GOSTDOWN+

(GD)

КОМПЛЕКС ПРОГРАММНЫХ СРЕДСТВ

**Описание программы**

|  |  |
| --- | --- |
| *Подп. и дата* |  |
| *Инв. № дубл.* |  |
| *Взам. Инв №* |  |
| *Подп. и дата* |  |
| *Инв. № подл.* |  |

Листов

2023

Аннотация

В этом программном документе приведено описание программы Gostdown+ – это переменная из преамбулы.

[Gostdown](https://gitlab.iaaras.ru/iaaras/gostdown) - замечательный проект для нас, несчастных, крепко завязанных на Microsoft Word. И все же в основной версии кое-чего важного ему не хватает: трушного single source, переменных и CI/CD. Но нет ничего невозможного для ленивого техписа, который хочет, чтобы оно все само!

Мой первоначальный фреймворк описан в двух предыдущих постах. Для этого поста я также подготовила демонстрационный репозиторий [demo-gostdown-plus]. Эту версия успешно отработала на последней версии pandoc (3.1) и pandoc-crossref (v0.3.15.1).

В чем отличия от основной версии:

1. Можно использовать один шаблон для всех документов: прикручен скрипт [update\_docx\_props](https://github.com/annjulyleon/doc-scripts/tree/main/update_docx_props).
2. Команда запуска pandoc в скрипте build.ps1 Gostdown модифицирована. Теперь можно использовать фильтры pandoc. Добавлены фильтры для включения файлов (include\_files.lua) и обработки переменных (metadata\_processor.lua).
3. Прикручена возможность собирать простой одностраничный HTML.
4. Отлажены Tasks для VSCode. Таски позволяют запускать скрипты без всяких bat-ников, что уменьшает дублирование. Один скрипт build.ps1 для всего.
5. Прикручена [проверка правописания с Vale](https://annjulyleon.github.io/docops/docops-spelling/). Эта очень полезная штука просто спасает мне жизнь при переносе информации с внутренней вики, отлично отлавливает сленг.
6. Чтобы любой чайник мог собрать документ прямо в GitLab, прикручен CI/CD (CI/CD конфигурация из репозитория Gostdown у меня не заработала).

Проехали.

Содержание

[1 Обновление свойств документа 4](#_Toc128645408)

[2 Добавляем single-source и переменные 6](#_Toc128645409)

[3 Одностраничный HTML по шаблону 8](#_Toc128645410)

[4 Таски 9](#_Toc128645411)

[5 Прикручиваем CI/CD на GitLab 10](#_Toc128645412)

[5.1 Установка и настройка gitlab-runner 10](#_Toc128645413)

[5.2 Исправляем ошибки и фиксим конфиги 12](#_Toc128645414)

[5.2.1 Кракозябры в консоли на GitLab 12](#_Toc128645415)

[5.2.2 Git, пути и директории 13](#_Toc128645416)

[5.2.3 Права, разрешения и доступ к COMObject 14](#_Toc128645417)

[5.3 Пример конфигурации 16](#_Toc128645418)

[5.4 Чистим историю пайплайнов 18](#_Toc128645419)

[6 Вместо эпилога 19](#_Toc128645420)

# Обновление свойств документа

В Gostdown для формирования документа docx используется шаблон, в котором задается титул, название документа, децимальный номер и прочие общие сведения. Если документов в проекте много, то держать в репозитории несколько шаблонов расточительно, особенно, если отличаются они только названием и децимальными номерами.

К счастью, у меня уже есть скрипт для обновления полей документов [update\_docx\_props](https://github.com/annjulyleon/doc-scripts/tree/main/update_docx_props). Понадобилась лишь доработка напильником, чтобы можно было вызывать скрипт для одного файла (эта версия лежит [вот тут](https://github.com/annjulyleon/doc-scripts/tree/main/update_docx_props/update_docx_props_single), рядышком).

Получается вот такой workflow:

1. Собрать документ. Документ сохранен в директорию /build (настраивается в таске).
2. Запустить таску «Update docx fields». Выбрать из списка документ (документ соответствует названию файла и директории).
3. Скрипт обновления свойств будет запущен для выбранного документа.

Настройка:

1. Создать в директории проекта папку /build.
2. В папку scripts сохранить скрипт update\_docx\_props.ps1.
3. В папке scripts/configs создать конфигурационный файл для документа. В конфигурационном файле указать свойства, которые нужно обновлять.

