МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение  
высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

**Кафедра информатики и автоматизации научных исследований**

Направление подготовки: «Прикладная информатика»

Профиль подготовки: «Прикладная информатика в области принятия решений»

**ОТЧЕТ**

по технологической (проектно-технологической) практике

на тему:

**«Автоматизация задач на примере GitHub Actions»**

**Выполнил:** студент группы 381907–1

Мокин Илья Владиславович

Подпись:

**Научный руководитель:**

Доцент, кандидат технических наук

Штанюк Антон Александрович

Подпись:

Нижний Новгород  
2022

Оглавление

[Введение 3](#_Toc96540002)

[Реализация приложения 3](#_Toc96540003)

[Организация GitHub Actions 6](#_Toc96540004)

[Заключение 8](#_Toc96540005)

[Список литературы: 10](#_Toc96540006)

[Приложения: 11](#_Toc96540007)

# Введение

В современном мире комплексность и сложность задач, которые нацелено решать ПО, приводит к большим временным затратам на его разработку. Одним же из основных способов сокращения этих затрат является автоматизация. Уместная даже при разработке простых проектов, она главным образом применима при выполнении сложных задач, которые можно разбить на подзадачи, некоторые из которых и поддаются автоматизации. Одной из таких задач является тестирование.

В данной работе пример автоматизации тестирования будет строиться на использовании GitHub и инструментов автоматизации, которые он предоставляет. Речь идет о механизме GitHub Actions, позволяющем выполнять различные задачи над содержимым репозиториев без вмешательства человека.

Введем следующую задачу:

Используя GitHub Actions, протестировать программный код пользователя, хранимый локально, используя тесты из его онлайн репозитория, после этого получить результаты работы GitHub Actions и тестов, оставляя онлайн репозиторий без изменений.

Для решения данной задачи создадим приложение на .NET Core 3.1, функционал которого будет описан далее и настроим GitHub Actions для репозитория пользователя.

Краткая суть приложения:

1 – Создать git репозиторий и добавить в него файлы для тестирования;

2 – Создать новую ветку в git репозитории и загрузить ее в онлайн репозиторий пользователя;

3 – Получить результаты работы GitHub Actions и тестов.

GitHub Actions же потребуется настроить так, чтобы можно было:

1 – Провести тестирование программного кода, отправленного пользователем;

2 – Получить доступ к результатам работы GitHub Actions и тестов;

3 – Отменить все изменения в репозитории.

# Реализация приложения

После копирования файлов с помощью функции Copy(), приведенной в приложениях, мы можем приступить к инициализации git репозитория и работе с ним. Для этого воспользуемся библиотекой LibGit2Sharp:

Repository.Init(gitRepoDir);

Repository repo = new Repository(gitRepoDir);

Repository.Init() аналогична команде git init. После инициализации репозитория мы создаем объект класса Repository для дальнейшей работы.

Используя класс Repository, мы можем исполнять все команды git в нашем репозитории в программном коде:

Commands.Stage(repo, "\*");

repo.Branches.Add("temp", repo.Commit("sampletext", new Signature(username, email, DateTimeOffset.Now), new Signature(username, email, DateTimeOffset.Now), new CommitOptions()));

Commands.Checkout(repo, "temp");

Commands.Stage() добавляет в репозиторий файлы, которые мы хотим протестировать. Repo.Branches.Add() создает новую ветку в репозитории, а Commands.Checkout() делает её текущей рассматриваемой.

После этого нам нужно связать локальную ветку и GitHub репозиторий. Для этого воспользуемся LibGit2Sharp аналогом “git remote add”:

repo.Network.Remotes.Add("origin2", url);

Remote remote = repo.Network.Remotes["origin2"];

repo.Branches.Update(repo.Branches["temp"], b => b.Remote = remote.Name,

b => b.UpstreamBranch = "temp");

Функция Branches.Update() связывает ветку с remote.

Перед тем, как “запушить” ветку в GitHub репозиторий, надо провести аутентификацию пользователя:

var creds = new UsernamePasswordCredentials()

{

Username = USERNAME,

Password = PASSWORD

};

CredentialsHandler credHandler = (\_url, \_user, \_cred) => creds;

var options = new PushOptions() { CredentialsProvider = credHandler };

В качестве “USERNAME” следует указать имя аккаунта GitHub, а в качестве пароля рекомендуется использовать Personal Access Token(PAT).

