SOFWERX

First Steps Toward CI/CD with WordPress, SiteGround, and GitHub

Lewis Morgan

Patrick Schippers

Jim Ladd

December 29, 2020

Contents

Introduction

Continuous Integration/Continuous Development (CI/CD) processes have been at the forefront of the software development industry for several years now. While CI/CD concepts are easy to grasp, they can be difficult to implement while maintaining schedules and achieving milestones. This paper describes our initial steps toward a CI/CD pipeline for our client facing websites that use WordPress and SiteGround. We incorporate GitHub's repository to maintain our source code while using GitHub's Actions service to automate the deployment to a staging environment hosted by SiteGround.

Background

The current deployment process for the SOFWERX's websites is very traditional. The coding and developer-based testing activities are performed on local environments. When the programmer deems the code ready, it is deployed to a remote staging environment hosted by SiteGround for a small set of internal users to review. Once their approval is received, the code is deployed to the production environment. The data maintained in the different databases follows a reverse path. The staging database is refreshed with data from the production environment and then the development database is refreshed with the data from the staging environment. A diagram of these processes is shown below:

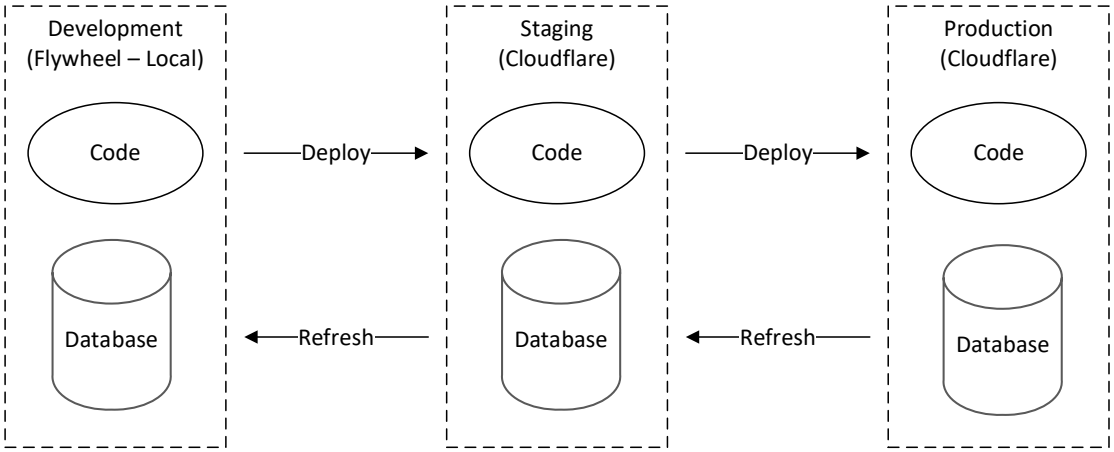

Figure 1 - Traditional development process

Not only is this process traditional, it is also very manual and tedious. The team at SOFWERX wanted to move toward automation with a Continuous Integration/Continuous Deployment (CI/CD) pipeline but we needed to do so in logical steps that didn't interfere with our existing schedules and milestones.

The Teamwerx website (teamwerx.org) within the SOFWERX realm provides a platform to host prize challenges and is currently not being used. This website is built with WordPress and hosted by SiteGround. Since this is the same technology and hosting service used by our primary website system, the Teamwerx website provides an excellent proof of concept base for migrating toward a CI/CD pipeline.

Automating the Staging Deployment Process

The goals of this initial effort were to 1) place the deployment artifacts under version control and 2) automate the deployment process from the development environment to the staging environment.

GitHub was selected for the repository service. It is a very popular selection in the industry and is well known within our company. In fact, SOFWERX currently has 283 repositories in GitHub. Unlike most of the other repositories, this one is private with access only to the in-house development team.

The process steps for the promotion to the staging environment include:

1. The user will checkout the develop branch from GitHub
2. The user will push changes to the remote repository.
3. The user will submit a pull request.
4. GitHub will perform a backup of the staging database.
5. GitHub will transfer the code to the staging environment.

A diagram of this process is shown below:

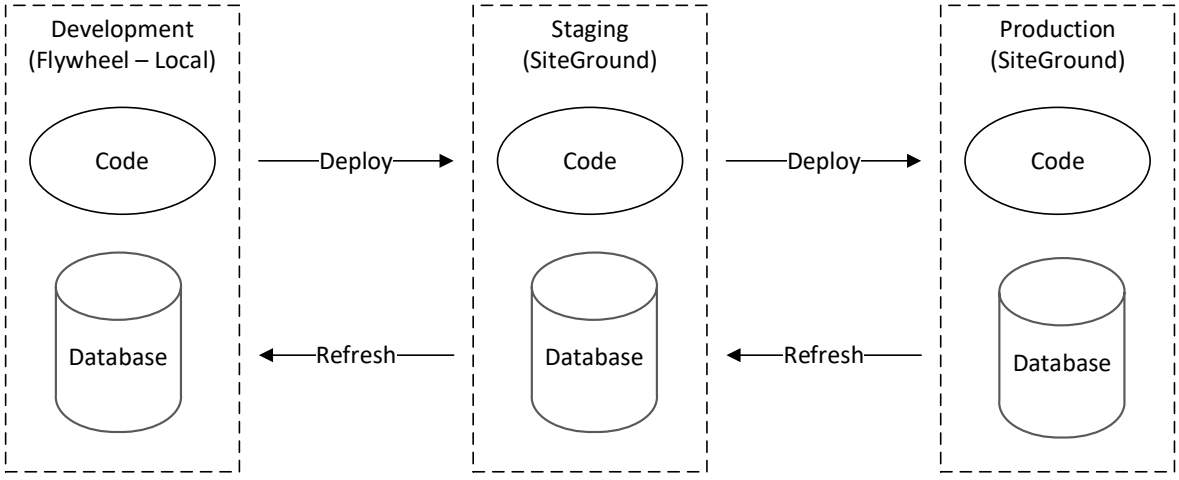

Figure 2 - Initial CI/CD pipeline model

GitHub provides a service called Actions that helps with the automation of tasks within the development process. The Actions service consists of event-driven workflows. Each workflow contains a job which consists of a set of steps. These steps control the sequence of the actions which perform the actual processing. A graphical representation of this model is shown below:

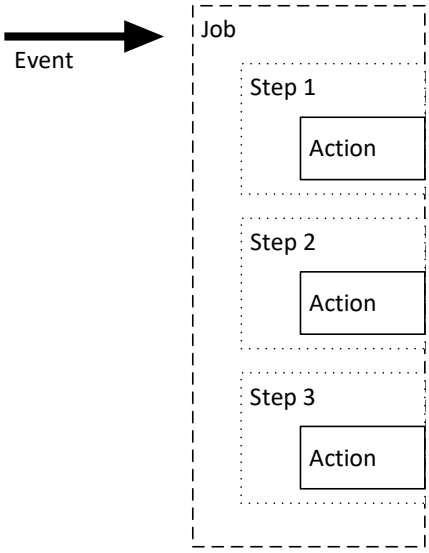

Figure 3 - GitHub workflow components

GitHub uses YAML (Yet Another Markup Language) syntax to define the events, jobs, and steps used in the automation. The structure of the workflow YAML file is very similar to the model presented above. The YAML file used in our project is presented in Appendix A of this document.

One of the key features of the GitHub Actions service is the use of "secrets". Secrets are variables that store sensitive data and can only be viewed by services such as the Actions service. Our YAML file has the following eleven secret types, grouped into three different sets.

```
SSH_PRIVATE_KEY
SSH_PASSPHRASE
SSH_HOST
SSH_PORT
SSH_USER

DB_STAGE_USER
DB_STAGE_PASSWORD
DB_STAGE_NAME

FTP_STAGE_SERVER
FTP_STAGE_USER
FTP_STAGE_PASSWORD
```

Our workflow is partitioned into two major jobs, ***backup*** and ***deploy***. The ***backup*** job creates a copy of the staging database and uploads this file as an artifact of the workflow instance. The major challenge with this job was to programmatically access the staging environment via SSH in order to make a SQL dump of the database. A config file used by the SSH software is dynamically created on the Ubuntu-based runner instance. The contents of the config file consist primarily of the SSH secrets. The SSH_ASKPASS feature of the SSH software bypasses the prompting for a passphrase. Finally, the job uses the "actions/upload-artifact" action to upload the backfile to GitHub as an artifact of this workflow instance.

The ***deploy*** job relies on the successful completion of the ***backup*** job as a pre-requisite condition. If the database was successfully archived, the ***deploy*** job uses the "actions/checkout" action to checkout the repository. The next step uses the "SamKirkland/FTP-Deploy-Action@3.1.1" action to upload the repository to the staging environment in SiteGround via FTP.

Looking Back and Moving Forward

This project proved to be more challenging than first envisioned. At this point in the adoption lifecycle of CI/CD pipeline technologies, we expected more knowledge to be available with the WordPress, GitHub, and SiteGround stack. However, the information that we uncovered was scattered and fragmented. After significant effort, we are pleased with the results of the initial effort and excited with continuing moving forward.

The next steps will be partitioned into two major tasks. The first is to automate the deployment from the staging environment into the production environment. The second is to reduce the amount of manual database management required. We wish to isolate the WordPress generated data from the user generated data so that deployment of new WordPress components will not impact the existing data structures or values.

Appendix A – YAML file

```
name: Staging Deployment

on:
  push:
    branches:
      - staging

jobs:
  # Database Backup
  backup:
    runs-on: ubuntu-latest
    name: Create Database Backup
    env:
      SSH_KEY: ${{ secrets.SSH_PRIVATE_KEY }}
      SSHPASS: ${{ secrets.SSH_PASSPHRASE }}
      SSH_HOST: ${{ secrets.SSH_HOST }}
      SSH_PORT: ${{ secrets.SSH_PORT }}
      SSH_USER: ${{ secrets.SSH_USER }}
    steps:
      # 1. Create a run timestamp output with id=time name=date
      - name: Get date timestamp
        id: time
        run: echo "::set-output name=date::$(date '+%Y-%m-%d_%H-%M-%S')"
      # 2. Configure SSH Access on GH CI to the staging machine
      - name: Configure SSH Access
        # Creates a ssh config file with the key/host/port/user
        run: |
          mkdir -p ~/.ssh/
          echo "$SSH_KEY" > ~/.ssh/staging.key
          chmod 600 ~/.ssh/staging.key
          cat >>~/.ssh/config <<END
          Host staging
            HostName $SSH_HOST
            User $SSH_USER
            Port $SSH_PORT
            IdentityFile ~/.ssh/staging.key
            StrictHostKeyChecking no
            LogLevel INFO
          END
      # 3. Create the $SSH_ASKPASS with script that just echos $SSHPASS value
to negate asking for passphrase during ssh connection
      - name: Create SSH_ASKPASS
        run: |
          cat > /tmp/ssh-askpass-script <<EOL
          #!/bin/bash
          echo $SSHPASS
          EOL
          chmod +x /tmp/ssh-askpass-script
```

```
      # 4. Sets display to :0.0 and uses setsid to ensure that ssh will use
$SSH_ASKPASS value. Connects to the staging defined in hosts file and runs
mysqldump.
      - name: Run mysqldump via SSH
        run: |
          DISPLAY=":0.0" SSH_ASKPASS="/tmp/ssh-askpass-script" setsid ssh
staging mysqldump --skip-add-drop-table --skip-extended-insert -u ${{
secrets.DB_STAGE_USER }} -p${{ secrets.DB_STAGE_PASSWORD }} ${{
secrets.DB_STAGE_NAME }} > backup-${{ steps.time.outputs.date }}.sql
      # 5. Save the copied backup as an artifact
      - name: Archive Database Backup
        uses: actions/upload-artifact@v2
        with:
          name: backup-${{ steps.time.outputs.date }}
          path: backup-${{ steps.time.outputs.date }}.sql
  # FTP Deploy
  deploy:
    runs-on: ubuntu-latest
    # Only attempt to deploy the files if the mysql server has backed up
    needs: backup
    name: Deploy via FTP
    steps:
      # 1. Checkout the Repository up to 20 commit history
      - uses: actions/checkout@v2
        with:
          fetch-depth: 20
      - name: FTP Deploy
        uses: SamKirkland/FTP-Deploy-Action@3.1.1
        with:
          ftp-server: ${{ secrets.FTP_STAGE_SERVER }}
          ftp-username: ${{ secrets.FTP_STAGE_USER }}
          ftp-password: ${{ secrets.FTP_STAGE_PASSWORD }}
          # The directory that will be uploaded. Anything that is not in this
folder will not be uploaded.
          local-dir: public
          # TODO: Remove dry run when finalized
          git-ftp-args: --dry-run
```