یک Workflow ساده با Github Action

تو این پست میخوام راه اندازی و تنظیم یک workflow ساده با Github Action رو توضیح بدم ، تو این آموزش روی مفاهیم پایه و اولیه تمرکز میکنم ، پس اگه حرفه ای هستید فکر نکنم براتون مفید باشه😉

قبلش لازمه یه سری مفاهیم اولیه ، نیازها و هدف این موضوع رو به صورت خلاصه بررسی کنیم :

⁉️ چی شد که تصمیم گرفتم این پست رو بنویسم

چه وقت‌هایی که خودم برنامه نویسی میکنم چه وقت‌های که برای پروژه های مختلف workflow طراحی میکنم زیاد با این مورد مواجه میشم که خیلی از بچه های فنی یا اصلا از workflow استفاده نمی‌کنند یا شناخت کاملی نسبت که کارهایی که میتونند با workflow انجام بدند ندارد و این موضوع باعث میشه کارها رو به روش‌های سخت تری انجام بدند
یه شب وسط یه مهمونی 😁 داشتم یه workflow رو به رفیقم شهریار نشون میدادم و در موردش صحبت میکردیم که گفت اینی که تو نوشتی خیلی خفنه و متاسفانه وقتی workflow های موجود رو میبینم ، هیچکدومشون حتی از این موارد ساده هم استفاده نمی کنند ، خوب میشه اگه همین موارد ساده رو هم تو لینکدین پست کنی ، حتما برای خیلی ها مفیده
(اینکه وسط یه مهمونی ما داشتیم یه workflow رو بررسی می کردیم هم تو جای خودش قابل بحثه😅)

بریم سراغ مفاهیم:

🚀 گیت‌هاب اکشن و مفهوم ورک‌فلو

گیت‌هاب اکشن یک سیستم CI/CD داخلی گیت‌هابه که به ما اجازه میده کارهای خودکار مثل بیلد، تست و استقرار رو روی مخازن گیت‌هاب اجرا کنیم. این ابزار به کمک **ورک‌فلوها**، فرایندهای توسعه رو ساده‌تر کرده و به ما کمک می‌کنه کارهای تکراری رو به صورت خودکار انجام بدیم.

✅ ورک‌فلو (Workflow) چیه؟

یک **ورک‌فلو** مجموعه‌ای از دستوراته که به صورت خودکار روی یک مخزن اجرا می‌شه. این دستورات در یک فایل `YAML` تعریف شده و میتونه تو شرایط مشخص مثل `push`، `pull request` یا یک زمان‌بندی خاص اجرا بشه.

🎯 چرا از ورک‌فلو استفاده کنیم؟

✅ خودکارسازی کارهای تکراری مثل بیلد، تست و استقرار
✅ اجرای تست‌ها قبل از ادغام کد و کاهش باگ‌ها
✅ انتشار خودکار پکیج‌ها و استقرار سریع‌تر
✅ اجرای اسکریپت‌ها روی سرور، Docker یا ماشین‌های مجازی

و کلی کار باحال دیگه که درنهایت فرایند ها رو ساده، سریع و بی اشتباه تر میکنه

📌 چه کارهایی رو میشه با ورک‌فلو انجام داد؟

1️⃣اجرای تست‌ها به صورت خودکار بعد از هر `push`
2️⃣بیلد و کامپایل پروژه تو محیط‌های مختلف (مثلاً Node.js، Python، Go)
3️⃣انتشار خودکار پکیج‌ها تو سرویس‌هایی مثل Docker Hub یا npm
4️⃣استقرار خودکار روی سرور یا سرویس‌های ابری مثل AWS و Heroku
5️⃣مدیریت خودکار مخزن، مانند برچسب‌گذاری (`Label`) روی `Issue` ها

خب تا اینجا مفاهیم کلی رو متوجه شدیم

شما میتونید workflow ها رو برای محیط های مختلف و پروژه های مختلف طراحی و اجرا کنید اما تو این پست من موضوع رو برای Github Action توضیح بدم تا در ساده ترین حالت برای دوستانی که ریپازیتوری های عمومی و خصوصی پروژه های شخصی کوچیک هم دارند قابل استفاده باشه

💡 یه پروژه ی خیالی

برای درک بهتر موضوع بیاید همینجا یه پروژه ی خیالی تعریف کنیم که خیلی ساده است ،
فرض میکنیم ما داریم یه اسکریپت Bash مینویسیم و هیچ تیمی وجود نداره ، فقط خودمونیم و خودمون
اما میخوایم هربار که تغییری رو پوش میکنیم ، یه نسخه از این تغییرات با توضیحاتش باقی بمونه و بتونیم بهش رجوع کنیم ، در کنارش ، یه ریلیز برای اون پوش ایجاد کنیم که بعضی فایل ها رو تولید کنه و تو ریلیز انجام بده یا حتی تغییرات رو روی سرور دیگه ای آپلود کنه