* <?xml version="1.0"?>  
  <configuration>  
   <customProperties>  
   <add key="DeviceName" value="Красивое название"/>  
   <add key="DocTitle" value="Описание программы"/>  
   <add key="DecimalNumber" value="АБВГ.00002 01 13"/>  
   <add key="DocSubtitle" value="Подзаголовок"/>  
   <add key="DocYear" value="2023"/>  
   </customProperties>  
   <builtinProperties>  
   <add key="Title" value="Красивое название. Описание программы"/>  
   <add key="Subject" value="Красивое название"/>  
   <add key="Keywords" value="description"/>  
   <add key="Comments" value="2023"/>  
   </builtinProperties>  
   <vsdProperties>  
   </vsdProperties>  
  </configuration>

1. Создать таску для VS Code в .vscode/tasks.json:

* {  
   "label": "Update docx fields",  
   "detail": "Update builded document",  
   "type": "shell",  
   "options": {  
   "cwd": "${workspaceFolder}/build/"  
   },  
   "command": "${workspaceFolder}\\scripts\\update\_docx\_props.ps1 -dir ${workspaceFolder}\\build -conf ${workspaceFolder}\\scripts\\configs\\${input:doc}.xml -filename ${input:doc}.docx",  
   "presentation": {  
   "echo": true,  
   "reveal": "always",  
   "focus": false,  
   "panel": "shared",  
   "showReuseMessage": false,  
   "clear": true  
   }  
   }

1. В таске используется [input variables](https://code.visualstudio.com/docs/editor/variables-reference#_input-variables). Их удобно использовать, например, для запуска одной таски для нескольких директорий:

* "inputs": [  
   {  
   "id": "doc",  
   "description": "Document Path: ",  
   "type": "pickString",  
   "options": [  
   "example\_doc",  
   "another\_doc"  
   ],  
   "default": "example\_doc"  
   },  
   ]

# Добавляем single-source и переменные

В общем я не думала, что мне опять понадобиться single-source, но ~~жизнь~~ требования технического задания распорядились иначе. 11 комплектов документов с большими общими кусками, кусками, отличающимися только названием ПО - без single source тут с ума сойдешь.

Вот как это работает:

1. Файлы с общими кусками текста лежат где-то в общей папке. В демонстрационном репозитории это docs\_gost/single\_source.
2. В файле или файлах основного документа (например, в docs\_gost/example\_doc) вставляем в нужные места файлы из docs\_gost/single\_source с помощью специального синтаксиса.
3. В шапке документа (в YAML-преамбуле) добавляем свои переменные (например, в docs\_gost/single\_source00\_begin.md добавлена переменная soft-name).
4. В тексте к переменной обращаемся вот так: ${soft-name}. При сборке сюда будет подставлено значение переменной из преамбулы.
5. Модифицированная команда pandoc в build.ps1 применяет все необходимые фильтры и обрабатывает результат. А затем передает уже основному скрипту build.ps1, который применяет шаблон и творит свою COMKbject-овскую магию.

В демонстрационном репозитории example\_doc имеет все перечисленный фичи: переменную, включенные файлы и разнообразные картинки. И да, в файлах-источниках тоже можно использовать переменные, они будут заменены на значение из основного файла.

К счастью, добрые люди уже разработали все необходимые фильтры, так что их осталось просто немного доработать.

Стандартный фильтр из репозитория pandoc [include-files](https://github.com/pandoc/lua-filters/tree/master/include-files) уже делает все, что нужно для счастья. В документе, в который хотим что-то вставить пишем (рисунок [1](#fig:2023-03-07-include)):

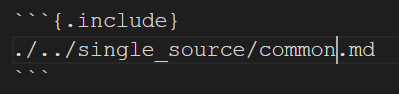


Рисунок 1 – Include

В команде pandoc нужно передать ключ -M include-auto , если нужно автоматически перенумеровывать заголовки. Также можно для каждого блока указывать [свои правила нумерации](https://github.com/pandoc/lua-filters/tree/master/include-files#manual-shifting). Я пока обхожусь -M include-auto.

Для переменных тоже уже написано немало фильтров. Я адаптировала вот этот [pandoc-curly-switch](https://github.com/cdivita/pandoc-curly-switch).

Минусы использования этих фильтров – не полная совместимость с mkdocs, с которой я пока не разбиралась. В mkdocs тоже есть [переменные](https://jimandreas.github.io/mkdocs-material/reference/variables/) и плагин [include](https://github.com/mondeja/mkdocs-include-markdown-plugin), так что вероятно их можно совместить с pandoc-фильтрами.

Команда pandoc с нашими фильтрами для теста:

pandoc (cat \*.md) -o some.docx -M include-auto --lua-filter include\_files.lua --lua-filter linebreaks.lua --lua-filter metadata\_processor.lua --filter pandoc-crossref --citeproc --reference-doc template.docx

А вот команда вызова pandoc в скрипте Gostdown. Как видно добавлена переменная $luafilter, в которой передается путь к папке с фильтрами .lua.

