Politechnika Śląska
Wydział Matematyki Stosowanej
Kierunek Informatyka
Studia stacjonarne I stopnia

Projekt inżynierski

Automatyzacja procesów w przemyśle IT

Kierujący projektem:
dr inż. Zdzisław Sroczyński

Autorzy:
Artur Kasperek
Michał Płonka
Patryk Musiol

Gliwice 2020

Projekt inżynierski:

Automatyzacja procesów w przemyśle IT

kierujący projektem: dr inż. Zdzisław Sroczyński

autorzy: Artur Kasperek, Michał Płonka, Patryk Musiol

Podpisy autorów pracy

.....

.....

.....

Podpis kierującego projektem

.....

Oświadczenie kierującego projektem inżynierskim

Potwierdzam, że niniejszy projekt został przygotowany pod moim kierunkiem i kwalifikuje się do przedstawienia go w postępowaniu o nadanie tytułu zawodowego: inżynier.

Data Podpis kierującego projektem

Oświadczenie autorów

Świadomy/a odpowiedzialności karnej oświadczam, że przedkładany projekt inżynierski na temat:
Automatyzacja procesów w przemyśle IT
został napisany przeze mnie samodzielnie.
Jednocześnie oświadczam, że ww. projekt:

- nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (j.t. Dz.U. z 2018 r. poz. 1191, z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym, a także nie zawiera danych i informacji, które uzyskałem/am w sposób niedozwolony,
- nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów wyższej uczelni lub tytułów zawodowych.
- nie zawiera fragmentów dokumentów kopiowanych z innych źródeł bez wyraźnego zaznaczenia i podania źródła,
- złożona w postaci elektronicznej jest tożsama z pracą złożoną w postaci pisemnej.

Podpisy autorów pracy

Artur Kasperek, nr albumu:1234, ...(podpis:).....
Michał Płonka, nr albumu:1234, ...(podpis:).....
Patryk Musiol, nr albumu:1234, ...(podpis:).....

Gliwice, dnia .....

Spis treści

Wstęp

TODO dodać wstęp

1. Automatyzacja a branża IT

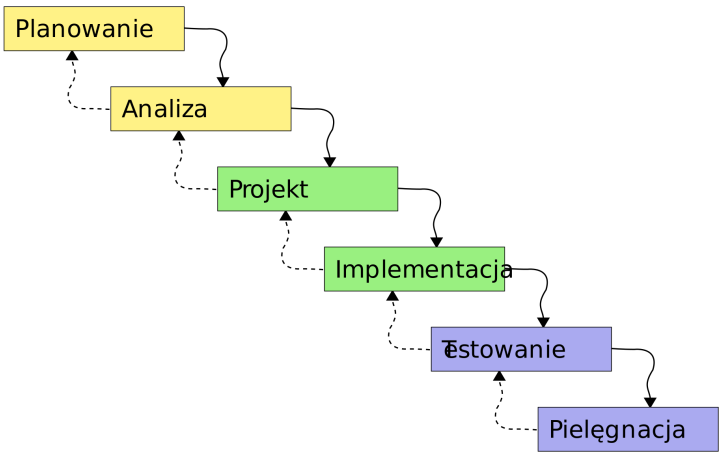
1.1. Charakteryzacja branży IT

W 2001 roku doszło do jednej z bardziej znaczących publikacji dla szeroko pojętego biznesu IT. Został wtedy opublikowany *Manifesto for Agile Software Development* autorstwa między innymi Kenta Becka, Roberta C. Martina oraz Martina Fowlera. Manifest ten opisywał rewolucyjne jak na tamte czasy praktyki [1]:

- Satysfakcja klienta dzięki wczesnemu i ciągłemu dostarczaniu oprogramowania,
- Zmiany wymagań mile widziane nawet na późnym etapie programowania,
- Częste dostarczanie działającej wersji oprogramowania (bardziej tygodnie niż miesiące),
- Bliska kooperacja między programistami a ludźmi zajmującymi się biznesem,
- Projekty powstają wokół zmotywowanych osób, którym należy ufać,
- Komunikacja w cztery oczy jest najlepszą formą komunikacji,
- Działający produkt jest najlepszym wskaźnikiem postępu prac,
- Zrównoważony rozwój, pozwalający na utrzymanie stałego tempa tworzenia aplikacji,
- Ciągła dbałość o doskonałość techniczną i dobry design,
- Prostota - sztuka projektowania systemu bez dużej komplikacji systemu,
- Najlepsze architektury, wymagania i designy powstają dzięki samoorganizującym się zespołom,
- Zespół regularnie zastanawia się, jak zwiększyć skuteczność i odpowiednio się dostosowuje.

Propozycje przedstawione przez autorów manifestu były dużą zmianą w stosunku do podejścia używanego powszechnie w tamtych czasach.

W latach 80 oraz 90 XX wieku popularnie stosowaną techniką była metodologia Waterfall, zobrazowana na rysunku 1. Poszczególne etapy projektowe były wykony-

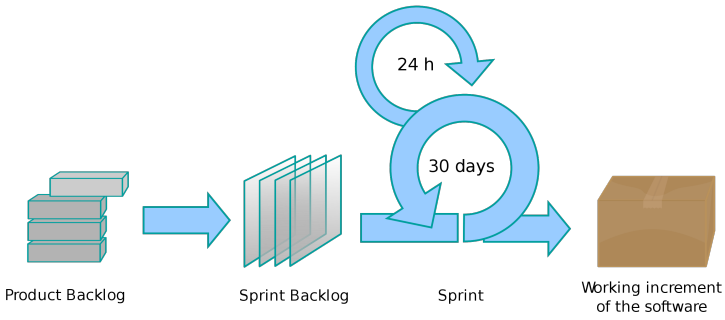


Rysunek 1: Metodologia Waterfall

wane tylko raz podczas procesu tworzenia oprogramowania. Z tego faktu wszelakie zmiany na późniejszym etapie projektowym były trudne w realizacji. Konkurencja na rynku oprogramowania komputerowego była na tyle niewielka, że producenci oprogramowania nie musieli przejmować się zanadto uwagami od użytkowników - to sprawiało, że Waterfall spełniał swoje zadania.

Pierwsze dziesięciolecie XXI wieku spopularyzowało jedno z największych osiągnięć ludzkości - Internet. Nowa rzeczywistość w której ludzkość coraz więcej czasu spędza przed urządzeniami elektronicznymi postawiła przez twórcami oprogramowania nowe wymagania. Dodatkowo coraz większe grono przedsiębiorców zaczyna dostrzegać w produkcji oprogramowania zyski. Użytkownicy zaczynają coraz bardziej spoglądać na przyjemny dla oka wygląd oprogramowania oraz jego prostotę. Metodyka zaproponowana przez autorów manifestu Agile zdaje się świetnie wpisywać w wizję tworzenia oprogramowania na miarę nowego tysiąclecia.