📂 ایجاد یک فایل ورک‌فلو

- داخل ریپازیتوری یه دایرکتوری به نام .github ایجاد میکنیم
- داخل دایرکتوری بالا یه دایرکتوری دیگه با نام workflows ایجاد میکنیم
- داخل دایرکتوری بالا یه فایل Yaml به نام دلخواه میسازیم ، مثلا release.yml

تا اینجا باید چیزی شبیه ساختار زیر داشته باشیم

```
+--- Simple project
|   +--- .github
|   |   +--- workflows
|   |   |   +--- release.yml
```

🛠️بخش های مختلف یک Workflow

همونطور که تا الان متوجه شدید workflow از ساختار Yaml استفاده میکنه پس همینجا لازمه یادآوری کنم :
دوست خوب من حواست به Indent ها باشه که به خطا نخوری 😉

1️⃣نام workflow که با name مشخص میشه

تو این بخش ما نام رو مشخص میکنیم و خیلی به سلیقه ی شما مربوط میشه اما سعی کنید اسمی بذارید که نفر بعدی که اون رو میخونه یا حتی خودتون بعدا گیج نشید و نفرین و لعنت نفرستید
مثال این بخش میتونه شکل زیر باشه

```yaml
name: Release Automation
```

2️⃣رویداد (Event) که با on مشخص میشه

تو این بخش مشخص میکنیم چه رویدادی باعث اجرای این workflow میشه ، در واقع تو این بخش trigger اجرای Workflow رو مشخص میکنیم . این رویداد به صورت خلاصه مشخص میکنه چه اتفاقی در کجا باعث اجرای Workflow میشه
تو پروژه‌ی خیالیمون، ما میخوایم هر بار که یک push روی برنچ Main اتفاق افتاد اجرا بشه پس بخش رویداد ما به شکل زیره

```yaml
on:
  push:
    branches:
      - main
```

3️⃣مجوزها که با permissions مشخص میشه

تو این بخش مشخص میکنیم که این workflow چه مجوزهایی در مورد ریپازیتوری داره که شما دقیقا میتونید توش مشخص کنید ، دسترسی انجام چه کارهایی رو داره ، حدود ۱۳ آپشن مختلف داره که هر کدوم میتونند یکی از سه حالت write ، read و none رو داشته باشند ، تو این پروژه ما احتیاجی به تمامش نداریم ، فقط میخوایم بتونیم محتوای ریپازیتوری رو تغییر بدیم پس خیلی ساده از دو تا از آپشن ها استفاده میکنیم تا workflow بتونه بعضی بخش ها رو بخونه یا حتی تغییر بده ، تو قسمت های بعدی متوجه میشید چرا به این دسترسی نیاز داشتیم

```yaml
permissions:
  contents: write
  id-token: write
```

4️⃣وظایف که با jobs مشخص میشه

این بخش ، هسته اصلی workflow هست و مشخص میکنه دقیقا چه وظایفی باید انجام بشه ، معمولا دو بخش اصلی داره که محیط اجرا و قدم های اجرا رو مشخص میکنیم ، در ادامه بخش های مختلف رو براتون توضیح میدم اما فعلا حالت کلی بخش jobs رو نگاه کنید که به شکل زیره

```yaml
jobs:
  create-release:
    runs-on: ubuntu-22.04
    steps:
      - name: step 1
        uses: actions/checkout@v3

      - name: step 2
        id: step_2
        run: |
          command

      - name: step 3
        id: step_3
        run: |
          command
```

حالا بریم بخش های مختلف jobs رو با هم بررسی کنیم

💻 محیط اجرا که با runs-on مشخص میشه

همونطور که احتمالا خودتون متوجه شدید اینجا مشخص میکنیم وظایف تو چه محیطی اجرا بشه ، محیط های اجرا متفاوت اما محدوده و لیست کاملش رو میتونید با جستجو پیدا کنید اما من معمولا از ubuntu 22 استفاده میکنم چون استیبل و سریعه ، پس به شکل زیر میشه

```yaml
    runs-on: ubuntu-22.04
```