&$exe $md -o $tempdocx -M include-auto --lua-filter $luafilter\include\_files.lua --lua-filter $luafilter\linebreaks.lua --lua-filter $luafilter\metadata\_processor.lua --filter pandoc-crossref --citeproc --reference-doc $template

# Одностраничный HTML по шаблону

mkdocs крут, но не всегда нужен полноценный сайт с документацией. Иногда нужен просто одностраничный HTML с навигацией, чтобы быстро отправлять инженерам по запросу. К счастью здесь ничего сложного придумывать не надо, так как pandoc уже все умеет, нужно только подобрать правильное ~~заклинание~~ команду.

Шаблон для pandoc можно взять готовый, я использую [easy-pandoc-templates](https://github.com/ryangrose/easy-pandoc-templates) by [ryangrose](https://github.com/ryangrose). Их легко можно адаптировать для своих нужд, скачать все необходимые ресурсы, чтобы шаблоны работали оффлайн.

Команда для конвертации кучки .md файлов в HTML:

pandoc -s $(Get-Content html\_include.txt) -f markdown --quiet --template=easy-template.html --lua-filter=include\_files.lua --lua-filter=linebreaks.lua --lua-filter=metadata\_processor.lua --filter pandoc-crossref --citeproc -o output.html --toc

В файле html\_include.txt на каждой строке перечислены все md файлы, которые хотим слепить в одностраничный HTML. В скрипте create\_simple\_html.ps1 эта команда параметризована, так что можно легко вызывать скрипт в тасках VSCode.

В таске VSCode команда для вызова получилась развесистая, так как помимо сборки документа, нужно также скопировать все картинки. Это делается автоматически таской. К сожалению, из таски не получится вызвать команду замены путей к картинкам (из-за экранирования PowerShell будет ругаться на неправильные паттерны регулярных выражений), поэтому эту команду придется выполнить вручную. В папке /build нужно выполнить команду вида:

foreach ($file in (Get-ChildItem .\web \*.html -rec)) {(Get-Content $file.PSPath) | Foreach-Object { $\_ -replace '..\\single\_source\\\_img\\', '\_img/' } | Set-Content $file.PSPath }

# Таски

Я уже немного писала [про таски](https://annjulyleon.github.io/docops/docops-gostdown/#tasks) для запуска стандартных батников Gostdown. Но в тасках можно вызывать скрипт build.ps1 напрямую и передавать ему все необходимые параметры. Это позволило убрать из проекта дублирующиеся скрипты и bat и создать одну единственную задачу для сборки всех документов.

Рассмотрим вот такую таску из демонстрационного проекта:

{  
 "label": "Build Doc",  
 "detail": "Build selected in docx\_include.txt to build docs",  
 "type": "shell",  
 "options": {  
 "cwd": "${workspaceFolder}/docs\_gost/${input:doc}/"  
 },  
 "command": "${workspaceFolder}\\scripts\\build.ps1 -md $(Get-Content docx\_include.txt) -template ${workspaceFolder}\\docs\_gost\\template.docx -luafilter ${workspaceFolder}\\scripts -docx ${workspaceFolder}\\build\\${input:doc}.docx -embedfonts",  
 "presentation": {  
 "echo": true,  
 "reveal": "always",  
 "focus": false,  
 "panel": "shared",  
 "showReuseMessage": false,  
 "clear": true  
 },  
 "group": {  
 "kind": "build",  
 "isDefault": false  
 }  
 }

Как видно, это просто команда из build.bat скрипта Gostdown. Добавлен параметр -luafilter, в котором указан путь к фильтрам pandoc. Кроме того, обратите внимание, чтобы отработали правильно все скрипты, рабочая директория должна быть установлена в корне собираемого документа (параметр options.cwd). Список файлов .md, которые нужно собрать, теперь лежат в файле docx\_include.txt в корне документа.

В таске используются переменные [input variables](https://code.visualstudio.com/docs/editor/variables-reference#_input-variables), о которых уже говорилось [выше](#обновление-свойств-документа). Таким образом, эту таску можно запускать для всех документов: при запуске пользователю будет предложено выбрать документ для сборки.

# Прикручиваем CI/CD на GitLab

Итак, наконец-то я добралась до уровня с боссом CI/CD. Пришлось просмотреть немало всяких прохождений, но я справилась). Теперь в репозитории проекта любой менеджер или разработчик может собрать себе docx и скачать артефакты. И теперь техпису не нужно таскать с собой в отпуск ноутбук!

