Helion

<packt>

Akcje GitHuba

Receptury

Praktyczny przewodnik po automatyzacji
i usprawnianiu procesu tworzenia oprogramowania

MICHAEL KAUFMANN

Tytuł oryginału: GitHub Actions Cookbook: A practical guide to automating repetitive tasks and streamlining your development process

Tłumaczenie: Piotr Rakowski

ISBN: 978-83-289-2046-0

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/akgihu
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

Rozdział 3

Tworzenie akcji GitHuba

Teraz, gdy umiesz już tworzyć przepływy pracy i skorzystałeś z niektórych akcji z Marketplace, nadszedł czas, aby w pełni zrozumieć, czym są akcje i jak działają. W tym rozdziale wyjaśnię różne typy akcji i omówię następujące receptury, abyś wiedział, jak samodzielnie pisać akcje:

- Tworzenie akcji kontenera Dockera
- Dodawanie parametrów wyjściowych i korzystanie z podsumowań zadań
- Tworzenie akcji TypeScript
- Tworzenie akcji złożonej
- Udostępnianie akcji na Marketplace
- Najlepsze praktyki dotyczące tworzenia akcji niestandardowych

Dowiesz się, jak przekazywać parametry do akcji i używać parametrów wyjściowych w kolejnych krokach przepływu pracy. Pokażę również, jak zapisywać dane w dzienniku przepływu pracy i dodawać adnotacje do zmian w plikach z poziomu akcji oraz jak tworzyć bogate podsumowania zadań.

Wymagania techniczne

W poniższych recepturach będę używał VS Code. Będziesz potrzebować jego wersji i lokalnego klienta git, aby śledzić postępy.

Jeśli chcesz uruchomić obraz Dockera, który udostępniamy jako akcję lokalnie, będziesz również potrzebować Dockera zainstalowanego na swoim komputerze. Jeśli chcesz, możesz również użyć GitHub Codespaces.

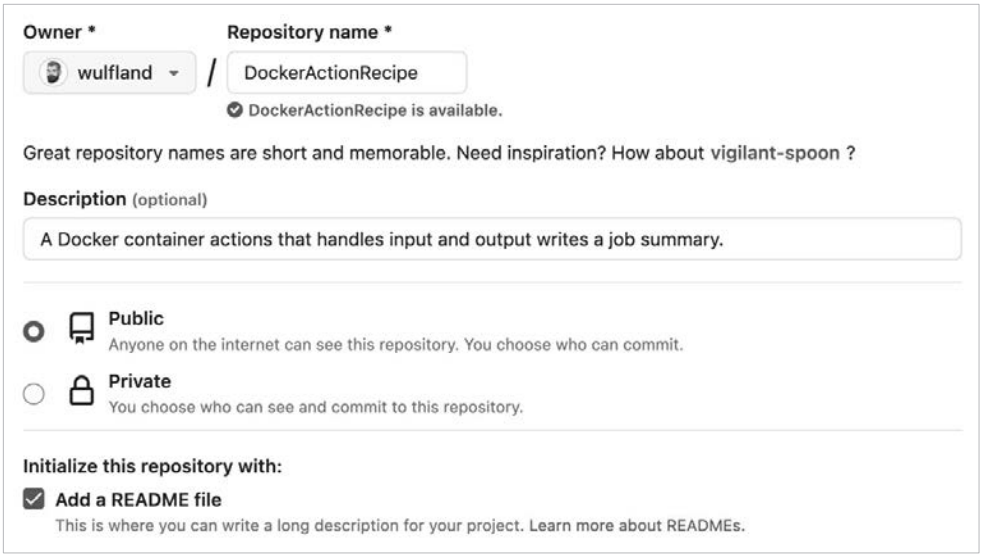
W przypadku akcji TypeScript musisz mieć zainstalowaną w miarę nowoczesną wersję *Node.js*. Jeśli używasz menedżera wersji, takiego jak `nodenv` lub `nvm`, możesz uruchomić `nodenv install` w katalogu głównym repozytorium, aby zainstalować wersję określoną w *package.json*. W przeciwnym razie powinna działać wersja 20.x lub nowsza. Sprawdź plik *README.md* na stronie *https://github.com/actions/typescript-action*, aby uzyskać zaktualizowane wymagania. Jeśli nie chcesz instalować Node.js, po prostu użyj GitHub Codespaces dla tej receptury.

Tworzenie akcji kontenera Dockera

W tej recepturze utworzysz prostą **akcję kontenera Dockera** na podstawie pliku Dockera (ang. *Dockerfile*) i użyjesz jej w przepływie pracy **ciągłej integracji** (**CI**), który będzie uruchamiał akcję z poziomu przepływu pracy za każdym razem, gdy coś zmienisz.

Przygotuj się!

Utwórz nowe repozytorium, o nazwie *DockerActionRecipe*. Uczyń je publicznym, aby nie zużywać żadnych minut akcji, i zainicjalizuj plikiem README (zobacz rysunek 3.1).



Rysunek 3.1. Tworzenie nowego repozytorium dla akcji kontenera Dockera

Sklonuj repozytorium lokalnie i otwórz je w VS Code lub GitHub Codespaces.

Jak to zrobić?

1. Utwórz nowy plik o nazwie *Dockerfile* w katalogu głównym repozytorium. Dodaj następującą zawartość do pliku:

   ```
   # Obraz kontenera, który uruchamia kod
   FROM alpine:latest
   CMD echo "Witaj, świecie"
   ```

   Spowoduje to utworzenie obrazu opartego na najnowszym obrazie Alpine i dodanie warstwy, która wypisuje `"Witaj, świecie"` do konsoli.

2. Uruchom lokalnie kontener Dockera za pomocą następującego polecenia:

   ```
   $ docker run $(docker build -q .)
   ```

Spowoduje to utworzenie obrazu (`docker build`) i uruchomienie go (`docker run`). Powinieneś zobaczyć napis `Witaj, świecie` w konsoli.

3. Dla większej elastyczności przenieśmy kod skryptu do własnego pliku. Utwórz nowy plik o nazwie *entrypoint.sh* i dodaj następującą zawartość:

```
#!/bin/sh -l
echo "Witaj, świecie"
```

4. Teraz dostosuj plik *Dockerfile* tak, aby wykonywał skrypt zamiast bezpośredniego zapisu do konsoli. Skopiuj plik skryptu do katalogu głównego kontenera, a następnie użyj go jako punktu wejścia:

```
FROM alpine:3.10
COPY entrypoint.sh /entrypoint.sh
RUN chmod +x entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
```

Zauważ, że dodałem polecenie `chmod +x entrypoint.sh`, aby skrypt był wykonywalny. W przeciwnym razie próba uruchomienia kontenera lokalnie zakończy się niepowodzeniem i komunikatem `exec: "/entrypoint.sh": permission denied`. We wszystkich systemach uniksowych można po prostu uruchomić `chmod +x entrypoint.sh` lokalnie, a atrybut zostanie dołączony do pliku podczas zatwierdzania na platformie Git. W systemie Windows można użyć Gita, aby ustawić uprawnienia do pliku:

```
$ git add entrypoint.sh
$ git update-index --chmod=+x entrypoint.sh
```

Uruchom ponownie kontener Dockera. Powinieneś ponownie zobaczyć `Witaj, świecie` — tym razem z pliku skryptu:

```
$ docker run $(docker build -q .)
```

5. W akcji zamierzamy wykorzystać parametr wejściowy. Dlatego sparametryzujemy nasz skrypt. Zastąp słowo *świecie* argumentami, które zostały przekazane do Dockera (`$@` dla wszystkich argumentów):

```
#!/bin/sh -l
echo "Witaj, $@"
```

Spróbuj ponownie uruchomić kontener lokalnie i podaj kilka słów. Kontener wydrukuje wynik w następujący sposób:

```
$ docker run $(docker build -q .) foo bar
> Witaj, foo bar
```

6. Następnie dodaj nowy plik do repozytorium o nazwie *action.yml* i dodaj poniższe dane wejściowe:

```
name: 'Receptura akcji Dockera'
description: 'Pozdrów kogoś'
inputs:
     who-to-greet:
       description: 'Kogo pozdrowić'
        required: true
        default: 'świecie'
```

```
runs:
      using: 'docker'
      image: 'Dockerfile'
      args:
        - ${{ inputs.who-to-greet }}
```

7. W tym momencie akcja jest gotowa. Aby ją przetestować, dodamy lokalny plik przepływu pracy o nazwie *.github/workflows/ci.yml*, który będzie uruchamiany przy każdym wypchnięciu. Pobierze on repozytorium i wykona akcję z niestandardowym parametrem wejściowym:

```
name: Akcja CI

on: [push]

jobs:
   ci:
     runs-on: ubuntu-latest
     steps:
         - uses: actions/checkout@v4.1.1
         - name: Uruchom moją własną akcję kontenera
           uses: ./
           with:
               who-to-greet: "@wulfland"
```

Odwoływanie się do działań lokalnych

Zauważ, że odwołujemy się do akcji poprzez lokalną ścieżkę, ./ — to jest powód, dla którego musimy najpierw użyć akcji checkout. Przepływ pracy będzie korzystał z tej samej wersji, na której on działa. Możesz również odwołać się do akcji normalnie, określając `<właściciel>/DockerActionRecipe@main` – w taki sam sposób, w jaki odwołałbyś się do niej z innego repozytorium.

8. Zatwierdź i wypchnij wszystkie zmiany. Wyzwalacz push automatycznie uruchomi przepływ pracy i będziesz mógł sprawdzić wynik akcji. Powinien on wyglądać jak na rysunku 3.2.

   Sprawdź dane wyjściowe akcji za pośrednictwem demona Dockera i parametr, który został przekazany do akcji.

