I. GitHub Actions Komponenten

GitHub Actions ist eine Automatisierungsplattform von GitHub, die speziell dafür entwickelt wurde, Workflows für Continuous Integration (CI) und Continuous Deployment (CD) sowie andere Automatisierungsaufgaben direkt im GitHub-Repository zu erstellen. Die Plattform besteht aus verschiedenen Komponenten, die gemeinsam dazu beitragen, Workflows flexibel und skalierbar zu gestalten. Hier sind die Hauptkomponenten von GitHub Actions:

1. Workflows

Workflows sind Automatisierungsprozesse, die in einer Datei namens, .github/workflows/<workflow-name>.yml definiert werden.

Sie bestehen aus einer Reihe von Anweisungen, die ausgeführt werden, wenn bestimmte Ereignisse eintreten.

Eigenschaften:

- Werden in YAML geschrieben.
- Können beliebig viele Jobs enthalten.
- Werden durch Events getriggert (z. B. Push, Pull Request).

2. Events

Events sind Auslöser, die einen Workflow starten.

Beispiele:

- **Repository-Aktivitäten**: push, pull_request, issues, release.
- **Zeitgesteuerte Trigger**: schedule (mit Cron-Syntax).
- **Externe Ereignisse**: workflow_dispatch (manuelles Triggern) oder Webhooks.
- **Flexibilität**: Ermöglicht es, Workflows auf spezifische Bedingungen zuzuschneiden.

3. Jobs

Jobs sind Einheiten innerhalb eines Workflows, die unabhängige Aufgaben ausführen.

Eigenschaften:

- Werden parallel ausgeführt, sofern keine Abhängigkeiten definiert sind.
- Können Abhängigkeiten zu anderen Jobs haben, z. B.: needs: [build]
- Bestehen aus mehreren Schritten (steps).

4. Steps

Schritte innerhalb eines Jobs, die Befehle oder Aktionen ausführen.

Eigenschaften:

- Werden sequenziell innerhalb eines Jobs ausgeführt.
- Können Skripte oder vorgefertigte Aktionen verwenden.
- Beispiel:

  steps:

  - name: Checkout Code

    uses: actions/checkout@v3

  - name: Run Tests

    run: npm test

Name	Uses	Run
Checkout Code	actions/checkout@v3	
Run Tests		npm test

5. Actions

Wiederverwendbare Komponenten, die eine spezifische Aufgabe in einem Workflow ausführen.

Arten von Actions:

- **Docker-Actions**: Aktionen, die in einem Docker-Container laufen.
- **JavaScript-Actions**: Direkt in Node.js geschrieben.
- **Composite-Actions**: Kombination aus mehreren Befehlen und anderen Actions.
- **Verwendung**: Actions können aus dem GitHub-Marktplatz (GitHub Marketplace) bezogen oder selbst erstellt werden.

6. Runners

Server, die die Jobs eines Workflows ausführen.

Arten von Runners:

- **GitHub-hosted Runners**: Von GitHub bereitgestellte und verwaltete Server (Linux, Windows, macOS).
- **Self-hosted Runners**: Benutzerdefinierte Runners, die auf eigenen Servern laufen.
- **Funktion**: Führen die in den Steps definierten Befehle aus.
- **Konfiguration**: Kann spezifische Umgebungen oder Hardwareanforderungen erfüllen.

7. Artifacts

Dateien, die von Workflows generiert und gespeichert werden, um sie später wiederzuverwenden.

Beispiele:

- Build-Artefakte (z. B. ausführbare Dateien).
- Testergebnisse.
- **Speicherung**: Mit der upload-artifact-Action hochgeladen und mit download-artifact heruntergeladen.

8. Secrets

Sichere Variablen, die sensible Daten wie API-Schlüssel oder Tokens enthalten.

Verwendung:

- Werden in den Repository-Einstellungen definiert.
- Im Workflow als Umgebungsvariablen zugänglich:
- env:
- MY_SECRET: ${{ secrets.MY_SECRET }}

9. Environments

Konfigurationsumgebungen für Workflows (z. B. Entwicklung, Test, Produktion).

Eigenschaften:

- Können Secrets und Genehmigungen pro Umgebung enthalten.
- Beispiel für eine Produktionsumgebung:
- environment: production

10. Context

Kontextinformationen, die in Workflows verwendet werden können.

Beispiele:

- **GitHub-Context**: ${{ github.repository }} enthält den Repository-Namen.
- **Job-Context**: ${{ job.status }} gibt den Status eines Jobs an.
- **Runner-Context**: ${{ runner.os }} gibt das Betriebssystem des Runners an.

GitHub Actions bietet durch die Kombination dieser Komponenten eine mächtige und flexible Plattform zur Automatisierung von Softwareentwicklungsprozessen.

II. Erstellung einer einfachen CI/CD-Pipeline

A. Test Workflow erstellen

Der YAML-Workflow beschreibt eine Pipeline für die kontinuierliche Integration (CI), um den Codestil in einem Laravel-Projekt zu testen und zu pflegen. Hier eine Erklärung der drei großen Teile :

1. Name Section

Im Feld name können Sie dem Workflow einen Titel geben, damit er in der GitHub Actions-Oberfläche leichter zu erkennen ist.

```yaml
name: "Testen und Verwalten des Codestils in einem Laravel-Projekt."
```

2. On Section

Der Abschnitt on legt die Ereignisse fest, die den Workflow auslösen. Er kann manuelle Auslöser (workflow_dispatch), bei push on main branch oder den Aufruf aus einem anderen Workflow (workflow_call) umfassen.

```yaml
on:
  # Der Workflow kann manuell über die GitHub-Benutzeroberfläche gestartet
werden.
  workflow_dispatch:
    inputs:
      name:
        description: "Specify as Job name"
        required: false
        type: string
        default: "Run a Laravel Tests"

  # Der Workflow kann von einem anderen Workflow aufgerufen werden.
  workflow_call:
    inputs:
      name:
        description: "Specify as Job name"
        required: false
        type: string
        default: "Run a Laravel CS test"
      code_directory:
        description: "Verzeichnis, das überprüft werden soll"
        required: false
        type: string
      php_version:
        description: "PHP-Version für die Anwendung"
        required: false
        type: string
        default: '8.1'
```

```
  # Der Workflow wird automatisch ausgelöst, wenn Änderungen in den Branch
"main" gepusht werden.
  push:
    branches:
      - main
```

3. Job Section

Der Abschnitt Jobs enthält Anweisungen zum Ausführen bestimmter Aufgaben. Jeder Job beinhaltet :

- Einen Namen (name), der oft über die Einträge eingestellt wird.
- Eine virtuelle Maschine zur Ausführung (runs-on).
- Eine Reihe von Schritten (steps).

```
jobs:
  laravel-tests:
    name: ${{ inputs.name }}
    runs-on: ubuntu-latest
    steps:
    - uses: shivammathur/setup-php@15c43e89cdef867065b0213be354c2841860869e
      with:
        php-version:  ${{ inputs.php_version }}
    - uses: actions/checkout@v4

    - name: Copy .env
      run: php -r "file_exists('.env') || copy('.env.example', '.env');"

    - name: Install Dependencies
      run: |
        composer update
        composer install -q --no-ansi --no-interaction --no-scripts --no-
progress --prefer-dist

    - name: Generate key
      run: php artisan key:generate

    - name: Directory Permissions
      run: chmod -R 777 storage bootstrap/cache

    - name: Install PHP CodeSniffer
      run: composer global require squizlabs/php_codesniffer

    - name: Run PHP Code Style Test
      run: ~/.composer/vendor/bin/phpcs --standard=PSR12 --report=checkstyle -
-report-file=$(pwd)/phpcs-report.xml .
```

```
    - name: Fix PHP Code Style
      run: ~/.composer/vendor/bin/phpcbf --standard=PSR12 --report=checkstyle
--report-file=phpcs-report.xml app/

    - name: Upload PHPCS Report as Artifact
      uses: actions/upload-artifact@v3
      with:
        name: phpcs-report
        path: phpcs-report.xml
```

B. Build Workflow erstellen

Diese GitHub Actions YAML-Konfiguration beschreibt eine CI/CD-Pipeline zum Erstellen und Übertragen eines Docker-Images an Docker Hub, wenn Code in den Entwicklungszweig übertragen wird

```yaml
name: "Build and Push Artefact to Docker Hub"

env:
  TAG_NAME: v1.1.0
  APP_NAME: laravel_admin

on:
  push:
    branches:
      - develop

jobs:
  build-and-push:
    name: Build and Push Docker Image
    runs-on: ubuntu-latest

    steps:
    # Vérifiez le code source
    - name: Checkout repository
      uses: actions/checkout@v4

    # Configuration Docker Login
    - name: Log in to Docker Hub
      uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}

    # Construire l'image Docker
```

```
    - name: Build Docker Image
      run: |
        docker build -t ${{ secrets.DOCKER_USERNAME }}/$APP_NAME:$TAG_NAME .

    # Pousser l'image sur Docker Hub
    - name: Push Docker Image to Docker Hub
      run: |
        docker push ${{ secrets.DOCKER_USERNAME }}/$APP_NAME:$TAG_NAME
```

B: Deploy Workflow erstellen

Diese GitHub Actions YAML-Konfiguration definiert einen CI/CD-Workflow für die Bereitstellung einer Laravel-Anwendung in einer Staging- (oder Produktions-) Umgebung. Er kann entweder durch das Pushen von Code in den Entwicklungszweig oder manuell über workflow_dispatch ausgelöst werden.

```yaml
name: "Deploy on staging env"

env:
  DIR_NAME: sosk_workshop
  SSH_PRIVATE_KEY: ${{ secrets.SSH_PRIVATE_KEY }}
  APP_NAME: "laravel-admin"

on:
  push:
    branches:
      - develop

  workflow_dispatch:

jobs:
  deploy:
    runs-on: ubuntu-latest
    environment:
      name: 'production'

    steps:
      # Clone the repository
      - name: Check code
        uses: actions/checkout@v3

      # Install Docker on the GitHub runner
      - name: Set up Docker
        uses: docker/setup-buildx-action@v1

      # Docker login to Docker Hub
      - name: Log in to Docker Hub
```

```
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      # Set up SSH key for VM connection
      - name: Set up SSH key
        run: |
          echo "$SSH_PRIVATE_KEY" > private_key
          chmod 600 private_key

      # Deploy the Laravel app via Docker on Azure VM
      - name: "Deploy Laravel app via Docker on Azure VM"
        env:
          VM_USER: ${{ secrets.VM_USER }}
          VM_IP: ${{ secrets.VM_IP }}
        run: |
          ssh -T -i private_key -o StrictHostKeyChecking=no $VM_USER@$VM_IP
'echo "SSH connection successful"'

          ssh -T -i private_key -o StrictHostKeyChecking=no $VM_USER@$VM_IP <<
EOF
            mkdir -p /home/$VM_USER/$DIR_NAME
            cd /home/$VM_USER/$DIR_NAME
            docker pull ${{ secrets.DOCKER_USERNAME }}/ $APP_NAME:v1.0.0
            docker build -t $APP_NAME:v1.0.0 .
            docker run -d -p 8000:8000  $APP_NAME:v1.0.0
          EOF
```

III. Zusammenfassend

Zusammenfassend lässt sich sagen, dass der Test-, Build- und Deployment-Workflow mit GitHub Actions eine einfache und sichere Verwaltung des gesamten Anwendungslebenszyklus ermöglicht. Er vereinfacht die Durchführung von Tests, die Erstellung von Docker-Images und die Bereitstellung auf einer virtuellen Maschine und gewährleistet so schnelle und konsistente Bereitstellungen. Diese Automatisierung reduziert menschliche Fehler, erhöht die Effizienz und beschleunigt die Aktualisierung von Anwendungen in der Produktion.