Gitlab’овский CI/CD состоит из двух важных компонентов и двух конфигов:

* gitlab-runner нужно установить на машину с Windows, на которой будет выполнена сборка. Эта служба выполняет все нужные команды для сборки нашего проекта;
* в проекте на сайте Gitlab должна быть включена функция CI/CD;
* в корне проекта должен существовать файл .gitlab-ci.yml с описанием порядка сборки (команды). Все это будет передано gitlab-runner’у на исполнение;
* рядом с исполняемым файлом gitlab-runner должен лежать конфиг config.toml, в котором описаны настройки подключения к Gitlab.

Все нижеописанные непотребства я проделывала на selfhosted Gitlab/Gitlab runner версии 15.8, Windows 10, Microsoft Word 2021. Предполагается, что у вас есть права владельца проекта на Gitlab’е.

Кроме того, на машине, которую будете использовать для сборки, должен быть установлен весь набор программ: Git, pandoc/pandoc-crossref, Microsoft Word, шрифты. Рекомендую также поставить расширение [Gitlab Workflow](https://marketplace.visualstudio.com/items?itemName=GitLab.gitlab-workflow) для VSCode для валидации конфигов и запуска пайплайнов прямо из VSCode.

## Установка и настройка gitlab-runner

В официальной документации присутствует [туториал](https://docs.gitlab.com/runner/install/windows.html) по установке и настройке gitlab-runner, однако прямо скажу, не очень он отражает суровую действительность и количество проблем, с которыми пришлось столкнуться.

1. Для начала активируем для проекта CI/СD (*Settings* -> *General* -> *Visibility, project features, permissions* -> *Expand* -> включить CI/CD).
2. В разделе *Setting* -> *CI/CD* -> *Runners (Expand)* есть информация, которую нужно указывать при регистрации gitlab-runner ниже (token и адрес). Если у вас selfhosted Gitlab на докерах, то адрес может быть с HTTP, даже если у вас HTTPS. Если при регистрации gitlab-runner появляется ошибка 401, то вам нужен HTTPS.
3. Делаем по туториалу [Install Gitlab Runner on Windows](https://docs.gitlab.com/runner/install/windows.html), ну по крайней мере первые пункты совпадают ). На машине создаем папку без кириллицы (D:\gitlab-runner). Достаточно одного раннера на все проекты.
4. Сохранить туда .exe, ([вот файл](https://gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-windows-amd64.exe)). Переименовать exe в gitlab-runner.exe (для удобства, можно как угодно назвать).
5. Если у вас self-signed сертификат в Gitlab’e, экспортируйте его с сайта и положите в рабочую папку раннером (например, файл gitlab.blabla.ru.crt).
6. Запустить от администратора консоль (обычную, не PowerShell) и перейти в папку с раннером (d:, cd D:\D:\gitlab-runner).
7. Выполнить в консоли следующую команду. В TOKEN надо указать токен проекта со страницы *Setting* -> *CI/CD* -> *Runners (Expand)*, а в tls-ca-file – **абсолютный путь** к сертификату. В executor можно указать несколько оболочек (https://docs.gitlab.com/runner/executors/), мне нужен только shell:

* gitlab-runner register --non-interactive --registration-token TOKEN --url https://gitlab.blabla.ru/ --tls-ca-file D:\gitlab-runner\gitlab.blabla.ru.crt --executor shell

1. Раннер будет зарегистрирован и появится на странице *Setting* -> *CI/CD* -> *Runners (Expand)*.
2. Открыть конфигурационный файл config.toml (появится рядом с exe раннера после регистрации) и поменять pwsh в ключе shell на powershell (иначе при запуске пайплайна будет ошибка, что путь не найден).
3. Установить службу командой gitlab-runner.exe install и запустить gitlab-runner.exe start. Если вы измените конфиг config.toml, то службу нужно перезапустить.
4. После этого на странице *Setting* -> *CI/CD* -> *Runners* раннер должен стать зелененьким. Если не стал, значит что-то не так, и джобы не запустятся.
5. В “Службах” в Windows перейти в свойства службы gitlab-runner -> *Вход в систему* > *галка “Разрешить взаимодействие с рабочим столом”*.

В общем все готово, и можно выполнить тестовую джобу из [туториала](https://docs.gitlab.com/ee/ci/quick_start/). Для этого нужно скопировать содержимое тестового примера в файл .gitlab-ci.yml в корне проекта и закоммитить. В данном случае джоба будет запущена автоматически, сразу после коммита, и появится на в разделе *CI/CD* -> *Jobs*. В консоли будут выведены команды echo.