Jedną z bardziej znanych implementacji Agile jest metodologia o nazwie Scrum. Zakłada się w niej, że oprogramowanie powstaje w procesie kolejnych inkrementacji. Każda iteracja jest nazywana sprintem. Sprint ma z góry zdefiniowane ramy czasowe w których będzie on trwał. Na podstawie różnych czynników biznesowych



Rysunek 2: Framework Scrum

opiekun projektu decyduje które zadania powinny trafić do danego sprintu, a które są mniej priorytetowe i mogą pozostać w tzw. backlogu. Efektem końcowym danego sprintu jest działający produkt, który jest wzbogacony o rzeczy dodane podczas trwania sprintu. Scrum sam w sobie nie narzuca, ile powinien trwać dany sprint, czy też jaki system powinien być stosowany do śledzenia zadań. Wszystko zależy od preferencji danego zespołu programistycznego. Integralną częścią każdego sprintu jest retrospektywa. Na takim spotkaniu zespół dyskutuje jakie zmiany należy dokonać w procesie, by uefektywnić pracę. Dzięki elastycznemu podejściu i możliwości ulepszania procesu Scrum wydaje się być dobrym rozwiązaniem dla zespołów, które wypuszczają oprogramowanie regularnie oraz zmieniają je na podstawie opinii użytkowników.

1.2. Agile a automatyzacja

Spełnienie wymagań wymienionych w manifeście Agile wydaje się być trudne w kontekście częstego wypuszczania działającej wersji. Oczekuje się tego, aby zespół deweloperski regularnie publikował działającą wersję podglądową oprogramowania dla osób nietechnicznych. Problem ten można rozwiązać na co najmniej dwa sposoby:

- Manualny - Członek zespołu deweloperskiego regularnie według wymagań zajmuje się budowaniem wersji podglądowej oraz udostępnia ją osobom zainteresowanym,
- Automatyczny - Zespół deweloperski ustawia automatyczne procesy, które na serwerze budującym tworzą wersję podglądową aplikacji oraz publikują ją dla

osób zainteresowanych.

Proces automatyczny jest preferowanym sposobem publikacji oprogramowania. Ma on kilka zalet nad sposobem manualnym. Nie tracimy czasu specjalisty, który musiałby poświęcić go na zbudowanie i publikację aplikacji. Drugą zaletą jest fakt, że serwer za każdym razem robi te same kroki podczas procesu budowania. Tym sposobem wykluczamy możliwość popełnienia błędu przez człowieka.

Dobre praktyki związane z częstym budowaniem podglądowej wersji oprogramowania są określane jako DevOps. Len Bass, Ingo Weber oraz Liming Zhu w swojej książce [2] określają DevOps jako zbiór praktyk których celem jest zmniejszenie czasu publikacji zmian na serwerze produkcyjnym przy jednoczesnej trosce o wysoką jakość. W praktyce często członkiem zespołu deweloperskiego jest tzw. DevOps. Jego zadaniem jest automatyzacja wszelakich procesów oraz często także utrzymanie środowiska produkcyjnego. Osoba na tym stanowisku powinna się cechować dobrą znajomością systemu operacyjnego, który jest używany na serwerach produkcyjnych oraz deweloperskich. Ponadto powinna być zorientowana w różnych rozwiązaniach chmurowych które współcześnie są coraz częściej używane.

1.3. Git - kamień milowy dla deweloperów

Ciężko byłoby sobie wyobrazić obraz dzisiejszego przemysłu IT, gdyby nie system kontroli wersji Git. Jego autorem jest Linus Torvalds. Co ciekawe stworzył on Git'a jako dodatkowy projekt, który miał pomóc w pisaniu jądra Linuxa. Dzięki Git'owi każdy członek zespołu deweloperskiego ma dostęp do wspólnego repozytorium, gdzie każdy może publikować swoje zmiany. Jedną z ważniejszych funkcji Git'a jest możliwość tworzenia własnych rozgałęzień kodu, gdzie dany programista wysyła swoje zmiany. Dalej w procesie merge'owania jest możliwe połączenie zmian danego dewelopera z kodem innych programistów. Dzięki tej cesze Git nadaje się świetnie do wszelakich projektów programistycznych, w których pracuje kilku programistów równolegle. Z biegiem lat Git stał się standardem.

Powszechnie znaną dobrą techniką związaną z strukturą tego, co mamy na Gicie jest tzw. *Git Workflow*. Wzorzec ten sugeruje, by używać 2 głównych gałęzi. Jedną, która będzie odzwierciedlała finalną, zdatną do użytku wersję oraz drugą, gdzie znajduje się wersja deweloperska. Pośrednie gałęzie służą do implementacji

poszczególnych nowych funkcjonalności oraz do synchronizacji wersji deweloperskiej z gałęzią produkcyjną. Dzięki takiemu podejściu proces deployment'u na środowisko produkcyjne oraz testowe staje się o wiele prostszy, ponieważ mamy dwie gałęzie, które odzwierciedlają te środowiska. Na rysunku 4 możemy podejrzeć szczegółowy schemat, jak powinno wyglądać repozytorium Git'owe korzystające z wzorca *Git Workflow*

Rysunek 3: Git Workflow

W pierwszym dziesięcioleciu XXI wieku popularne stały się rozwiązania SaaS (z ang. *oprogramowanie jako usługa*) takie jak GitHub, GitLab czy też BitBucket. Dzięki takiemu rozwiązaniu nie musimy się przejmować utrzymaniem własnego serwera Git. Dodatkowo platformy SaaS zapewniają nam wszelakie aktualizacje, które usprawniają system. Platformy takie jak GitHub zapewniają narzędzia, które ułatwiają proces tworzenia oprogramowania. Narzędzia te są dopasowane, aby działać dobrze z naszym repozytorium. Przykładem takiego rozwiązania jest *GitHub Pages*. Technologia ta pozwala publikować stronę www na podstawie plików, które są częścią repozytorium. Użytkownik definiuje, na której gałęzi oraz w którym folderze znajdują się pliki ze stroną internetową. Od tego momentu GitHub automatycznie stwarza nam stronę internetową dostępną pod subdomeną *nazwauzytkownika.github.io*.

1.4. Ciągła integracja

Programiści podczas tworzenia nowych funkcjonalności powinni się upewnić, że ich zmiany nie zepsują tego, co już istnieje. Jednym z takich sposobów jest odpalenie testów. Najbardziej znanymi typami testów są:

- testy jednostkowe - testy te skupiają się na testowaniu funkcji oraz klas w danym projekcie. Są najprostszą formą testowania podczas której zastępuje się wszelkie trzecie zależności tzw. *mock'ami*,
- testy integracyjne - są to testy, w których odpalany jest testowany projekt oraz wybrany projekt trzeci, z którym chcemy sprawdzić poprawność działania,
- testy e2e - testy te wymagają działającej w pełni aplikacji wraz z wszystkimi zależnymi projektami. Zazwyczaj sprawdzają najbardziej znaczące funkcjonalności naszej aplikacji.

Pisanie testów jest integralną częścią pracy programisty. Ich jakość oraz ilość jest znaczącym wskaźnikiem mówiącym o stanie danego projektu. Pozwalają one na tworzenie oprogramowania, które powinno mieć mniej błędów. Ciągła integracja przenosi odpowiedzialność za odpalanie testów z programisty na serwer ciągłej integracji.