Jak to działa?

Istnieją trzy rodzaje akcji:

- akcje kontenera Dockera,
- akcje JavaScript,
- akcje złożone.

Akcje kontenera Dockera działają tylko w systemie Linux, podczas gdy **akcje JavaScript** i **akcje złożone** (ang. *composite actions*) mogą być używane na dowolnej platformie.

Rysunek 3.2. Wynik akcji w przepływie pracy

Wszystkie akcje są definiowane przez plik o nazwie *action.yml* (lub *action.yaml*), który zawiera metadane definiujące daną akcję. Plik ten nie może mieć innej nazwy, co oznacza, że akcja musi znajdować się we własnym repozytorium lub folderze. Sekcja `run` w pliku *action.yml* określa typ akcji.

Akcje kontenera Dockera zawierają wszystkie swoje zależności w kontenerze i dlatego są bardzo spójne. Pozwalają one na tworzenie akcji w dowolnym języku — jedynym ograniczeniem jest to, że muszą one działać w systemie Linux. Akcje kontenera Dockera są wolniejsze niż akcje JavaScript ze względu na czas potrzebny na pobranie lub zbudowanie obrazu i uruchomienie kontenera.

Akcje kontenera Dockera mogą odwoływać się do obrazu w rejestrze kontenerów, takim jak Docker Hub lub GitHub Packages, albo mogą w czasie wykonywania kodu zbudować plik *Dockerfile*, który dostarczasz razem z innymi plikami akcji. W takim przypadku musisz określić plik *Dockerfile* jako nazwę obrazu w pliku *action.yml*.

Krok dalej

Akcje kontenera są bardzo potężne, ponieważ można je pisać w dowolnym języku. Możesz zwracać parametry wyjściowe do przepływu pracy, zapisywać komunikaty w dzienniku przepływu pracy, dodawać adnotacje do plików w pull requestach i pisać wyczerpujące podsumowania zadań. W następnej krótkiej recepturze dodamy parametry wyjściowe i dopiszemy treść do podsumowania zadania.

Dodawanie parametrów wyjściowych i korzystanie z podsumowań zadań

W tej recepturze dodamy parametr wyjściowy do akcji, który może zostać użyty w kolejnych krokach, a także dopiszemy treść do podsumowania zadania przepływu pracy.

Przygotuj się!

Aby kontynuować tę recepturę, należy zakończyć poprzednią.

Jak to zrobić?

1. Otwórz plik *action.yml* i dodaj poniższy kod tuż pod sekcją `inputs`, ale przed sekcją `runs`:

   ```
   outputs:
      answer:
        description: 'Odpowiedź na wszystko (zawsze 42)'
   ```

   Kod ten definiuje jedno wyjście o identyfikatorze `answer`.

2. Następnie otwórz plik *entrypoint.sh* i dodaj następujący wiersz na końcu pliku:

   ```
   echo "answer=42" >> $GITHUB_OUTPUT
   ```

   Spowoduje to ustawienie wartości wyjściowej dla `answer` na `42`.

3. Teraz dodaj poniższe wiersze na końcu pliku *entrypoint.sh*, aby dopisać trochę kodu w językach Markdown i HTML do podsumowania tego kroku:

   ```
   echo "### Hello $@! :rocket:" >> $GITHUB_STEP_SUMMARY
   echo "<h3> The answer from Deep Thought is 42 :robot:</h3>" >>
   $GITHUB_STEP_SUMMARY
   ```

4. Zanim zatwierdzimy zmiany, musimy dostosować plik przepływu pracy, *.github/workflows/ci.yml*, tak aby używał parametru wyjściowego. W następujący sposób dodaj identyfikator `my-action` do kroku, który wykonuje akcję:

   ```
   - name: Run my own container action
     id: my-action
     uses: ./
     with:
        who-to-greet: "@wulfland
   ```

   Dodaj kolejny krok, który wysyła wynik do dziennika przepływu pracy:

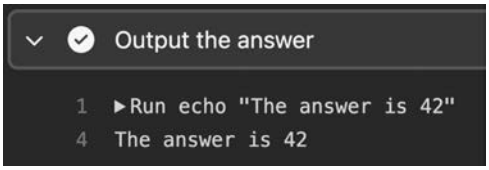
   ```
   - name: Output the answer
     run: echo "The answer is ${{ steps.my-action.outputs.answer }}"
   ```

5. Aby zakończyć kompilację *CI* niepowodzeniem, gdy z akcji kontenera Dockera zostanie zwrócony nieoczekiwany wynik, musimy dodać nowy krok, który zostanie wykonany tylko wtedy, kiedy wynik nie jest oczekiwany. Zwrócenie

niezerowej wartości (na przykład `exit 1`) wskaże przepływowi pracy, że krok się nie powiódł. Możemy użyć adnotacji pliku, aby wskazać, gdzie znajduje się błąd (w taki sam sposób jak w rozdziale 2.):

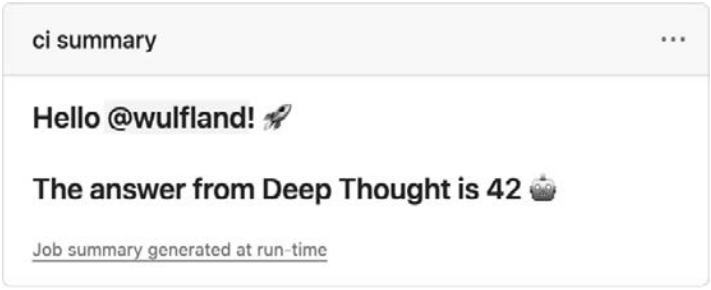
```
- name: Test the container
  if: ${{ steps.my-action.outputs.answer != 42 }}
  run: |
     echo "::error file=entrypoint.sh,line=4,title=Error in container::The
     ↪answer was not expected"
      exit 1
```

6. Zatwierdź i wypchnij wszystkie zmiany. Kompilacja zostanie uruchomiona automatycznie i powinna zakończyć się powodzeniem. Sprawdź dziennik przepływu pracy i upewnij się, że parametr wyjściowy został poprawnie przekazany do kolejnych kroków (zobacz rysunek 3.3).



Rysunek 3.3. Dane wyjściowe akcji kontenera Dockera

Spójrz również na podsumowanie zadania na stronie podsumowania, która wyrenderowała plik Markdown/HTML (zobacz rysunek 3.4).



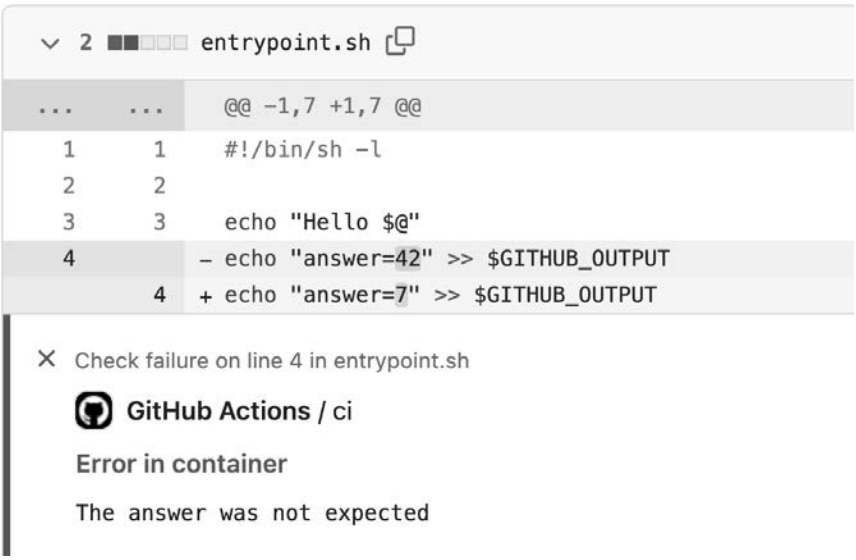
Rysunek 3.4. Podsumowanie zadania na stronie podsumowania przepływu pracy

7. Na koniec chcemy się upewnić, że kompilacja *CI* zakończy się niepowodzeniem, jeśli zwrócona zostanie nieoczekiwana wartość. Otwórz *entrypoint.sh* i zmień `42` na coś innego (na przykład `7`).

   Przełącz się do innej gałęzi, zatwierdź ją i wypchnij, a następnie utwórz nowy pull request:

```
$ git switch -c fail-ci-build
$ git commit -m "Fail CI build"
$ git push -u origin fail-ci-build
$ gh pr create --fill
```

   Sprawdzanie pull requesta zakończy się niepowodzeniem i zostanie dodana adnotacja do odpowiedniego pliku (zobacz rysunek 3.5).



Rysunek 3.5. Kompilacja CI zakończy się niepowodzeniem pull requesta, jeśli dane wyjściowe nie są oczekiwane.

Zwróć uwagę, jak GitHub zaznacza zmianę i dodaje adnotację do konkretnego wiersza.

Jak to działa?

Teraz wyjaśnię, jak działa ten kod.

Pliki środowiskowe

Przekazywanie wartości wyjściowych do kolejnych kroków i zadań działa poprzez potokowanie pary *nazwa* – *wartość* do **pliku środowiskowego**, czyli *$GITHUB_OUTPUT*:

```
echo "{nazwa}={wartość}" >> "$GITHUB_OUTPUT"
```

Operator >> dołącza parę *nazwa* – *wartość* do końca pliku. Ścieżka i nazwa pliku są przechowywane w zmiennej środowiskowej `$GITHUB_OUTPUT`. Dostęp do zmiennej `output` można uzyskać za pomocą właściwości `output` kroku w kontekście `steps`:

```
"${{ steps.<id_kroku>.outputs.<nazwa> }}"
```

Dane wyjściowe są ciągami znaków Unicode, a ich rozmiar nie może przekraczać 1 MB. Suma wszystkich danych wyjściowych w przepływie pracy nie może przekroczyć 50 MB.