Что делает джоба? Выкачивает с помощью Git наш проект из репозитория и применяет перечисленный в .gitlab-ci.yml команды. Артефакты, которые получились в результате, пакует и передает Gitlab’у. Артефакты можно скачать в результатах работы или с вкладки Release (если она была настроена).

Ошибки раннера можно смотреть в Windows в Просмотре Событий или в PowerShell Get-WinEvent -ProviderName gitlab-runner.

## Исправляем ошибки и фиксим конфиги

Как вы понимаете, если бы все было так просто, мы бы тут не собирались. Первая проблема, с которой я столкнулась – кракозябры в консоли на GitLab. Потом отказывался запускаться скрипт. Потом скрипт не мог найти нужные пути. Короче, без долгих прелюдий перехожу сразу к списку фиксов возможных проблем.

### Кракозябры в консоли на GitLab

Чтобы gitlab-runner не писал кракозябры в скрипт .gitlab-ci.yml в конфиг работы нужно добавить CHCP 65001:

build-job:  
 before\_script:  
 - CHCP 65001  
 stage: build  
 script:

Это еще не все. На машине, на которой стоит gitlab-runner нужно переключить региональные настройки на Английский (США): Региональные параметры -> Дополнительные параметры … -> Региональные стандарты - Изменение форматов даты, времени и чисел -> Дополнительно -> Изменить язык системы -> выбрать Английский (США).

Это тоже еще не все, так как при запуске скриптов build.ps1 и update\_docx\_props.ps1 могут возникнуть проблемы с кодировкой. Эти скрипты нужно открыть в блокноте Notepad++ и пересохранить в кодировку UTF-8 BOM. В демонстрационном репозитории все уже в правильной кодировке.

### Git, пути и директории

Если вы, как я, до этого никогда не имели дела с настройкой CI/CD (кроме как GitHub, который все сам делает), то можно зависнуть на очень простых ошибках.

Иногда, при скачивании с Git работа может упасть с ошибкой авторизации (**Authentication error**). Это происходит рандомно у разных людей. Чтобы исправить ошибку, нужно в конфигурационный файл gitlab-runner config.toml под параметром url добавить еще одну строку с параметром clone\_url. URL указать точно такой же как в url. Магия, что сказать.

Чтобы запускать наши скрипты, нам полезно будет иметь фиксированные директории. По умолчанию gitlab-runner скачивает репозиторий в папку с названием из рандомного набора букв и цифр (например, build/123hjksd898231). В конфиг config.toml в раздел [runners.custom\_build\_dir] добавить строчку enabled = true.

Вот как примерно выглядит конфиг после всех фиксов:

concurrent = 1  
check\_interval = 0  
shutdown\_timeout = 0  
  
[session\_server]  
 session\_timeout = 1800  
  
[[runners]]  
 name = "docbuilder"  
 url = "https://gitlab.blabla.ru/"  
 clone\_url = "https://gitlab.blabla.ru/"  
 id = 6  
 token = "SOMETOKEN"  
 token\_obtained\_at = 2023-02-16T11:50:04Z  
 token\_expires\_at = 0001-01-01T00:00:00Z  
 tls-ca-file = "D:\\gitlab-runner\\gitlab.blabla.ru.crt"  
 executor = "shell"  
 shell = "powershell"  
 [runners.custom\_build\_dir]  
 enabled = true  
 [runners.cache]  
 MaxUploadedArchiveSize = 0  
 [runners.cache.s3]  
 [runners.cache.gcs]  
 [runners.cache.azure]

Эта настройка позволит в конфигурации .gitlab-ci.yml задавать путь, куда будет склонирован проект:

variables:  
 GIT\_CLONE\_PATH: $CI\_BUILDS\_DIR/$CI\_PROJECT\_NAME

Код будет склонирован в папку /build/[название проекта с Gitlab]. Можно также указать вместо $CI\_PROJECT\_NAME любое имя.

Когда будете формировать шаги по сборке gostdown’ом очень рекомендую использовать команды перехода по директориям перед выполнением скриптов. Скрипты не будут выполнятся, если их путь начинается с переменной (./$CI\_BUILDS\_DIR/test/build.ps1 будет ругаться на You must provide a value expression following the '/' operator., Unexpected token, Missing closing '}' in statement block or type definition и все подряд).

Кроме того, скрипт build.ps1 необходимо запускать из директории документа, который вы собираете. В противном случае, pandoc сначала не найдет файлов, потом не найдет include, потом вообще обидится и упадет с ошибкой. Так что если что-то не собирается, и возникают непонятные ошибки, сначала проверьте все пути!

