioski**

Projekt opiera się na wykorzystaniu kontenerów Docker do zapewnienia izolacji i przenośności poszczególnych komponentów. Kontenery pozwalają na łatwą replikację, skalowanie i uruchomienie aplikacji w różnych środowiskach. Dodatkowo, projekt jest zorganizowany wokół mikrousług, gdzie każda funkcjonalność jest umieszczona w osobnym kontenerze. Dzięki temu można niezależnie zarządzać, aktualizować i skalować poszczególne składniki aplikacji.