Innym przypadkiem użycia plików środowiskowych jest ustawienie zmiennych środowiskowych dla kolejnych kroków w zadaniu. Ścieżka do odpowiedniego pliku środowiskowego jest przechowywana w `$GITHUB_ENV`. Wystarczy dołączyć kolejną parę *nazwa* – *wartość* na końcu pliku, jak widać na poniższym przykładzie:

```
echo "{nazwa_zmiennej_środowiskowej}={wartość}" >> "$GITHUB_ENV"
```

Należy pamiętać, że w nazwie rozróżniana jest wielkość liter! Oto kompletny przykład tego, jak ustawić zmienną środowiskową w jednym kroku i uzyskać do niej dostęp w kolejnym kroku przy użyciu kontekstu `env`:

```
steps:
  - name: Ustaw wartość
    id: step_one
    run: |
      echo "action_state=yellow" >> "$GITHUB_ENV"

  - run: |
      echo "${{ env.action_state }}" # Spowoduje to wyświetlenie wartości 'żółty'
```

Podsumowanie zadań

Dla każdego zadania w przepływie pracy możesz ustawić niestandardowy kod w języku Markdown. Wyrenderowany Markdown będzie wyświetlany na stronie *podsumowania* przepływu pracy. Podsumowania zadań mogą być używane do wyświetlania treści, takich jak wyniki testów lub pokrycia kodu, dzięki czemu osoba przeglądająca wynik przepływu pracy nie musi przechodzić do dzienników lub systemu zewnętrznego.

Podsumowania zadań obsługują kod w języku Markdown w stylu GitHuba. Ale ponieważ język Markdown to HTML, możesz również wyprowadzić HTML do pliku podsumowania zadania. Zauważ, że w tej recepturze moja nazwa użytkownika GitHuba, `@wulfland`, jest linkiem do mojego profilu z podglądem i że wszystkie emotikony GitHuba są obsługiwane.

Dodanie wyników z danego kroku do podsumowania zadania można osiągnąć poprzez dołączenie kodu w języku Markdown do poniższego pliku:

```
echo "{markdown content}" >> $GITHUB_STEP_SUMMARY
```

Kroki są odizolowane i ograniczone do 1 MiB (1,04858 MB), dzięki czemu zniekształcony kod w Markdownie z pojedynczego kroku nie może przerwać renderowania kodu w Markdownie dla kolejnych kroków. W podsumowaniu może zostać zapisanych tylko 20 kroków; dane wyjściowe każdego kolejnego kroku nie będą widoczne.

W następnej recepturze użyjemy zestawu narzędzi do dopisania bardziej złożonych rzeczy do podsumowań zadań.

Pełne informacje na temat plików środowiskowych i podsumowań zadań możesz znaleźć na stronie *https://docs.github.com/en/actions/using-workflows/workflow-commands-for-github-actions?tool=bash#environment-files*.

Wyrażenia i wykonywanie warunkowe

W rozdziałach 1. i 2. używaliśmy wyrażeń (`${{ ... }}`) do wyprowadzania wartości z obiektów kontekstowych. W tej recepturze użyliśmy wyrażenia z właściwością `if` dowolnego kroku, aby wykonywać ten krok warunkowo:

```
- name: Test the container
  if: ${{ steps.my-action.outputs.answer != 42 }}
```

Ten krok zostanie wykonany tylko wtedy, *gdy* wartość wyjścia `answer` akcji o identyfikatorze `my-action` w kontekście `steps` nie jest równa `42`.

Właściwość `if` istnieje również dla `jobs` (zadań), po to aby wykonywać je warunkowo.

W przypadku użycia wyrażeń we właściwości `if` można opcjonalnie pominąć składnię wyrażenia `${{ }}`, ponieważ akcje GitHuba automatycznie oceniają warunek `if` jako wyrażenie.

W przypadku wykonania warunkowego wyrażenie musi zwracać `true` lub `false`. Do pisania wyrażeń i porównywania kontekstu z wartościami statycznymi można użyć operatorów przedstawionych w tabeli 3.1.

Tabela 3.1. Operatory dla wyrażeń

Operator	Opis
`( )`	Grupowanie logiczne
`[ ]`	Indeks
`.`	Wyłączenie referencji do właściwości
`!`	Negacja
`< , <=`	Mniejszy niż, mniejszy niż lub równy
`> , >=`	Większy niż, większy niż lub równy
`==`	Równy
`!=`	Nie jest równy
`&&`	I (spójnik *i*)
`||`	Lub

GitHub oferuje zestaw wbudowanych funkcji, których można używać w wyrażeniach. Mogą one pomóc w wyszukiwaniu ciągów znaków, formatowaniu danych wyjściowych lub pracy z tablicami. Lista dostępnych funkcji znajduje się w tabeli 3.2.

Krok dalej

Istnieją również specjalne funkcje do sprawdzania statusu bieżącego zadania. W poniższym przykładzie ostatni krok zostanie wykonany tylko wtedy, gdy poprzedni krok zadania się nie powiedzie — co oznacza, że zwróci wartość niezerową:

```
steps:
  - run: exit 1
  - name: The job has failed
    if: ${{ failure() }}
```

Lista dostępnych funkcji sprawdzania statusu zadania znajduje się w tabeli 3.3.

Tabela 3.2. Funkcje wbudowane w GitHuba dla wyrażeń

Funkcja	Opis
`contains(zakres, ciąg)`	Zwraca wartość `true`, jeśli *`zakres`* zawiera *`ciąg`*. Przykłady: `contains('Witaj, świecie', 'taj')` zwraca wartość `true`. `contains(github.event.issue.labels.*.name, 'bug')` zwraca wartość `true`, jeśli zgłoszenie związane ze zdarzeniem ma etykietę *bug* (błąd).
`startsWith(zakres, ciąg)`	Zwraca wartość `true`, gdy *`zakres`* zaczyna się od *`ciąg`*.
`endsWith(zakres, ciąg)`	Zwraca wartość `true`, gdy *`zakres`* kończy się na *`ciąg`*.
`format(ciąg, w0, w1, …)`	Podstawia *`wartości`* do *`ciągu`*. Przykład: `format('Witajcie, {0} {1} {2}', 'Bolku', 'i', 'Lolku')` zwraca `'Witajcie, Bolku i Lolku'`.
`join(tablica, opcjonS)`	Wszystkie wartości w tablicy (*`tablica`*) są łączone w ciąg znaków. W przypadku podania opcjonalnego separatora, *`opcjonS`*, jest on wstawiany pomiędzy połączone wartości. W przeciwnym razie używany jest domyślny separator ,.
`toJSON(wartość)`	Zwraca elegancko przedstawioną w JSON-ie reprezentację *`wartości`*.
`fromJSON(wartość)`	Zwraca obiekt JSON lub typ danych JSON dla *`wartości`*.
`hashFiles(ścieżka)`	Zwraca pojedynczy hash dla zestawu plików pasujących do wzorca *`ścieżki`*.

Tabela 3.3. Funkcje sprawdzania statusu zadania przepływu pracy

Funkcja	Opis
`success()`	Zwraca wartość `true`, jeśli żaden z poprzednich kroków nie zakończył się niepowodzeniem lub nie został anulowany.
`always()`	Zwraca wartość `true`, nawet jeśli poprzedni krok został anulowany, i powoduje, że krok i tak zawsze zostanie wykonany.
`cancelled()`	Zwraca tylko wartość `true`, jeśli przepływ pracy został anulowany.
`failure()`	Zwraca wartość `true`, jeśli poprzedni krok zadania zakończył się niepowodzeniem.

Funkcje te mogą również pomóc w warunkowym wykonywaniu niektórych kroków i na przykład w przeprowadzaniu czyszczenia. Więcej informacji na temat wyrażeń warunkowych można znaleźć na stronach *https://docs.github.com/en/actions/learn-github-actions/expressions* i *https://docs.github.com/en/actions/using-jobs/using-conditions-to-control-job-execution*.

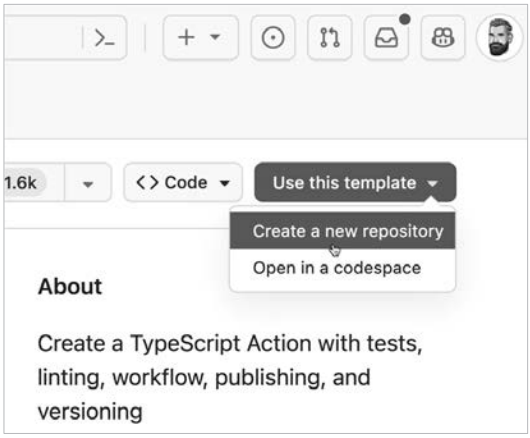
Tworzenie akcji TypeScript

W tej recepturze utworzysz podstawową **akcję TypeScript** na podstawie szablonu, zbudujesz ją i opublikujesz, a następnie użyjesz jej w przepływie pracy.

Przygotuj się!

Upewnij się, że masz zainstalowaną w miarę nowoczesną wersję Node.js (*https://nodejs.org/en/download*). Wykonaj następujące kroki:

1. Przejdź do *https://github.com/actions/typescript-action* i kliknij *Use this template/Create a new repository* (użyj tego szablonu/utwórz nowe repozytorium) w prawym górnym rogu repozytorium (zobacz rysunek 3.6).