Ну и теперь можно использовать push:

repo.Network.Push(remote, @"refs/heads/temp", options);

В связи с тем, что в репозитории не существует ветки “temp”, данная команда создаст новую ветку.

Для облегчения многократных запусков приложения предусмотрено удаление git репозитория:

var di = new DirectoryInfo(@"C:\testDirectory2");

Delete(di);

Directory.Delete(@"C:\testDirectory2", true);

Функция Delete(), приведенная в приложениях, удаляет атрибут read-only, позволяя удалить репозиторий.

После создания в GitHub репозитории ветки, содержащей код для тестирования, нам остается ждать окончания работы GitHub Actions. В приложении это реализовано посредством простого Thread.Sleep().

После определенного времени мы можем приступать к получению результатов работы GitHub Actions и тестирования.

Для доступа к этим данным воспользуемся GitHub API, позволяющим скачать так называемые “artifacts”, которые GitHub Actions может оставлять после своей работы. Для скачивания воспользуемся классом “HttpClient”:

HttpClient clienthttp;

clienthttp = new HttpClient();

clienthttp.BaseAddress = new Uri("https://api.github.com");

clienthttp.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Token", PASSWORD);

clienthttp.DefaultRequestHeaders.Add(

"User-Agent",

"Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2228.0 Safari/537.36");

clienthttp.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));

По завершении настройки HttpClient мы можем скачать артефакт, который описывает статус нужного нам workflow. Но для начала нам нужно узнать адрес необходимого артефакта. Для этого используем api для обзора всех артефактов репозитория:

query = "https://api.github.com/repos/user/reponame/actions/artifacts";

HttpResponseMessage response = await clienthttp.GetAsync(query);

string responseBody = await response.Content.ReadAsStringAsync();

Таким образом мы получаем .json, который нам еще предстоит десериализировать. Для этого воспользуемся библиотекой NewtonsoftJson:

var RCR = JsonConvert.DeserializeObject<Root>(responseBody);

Теперь, зная название нужного артефакта, можно его скачать:

var response2 = await clienthttp.GetAsync(@concurl + "?filename=conc.zip");

using (var stream = await response2.Content.ReadAsStreamAsync())

{

var fileInfo = new FileInfo("conc.zip");

using (var fileStream = fileInfo.OpenWrite())

{

await stream.CopyToAsync(fileStream);

}

}

Переменная “concurl” отвечает за url адрес артефакта.

Артефакты можно скачать только в формате .zip, поэтому их еще предстоит распаковать:

ZipFile.ExtractToDirectory("conc.zip", "conc.txt");

Если workflow завершился успешно, а это найдет отражение в артефакте, мы можем скачать результат теста:

string text = System.IO.File.ReadAllText(@"conc.txt\conclusion.txt");

if (text.Contains("0+0")|| text.Contains("0+2"))

{

try

{

var response1 = await clienthttp.GetAsync(@durl + "?filename=test-results.zip");

using (var stream = await response1.Content.ReadAsStreamAsync())

{

var fileInfo = new FileInfo("test-results.zip");

using (var fileStream = fileInfo.OpenWrite())

{

await stream.CopyToAsync(fileStream);

}

}

Console.WriteLine("Результат тестирования получен");

}

}

If (text.Contains(“0+0”)||text.contains(“0+2”)) просто проверяет содержимое артефакта на наличие в нем ключей, удовлетворяющих положительному завершению workflow.

# Организация GitHub Actions

Теперь поговорим о GitHub Actions. Как уже было сказано, этот инструмент позволяет нам совершать различные действия над объектами, находящимися в репозитории, к которому у нас есть доступ, автоматически. Эти действия могут совершаться при любом событии, произошедшем в вашем репозитории. Для создания workflow – элемента GA, выполняющего последовательность операций – нам нужно в папке .github/workflows нашего репозитория создать файл расширения .yml, который и будет описывать работу workflow.