🏗️ مراحل انجام وظایف یا همون ریز وظایف که با steps مشخص میشه

میشه گفت اصل داستان همینجاست ، و جادوی اصلی تو step ها اتفاق میفته ، اینجاست که شما تک تک مراحل مورد نظرتون برای انجام رو نامگذاری و تعریف میکنید ، هر step آپشن های مختلفی داره که بعضیاشون ضروری هستند و بعضی دیگه نه ، مواردی مثل name که نامش رو مشخص میکنه id که یک شناسه ی یکتا بهش میده uses که مشخص میکنه از چه چیزی استفاده میکنید و run که اصل عملیات رو نشون میده

📌 نگران نشید! 😉

ممکنه به خودتون بگید : ((ای بابا ، تا همین جاش کلی کار داشت ، من دیگه نمیکشم)

اما نگران نباشید ، شما قدم اصلی رو وقتی برداشتید که این مطلب به نظرتون جالب اومد، اگه به اینجای آموزش رسیدید، پس این آموزش مخصوص خود خود شماست ، و به سادگی از پسش بر میاید، نفس عمیق بکشید تا مراحل بعدی رو با مثال پیش بریم، و فقط لذت ببرید

📦 کد اصلی

برای راحتی بیشتر، من معمولا از هر اسکریپت Bash که مینویسم، چند تا متغیر اضافه میکنم که بعدا ازشون تو workflow و حتی جاهای دیگه استفاده کنم مواردی مثل version و تاریخ آپدیت و....

بسته به زبان مورد نظر و سلیقه میتونید ساختار خودتون رو داشته باشید

ابتدای پروژه ی خیالی ما با این شکله

```
#!/bin/bash
# Simple Bash script to show how github action work
AUTHOR="ALIREZA.ZOLFAGHAR"
VERSION="2.9.0"
VERSION_UPDATED="2025-02-21"
UPLOAD_TO_ALIREZAZ=true
```

⚙️ طراحی step ها

همونطور که همه میدونیم ، بهترین نوع یادگیری و تمرین ، اونیه که در حال انجام کار انجام میشه پس به جای توضیحات در مورد آپشن ها و شرایطشون ، بیاید شیرجه بزنیم توش و در عمل step های مختلف پروژه ی خیالمون رو بنویسیم

1️⃣دریافت محتوای ریپازیتوری تو محیط رانر

لازمه ما به تمام محتوای push شده تو محیط runner که همون ubuntu 22 هست دسترسی داشته باشیم ، پس اولین step رو اینجوری تنظیم میکنیم تا محتوای ریپو رو بیاریم داخل رانر

```
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3
```

در واقع ما داریم برای این کار از یک action آماده به نام checkout استفاده میکنیم که کارش همینه و دیگه نیازی نیست خودمون از اول بنویسیمش

اکشن های آماده ی زیادی وجود داره که شما میتونید سر فرصت توشون بچرخید و باهاشون کارهای خودتون رو ساده تر انجام بدید ، برای دیدن لیست اکشن های آماده لینک زیر شروع خوبیه:

https://github.com/marketplace?type=actions

2️⃣خواندن اطلاعات از کد

تو این پروژه خیالی لازمه ما سه تا چیز رو از code بخونیم : نسخه ی برنامه ، تاریخ آپدیت ، و متغیر آپلود، این کار رو با دستورات لینوکسی انجام میدیم

```
- name: Extract version number
  id: extract_version
  run: |
    VERSION=$( grep VERSION= simple-script.sh | head -n 1 | awk -F'"' '{print $2}')
    echo "VERSION=$VERSION" >> $GITHUB_ENV
    UPDATE=$( grep VERSION_UPDATED= simple-script.sh | head -n 1 | awk -F'"' '{print $2}')
    echo "UPDATE=$UPDATE" >> $GITHUB_ENV
    UPLOAD=$( grep UPLOAD_TO_ALIREZAZ simple-script.sh | head -n 1 | awk -F '=' '{print $2}')
    echo "UPLOAD=$UPLOAD" >> $GITHUB_ENV
```

سه متغیر VERSION، UPDATE و UPLOAD تعریف کردیم و با دستورات ساده ی لینوکسی مثل grep و awk مقدار این متغیر ها رو از کد اصلی اسکریپت که فایل simple-script.sh باشه دریافت کردیم و آخرش این متغیر ها رو به GITHUB_ENV فرستادیم تا به عنوان متغیر محیطی گیت هاب شناخته بشند و در ادامه workflow ازشون برای اهدافمون استفاده کنیم

3️⃣دریافت اطلاعات کامیت

برای خوانایی بیشتر، من معمولا عنوان و توضیحات Commit رو هم از خود گیت هاب میگیرم تا به صفحه ی release اضافه کنم ، اما این مورد ضروری نیست و شما میتونید ازش بگذرید

```
- name: Get Commit Details
  id: commit_details
  run: |
    COMMIT_TITLE=$(git log -1 --pretty=%s)
    COMMIT_BODY=$(git log -1 --pretty=%b)
    echo "COMMIT_TITLE=$COMMIT_TITLE" >> $GITHUB_ENV
    echo "COMMIT_BODY=$COMMIT_BODY" >> $GITHUB_ENV
```

عنوان و توضیحات COMMIT رو از خود گیت‌هاب میگیریم
و مثل دفعه ی قبل به متغیرهای محیطی گیت‌هاب میفرستیم تا بعدا بتونیم ازش استفاده کنیم
به این موضوع دقت کنید که ممکنه کاراکتر های خاصی تو عنوان و توضیحات COMMIT شما باشه که این بخش نتونه به درستی عمل کنه ، پس یا باید از یه شیوه ی استانداردی برای عنوان و توضیحات کامیت استفاده کنید یا باید شیوه استخراج این دو مورد رو تغییر بدید و چیزی مناسب با عادت خودتون رو استفاده کنید
اما در حالت کلی ، چیزی که اینجا براتون نوشتم کار میکنه و در اکثر موارد لازم نیست تغییرش بدید مگر اینکه بخواید اطلاعات جذابتری رو هم از commit استخراج کنید و تو release ثبت کنید

4 ساخت release notes

ما میخوایم تمام این اطلاعاتی که تو مراحل قبلی دریافت کردیم رو جایی به یادگار بگذاریم 😁 تا آیندگان ، وقتی به این پروژه رجوع میکنند ، یه نسخه ی کامل از فایل های اون نسخه از پروژه داشته باشند ، دوما شماره ی نسخه و توضیحات مربوط به commit این نسخه رو ببینند، برای این هدف ، چی بهتر از صفحه ی release وجود داره؟

پس ما تمام این موارد رو تو صفحه ی ریلیز برای هر ورژن منتشر می‌کنیم، و برای این توضیحات release note میسازیم ، که شکل کلی step به این صورت میشه

```
- name: Create release notes
  id: generate_notes
  run: |
    echo "# Version: $VERSION" > release_notes.md
    echo "# Updated: $UPDATE" >> release_notes.md
    echo "# Upload : $UPLOAD  " >> release_notes.md
    echo "If true, it will upload to External Server" >> release_notes.md
    echo "" >> release_notes.md
    echo "## Commit Details" >> release_notes.md
    echo "**Title:** $COMMIT_TITLE" >> release_notes.md
    echo "**Description:** $COMMIT_BODY" >> release_notes.md
    echo "" >> release_notes.md
    echo "## Installation" >> release_notes.md
    echo "curl -k  https://github.com/${{ github.repository}}/releases/download/$VERSION/simple-script.sh | bash" >> release_notes.md
    echo "" >> release_notes.md
```

همونطور که مشخصه ما اومدیم با چند تا دستور **echo** محتوای مورد نظرمون به فایلی با نام **release_notes.md** اضافه کردیم و تو مرحله پایانه به گیت‌هاب میگیم از این فایل به عنوان توضیحات ریلیز استفاده کنه ، تمام متغیر ها رو تو مراحل قبل استخراج کردیم و به متغیر محیطی گیت‌هاب دادیم تا همینجا ازش استفاده کنیم

فقط ممکنه یکی از خطوط به نظر گیج کننده باشه :

```
curl -k  https://github.com/${{ github.repository }}/releases/download/$VERSION/simple-script.sh
```

ما این خط رو اضافه کردیم که توضیحات هر نسخه یه آموزش ساده ی نصب مخصوص هر نسخه رو هم داشته باشه و به همین خاطر لینک دانلود رو به صورت داینامیک ساختیم

گیت هاب خودش آدرس ریپو رو جایگزین متغیر **github.repository** میکنه و شماره ی نسخه ای که از اسکریپت استخراج کرده رو هم جایگزین **VERSION** میکنه تا لینک هر ورژن به صورت داینامیک و مخصوص همون ورژن ساخته بشه

اگه براتون سواله که سایر قسمت های لینک رو از کجا آوردیم ، باید بگم این آدرسی هست که گیت‌هاب به صورت پیشفرض به هر ریلیز شما میده ، کافیه release یه tag داشته باشه ، که اینجا تگ ما همون version هست که از داخل code استخراج شده

و باید دقت کنید تو این مثال ، ما آدرس فایل پروژه رو مستقیم از توی ریپو ندادیم ، چون در اون صورت همیشه به آخرین نسخه ی فایل اشاره می کرد ، در حالی که ما میخواستیم به نسخه ی خاصی از فایل اشاره کنیم و به یادگار بگذاریم