Rysunek 3.6. Tworzenie nowego repozytorium na podstawie szablonu akcji TypeScript

2. Wybierz swoje konto GitHuba w polu *Owner* (właściciel), nazwij je `TypeScriptActionRecipe` (receptura akcji TypeScript), pozostaw jego widoczność ustawioną na *Public* (publiczna) i kliknij *Create repository* (utwórz repozytorium).
3. Sklonuj repozytorium lokalnie i otwórz folder w VS Code.

Jak to zrobić?

1. Otwórz terminal i przejdź do katalogu głównego repozytorium. Zainstaluj wszystkie niezbędne zależności:

   ```
   $ npm install
   ```

   Repozytorium zawiera kilka testów jednostkowych. Uruchom je, aby sprawdzić, czy wszystko jest w porządku:

   ```
   $ npm test
   ```

2. Otwórz plik *action.yml* i zaktualizuj metadane dla `name`, `description` i `author`:

```
name: 'Receptura akcji TypeScript'
escription: 'Odczekuje kilka milisekund, zapisuje niesamowite podsumowanie
↪zadania na wyjściu przepływu pracy i zwraca bieżącą datę i godzinę.'
author: 'Michael Kaufmann'
```

   Zignoruj na razie branding — ale zauważ, że akcja ma zdefiniowany parametr wejściowy (`milisekundy`) i parametr wyjściowy (`time`):

```
# Zdefiniuj tutaj swoje dane wejściowe
inputs:
  milliseconds:
  description: 'Twój opis wejściowy tutaj'
  required: true
  default: '1000'

# Zdefiniuj tutaj swoje dane wyjściowe.
outputs:
  time:
    description: 'Twój opis wyjściowy tutaj'
```

3. Otwórz plik *src/main.ts* i znajdź funkcję `run`:

```
export async function run(): Promise<void> {
```

   Zauważ, że używa ona zestawu narzędzi (`@actions/core`; zobacz *https://github.com/actions/toolkit*) do odczytywania parametru wejściowego:

```
const ms: string = core.getInput('milisekundy')
```

   Używa również zestawu narzędzi do pisania komunikatów debugowania. Jest to podobne do polecenia workflow `echo "::debug::{`*`komunikat_debugowania`*`}"`, którego używaliśmy w rozdziale 2.:

```
core.debug(`Waiting ${ms} milliseconds ...`)
```

   To samo dotyczy ustawiania parametru wyjściowego. Jest to to samo co zapis do pliku środowiskowego *GITHUB_OUTPUT*:

```
echo "answer=42" >> $GITHUB_OUTPUT
```

   W przypadku zestawu narzędzi wygląda to następująco:

```
core.setOutput('czas', new Date().toTimeString())
```

4. Następnym krokiem jest napisanie podsumowania pracy pod `core.setOutput` w funkcji `run`. Zacznij od `core.summary` i zwróć uwagę, że masz autouzupełnianie, które pomoże Ci we wszystkich dostępnych funkcjach i składni. Dodaj nagłówek `h2`:

```
// Napisz zaawansowane podsumowanie zadania
core.summary
  .addHeading('Advanced Job Summary', 'h2')
```

   Obiekt `core.summary` ma płynny interfejs. Oznacza to, że można dodawać nowe metody bezpośrednio do wyjścia poprzedniej. Dodaj obraz i ustaw rozmiar na 64×64:

```
.addImage(
  'https://octodex.github.com/images/droidtocat.png',
  'Droidtocat',
     {
       width: '64',
       height: '64'
      }
   )
```

Dodaj tabelę z danymi:

```
.addTable([
  [
    { data: 'Plik', header: true },
    { data: 'Wynik', header: true }
   ],
   ['foo.js', 'Pass ✅'],
   ['bar.js', 'Fail ❌'],
   ['test.js', 'Pass ✅']
])
```

I prosty link:

```
.addLink('Mój niestandardowy link', 'https://writeabout.net')
```

Podsumowanie należy zakończyć za pomocą funkcji `write`, aby zapisać bufor do pliku środowiskowego:

```
.write()
```

Kod można sprawdzić tutaj: *https://github.com/wulfland/TypeScriptActionRecipe*.

5. Spakuj TypeScript do dystrybucji. Jest to ważny krok, który należy wykonać za każdym razem, gdy modyfikuje się plik *.ts*!

```
$ npm run bundle
```

6. Zatwierdź zmiany. Spowoduje to automatyczne uruchomienie niektórych przepływów pracy — a podczas ich działania możesz wykorzystać wolny czas, aby zobaczyć, co robią. Przepływ pracy *.Github/workflows/linter.yml* działa na wszystkich żądaniach push i pull do `main` i będzie lintować bazę kodu:

```
- name: Lint Code Base
  id: super-linter
  uses: super-linter/super-linter/slim@v5
  env:
      DEFAULT_BRANCH: main
      GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
      TYPESCRIPT_DEFAULT_STYLE: prettier
      VALIDATE_JSCPD: false
```

Jeśli to się nie powiedzie, prawdopodobnie zapomniałeś uruchomić `npm run bundle` przed zatwierdzeniem zmian. Możesz również uruchomić `prettier` lokalnie, aby upewnić się, że przestrzegasz standardów lintingu:

```
$ npx prettier . --check
$ npx prettier . --write
```

Przepływ pracy *.github/workflows/codeql-analysis.yml* będzie również uruchamiany przy każdym żądaniu push i pull do `main` — ale także raz w tygodniu. Skanuje on kod pod kątem luk w zabezpieczeniach.

Check dist *(.github/workflows/check-dist.yml*) to prosty przepływ pracy, który uruchamia `npm run bundle` i porównuje dane wyjściowe z folderem *dist*. Nie powiedzie się, jeśli zapomnisz spakować TypeScript za pomocą małego i prostego skryptu:

```
- name: Compare Expected and Actual Directories (porównanie oczekiwanych
↪i rzeczywistych katalogów)
  id: diff
  run: |
    if [ "$(git diff --ignore-space-at-eol --text dist/ | wc -l)" -gt "0"
    ↪]; then
       echo "Wykryto niezatwierdzone zmiany po kompilacji. Zobacz status
       ↪poniżej:"
       git diff --ignore-space-at-eol --text
       dist/ exit 1
     fi
```

Ostatni plik to **kompilacja CI** (*.github/workflows/ci.yml*). Składa się z dwóch zadań. Jedno instaluje wszystkie zależności i uruchamia testy jednostkowe, podczas gdy drugie wykonuje akcję i wykorzystuje dane wyjściowe, tak jak zrobiliśmy to w rozdziale 2.:

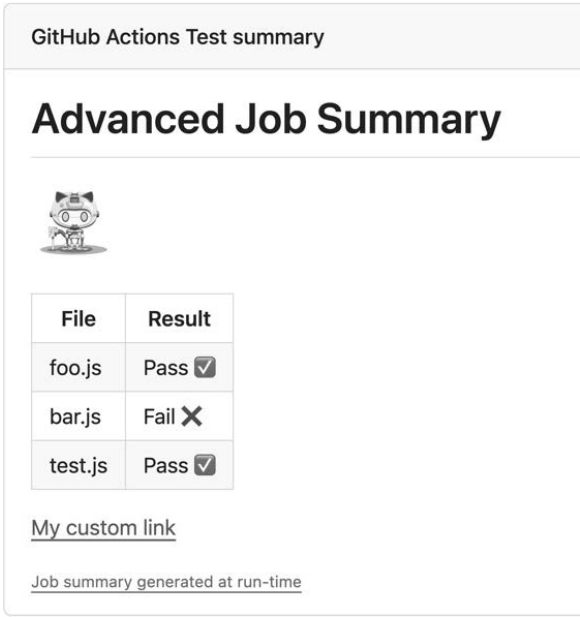
```
- name: Test Local Action (przetestuj lokalną akcję)
  id: test-action
  uses: ./
  with:
    milliseconds: 1000

- name: Print Output (wyświetl dane wyjściowe)
  id: output
  run: echo "${{ steps.test-action.outputs.time }}"
```

Zadanie `test-typescript` nie powiedzie się, ponieważ nie dostosowaliśmy testów jednostkowych, ale drugie zadanie powinno się powieść. W tym momencie możesz sprawdzić podsumowanie zadania — powinno wyglądać jak na rysunku 3.7.

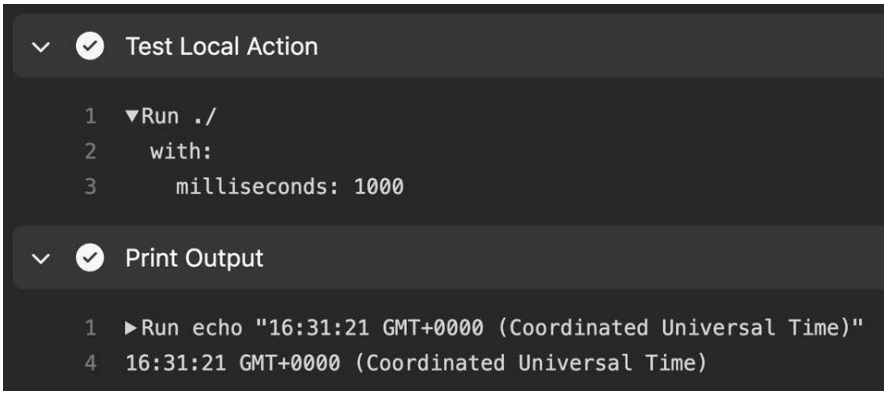
Sprawdź także wartość parametru wyjściowego w dzienniku przepływu pracy (zobacz rysunek 3.8).