Для организации работы GitHub Actions в наших целях вместе с кодом на проверку нам еще предстоит передать файл .yml, который поможет запустить нужные нам workflow. Главное, что нужно знать об этом файле, это то, что он запускает первый workflow (назовем его “OnCreate”) в момент образования новой ветки. В самом же workflow помимо этого ничего не делается. Он нужен для того, чтобы workflow, описанные в main ветке репозитория начали свою работу.

Перейдем к рассмотрению workflow “Copy”, используемого при решении нашей задачи. Как уже было сказано, он начинает свою работу после OnCreate:

on:

workflow\_run:

workflows: ['OnCreate']

types:

- completed

Его основная цель – перенести проект с тестами в новую ветку:

- name: Copycat1

uses: andstor/copycat-action@v3

with:

personal\_token: ${{ secrets.TOKEN }}

src\_path: /tests/.

dst\_path: /.

dst\_owner: user

dst\_repo\_name: repo

dst\_branch: temp

src\_branch: main

В нем используется готовое решение “adstor/copycat-action”, которое и осуществляет копирование.

По завершении работы этого workflow(механизм устроен точно так же) начинает свою работу следующий – “BuildAndTest”.

Как понятно из названия, в нем происходит сборка и тестирование программного кода с помощью тестов из репозитория:

- name: Build

continue-on-error: true

id: step3

run: |

dotnet build ./myNewFolder/t.sln

- name: Test

continue-on-error: true

id: step4

run: |

dotnet test XUnitTestProject1.csproj --logger "trx;LogFileName=test-results.trx"

Кроме того, в этом workflow собирается информация о пройденных шагах:

- name: WorkflowStatus1

id: conclusion

if: |

steps.step1.outcome == 'success' &&

steps.step2.outcome == 'success' &&

steps.step3.outcome == 'success' &&

steps.step4.outcome == 'success' &&

steps.step5.outcome == 'success'

shell: bash

run: |

expr 0+0 > conclusion.txt

Код, описывающий ситуацию, когда в workflow возникает ошибка, и ситуацию, когда workflow был верен, но код не прошел тесты, можно найти в приложениях.

Сбор информации нужен для того, чтобы использовать её как артефакт, то есть объект, создаваемый workflow, который можно использовать как в других workflow, так и вне GitHub Actions.

Загрузка артефакта в репозиторий совершается следующим образом:

- name: Upload conclusion

uses: actions/upload-artifact@v2

with:

name: conc

path: ./\*\*/\*.txt

По завершении “BuildAndTest” запускается следующий workflow, который публикует результат тестирования(в нашей задаче не используется) и удаляет ветку, добавленную ранее:

jobs:

report:

runs-on: ubuntu-latest

steps:

- uses: dorny/test-reporter@v1.5.0

with:

artifact: test-results

name: dotnet-trx

path: ./\*\*/TestResults/\*.trx

reporter: dotnet-trx

deletebranch:

runs-on: ubuntu-latest

steps:

- name: Delete branch

uses: dawidd6/action-delete-branch@v3

with:

github\_token: ${{github.token}}

branches: temp

Функции dorny/test-reporter и dawidd6/action-delete-branch являются готовыми решениями, которые выводят результат тестов и удаляют ветку соответственно.

Из особенностей работы workflow следует отметить возможность выбора операционной системы, на которой будут выполняться действия. Для каждой “job” из каждого workflow нужно задать операционную систему(при выполнении задания использовалась ubuntu), от которой зависит, например, скорость выполнения workflow.

В качестве триггеров workflow можно использовать schedule. Однако этот триггер сильно зависит от нагрузки серверов. В случае, когда сервера перегружены он может сработать с запозданием, а иногда и вовсе быть проигнорирован.

# Заключение

Таким образом, был рассмотрен пример автоматизации тестирования с использованием GitHub Actions.

Несмотря на то, что организация автоматизации тех или иных этапов разработки, например тестирования, сама по себе занимает время, она может сэкономить его в долгосрочной перспективе и, что важнее, предотвратить ошибки, связанные с многократным выполнением человеком одних и тех же действий.

# Список литературы:

Документация GitHub по GitHub Actions [Электронный ресурс]: руководство пользователя/GitHub, Inc.// docs.GitHub: [сайт]. – Режим доступа: <https://docs.github.com/en/actions>;

Библиотека LibGit2Sharp [Электронный ресурс]: динамически подключаемая библиотека/LibGit2// GitHub: [сайт]. – Режим доступа: <https://github.com/libgit2/libgit2sharp>;

Библиотека Newtonsoft.Json [Электронный ресурс]: динамически подключаемая библиотека/Newtonsoft// GitHub: [сайт]. – Режим доступа: <https://github.com/JamesNK/Newtonsoft.Json>;

# Приложения:

Код функций:

Copy:

static void Copy(string sourceDir, string targetDir)

{

Directory.CreateDirectory(targetDir);

foreach (var file in Directory.GetFiles(sourceDir))

File.Copy(file, Path.Combine(targetDir, Path.GetFileName(file)));

foreach (var directory in Directory.GetDirectories(sourceDir))

Copy(directory, Path.Combine(targetDir, Path.GetFileName(directory)));

}

Delete:

static void Delete(DirectoryInfo deleteDir)

{

deleteDir.Attributes &= ~FileAttributes.ReadOnly;

foreach (var dir in deleteDir.GetDirectories())

{

Delete(dir);

dir.Attributes &= ~FileAttributes.ReadOnly;

}

foreach (var fi in deleteDir.GetFiles())

{

fi.Attributes = FileAttributes.Normal;

}

}

Код workflow:

BuildAndTest:

name: BuildAndTest

on:

workflow\_run:

workflows: ['Copy']

types:

- completed

workflow\_dispatch:

jobs:

build:

runs-on: ubuntu-latest

strategy:

matrix:

dotnet-version: [ '3.1.x' ]

steps:

- uses: actions/checkout@v2

with:

ref: temp

- name: Setup dotnet

uses: actions/setup-dotnet@v1.7.2

with:

dotnet-version: ${{ matrix.dotnet-version }}

- name: Create folder

run: |

mkdir myNewFolder

- name: Setup NuGet.exe for use with actions

uses: NuGet/setup-nuget@v1.0.5

- name: Create sln

continue-on-error: true

id: step1

working-directory: ./myNewFolder

run: dotnet new sln --name t

- name: Add csproj

continue-on-error: true

id: step2

run: |

dotnet sln ./myNewFolder/t.sln add \*\*/TestConsoleApp.csproj

dotnet add XUnitTestProject1.csproj reference \*\*/TestConsoleApp.csproj

dotnet sln ./myNewFolder/t.sln add XUnitTestProject1.csproj

nuget restore ./myNewFolder/t.sln

- name: Build

continue-on-error: true

id: step3

run: |

dotnet build ./myNewFolder/t.sln

- name: Test

continue-on-error: true

id: step4

run: |

dotnet test XUnitTestProject1.csproj --logger "trx;LogFileName=test-results.trx"

- name: Publish Unit Test Results

continue-on-error: true

id: step5

uses: actions/upload-artifact@v2

if: always()

with:

name: test-results

path: ./\*\*/TestResults/\*.trx

- name: WorkflowStatus1

id: conclusion

if: |

steps.step1.outcome == 'success' &&

steps.step2.outcome == 'success' &&

steps.step3.outcome == 'success' &&

steps.step4.outcome == 'success' &&

steps.step5.outcome == 'success'

shell: bash

run: |

expr 0+0 > conclusion.txt

- name: WorkflowStatus2

if: |

steps.step1.outcome != 'success' ||

steps.step2.outcome != 'success' ||

steps.step3.outcome != 'success' ||

steps.step5.outcome != 'success'

shell: bash

run: |

expr 0+1 > conclusion.txt

- name: WorkflowStatus3

if: |

steps.step1.outcome == 'success' &&

steps.step2.outcome == 'success' &&

steps.step4.outcome != 'success' &&

steps.step3.outcome == 'success' &&

steps.step5.outcome == 'success'

shell: bash

run: |

expr 0+2 > conclusion.txt

- name: Upload conclusion

uses: actions/upload-artifact@v2

with:

name: conc

path: ./\*\*/\*.txt