5⃣ساخت یک فایل Zip از تمام ریپازیتوری

پروژه ی خیالی ما، یه اسکریپت ساده ی bash یه فایل pdf داره که همین آموزش هست و از همه مهم تر یه فایل release که workflow هست و ما هر سه فایل رو به release اضافه میکنیم چون میخوایم بعدا به اونها لینک مستقیم داشته باشیم ، اما تو یه پروژه ی واقعی ما فایل و دایرکتوری های دیگه ای داریم که لینک مستقیم لازم ندارند اما نیاز داریم تا به عنوان نسخه پشتیبان از اونها نگهداری کنیم، پس برای اینکه بکاپ کامل باشه ، ما از تمام محتوای ریپازیتوری هم یک فایل zip می‌سازیم و به ریلیز اضافه می‌کنیم

```
    - name: Create simple-github-action.zip
      run: |
        zip -r simple-github-action.zip . -x ".git/*"
```

این فایل میتونه هر اسمی داشته باشه و با یک دستور لینوکسی ساده ساخته میشه ، اینجا من zip رو انتخاب کردم چون اطمینان داشتم تو محیط رانر به صورت پیشفرض نصب هست اما شما میتونید هر پسوند دیگه ای مثل tar.gz |.tar.zst | tar و... انتخاب کنید ، فقط باید مطمئن بشید پیش نیاز هاش روی رانر نصبه و اگه نصب نبود از قبل با یه دستور apt نصبش کنید
موضوع بعدی اون سوئیچ x هست که شاید براتون سوال ایجاد کنه
باید بگم اون رو گذاشتم تا بدونید محدودیتی برای دستورات نداریم و هر کاری تو یک لینوکس عادی انجام میدیم ، اینجا هم قابل انجامه ، با این آپشن من مشخص کردم یک نسخه از همه چیز داخل فایل zip قرار بگیره به غیر از محتوای پوشه ی .git

6⃣ساخت و انتشار release

تا اینجا ما هر اطلاعاتی می خواستیم رو از پروژمون استخراج کردیم، و حتی براش release note ساختیم و دیگه همه چیز آماده است تا موشکی که ساختیم رو به هوا بفرستیم ، برای ساخت ریلیز میایم از یکی دیگه از اکشن های آماده ی گیت‌هاب به نام **action-gh-release** استفاده میکنیم که از ما عنوان ریلیز، tag و فایل های اون رو می‌گیره و منتشرش می‌کنه
فایل های که داریم اینها بودند

دو فایل از ریپازیتوری

- **simple-script.sh**
- **simple-github-action.pdf**

یک فایل zip که تو ریپازیتوری نیست و ما هربار خودمون میسازیمش:

- **simple-github-action.zip**

و یک فایل release notes که اون رو هم هر بار میسازیم تا توضیحات توش باشه

- **release_notes.md**

شکل این مرحله اینطوری میشه:

```
      - name: Create release
        uses: softprops/action-gh-release@v1
        with:
          name: ${{ env.VERSION }}
          tag_name: ${{ env.VERSION }}
          body_path: ./release_notes.md
          files: |
            simple-script.sh
            simple-github-action.pdf
            simple-github-action.zip
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

- ما VERSION رو که تو مراحل قبل از code استخراج کردیم هم به جای نام release معرفی میکنیم و هم به جای tag
- فایل .md که تو مراحل قبل ساختیم رو هم به عنوان متن ریلیز معرفی میکنیم
- سه فایل دیگه رو هم به عنوان فایل هایی که باید تو release قرار داده بشند اعلام می کنیم

بخش مهم از این step قسمت env هست :
تو این workflow شما مثل یک کاربر دارید با گیت‌هاب تعامل میکنید ، تمام این کارها رو میتونستید به صورت دستی انجام بدید و در نهایت به صفحه release از ریپازیتوری برید و عنوان و توضیحات رو بنویسید و فایل ها رو ضمیمه کنید و در نهایت منتشرش کنید ، حالا که قراره workflow به جای شما انجامش بده ، لازمه دسترسی انجامش رو داشته باشه ، پس با این تیکه از code شما به گیت‌هاب می‌فهمونید که خودکار دسترسی لازم رو اعطاء کنه

✅ تمومه ! بریم تست کنیم

کار تمومه ، الان فقط کافیه یک push جدید روی این ریپازیتوری بفرستیم و بشینیم نگاه کنیم که چطور Github Action به جای ما بقیه ی کارها رو انجام میده و لذت ببریم