Poprawianie testów jednostkowych

Nie zamieściłem receptury opisującej dostosowanie testów jednostkowych w tej książce, ponieważ książka dotyczy akcji GitHuba, a nie TypeScriptu. Jeśli jednak chcesz naprawić testy, możesz zacząć od zapoznania się z przykładem na stronie *https://github.com/wulfland/TypeScriptActionRecipe/blob/main/__tests__/main.test.ts*.



Rysunek 3.7. Podsumowanie zadania utworzone przy użyciu zestawu narzędzi



Rysunek 3.8. Dane wyjściowe akcji TypeScript w dzienniku przepływu pracy

7. Jeśli wrócisz do *main.ts*, na końcu zobaczysz, że akcja zakończy się niepowodzeniem, gdy napotka błąd. Robi to za pomocą `core.setFailed` zamiast zwracania niezerowej wartości:

```
} catch (error) {
    // Niepowodzenie przepływu pracy w przypadku wystąpienia błędu
    if (error instanceof Error) core.setFailed(error.message)
  }
```

8. Utwórz nową gałąź o nazwie `fail-ci-build` i przełącz się na nią:

```
$ git switch -c fail-ci-build
```

Dodaj niżej podany blok kodu do bloku *catch* przed wierszem zawierającym `core.setFailed`:

```
core.error('Something bad happened', {
      title: 'Bad Error',
      file: '.github/workflows/ci.yml',
      startLine: 59,
      startColumn: 11,
      endColumn: 23
})
```

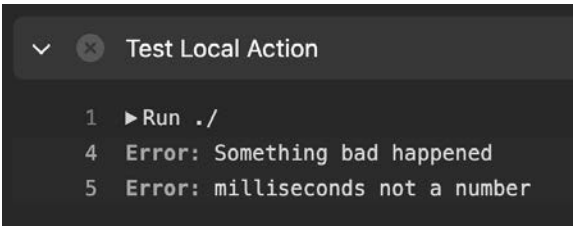
W *.github/workflows/ci.yml* zmodyfikuj argument `milliseconds` na coś, czego nie można przekonwertować na liczbę całkowitą:

```
      - name: Test Local Action // Przetestuj lokalną akcję
        id: test-action
        uses: ./
        with:
          milliseconds: xxx
```

Spakuj TypeScript, zatwierdź zmiany i utwórz pull request:

```
$ npm run bundle
$ git add .
$ git commit -m "Fail CI build"
$ git push –set-upstream origin fail-ci-build
$ gh pr create --fill
```

9. Zapoznaj się z utworzonym pull requestem i zwróć uwagę na dane wyjściowe w dzienniku z `core.setFailed` i `core.error` (zobacz rysunek 3.9).



Rysunek 3.9. Dane wyjściowe nieudanej akcji w dzienniku przepływu pracy

Należy również pamiętać, że adnotacja jest wyświetlana w pliku przepływu pracy (zobacz rysunek 3.10).

Narzędzia związane z TypeScriptem i zestaw narzędzi sprawiają, że tworzenie akcji GitHuba jest wygodne i bardzo pomocne w tworzeniu wysokiej jakości akcji.

Jak to działa?

Akcje TypeScript nie różnią się tak bardzo od akcji kontenera — po prostu działają w środowisku Node.js zamiast w kontenerze Dockera. TypeScript to tylko warstwa na JavaScripcie, która jest transpilowana do JavaScriptu (do folderu */dist*) po uruchomieniu

Rysunek 3.10. Adnotacje z zestawu narzędzi

`npm run bundle`. Jeśli nie znasz jeszcze języka TypeScript, wszystkie te narzędzia mogą wydawać się przytłaczające, ale ułatwiają rozpoczęcie pracy. Autouzupełnianie i IntelliSense w VS Code, automatyczne lintowanie i formatowanie, testy jednostkowe z tzw. mockowaniem — istnieje wiele narzędzi, które pomogą Ci napisać dobry kod.

Krok dalej

To tylko zarys tego, co można zrobić za pomocą zestawu narzędzi (*https://github.com/actions/toolkit*). Zestaw ten może również pomóc w pracy z API GitHub REST lub GraphQL, tokenami OICD i wieloma innymi. Jeśli planujesz robić coś więcej z akcjami GitHuba, warto nauczyć się przynajmniej trochę języka TypeScript, aby móc wykorzystać potężne możliwości tego zestawu narzędzi.

Tworzenie akcji złożonej

Trzeci typ akcji, oprócz akcji kontenera Dockera i akcji JavaScript/TypeScript, to **akcje złożone** (ang. *composite actions*). Akcje złożone są opakowaniem dla innych akcji. W tej recepturze utworzysz prostą akcję złożoną i użyjesz jej w przepływie pracy — za jednym razem przy użyciu skryptu bash, a za drugim razem za pomocą skryptu GitHuba.

Przygotuj się!

Utwórz nowe repozytorium, o nazwie *CompositeActionRecipe*. Uczyń je publicznym, aby nie zużywać żadnych minut akcji, i zainicjalizuj plikiem README. Sklonuj repozytorium lokalnie i otwórz je w VS Code lub otwórz w GitHub Codespaces.

Jak to zrobić?

1. Dodaj nowy plik o nazwie *action.yml* do katalogu głównego repozytorium. Dodaj nazwę i opis:

   ```
   name: 'Receptura akcji złożonej'
   description: 'Pozdrawia użytkownika i zwraca 42.'
   ```

2. Dodaj dane wejściowe o nazwie `who-to-greet` i dane wyjściowe o nazwie `answer`. Zauważ, że potrzebujesz identyfikatora kroku, aby uzyskać dostęp do danych wyjściowych. Dodamy go w następnym kroku:

   ```
   inputs:
     who-to-greet:
       description: 'Kogo pozdrowić'
       required: true
       default: 'Świat'
   outputs:
     answer:
       description: "Odpowiedź na życie, wszechświat i wszystko"
       value: ${{ steps.deep-thought.outputs.answer }}
   ```

3. Następnie dodaj sekcję `runs`, ustaw `using` na `composite` i dodaj jeden krok, który wykonuje skrypt *bash*. Zauważ, że w akcjach złożonych musisz określić powłokę i nie możesz polegać na powłoce domyślnej:

   ```
   runs:
      using: "composite"
      steps:
        - name: Awesome bash script action // Niesamowita akcja skryptu bash
          id: deep-thought
          shell: bash
          run: |
             echo "Witaj '${{ inputs.who-to-greet }}'".
             echo "answer=42" >> $GITHUB_OUTPUT
             echo "Na razie i dzięki za wszystkie rybki".
   ```

   Używamy danych wejściowych do zapisania komunikatu powitalnego w dzienniku przepływu pracy i ustawiamy parametr wyjściowy na `42` w taki sam sposób, jak zrobiliśmy to w akcji kontenera. Jedyną różnicą jest to, że wykonujemy skrypt bezpośrednio na runnerze przepływu pracy, a nie w kontenerze.

4. Dodaj nowy plik przepływu pracy o nazwie *.github/workflows/ci.yml* i skonfiguruj go tak, aby działał dla `push` i `pull_requests`:

   ```
   name: CI Workflow (przepływ pracy CI)
   on: [push, pull_request].
   ```

Dodaj zadanie, które sprawdza repozytorium i uruchamia akcję złożoną z parametrem wejściowym.

Nadaj krokowi identyfikator, aby uzyskać dostęp do danych wyjściowych, na przykład:

```
jobs:
    ci-job:
        runs-on: ubuntu-latest
        steps:
            – uses: actions/checkout@v4.1.1
            – name: Run my own composite action (uruchom moją własną akcję
            ↪złożoną)
              id: my-action
              uses: ./
              with:
                who-to-greet: "@wulfland
```

Następnie dodaj dodatkowy krok, który wypisze dane wyjściowe do konsoli:

```
            - name: Output the answer (wypisz odpowiedź)
              run: echo "Odpowiedź to ${{ steps.my-action.outputs.answer }}"
```

Pojęcia te powinny być już znane z innych typów akcji.

5. Zatwierdź i wypchnij zmiany. Spowoduje to uruchomienie przepływu pracy; możesz sprawdzić dziennik przepływu pracy, aby zobaczyć swoją akcję w działaniu.

Jak to działa?

Akcje złożone to opakowania dla kroków i innych akcji. Można ich używać do łączenia wielu poleceń uruchamiania lub akcji — niezależnie od tego, czy są one własnością użytkownika, czy pochodzą z platformy handlowej — i można podać wartości domyślne dla innych akcji użytkownikom w organizacji.

Akcje złożone mają kroki w sekcji *runs* pliku *action.yml* — tak jak w normalnym przepływie pracy. Dostęp do argumentów wejściowych można uzyskać za pomocą kontekstu `inputs`. Dostęp do parametrów wyjściowych można uzyskać za pomocą kontekstu `outputs` kroku w kontekście `steps`.

Krok dalej

Akcje złożone są wykonywane bezpośrednio na runnerze przepływu pracy. Ponieważ każdy runner ma zainstalowany Node.js, można również uruchamiać JavaScript i TypeScript w akcji złożonej. Użyjemy akcji o nazwie `github-script`, aby wykorzystać moc zestawu narzędzi w akcji złożonej. Oczywiście możesz również użyć tej akcji bezpośrednio w przepływach pracy.

Użycie skryptu github w akcji złożonej w celu dodania komentarza do zgłoszenia

W tej recepturze użyjemy akcji o nazwie `github-script`, aby wykorzystać moc zestawu narzędzi w akcji złożonej.

Jak to zrobić?