### Права, разрешения и доступ к COMObject

Если вы сразу после установки и успешного выполнения джобы из туториала попробуете запустить скрипт Gostdown, то ничего у вас не выйдет. Скрипт свалится вот на этой строчке: $word = New-Object -ComObject Word.Application.

Нет, у меня сначала все получилось, потому что я запускала все на своем ноутбуке для тестирования. Но при переносе пайплайна на отдельный сервер вылезли ошибки, связанные с правами запуска службы и разрешениями COMObject.

Такая горемыка была не я одна, про это есть тред на [stackoverflow](https://stackoverflow.com/questions/74277833/powershell-script-doesnt-work-with-docx-files-as-com-objects-while-being-run-vi). В одиноком ответе к этой проблеме содержится способ решения.

1. В “Службах” найти gitlab-runner -> *Свойства* на вкладке *Вход в систему* поставить переключатель в *С учетной записью* и указать имя пользователя и пароль (я указала пользователя с административными правами, под которым ставила pandoc, word и все остальное).
2. Выберите *Пуск* -> *Выполнить* (или нажмите Win + R) -> введите MMC -32 и нажмите ОК.
3. Выберите *Файл* -> *Добавить/удалить оснастку* ->\_ добавьте Службы компонентов\_. Сохраните консоль.
4. Запустите сохраненную консоль с правами администратора.
5. Раскройте дерево *Службы компонентов* до *Мой компьютер* и выберите *Настройка DCOM*.
6. Найдите Microsoft Word 97 - 2003 Document -> ПКМ -> \_Свойства.
7. На вкладке *Удостоверение* выберите *Указанный пользователь* и укажите учетные данные пользователя (такой же как на шаге 1).
8. На вкладке *Безопасность* > *Разрешения на доступ* выбрать *Настроить* и добавить явно своего пользователя и выдать ему все разрешения. Также у меня выданы разрешения в разделе *Разрешения на настройку*. Это риск безопасности, так что выполняйте такую настройку только в доверенной сети.

Вот, теперь должно работать. Имейте в виду, что теперь работа не будет запускаться, если выполнен интерактивный вход под тем же пользователем (то есть вы залогинены, например, по удаленному рабочему столу). Для тестирования можно переключить в пункте 7 удостоверение на интерактивного пользователя, тогда служба будет запускаться с правами залогиненного пользователя.

Еще несколько моментов:

1. Если работа зависла на этапе взаимодействия с Word, то все дальнейшие запуски и работы завершаться ошибкой, так как процесс Word будет продолжать висеть. Чтобы этого не происходило, можно добавлять в конфиг работы команду Stop-Process -Name "WINWORD" перед или после выполнения скрипта (лучше до).
2. Если работа завершилась ошибкой, на этапе взаимодействия с template.docx, то следующее открытие Word может повиснуть с диалоговым окном, которое нужно закрыть вручную. Как это обойти, я пока не поняла, но больше работы у меня не зависали, так что это редкая ошибка.

## Пример конфигурации

Так как в демонстрационном проекте на GitHub показать сборку затруднительно, я покажу рабочий конфиг работы вывод консоли (рисунки [3](#fig:2023-03-07-job-console),[2](#fig:2023-03-07-job-result)).

Ниже конфиг .gitlab-ci.yml для сборки одного документа (в данном случае - программа и методика испытаний, её чаще всего требуется собирать). Работа собирает документ, обновляет поля и отправляет артефакты работы в Gitlab.

Keyword workflow обрабатывается перед работами и определяет, будет ли генерироваться пайплайн и при каких условиях (см. [документацию](https://docs.gitlab.cn/14.0/ee/ci/yaml/README.html#workflow)). В данном случае, пайплайн будет сгенерирован только, если в commit message содержится ключевой флаг -build.