Testy to nie jedyna rzecz, która może być sprawdzana za pomocą ciągłej integracji. W procesie tym jest ważne, by zweryfikować jakość nowego kodu stworzonego przez developera. Przykładowymi rzeczami, które możemy zrobić podczas procesu ciągłej integracji jest:

- sprawdzenie procentowego pokrycia testami projektu. Możemy na tej podstawie nie pozwolić na zmergowanie zmian, jeżeli przekroczymy z góry ustaloną procentową ilość kodu, która nie jest pokryta testami,
- sprawdzenie czy kod jest poprawnie sformatowany według ustalonych reguł. Proces ten nazywa się powszechnie *lintowaniem*. Dzięki temu unikniemy problemu, w którym kod będzie różnie sformatowany w zależności dewelopera piszącego kod,
- testowanie aplikacji na niestandardowych systemach operacyjnych. Dzięki temu możemy się upewnić, że nasza aplikacja zadziała na mniej standardowych

konfiguracjach. Jest to szczególnie istotne, jeżeli deweloperzy oraz testerzy nie skupiają się na testowaniu na danym systemie operacyjnym,

- sprawdzanie czy zależności trzecie, które są używane w projekcie nie mają żadnych podatności związanych z bezpieczeństwem aplikacji. Z racji tego możemy na wczesnym etapie wychwycić takie problemy i odpowiednio szybko zaaktualizować te biblioteki,

- zbudowanie obrazów dockerowych. Taki krok pozwala osobom znającym dockera bezproblemowe przetestowanie każdej gałęzi w naszym projekcie.

Dobór rzeczy, które chcemy zautomatyzować zależy od indywidualnych potrzeb danego projektu. Za duża komplikacja procesu ciągłej integracji może przynieść więcej szkody niż korzyści. Należy pamiętać, że programista musi czekać na to, aż proces ciągłej integracji się wykona. Więcej o testach opisane jest w rozdziale 5.

1.5. Ciągłe dostarczanie

Podczas gdy ciągła integracja skupia się na upewnieniu, czy dane zmiany przechodzą testy, tak zadaniem procesów ciągłego dostarczania jest zbudowania aplikacji oraz umożliwienie jej późniejszej manualnej publikacji na środowisku produkcyjnym. Każdy projekt ma swoją własną specyfikę i proces ciągłego dostarczania jest inaczej ustawiany. Punktem wspólnym jest to, że człowiek decyduje, kiedy release'a oprogramowania powinien nastąpić.

Zanalizujmy typowy przykład, jak może wyglądać proces ciągłej integracji. Mamy aplikację webową, która komunikuje się z bazą danych oraz wystawia API. Istnieją dwa środowiska gdzie działa ten serwis:

- środowisko produkcyjne - jest to środowisko, które korzysta z produkcyjnej bazy i jest wykorzystywane przez klientów. Jest sprawą priorytetową, by środowisko to działało bez przestojów,

- środowisko staging'owe - środowisko to służy do testowania wersji deweloperskiej naszego serwisu. Możemy pozwolić sobie na to, by środowisko to miało przestoje od czasu do czasu.

Struktura gałęzi na Gicie jest prosta. Mamy dwa główne branche.

- master - główna gałąź, gdzie jest trzymana najnowsza wersja stabilna serwisu,
- staging - gałąź deweloperska zawierająca zmiany, które nie były jeszcze dobrze przetestowane przez testerów.

Rysunek 4: Repozytorium z branchami master oraz staging

Struktura na Gicie dobrze odzwierciedla jakie środowiska mamy. Możemy wykorzystać ten fakt i za pomocą serwera ciągłego dostarczania budować aplikację i automatycznie ją instalować na serwerze stagin'owym za każdym razem, gdy ktoś zrobi commit'a na gałęzi staging. Dzięki takiemu podejściu oszczędzamy czas testerów. Nie muszą oni czekać na techniczną osobę, która zaktualizuje środowisko staging'owe.

By dalej ułatwić proces wypuszczania oprogramowania możemy zrobić podobne kroki jak przy staging'u dla wersji produkcyjnej. Za każdym razem, gdy ktoś wrzuca nowe zmiany na gałąź master możemy zbudować obraz (więcej o tworzeniu obrazów w rozdziale 2) z serwisem i wysłać go do rejestru obrazów. Tym sposobem kwestia wypuszczenia oprogramowania na serwer produkcyjny zostaje po stronie DevOps'a/administratora, który manualnie może wywołać proces aktualizacji danego serwisu.

1.6. Ciągła dystrybucja

Ciągła dystrybucja jest rozszerzeniem ciągłego dostarczania. Proces ten zakłada, że zmiany, które znajdują się na naszej głównej gałęzi na repozytorium odzwierciedlają stan, który jest na serwerze produkcyjnym. Możemy to osiągnąć poprzez proces budujący aplikacje oraz aktualizujący serwer produkcyjny za każdym razem,

gdy ktoś wrzuci nowego commit'a. Podejście to wymaga samodyscypliny i dobrej organizacji w zespole deweloperskim. Zespół musi być świadomy tego, że jeżeli dane zmiany nie są zanadto przetestowane, może to spowodować problemy na środowisku produkcyjnym, które jest kluczowe dla biznesu. W skrajnym przypadku może okazać się, że środowisko produkcyjne będzie działało z problemami przez dłuższy czas i dane, które trzymamy w bazie zostaną "zepsute" przez wadliwy kod.

Problemy te możemy zminimalizować przez stosowanie następujących dobrych praktyk:

- Wsteczna kompatybilność - jeżeli dana wersja ma w sobie błąd, administrator serwera w każdej chwili będzie w stanie cofnąć działającą wersję aplikacji do wersji, która nie miała problemów,
- Orkiestrator - nowoczesne orkiestratory takie jak np. Kubernetes (o orkiestratorach traktuje rozdział 2) umożliwiają zrobienie tzw. *roll back*(cofnięcia do wcześniejszej wersji) w prosty sposób. Dzięki temu czas, w którym klienci doświadczyli problemów będzie stosunkowo krótki,
- Kopia zapasowa bazy danych - dzięki kopii możemy być pewni, że w przypadku "zepsucia" danych lub ich utraty przez błąd w programie będziemy w stanie je odzyskać,
- Testy e2e - dodanie testów całościowych systemu jest ważną rzeczą w kontekście ciągłej dystrybucji. Odpalając testy e2e na wersji deweloperskiej aplikacji możemy być bardziej pewni, że zmiany, które zostały dodane, nie psują istniejących funkcjonalności.

Dzięki wdrożeniu procesu ciągłej dystrybucji czas między stworzeniem danego kodu i wysłaniem go na repozytorium a uruchomieniem go na środowisku produkcyjnym zmniejsza się. Dodatkowo nie musimy poświęcać czasu administratora by zaktualizować środowisko produkcyjne. Należy pamiętać, że bez odpowiedniej dyscypliny programistów oraz braku stosowania dobrych praktyk proces ciągłej dystrybucji może spowodować więcej problemów niż korzyści. Zazwyczaj proces ten jest stosowany przez doświadczone zespoły deweloperskie.

1.7. GitOps - czym jest?

GitOps jest zbiorem dobrych praktyk, opisująca prawidłowy sposób wypuszczania aplikacji na środowisko deweloperskie jak i też produkcyjne. Jest to swoiste rozszerzenie procesu ciągłego dowożenia. Metodologia ta zachęca do trzymania plików konfiguracyjnych danego orkiestratora na repozytorium Git'owym. Dzięki takiej praktyce zyskujemy:

- Możliwość weryfikacji jakości zmian, tzw. *Code Review*. Dlatego że konfiguracja jest trzymana na Gicie, inni deweloperzy mogą ocenić, czy nasze zmiany są właściwe. Jest to dość duża zmiana względem tradycyjnego modelu, gdzie administrator sam decydował o zmianach w konfiguracji,
- Historię zmian - możemy dzięki temu przejrzeć, jak w przeszłości aplikacja była skonfigurowana,
- Przejrzystość systemu - każdy członek techniczny może podejrzeć to, w jaki sposób aplikacja działa na serwerze deweloperskim/produkcyjnym. W modelu tradycyjnym przeciętni członkowie nie wiedzą jak wygląda konfiguracja serwera - dostęp ma tylko administrator,
- Automatyzację procesu wypuszczania, czyli ciągłe dowożenie. Jeżeli projekt korzysta z praktyk GitOps, z definicji spełnia proces ciągłego dowożenia. Każda zmiana na gałęzi deweloperskiej/produkcyjnej powoduje to, że wypuszczamy nową wersję oprogramowania na serwerze.

Termin GitOps został spopularyzowany przez orkiestrator Kubernetes. Pliki konfiguracyjne Kubernetesa są plikami w formacie YAML. Mocna decentralizacja sposobu, w jakim działa Kubernetes pozwala na trzymanie plików konfiguracyjnych pojedynczych jednostek logicznych w osobnych plikach. Jest to cecha ważna dla projektów opartych o architekturę mikroserwisową. W takich projektach mamy kilkanaście serwisów odpowiedzialnych za różne aspekty biznesowe aplikacji. Każdy taki serwis posiada własne repozytorium kodu w Gicie. Dzięki możliwości decentralizacji plików konfiguracyjnych Kubernetesa możemy dla każdego mikroserwisu trzymać te pliki osobno - każdy mikroserwis ma tylko pliki konfiguracyjne dotyczące samego siebie. Zapewnia to większą separację logiczną. Możność aplikowania na klaster Kubernetesowy pojedynczego pliku YAML, który stanowi tylko częściową

konfigurację całego systemu, pozwala na aktualizację tylko pojedyńczego serwisu na klastrze. Dzięki temu poszczególne serwisy stają się bardziej niezależne od siebie.

Załóżmy, że tworzymy aplikację opartą na mikroserwisach. Jednym z elementów tej aplikacji jest serwis zarządzający użytkownikami. Serwis ten zajmuje się logiką związaną z zarządzaniem użytkownikami. Jest on stworzony w języku JavaScript i swoją usługę wystawia na porcie 3000. Jego kod wygląda następująco:

```
const express = require('express')
const app = express()
const port = 3000
const users = [
    {
        name: 'Jan',
        email: 'jan@gmail.com',
    },
    {
        name: 'Kasia',
        email: 'kasia@gmail.com',
    }
]

app.get('/get-all-users', (req, res) => {
    res.send(users)
})

app.listen(port, () => {
    console.log(`Users service listening at http://localhost:${
        port}`)
})
```

Listing 1: Serwis zarządzający użytkownikami

Jednym z elementów tego serwisu jest plik konfiguracyjny Kubernetesa, który zajmuje się zdefiniowaniem usługi, jaki nasz serwis ma oferować. Usługa to jeden z obiektów Kubernetesa, który umożliwia przesłanie ruchu sieciowego do instancji z usługą. Plik konfiguracyjny wygląda następująco:

```
kind: Service
apiVersion: v1
metadata:
  name: user-service
```

```
  namespace: production
  labels:
    run: user-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-internal:
       0.0.0.0/0
spec:
  selector:
    app: UserService
  ports:
  - protocol: TCP
    port: 80
    targetPort: 3000
  type: LoadBalancer
```

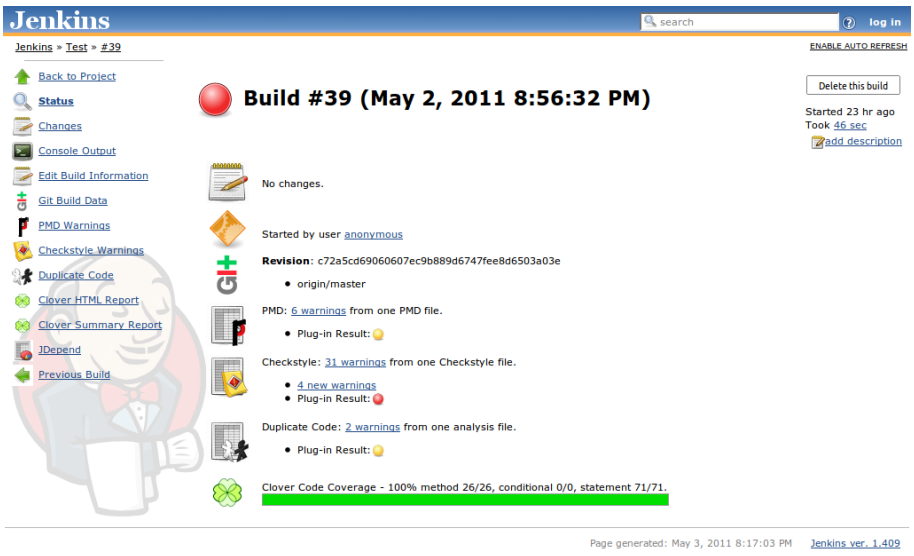
Okazuje się, że z jakiegoś powodu chcielibyśmy zmienić port, na którym nasz serwis z użytkownikami nasłuchuje. W tym celu musimy zmienić plik JavaScript'owy oraz plik YAML z konfiguracją Kubernetesa. Dzięki temu, że obydwa pliki są częścią jednego repozytorium możemy taką zmianę wykonać za pomocą jednego pull request'a. Nie musimy do całej akcji zmiany portu angażować administratora klastra Kubernetesowego - administrator jedyne przegląda, czy nasze zmiany są poprawne. Po zmergowaniu aktualizacja środowiska na klastrze następuje automatycznie. Przebiega to przy użyciu procesu ciągłego dowożenia.

1.8. Serwery automatyzujące SaaS kontra self-hosted

Wczesne lata rozwoju oprogramowania służącego do ciągłej integracji stały pod znakiem Jenkinsa. Jenkins to oprogramowanie, które jest zainstalowane na serwerze i pozwala tworzyć potoki automatyzujące. Może on służyć do stworzenia procesów ciągłej integracji, ciągłego dostarczania oraz ciągłego dowożenia. Jego charakterystyczną cechą jest to, że został on przygotowany do uruchomienia na własnym serwerze - tzw. self-hosted. Instalacja oraz utrzymanie Jenkinsa wymaga czasu specjalisty. Przewaga nad rozwiązaniami typu SaaS (z ang. *oprogramowanie jako usługa*) jest taka, że mamy większą kontrolę nad tym jak Jenkins działa. Jenkins to nie jedyne rozwiązanie self-hosted do automatyzacji:

- CruiseControl - powstał w 2001 roku. Skupiał się na automatyzacji projektów tworzonych w języku Java,

- Hudson - prekursor Jenkinsa który zakończył swój żywot w 2015. Był tworzony jako alternatywa do CruiseControl. W roku 2011 został stworzony fork, którym teraz jest Jenkins,
- GitLab CI - jako jeden z nielicznych projektów oferuje możliwość wyboru i używania w formie self-hosted oraz w formie SaaS, dostępnej pod adresem *gitlab.com*. Oprogramowanie to wyróżnia integracja z repozytorium kodu, dzięki której w kilku krokach możemy dodać automatyczne budowanie do naszego projektu. Kod całej platformy GitLab jest dostępny w formie open source.



Rysunek 5: Interfejs webowy Jenkinsa

Lata 10 XXI wieku przyczyniły się do popularyzacji rozwiązań typu SaaS, w których zespół deweloperski nie musi już utrzymywać własnego serwera z zainstalowanym tam oprogramowaniem. Dzięki SaaS utrzymaniem serwera zajmuje się strona trzecia, która za pewną opłatą zapewnia dostęp do aplikacji - zostawiając po swojej stronie sprawy związane z utrzymaniem oraz aktualizacją. Model ten stał się niezwykle popularny dla startup'ów. Firmy takie zazwyczaj posiadają tylko kilkoro ludzi technicznych. Użycie oprogramowania w formie SaaS pozwala takim zespołom skupić się na tworzeniu kodu, który ma dostarczać wartość biznesową. Do najpopularniejszych rozwiązań SaaS do automatyzacji należą:

- Travis CI - jest to serwis który zapewnia integrację z repozytorium trzymanym na GitHubie oraz BitBuckecie. Jego kod jest oferowany w formie open

source - aczkolwiek jego samodzielna instalacja na własnym serwerze jest dość wymagająca [3],

- GitLab CI - może być używany w postaci SaaS. Oferuje on jedynie integrację z repozytorium hostowanym przez GitLaba,
- GitHub Actions - rozwiązanie do automatyzacji zaproponowane przez GitHuba. Zapewnia integrację z repozytorium kodu hostowanym na GitHubie. Jest zamknięto-źródłowym projektem,
- CircleCI - rozwiązanie podobne do Travis CI. Różnica polega na tym, że CircleCI zapewnia integrację tylko z repozytorium hostowanym na GitHubie. Dodatkowo jego kod nie jest dystrybuowany w formie open source - jest zamknięty.

2. Wirtualizacja i orkiestracja

myślę, że opisanie tego jak wirtualizacja pomaga w CI/CD zasługuje na swój rozdział. Szczególnie mógłbym ten rozdział poświęcić dockerowi z faktu, że zdominował rynek. Warto tutaj byłoby napisać też o rozwiązaniach które były używane w przeszłości a zostały wyparte jak vagrant. Dodatkowo w rozdziale mozemy opisać na czym polega orkiestracja - Kubernates, docker swarm

3. Hudson oraz Jenkins - klasyczne narzędzia do CI/CD

tutaj mógłbym zamieścić trochę informacji historycznych jak język java zrewolucjonował inżynierię oprogramowania i dalej rozwinąć, że pokłosiem tego było powstanie Jenkinsa. Myślę, że mógłbym tutaj opisać najbardziej uniwersalne rzeczy które są w nim używane. Wydaje mi się że mógłbym poszukać alternatyw do Jenkinsa by rozszerzyć ten dział. Jako część tego rozdziału mógłbym zawrzeć problem gdzie konieczne jest użycie Jenkinsa i to byłoby zrobione jako część praktyczna. Wbrew pozorom do niektórych zastosować Jenkins nadal jest używany, np do środowisk gdzie QA musi odpalać środowisko z różnymi parametrami

4. Platformy SaaS z wbudowanym CI/CD

4.1. Z czego wynika popularność platform SaaS?

Platformy takie jak GitHub, CircleCI czy też BitBucket zasłynęły z tego, że oferowały możliwość hostowania naszego repozytorium Git'owego w chmurze. Z biegiem czasu zaczęły one oferować dodatkowe usługi. Serwisy te to już nie tylko miejsce, które umożliwia hostowanie naszego repozytorium kodu - pozwalają one na takie rzeczy jak:

- Odpalanie procesów automatyzujących (GitHub, GitLab, BitBucket) - dzięki możliwości tworzenia własnych procesów automatyzujących mamy możliwość zaimplementować własny proces ciągłej integracji, ciągłego dowożenia czy też ciągłego dostarczania,

- Odpalanie manualne predefiniowanych procesów (GitLab) - jest to dość prosta funkcjonalność, która daje sporo możliwości. GitLab dostarcza interfejs, w którym możemy uruchomić dowolny pipeline (z ang. *potok*) manualnie - dodatkowo możemy podać argumenty w postaci pól tekstowych. Przykładem, gdzie ta funkcja GitLab'a byłaby użyteczna, jest aplikacja webowa, która zabiera dużo zasobów. Użycie procesu ciągłego dostarczania, by uruchamiać wersję staging'ową oznaczałoby, że aplikacja ta zbędnie używałaby zasoby na serwerze. Dzięki funkcji odpalania manualnego pipeline'ów moglibyśmy uruchamiać aplikację tylko wtedy, kiedy chcielibyśmy przetestować jak działa,

- Integracja z Kubernetes'em (GitLab) - GitLab upraszcza proces release'u naszego oprogramowania dzięki integracji z Kubernetes'em. Możemy podpiąć istniejący już klaster albo stworzyć nowy. GitLab może stworzyć taki klaster za nas pod warunkiem, że będzie on działał na chmurze AWS albo Google Cloud,

- Publikacje strony statycznej (GitHub) - GitHub umożliwia publikacje danego folderu z plikami statycznymi strony internetowej, który jest częścią repozy-

torium. Dzięki temu możemy uprościć proces publikacji naszej strony internetowej,

- Hosting obrazów Docker'owych oraz paczek/bibliotek (GitHub, GitLab, BitBucket) - dzięki hostingowi obrazów Docker'owych możemy uprościć proces późniejszego odpalenia naszego kodu na serwerze. Dodatkowo możemy hostować na tych platformach paczki popularnych języków. Wspierane są między innymi: NPM (nodeJS), Maven (Java), NuGet (.NET), RubyGems (Ruby),
- Śledzenia zadań (GitHub, GitLab, BitBucket) - każda ze znaczących platform ma wbudowane w sobie oprogramowanie do śledzenia zadań. Dzięki temu możemy stworzyć prostą metodykę Agile'ową używając tego, co zapewnia nam dana platforma. Z reguły oprogramowanie to jest mocno ograniczone i nadaje się do prostszych projektów. Zespoły, które mają większe wymagania, muszą szukać oddzielnego oprogramowania,
- Audyt bezpieczeństwa bibliotek (GitHub, GitLab) - dzięki tej funkcjonalności możemy się dowiedzieć, czy nasze zależności trzecie nie posiadają luk bezpieczeństwa. GitHub lub GitLab w przypadku wykrycia takiego problemu wysyła maila z powiadomieniem oraz tworzy automatyczną poprawkę,
- Sponsoring (GitHub) - GitHub posiada wbudowaną opcję która włącza sponsoring naszego projektu. Dzięki temu na stronie głównej repozytorium pojawia się ikonka serca, która pozwala wesprzeć dany projekt pieniężnie. Jest to ciekawa opcja dla projektów open source.

Powyższe funkcjonalności czynią te platformy narzędziami all-in-one (z ang. wszystko w jednym). Dzięki temu, że są one oferowane jako serwisy SaaS, czas który musimy poświęcić na utrzymanie i instalację ogranicza się do zera.