1. Usuń krok uruchamiania zawierający skrypt *bash* z pliku *action.yml* i zastąp go akcją `github-script`:

   ```
   - name: Awesome github script action // Niesamowity skrypt akcji GitHuba
     uses: actions/github-script@v6
     with:
       script: |
   ```

2. Użyj zestawu narzędzi, aby odczytać parametr wejściowy, zapisać powitanie do dziennika i ustawić parametr wyjściowy. Powinno to być znane z receptury „Tworzenie akcji TypeScript”:

   ```
   var whoToGreet = core.getInput('who-to-greet')
   core.notice(`Hello ${whoToGreet}`)
   core.setOutput('answer', 42)
   ```

3. Teraz dodamy dodatkową funkcjonalność. Jeśli przepływ pracy został wyzwolony przez zdarzenie typu zgłoszenie (`issues`), dodamy do tego zgłoszenia komentarz. Sprawdź, czy zdarzenie, które wyzwoliło przepływ pracy, było w rzeczywistości zgłoszeniem (`issues`) i czy użyliśmy `github.rest.issues` do utworzenia komentarza:

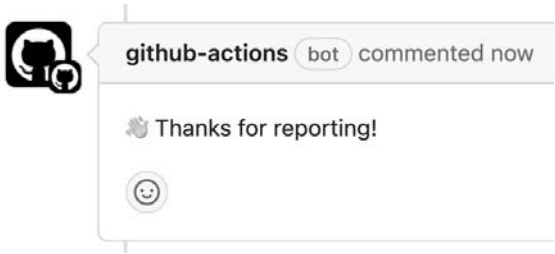
   ```
   if (context.eventName === 'issues') {
      github.rest.issues.createComment({
        issue_number: context.issue.number,
        owner: context.repo.owner,
        repo: context.repo.repo,
         body: '👏 Thanks for reporting!'
      })
   }
   ```

4. W przepływie pracy *CI* dodaj dodatkowy wyzwalacz, który uruchamia przepływ pracy po otwarciu nowego zgłoszenia:

   ```
   on:
      issues:
        types: [opened]
   ```

5. Zatwierdź i wypchnij zmiany.

6. Utwórz nowe zgłoszenie w repozytorium w sekcji *Issues/New issue* (zgłoszenia/nowe zgłoszenie) (`issues/new`), nadaj mu tytuł i je zapisz. Przepływ pracy uruchomi się i doda komentarz do tego zgłoszenia, jak pokazałem na rysunku 3.11.



Rysunek 3.11. Akcja github-script komentująca nowe zgłoszenie

Akcja `github-script` to świetny sposób na prototypowanie różnych rzeczy z wykorzystaniem możliwości zestawu narzędzi i bez konieczności tworzenia osobnej akcji.

Jak to działa?

Akcja `github-script` ułatwia szybkie napisanie skryptu w przepływie pracy lub akcji złożonej, która korzysta z API GitHuba i kontekstu uruchamiania przepływu pracy. Akcja ta wykorzystuje dane wejściowe o nazwie `script`, które zawierają treść asynchronicznego wywołania funkcji. Akcja przekazuje następujące argumenty:

- `github` — wstępnie uwierzytelniony klient *octokit/rest.js* z wtyczkami paginacji.
- `context` — obiekt zawierający kontekst przebiegu przepływu pracy.
- `core` — referencja do pakietu *@actions/core*.
- `glob` — referencja do pakietu *@actions/glob*.
- `io` — referencja do pakietu *@actions/io*.
- `exec` — referencja do pakietu *@actions/exec*.
- `fetch` — referencja do pakietu *node-fetch*.
- `require` — tworzy *wrapper proxy* wokół normalnego Node.js. Argument ten powoduje, że są wymagane względne ścieżki do bieżącego katalogu roboczego oraz zainstalowanie pakietów *npm* w bieżącym katalogu roboczym.

Ponieważ skrypt jest tylko ciałem funkcji, wartości te będą już zdefiniowane, więc nie trzeba ich importować i można ich użyć bezpośrednio, tak jak zrobiliśmy to z `core`, `context` i `github` w przykładzie.

Aby dowiedzieć się więcej o `github-script`, odwiedź *https://github.com/actions/github-script*.

Krok dalej

Edytowanie i debugowanie w pliku YAML nie są na najlepszym poziomie. Możesz jednak użyć pliku *script* w swojej akcji w następujący sposób:

```
with:
  script: |
    const script = require('./path/to/script.js')
    await script({github, context, core})
```

W ten sposób można połączyć prostotę akcji `github-script` z lepszym doświadczeniem w tworzeniu kodu w JavaScripcie.

Akcja `github-script` to świetny sposób na szybkie wypróbowanie czegoś i stworzenie roboczej koncepcji dla niektórych integracji. Dzięki akcjom złożonym można stopniowo umieszczać je w blokach konstrukcyjnych, które łatwo udostępniać. Akcje złożone są również łatwym sposobem na pakowanie funkcjonalności wielokrotnego użytku. W miarę rozwoju rozwiązania prawdopodobnie warto pomyśleć o przeniesieniu go do akcji TypeScript lub akcji kontenera Docker, aby zapewnić jego łatwiejsze utrzymywanie.

Udostępnianie akcji na Marketplace

Siłą akcji GitHuba jest społeczność — a dzielenie się wynikami pracy oznacza troskę o innych. Dlatego właśnie platforma GitHub Marketplace odgrywa kluczową rolę w usprawnianiu przepływów pracy opartych na społeczności. W tej recepturze dodasz branding i inne metadane do jednej z akcji i udostępnisz ją na platformie Marketplace.

Przygotuj się!

W tej recepturze użyję akcji kontenera Dockera, którą utworzyliśmy wcześniej — ale możesz również użyć akcji TypeScript lub akcji złożonej. Nie ma to znaczenia. Dopóki akcja znajduje się we własnym publicznym repozytorium, będzie działać.

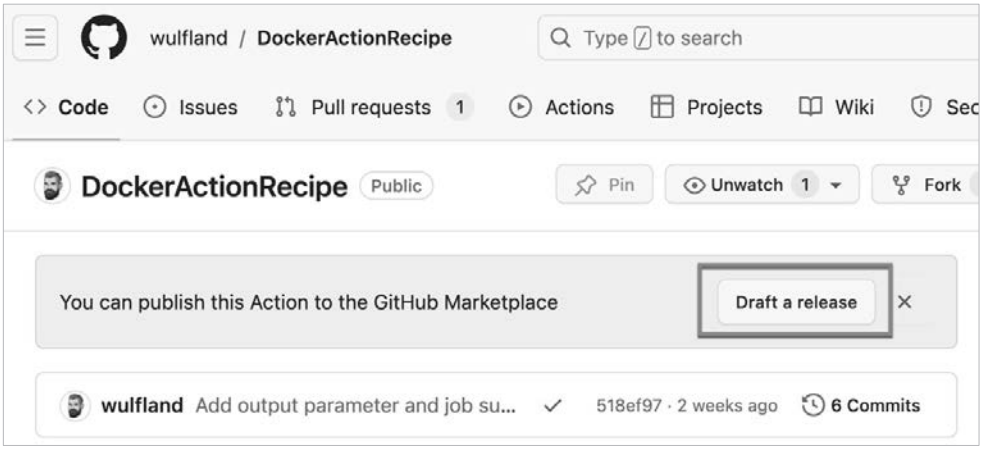
Jak to zrobić?

1. Przejdź do katalogu głównego repozytorium w przeglądarce. GitHub wykryje, że repozytorium zawiera plik *action.yml*, i zaproponuje opublikowanie wydania w niebieskim banerze (zobacz rysunek 3.12).

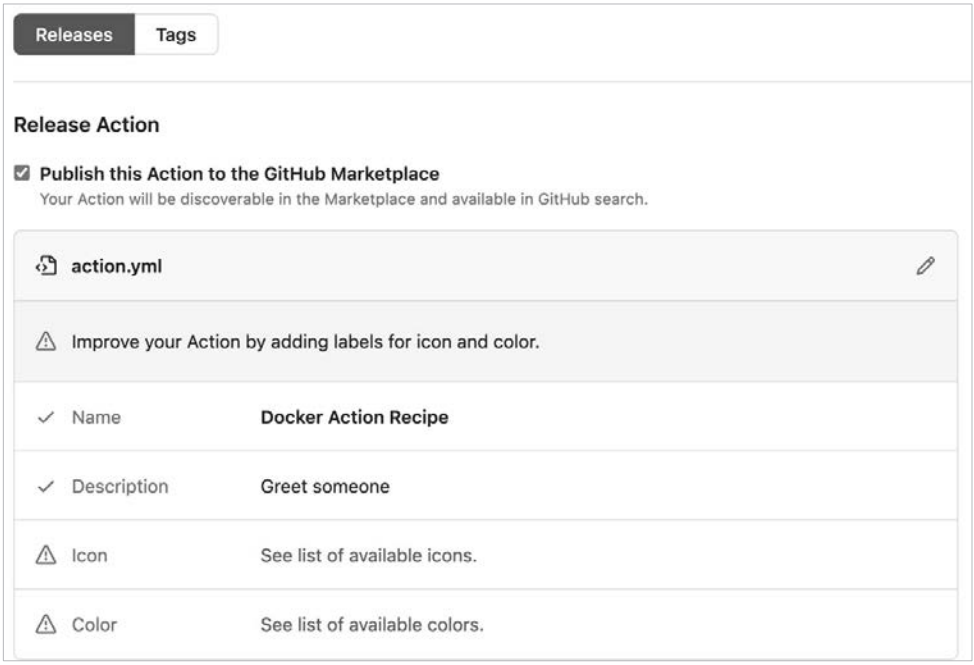
   Jest to to samo co przejście do *Releases* (wydania) i kliknięcie *Draft a new release* (projekt nowego wydania) (`/releases/new`).

2. W oknie dialogowym GitHub wyświetli kilka ostrzeżeń, które pomogą ulepszyć akcję na Marketplace (zobacz rysunek 3.13).

   Kliknij link, aby zobaczyć dostępne ikony i kolory, i wybierz któreś z nich. Ja wybiorę `bell` (dzwoneczek) i `purple` (fioletowy).



Rysunek 3.12. Przygotowanie wydania w celu opublikowania akcji na Marketplace

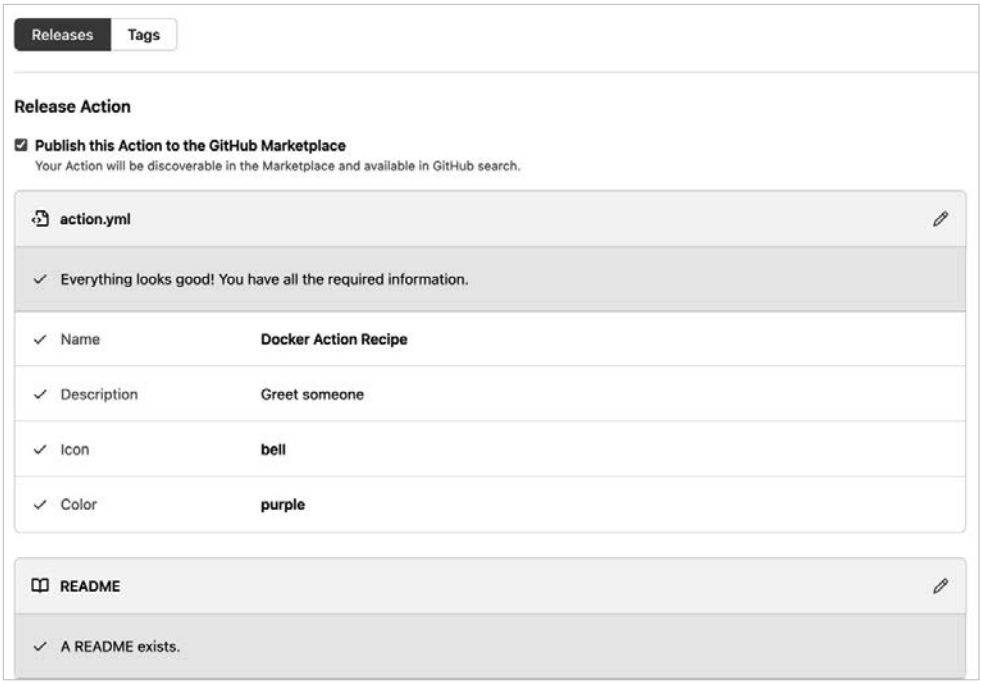


Rysunek 3.13. Wytyczne dotyczące publikowania akcji na Marketplace

3. Otwórz plik *action.yml* w VS Code i dodaj informacje o brandingu i autorze:

```
name: 'Docker Action Recipe'
description: 'Greet someone'
branding:
   icon: bell
   color: purple
author: 'Michael Kaufmann'
```

4. Dobrze utworzony plik *README.md* jest ważny dla akcji dostępnej na Marketplace. Dodaj sekcję dla danych wejściowych i wyjściowych oraz przykład użycia. Pobierz plik z GitHuba (*https://github.com/wulfland/DockerActionRecipe*). Informacje zawarte w tym pliku na pewno okażą się pomocne.
5. Zatwierdź i wypchnij zmiany.
6. Wróć do przeglądarki i odśwież okno nowego wydania. Procedura sprawdzania nie powinna teraz wyświetlać żadnych ostrzeżeń i powinno to wyglądać tak jak na rysunku 3.14.

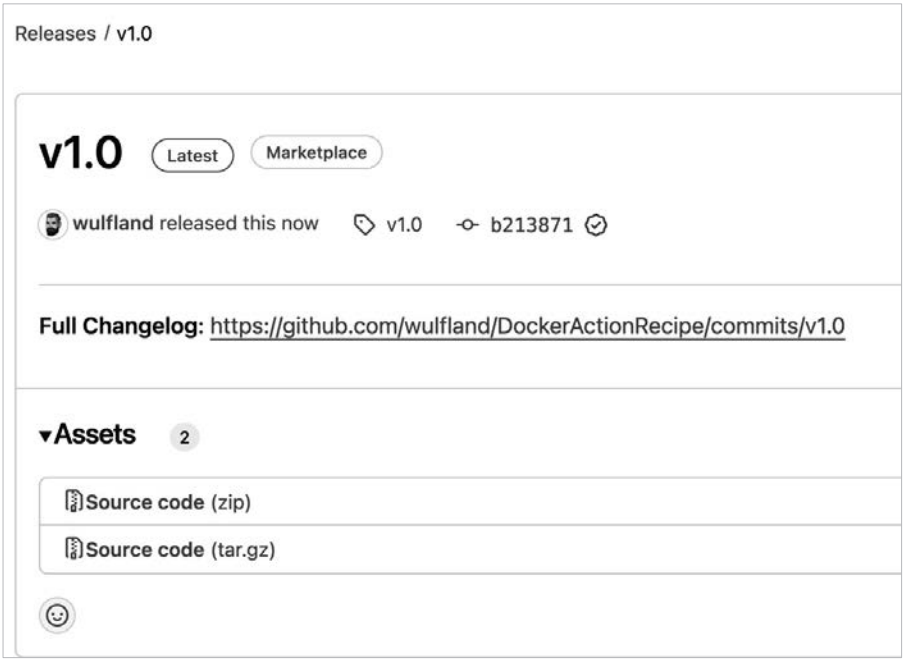


Rysunek 3.14. Sprawdzenie powiodło się, ponieważ ma unikalną nazwę, branding, opis i plik README

7. Kliknij *Choose a tag* (wybierz tag), wpisz **`v1.0`** i kliknij *Create new tag* (utwórz nowy tag) (zobacz rysunek 3.15).

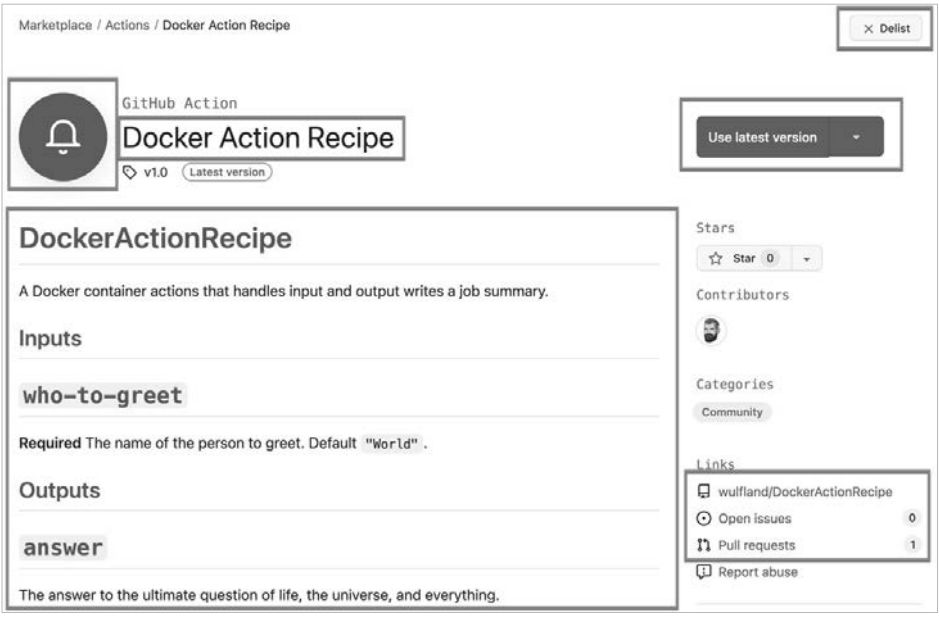
Rysunek 3.15. Tworzenie lub wybieranie tagu w celu utworzenia wydania

8. Dodaj `v1.0` jako tytuł wydania. Zauważ, że możesz automatycznie wygenerować notatki do swojego wydania. Jednak wyniki będą bardzo dobre tylko wtedy, gdy pracujesz z pull requestami. Możesz również dodać opis ręcznie.
9. Kliknij *Publish release* (opublikuj wydanie). W wydaniu zostaną wyświetlone etykiety *Marketplace* i *Latest* (zobacz rysunek 3.16).

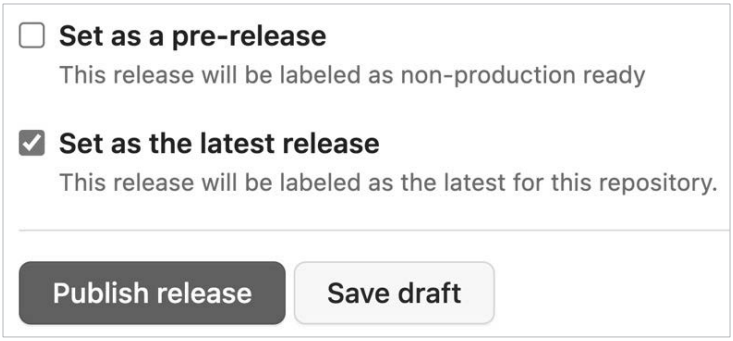


Rysunek 3.16. Zawartość wydania w repozytorium

10. Kliknij etykietę *Marketplace*. Spowoduje to przejście do Twojej akcji na Marketplace.
11. Zwróć uwagę na ikonę z kolorem określonym w `branding` w pliku *action.yml*, obok nazwy akcji. Plik *README.md* repozytorium zajmie największą część plików akcji. Po prawej stronie znajdują się: przycisk do *Delist* (usunięcia) plików akcji z bazy (usunięcia jej z Marketplace), selektor wersji i ważne linki. Pierwszy z nich przeniesie Cię z powrotem do Twojego repozytorium (zobacz szczegóły ekranu akcji GitHuba na rysunku 3.17).
12. Wróć do repozytorium i utwórz nowe wydanie o nazwie `v1.1`, powtarzając te same kroki. Zwróć uwagę, że tym razem okno dialogowe ma ustawioną flagę najnowszego wydania (zobacz rysunek 3.18). Jeśli o tym zapomnisz, GitHub nie oznaczy wydania jako najnowsze — niezależnie od numeru wersji wybranego jako etykieta.

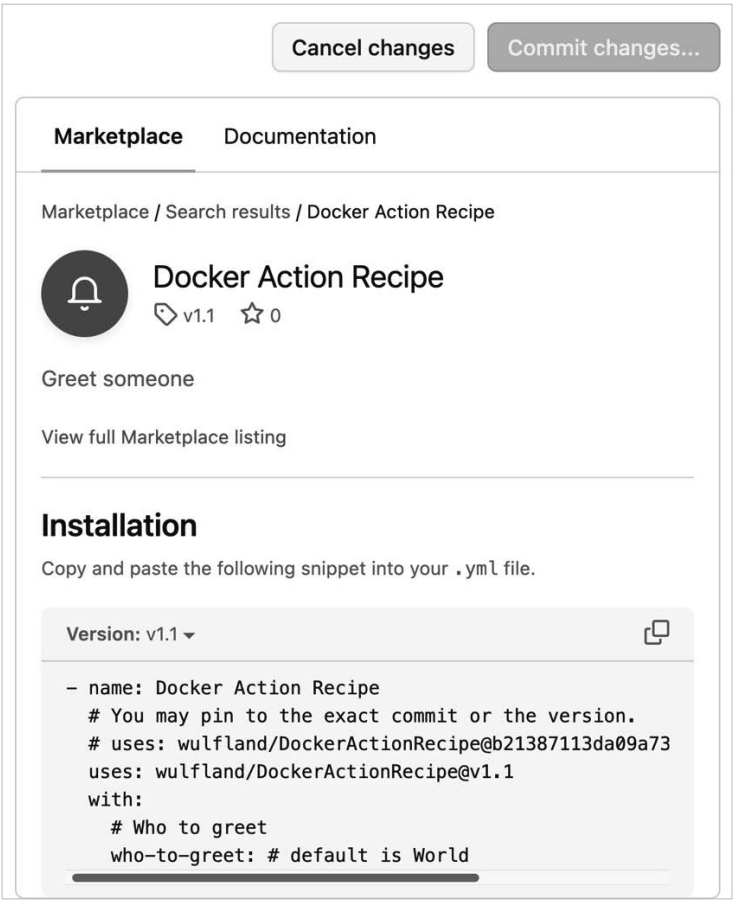


Rysunek 3.17. Akcja umieszczona na Marketplace



Rysunek 3.18. Podczas tworzenia lub edytowania wydania należy ręcznie ustawić je jako najnowsze

13. Na koniec utwórz nowy plik przepływu pracy lub edytuj istniejący przepływ pracy w przeglądarce. Po prawej stronie wpisz nazwę swojej akcji w oknie Marketplace. Twoja akcja powinna zostać natychmiast znaleziona. Zwróć uwagę na instrukcje instalacji, które GitHub automatycznie tworzy na podstawie Twojego wydania (zobacz rysunek 3.19).
14. Usuń akcję z listy, jeśli nie chcesz utrzymywać jej na Marketplace (zobacz rysunek 3.17).



Rysunek 3.19. Akcja w edytorze przepływu pracy

Jak to działa?

GitHub Marketplace jest zbudowany na bazie **wydań GitHuba** (zobacz *https://docs.github.com/en/repositories/releasing-projects-on-github*), które są zbudowane na bazie **tagów git**.

Tagiem może być dowolny tekst, ale zachęcam do korzystania z wersjonowania semantycznego, aby nadać wersjom znaczenie.

Wersjonowanie semantyczne jest to formalna konwencja określania numerów wersji oprogramowania. Składają się one z części o różnych znaczeniach. Przykłady semantycznych numerów wersji to `1.0.0` i `1.5.99-beta`. Format jest następujący:

```
<wersja_główna>.<wersja_pomniejsza>.<łata>-<wersja_wstępna>
```

Przyjrzyjmy się temu bliżej:

- **Wersja główna** — identyfikator numeryczny, który jest zwiększany, jeśli wersja nie jest wstecznie kompatybilna i zawiera przełomowe zmiany. Z aktualizacją do nowej wersji głównej należy obchodzić się ostrożnie! Wersja główna zero jest przeznaczona do początkowego rozwoju oprogramowania.
- **Wersja pomniejsza** — identyfikator numeryczny, który jest zwiększany w przypadku dodania nowych funkcji, ale wersja jest wstecznie kompatybilna z poprzednią wersją i może być aktualizowana bez naruszania czegokolwiek, jeśli potrzebna jest nowa funkcjonalność.
- **Łata** (ang. *patch*) — identyfikator numeryczny, który jest zwiększany, jeżeli wydajesz poprawki bugów kompatybilne wstecz. Nowe łaty powinny być zawsze instalowane.
- **Wersja wstępna** — identyfikator tekstowy dołączany za pomocą łącznika. Identyfikator może zawierać wyłącznie znaki alfanumeryczne ASCII i łączniki (od `0` do `9`, od `A` do `Z`, od `a` do `z`, `-`). Im dłuższy tekst, tym mniejsza wersja wstępna (co oznacza `-alpha` < `-beta` < `-rc`). Numer wersji przedpremierowej jest zawsze mniejszy niż wersji normalnej (`1.0.0-alpha` < `1.0.0`).

Najlepszą praktyką jest poprzedzanie wersji semantycznych w wydaniach GitHuba przedrostkiem `v` (np. `v1.0`, `v1.0.1` itd.).

Pełna specyfikacja dotycząca wersji semantycznych znajduje się na stronie *https://semver.org/*.

Krok dalej

Wersjonowanie za pomocą tagów wiąże się z pewnym ryzykiem, ponieważ każda osoba z uprawnieniami do zapisu do repozytorium może zmodyfikować tag. Dlatego też, jako że jesteś opiekunem akcji GitHuba, zachęcamy Cię do korzystania z reguł ochrony tagów oprócz reguł ochrony gałęzi (zobacz *https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/managing-repository-settings/configuring-tag-protection-rules*). Reguła ochrony tagów dla tagów `v*` uniemożliwi wszystkim osobom, które nie mają uprawnień administratora do Twojego repozytorium, modyfikowanie tagów zaczynających się od `v`.

Jeśli chcesz zautomatyzować proces tworzenia wersji semantycznych dla swoich wydań oraz proces automatycznego tworzenia dobrych notatek do wydań, możesz użyć **commitów konwencjonalnych** (ang. *conventional commits*) (zobacz *https://www.conventionalcommits.org*). Commity konwencjonalne dodają prefiks do każdego commitu, wskazując, czy commit zawiera nową funkcję lub poprawkę (ang. *fix*) i czy jest to commit przełomowy, czy nie. Możesz połączyć to z narzędziem **GitVersion** (zobacz *https://gitversion.net/docs/*), aby automatycznie tworzyć wersje semantyczne dla swojego wydania. Więcej informacji na ten temat znajduje się w rozdziale 7. „Wydawaj oprogramowanie za pomocą akcji GitHuba”.


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Poznaj GitHuba — bijące serce społeczności open source!

GitHub stał się czymś więcej niż platformą do hostowania i udostępniania kodu. Za sprawą funkcji znanej jako GitHub Actions może posłużyć do zarządzania przepływami pracy, w tym do automatyzacji wszelkiego rodzaju powtarzalnych zadań inżynieryjnych, takich jak ciągła integracja, ciągłe wdrażanie czy też automatyczne przypisywanie zgłoszeń do określonej kategorii.

Dzięki tej książce nauczysz się tworzyć własne akcje i przepływy pracy wielokrotnego użytku, aby udostępniać bloki konstrukcyjne społeczności lub wewnątrz organizacji. Znajdziesz tu ponad trzydzieści receptur, które sprawią, że nabierzesz biegłości w tworzeniu i debugowaniu przepływów pracy GitHuba za pomocą Visual Studio Code, a także w korzystaniu z narzędzia GitHub Copilot. Zaprezentowane rozwiązania pomogą Ci zrozumieć, jak w praktyce zastosować techniki automatyzacji wdrażania kodu. Obejmują one tworzenie i testowanie oprogramowania i bezpieczne wdrażanie na platformach takich jak Azure, Amazon Web Services czy Google Cloud.

Dzięki recepturom nauczysz się:

- tworzyć przepływy pracy GitHub Actions za pomocą narzędzi takich jak VS Code i Copilot
- uruchamiać przepływy pracy na maszynach wirtualnych dostarczonych przez GitHub
- zabezpieczać przepływy pracy za pomocą GitHub Actions
- automatyzować przepływy pracy za pomocą zaawansowanych narzędzi GitHuba
- prowadzić wdrożenia etapowe lub pierścieniowe

Michael Kaufmann ma doświadczenie w pracy z DevOps, GitHubem i Azure. Jest laureatem wyróżnienia Microsoftu MVP w kategorii DevOps i GitHub. Założył firmę Xebia Microsoft Services, która wspiera klientów podczas wdrożeń rozwiązań chmurowych, metodyki DevOps i cyfryzacji. Chętnie dzieli się wiedzą jako prelegent na międzynarodowych konferencjach.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-2046-0
9 788328 920460
Cena: 69,00 zł
packt