variables:  
 GIT\_CLONE\_PATH: $CI\_BUILDS\_DIR/$CI\_PROJECT\_NAME  
  
workflow:  
 rules:  
 - if: $CI\_COMMIT\_MESSAGE =~ /-build$/  
  
Build PMI:  
 before\_script:  
 - CHCP 65001  
 stage: build  
 when: manual  
 script:  
 - echo "Starting building PMI..."  
 - mkdir build  
 - echo "Reading include file..."  
 - get-content $CI\_BUILDS\_DIR/$CI\_PROJECT\_NAME/docs\_gost/pmi/docx\_include.txt | foreach {"builds/$CI\_PROJECT\_NAME/docs\_gost/pmi/" + $\_} | out-file $CI\_BUILDS\_DIR/$CI\_PROJECT\_NAME/docs\_gost/pmi/include.txt  
 - echo "Building..."  
 - cd $CI\_BUILDS\_DIR/$CI\_PROJECT\_NAME/docs\_gost/pmi  
 - ./../../scripts/build.ps1 -md $(Get-Content $CI\_BUILDS\_DIR/$CI\_PROJECT\_NAME/docs\_gost/pmi/include.txt) -template $CI\_BUILDS\_DIR/$CI\_PROJECT\_NAME/docs\_gost/template.docx -luafilter $CI\_BUILDS\_DIR/$CI\_PROJECT\_NAME/scripts/ -docx $CI\_BUILDS\_DIR/$CI\_PROJECT\_NAME/build/pmi.docx -embedfonts  
 - sleep 5  
 - echo "Updating fields..."  
 - cd $CI\_BUILDS\_DIR/$CI\_PROJECT\_NAME  
 - ./scripts/update\_docx\_props.ps1 -dir $CI\_BUILDS\_DIR/$CI\_PROJECT\_NAME/build -conf $CI\_BUILDS\_DIR/$CI\_PROJECT\_NAME/scripts/configs/pmi.xml -filename pmi.docx  
 - sleep 5  
 artifacts:  
 paths:  
 - build/pmi.docx  
 expire\_in: 1 hour

Параметр expire\_in устанавливает время хранения собранного документа на Gitlab. По истечение этого срока, артефакт удаляется. Однако, по умолчанию GitLab в версии, начиная с 13, всегда сохраняет последний артефакт (это настраивается для каждого проекта в секции CI/CD).

Если нужно, чтобы работа запускалась автоматически только для мастера, а для всех остальных веток – только вручную, то в работу (или в workflow) можно добавить правило ([пример](https://gitlab.com/brendan-demo/default-or-manual-deploy)):

rules:  
 - if: $CI\_COMMIT\_BRANCH == $CI\_DEFAULT\_BRANCH  
 when: always  
 - if: $CI\_COMMIT\_BRANCH != $CI\_DEFAULT\_BRANCH  
 when: manual

В шаге скрипта Reading include file... выше выполняется операция по изменению путей в файле docx\_include.txt. По умолчанию в этом файле хранится список .md, которые мы хотим включить в результирующий .docx. Однако при запуске gitlab-runner пути в этом файле нужно изменить на относительные пути от gitlab-runner.exe, что и сделано в этом шаге.

На рисунке ниже видно, что отработал скрипт обновления полей документов: вопросительные знаки вместо русского языка из-за региональных настроек, в документе все будет установлено корректно.