Należy pamiętać, że użycie platform SaaS ma dwie strony medalu. Jeżeli nasz projekt jest publiczny, większość z powyższych usług jest oferowana za darmo. Natomiast jeżeli nasz projekt posiada kod, który jest prywatny, będziemy musieli uiścić odpowiednią opłatę. Zazwyczaj opłata ta jest uzależniona od liczby usług, z których będziemy korzystać oraz od minut automatycznych procesów, które zużyjemy. Platformę na której będziemy działać należy dobrać do indywidualnych potrzeb danego projektu - każda z nich oferuje swoiste ekskluzywne funkcjonalności. Jeżeli naszym

celem jest stworzenie statycznej strony, najlepszym wyborem wydaje się GitHub, który oferuje darmowy hosting. Z drugiej strony jeśli zależy nam na uruchamianiu procesów automatyzujących manualnie to powinniśmy spojrzeć w stronę GitLab'a. GitHub jest świetnym rozwiązaniem dla projektów open source, ponieważ większość dodatkowych usług dla takich projektów jest oferowana za darmo. Wynika to z polityki, która została obrana przez właściciela - Microsoft.

4.2. Strona statyczna - przykład użycia GitHub Actions i GitHub Pages

Załóżmy, że chcielibyśmy stworzyć stronę internetową, która pozwoliłaby nam przedstawić kim jesteśmy. Mamy następujące wymagania:

- Strona ma być statyczna. Dzięki temu będziemy mogli wykorzystać hosting plików, który często jest oferowany jako darmowa opcja,
- Strona powinna mieć co najmniej dwie podstrony, które zawierają wspólne sekcje: nagłówek oraz stopkę,
- Strona ma być dostępna w sieci za pomocą promowanego przez przeglądarki protokołu HTTPS.

By ułatwić sobie powyższe zadanie, użyjemy framework'a Gatsby.js. Jest to generator stron statycznych, który upraszcza tworzenie takowych stron. Framework używa popularną bibliotekę React, która pozwala korzystać z składni JSX (składnia mocno przypomina format HTML). Główną cechą Gatsby'iego jest to, że zapewnia integracje z popularnymi systemami takimi jak Wordpress [4]. Dzięki temu nie jesteśmy zmuszeni utrzymywać dedykowanego serwera PHP - Gatsby wygeneruje wszystkie możliwe podstrony podczas procesu budowania strony. Jest to szczególnie ciekawe rozwiązanie dla serwisów, które generują duży ruch. Znacznie łatwiej serwować pliki statyczne niż za każdym razem budować stronę, obciążając dodatkowo bazę danych.

Gatsby.js w postaci programu działającego w wierszu poleceń pozwala w prosty sposób pobrać startowy szablon z minimalnym zbiorem potrzebnych rzeczy. Użyjmy tego szablonu jako punktu wejściowego:

```
gatsby new static-website-with-ci-cd https://github.com/
    gatsbyjs/gatsby-starter-default
```

Listing 2: Pobierania szablonu startowego

Domyślny szablon posiada w dużej mierze gotowy layout z nagłówkiem oraz stopką. Jest on zdefiniowany w pliku 'src/components/layout.js'. Po małej przeróbce wygląda on następująco:

```
const Layout = ({ children }) => {
    const data = useStaticQuery(graphql`
      query SiteTitleQuery {
        site {
          siteMetadata {
            title
          }
        }
      }
    `)

    return (
      <>
        <Header siteTitle={data.site.siteMetadata?.title || `
            Title`} />
        <div
          style={{
            margin: `0 auto`,
            maxWidth: 960,
            padding: `0 1.0875rem 1.45rem`,
          }}
        >
          <main>{children}</main>
          <footer style={{
            marginTop: `2rem`
          }}>
            Stopka
          </footer>
        </div>
      </>
    )
```

```
}
```

Listing 3: Layout - komponent zawierający logikę związaną z layoutem strony

Dzięki temu, że zawartość sekcji *<main>* jest konfigurowalna za pomocą argumentu *children*, możemy współdzielić layout między różnymi podstronami.

Pierwszą stroną, którą chcielibyśmy stworzyć, jest strona domowa. Na niej wyświetlimy podstawowe informacje o nas oraz umieścimy link do strony, na której przedstawimy więcej danych o sobie. Kod tej strony jest stosunkowo prosty:

```
import React from "react"
import { Link } from "gatsby"
import Layout from "../components/layout"
import SEO from "../components/seo"
import MeImgSrc from "../images/ja.jpg"

const IndexPage = () => (
  <Layout>
    <SEO title="Home" />
    <div>
      <h2>Artur Kasperek - Programista oraz Student Politechniki
         Slaskiej</h2>
      <img src={MeImgSrc}/>
    </div>
    <div>
      <Link to="/detale">Wiecej o mnie</Link> <br />
    </div>
  </Layout>
)

export default IndexPage
```

Listing 4: index.js - plik zawiera treść strony domowej

Drugą stroną jest podstrona, która wyświetla więcej szczegółów. Podobnie jak na stronie domowej zawiera link do głównej strony. Zawartość wygląda następująco:

```
import React from "react"
import { Link } from "gatsby"

import Layout from "../components/layout"
import SEO from "../components/seo"
```

```
const SecondPage = () => (
  <Layout>
    <SEO title="Detale" />
    <h2>Doswiadczenie</h2>
    <ul>
      <li>JavaScript</li>
      <li>HTML</li>
      <li>CSS</li>
      <li>C++</li>
      <li>NodeJS</li>
    </ul>
    <h2>Szkoly</h2>
    <ul>
      <li>Politechnika Slaska</li>
      <li>I Liceum Ogolnoksztalcacego im. Stefana Zeromskiego w
          Zawierciu</li>
      <li>Gimnazjum nr 1 Zawierciu</li>
    </ul>
    <Link to="/">Powrot do strony domowej</Link>
  </Layout>
)

export default SecondPage
```

Listing 5: detale.js - plik zawiera treść strony z szczegółami

Strona może być teraz opublikowana. Do tego celu służy komenda *gatsby build*. Komenda ta buduje stronę i umieszcza ją w folderze *public*. Chcielibyśmy teraz w jakiś sposób opublikować stronę w sieci. W tym celu możemy skorzystać z usługi GitHub Pages. Wymogiem jest tutaj to, by nasz kod strony internetowej był repozytorium Git'owym trzymanym na GitHub'ie. Po tym, gdy nasz projekt jest już repozytorium Git'owym na GitHub'ie, możemy stworzyć definicję potoku ciągłego dowożenia. Chcielibyśmy, by za każdym razem, gdy nowy kod zostaje wysłany na gałąź *master*, strona się budowała i była publikowana w sieci.

GitHub Actions posiada własną składnię do definicji potoków automatyzujących. Najmniejszą jednostką, którą możemy zdefiniować jest *workflow* (z ang. *przepływ*). W dokumentacji [5] możemy dowiedzieć się, że *workflow* to konfigurowalny zautomatyzowany proces składający się z co najmniej jednego zadania oraz pliku YAML, który zawiera jego definicję. Warto podkreślić, że plik ten jest częścią naszego pro-

jektu - wynika z tego, że jest on wersjonowany wraz z kodem źródłowym i innymi plikami.

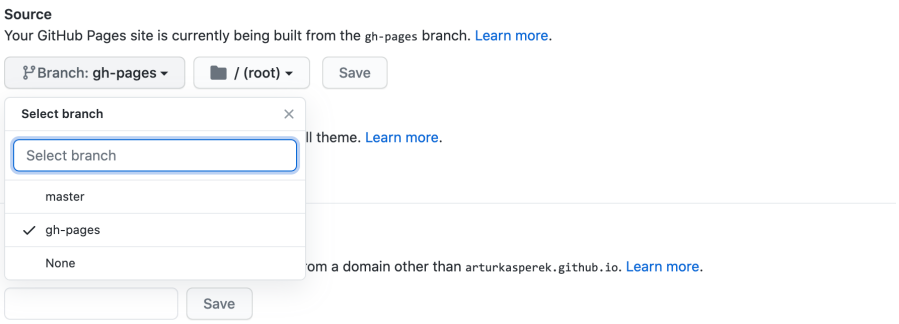
Plik YAML danego *workflow* zawiera informację o swojej nazwie, to kiedy ma się odpalić oraz definicję tego co ma wykonać. W naszym przypadku chcemy, by nasz *workflow* wykonywał się za każdym razem, gdy wrzucamy jakieś zmiany na master'a. Możemy to osiągnąć za pomocą poniższego fragmentu kodu:

```
name: Build and Deploy

on:
  push:
    branches: [ master ]
```

Listing 6: Nazwa oraz definicja wyzwalacza *workflow* budującego stronę

Kolejnym krokiem jest zdefiniowanie to, co nasz *workflow* ma robić. W naszym przypadku chcielibyśmy zbudować stronę oraz opublikować ją za pomocą GitHub Pages. W tym celu w ustawieniach projektu włączamy GitHub Pages oraz definiujemy, w jakim folderze oraz na jakiej gałęzi będą dostępne pliki statyczne strony. W naszym przypadku wybieramy gałąź *gh-pages* oraz główny folder. Dzięki temu



Rysunek 6: Ustawienia GitHub Pages

będziemy w stanie odseparować gałąź, gdzie trzymamy kod źródłowy strony, od zbudowanej statycznej strony. Jest to także ułatwienie dla potoku automatyzującego. Jeżeli chcielibyśmy zapisywać zbudowaną stronę na gałęzi *master*, oznaczałoby to, że nasz *workflow* wyzwalałby się w nieskończoność przez fakt, iż jest on uzależniony od tego, czy jakieś nowe zmiany zostały wysłane na gałąź *master*.

Definicja tego, co nasz proces ciągłego dowożenia ma robić, wygląda następująco:

```
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Use Node.js 14.x
      uses: actions/setup-node@v1
      with:
        node-version: 14.x
    - name: Install Dependencies
      run: |
        yarn install
    - name: Build website
      run: |
        yarn build
    - name: Deploy
      uses: JamesIves/github-pages-deploy-action@3.6.2
      with:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        BRANCH: gh-pages
        FOLDER: public
        CLEAN: true
```

Listing 7: Definicja zadań *workflow*'a budującego stronę

Jedną z rzeczy, które wyróżniają GitHub Actions nad innymi systemami ciągłej integracji/ciągłego dowożenia, jest możliwość definiowania tytułowych akcji. Akcje to nic innego jak sprytnie zapakowane programy lub też skrypty bash'owe, które pod spodem wykonują jakąś skomplikowaną rzecz. W zależności od argumentów środowiskowych możemy daną akcję dostosować do naszych potrzeb. Z faktu, że każdy może publikować własne akcje, nie jesteśmy ograniczeni do korzystania tylko z akcji dostarczonych przez twórców GitHub Actions. Dzięki takiemu podejściu mamy dostęp do bogatej bazy gotowych modułów.

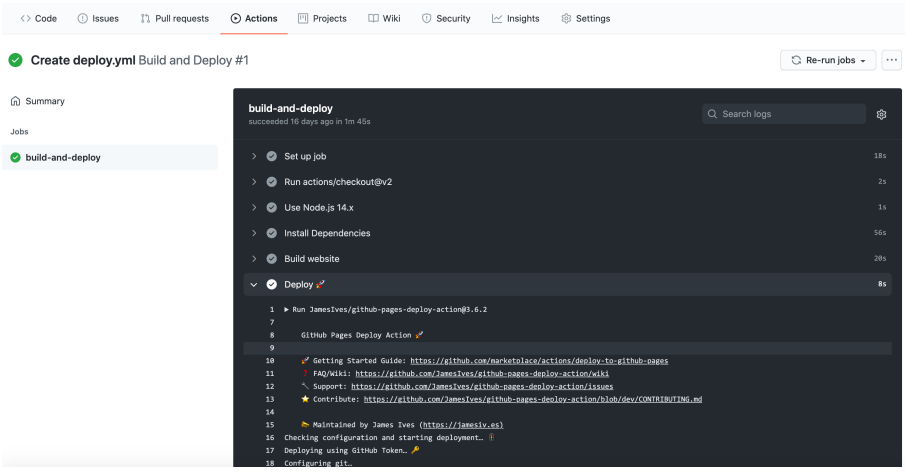
Wyjaśnijmy teraz, co dane linijki robią. W linii 3 definiujemy to, jaki system chcemy użyć jako bazowy. W naszym przypadku używamy Ubuntu, ponieważ jest najbardziej „lekkim" z wszystkich możliwych systemów. Od linijki 4 do końca mamy definicję kroków, co dany *job* (z ang. *zadanie*) ma wykonać. Pierwszymi krokami jest przełączenie się do danego branch'a oraz użycie akcji, która zainstaluje nam środowisko nodeJS w wersji 14. Od linii 10 do 15 zdefiniowane są dwa kroki, których celem

jest zainstalowanie zależności naszego projektu oraz jego zbudowanie. W tym celu używamy menadżera paczek *yarn*, który został zainstalowany dodatkowo podczas instalacji nodeJS. Kroki te pokazują, że nie jesteśmy uzależnieni tylko od użycia gotowych akcji. Za pomocą słowa kluczowego *run* możemy zdefiniować dowolną komendę bash'ową, która ma być wykonana w danym momencie. Warto podkreślić fakt, że zmienne środowiskowe, które były zdefiniowane w danym kroku, nie są dostępne dla kolejnych kroków. GitHub Actions odpala każdy krok w osobnej sesji bash'owej. Na ten moment nasza strona powinna być wybudowana i zapisana w folderze *public*.

Ostanim krokiem jest publikacja strony na gałęzi *gh-pages*. W tym celu używamy gotowej akcji o nazwie *JamesIves/github-pages-deploy-action* w wersji 3.6.2. Jest to przykład akcji, która została dostarczona przez trzeciego autora i jest udostępniona szerokiej publiczności. Akcja ta „pod spodem" za pomocą git'a wysyła dany folder na wyspecjalizowaną za pomocą parametru *BRANCH* gałąź. Dodatkowo musimy podać w parametrach akcji token dostępu do GitHuba. Robimy to za pomocą parametru *GITHUB_TOKEN*. Token ten jest automatycznie generowany przez GitHub'a podczas odpalania danego *workflow*'a i daje możliwość zapisywania zmian do repozytorium, gdzie *workflow* jest odpalane. Tym sposobem akcja dostaje niezbędne prawa do wysłania folderu z wybudowaną stroną na gałąź *gh-pages*, która jest częścią tego samego repozytorium. Dzięki temu, że wykorzystaliśmy gotową akcję do publikacji naszej strony, ograniczyliśmy długość kodu oraz zredukowaliśmy czas, który musielibyśmy poświęcić na stworzenie skryptu, który wysyła folder *public* na odpowiednią gałąź.

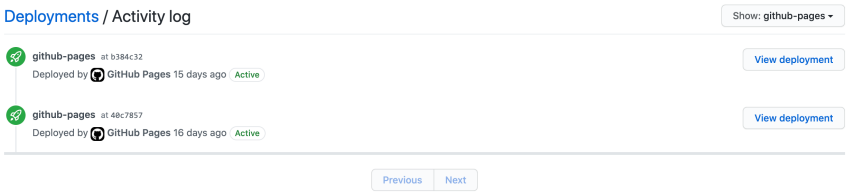
Teraz, gdy nasze repozytorium posiada plik konfiguracyjny GitHub Actions, w zakładce *Action* na GitHub'ie powinniśmy zauważyć zakolejkowane zadanie budowania strony. Sumarycznie pierwsze budowanie strony zajęło 1min 45sec. Najwięcej czasu zabrało zainstalowanie zależności - aż 56 sekund. Na szczęście GitHub zapewnił akcję, która pozwala cache'ować zależności. Warunkiem koniecznym do działania tego mechanizmu jest posiadanie pliku z informacjami o wersjach zależności jakich używamy. Nie powinno być to problemem, ponieważ większość współczesnych menadżerów zależności generuje taki plik. Implementując cache'owanie, proces automatyzujący będzie tylko instalował zależności, jeżeli jakaś ich wersja się zmieni - w reszcie przypadków proces używa plików z cache'a.

Po tym, gdy pliki z stroną „wylądowały" na gałęzi *gh-pages*, GitHub powinien



Rysunek 7: Widok na którym możemy zobaczyć szczegóły danego zadania

zakolejkować proces publikacji strony. Jest to rzecz, nad którą nie mamy kontroli.



Rysunek 8: Widok z publikacjami strony

GitHub wewnętrznie bierze pliki strony z gałęzi, którą ustawiliśmy, i publikuje to na swoich serwerach. Finalnie omawiana strona jest dostępna pod adresem *https://arturkasperek.github.io/static-website-with-ci-cd/*. GitHub pages samo w sobie nie ogranicza nas do używania domeny *github.io*. Jeżeli posiadamy własną domenę, możemy odpowiednio tak przekierować ruch do GitHub'a, że finalny użytkownik nie będzie wiedział, gdzie strona jest hostowana.

Dzięki zapewnieniu modułowości, GitHub Actions może pochwalić się dużą biblioteką akcji stworzoną przez społeczeństwo. Oto lista ciekawych akcji:

- jakejarvis/s3-sync-action - jest to akcja, która pozwala synchronizować dany folder repozytorium z folderem na usłudze hostingowej AWS S3. Może być użyteczna, jeżeli chcielibyśmy danej grupie deweloperów dać możliwość wysyłania plików na S3, bez konieczności dawania dostępu do panelu administratora AWS,

- repo-sync/github-sync - akcja ta potrafi synchronizować dane repozytorium z innym repozytorium dostępnym w sieci. Jest ona szczególnie użyteczna, gdy pracujemy nad fork'iem (fork to kopia projektu która rozwija się niezależnie względem oryginału) jakiegoś projektu i chcemy utrzymać zgodność z oryginałem. GitHub Actions pozwala na odpalanie danego procesu automatyzującego automatycznie na podstawie planera. Dzięki temu możemy wykorzystać tą akcję, by co jakiś czas synchronizowała nasze repozytorium z macierzystym projektem,

- release-drafter/release-drafter - akcja jest szczególnie użyteczna, jeżeli nasz projekt chce korzystać z dobrych praktyk, dotyczących publikacji oprogramowania. Akcja ta oblicza, jakie zmiany w kodzie nastąpiły od ostatniej publikacji i tworzy wstępną publikację na GitHub'ie z opisem zmian, które dokonaliśmy. Akcja ta jest oparta na commit'ach, dlatego warto by one były dobrze opisane. Jeżeli użyjemy odpowiednich prefixów jak *feature* oraz *bug*, akcja będzie w stanie lepiej sformatować opis publikacji,

- zaproxy/action-baseline - jest to akcja, która jest oparta o użycie narzędzia ZAP - to program, który analizuje nasz kod pod względem różnorakich luk bezpieczeństwa. Finalnie jeżeli akcja w wyniku swojego działania znajdzie jakieś podatności to tworzy automatycznie *issue* (system na GitHub'ie, który pozwala śledzić błędy). Jest to ciekawa opcja, jeżeli chcemy jak najbardziej zabezpieczyć się przed ewentualnymi atakami hackerów.

Wszystkie powyższe rzeczy moglibyśmy wykonać, używając akcji, która odpala skrypt bash'owy. Takie podejście jednak miałoby sporo wad - bylibyśmy zmuszeni spędzić sporo czasu, by wszystko zgrać tak jak chcemy. Dodatkowo prawdopodobnie nie pokrylibyśmy różnych przypadków brzegowych. Dzięki gotowym akcjom czas konfiguracji środowiska do automatyzacji skraca się do minimum, a my możemy się skupić na rozwiązywaniu innych problemów.

5. Testy a continues integration

mógłbym tutaj powiedzieć więcej co to są testy jednostkowe, integracyjne oraz e2e. Następnie mógłbym powiedzieć trochę więcej, że dzięki poprawnemu CI nasza aplikacja będzie miała mniej błędów oraz będzie trudniej o regresję. Tutaj mógłbym nawiązać do części praktycznej i opisać problem w którym przydają się testy e2e. Myślę, że mógłbym te testy napisać w niejakim frameworku cypress.js i skonfigurować to na GitHub actions,

6. Podsumowanie

myślę, że cały wydźwięk podsumowania powinien być nastawiony na to że dzięki CI/CD programiści mogą efektywnie działać, ich kod szybko znajduje się na wersji produkcyjnej i dzięki odpowiednio skonfigurowanemu workflow mogą być bardziej pewni, że ich zmiany nie zepsują istniejących funkcjonalności

Literatura

[1] Kent Beck; James Grenning; Robert C. Martin; Mike Beedle; Jim Highsmith; Steve Mellor; Arie van Bennekum; Andrew Hunt; Ken Schwaber; Alistair Cockburn; Ron Jeffries; Jeff Sutherland; Ward Cunningham; Jon Kern; Dave Thomas; Martin Fowler; Brian Marick (2001). "Principles behind the Agile Manifesto"

[2] Bass, Len; Weber, Ingo; Zhu, Liming (2015). DevOps: A Software Architect's Perspective

[3] Mathias Meyer (2015) "How We Improved the Installation and Update Experience for Travis CI Enterprise" https://blog.travis-ci.com/2015-06-19-how-we-improved-travis-ci-installation/

[4] Praca zbiorcza twórców GatsbyJS "Sourcing from WordPress" https://www.gatsbyjs.com/docs/sourcing-from-wordpress/

[5] Autor nieznany "GitHub Actions Reference" https://docs.github.com/en/free-pro-team@latest/actions/reference