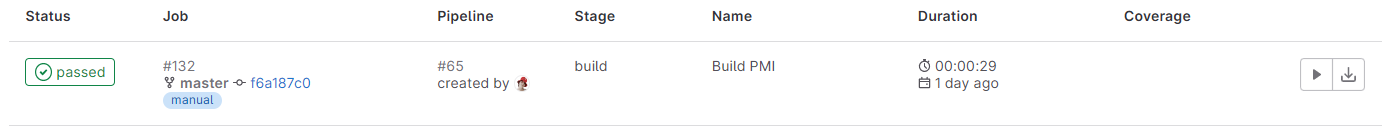


Рисунок 2 – Результат в списке работ

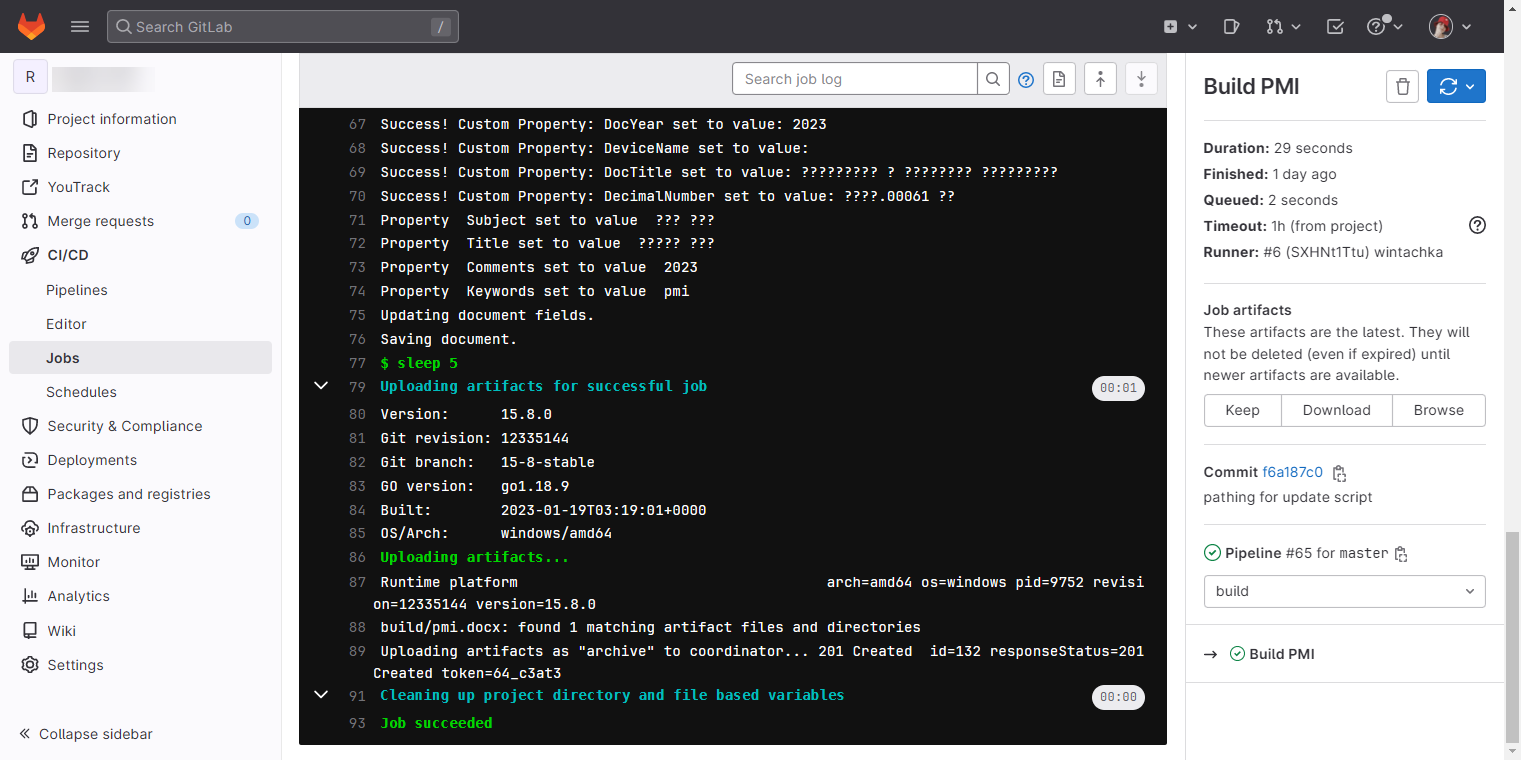


Рисунок 3 – Консоль

## Чистим историю пайплайнов

Пока я тестировала ci/cd, в проектах образовалась длинная история работ и пайплайнов. Вручную их удалять очень долго, а кнопки “почистить историю” в гитлаб еще не завезли. Поэтому рекомендую проводить тесты и настройку на тестовом проекте, который не жалко будет грохнуть. Если же вас, как и меня, бесит грязная история failed пайплайнов, то можно почистить историю скриптом на Python с библиотекой [python-gitlab](https://python-gitlab.readthedocs.io/en/stable/index.html):

import gitlab  
  
project\_id = 55  
gl = gitlab.Gitlab('https://gitlab.blabla.ru/', private\_token='TOKEN', api\_version='4',ssl\_verify=False)  
project = gl.projects.get(project\_id, obey\_rate\_limit=False)  
  
for pipeline in project.pipelines.list(as\_list=False):  
 pipeline.delete()

Если у вас selfhosted GitLab, то возможно придется немного доработать скрипт в зависимости от вашего SSL, мне хватило просто отключить проверку. В скрипте нужно указать ID проекта, URL и токен. Запустить скрипт 2-3 раза - и ура, история пайплайнов чиста и прекрасна, и позорные ошибки никто не увидит )

# Вместо эпилога

Полагаю, после всех пережитых приключений логичным следующим этапом должно быть прикручивание какого-нибудь ChatGPT или аналогичной сетки к документации. А то что, у нас зря что ли отдел машинного обучения сидит, ~~ерундой~~ важными проектами занимается? Уже есть интересные плагины для редактора ObsidianMD, которые делают неплохие summary - для научных отчетов и обзоров литературы просто огонь. Также есть немало задач, где требуется набросать несложную инструкцию по официальной документации (например, docker, linux и т.д.).

Как вы поняли, конечная цель, чтобы документация писалась сама, я потеряла работу и ушла работать дворником (рисунок [4](#fig:cat)). Их вроде не собираются автоматизировать, здесь пока люди дешевле технологий )



Рисунок 4 – Котик

Лист регистрации изменений

|  | Номера листов (страниц) | | | | Всего листов (страниц) в докум. | № документа | Входящий номер доп. Документа и дата | Подп. | Дата |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Изм. | изменен-ных | заменен-ных | новых | аннулиро-ванных |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |