Exploring Github
(https://dasarpai.com/dsblog/Exploring-Github)

14 minute read

Question: What is github package and what can I do with this?

GitHub Packages is a service that allows you to host and manage packages (e.g., code libraries, Docker containers) within GitHub. Here's a breakdown of what you can do or achieve with GitHub Packages:

1. Host and Share Packages

- **Languages Supported**: You can host packages for multiple programming languages, such as:
  - **npm** (JavaScript/Node.js)
  - **Maven** (Java)
  - **NuGet** (C#/.NET)
  - **RubyGems** (Ruby)
  - **Python** (PyPI)
  - **Docker** (Container images)
- **Share Packages**: You can share packages publicly or privately within an organization or with collaborators.

2. Seamless Integration with GitHub Repositories

- You can tightly integrate packages with GitHub repositories. This allows for automated versioning, releasing, and managing packages in tandem with your code changes.
- **Release Automation**: Automate package publishing during a GitHub release or via CI/CD pipelines (GitHub Actions).

3. Versioning and Dependency Management

- GitHub Packages supports semantic versioning. This means you can publish new versions of your package and manage dependencies within your projects.
- **Dependency Graph**: View dependencies in your repository, understand their relationships, and check for vulnerabilities.
- **Link with Code**: GitHub Packages is linked with repositories, so you can easily trace which version of your package is linked with which repository or commit.

4. Private Package Hosting

- Host private packages that are accessible only to specific people, teams, or organizations.
- **Role-based Access Control**: Control who can view, download, or publish to your packages using GitHub's role-based access.

5. Container Hosting (Docker and OCI Images)

- Use GitHub Packages as a Docker registry to store and manage your Docker images.
- **Build & Deploy Containers**: Automate the building and publishing of Docker images via GitHub Actions and deploy them using platforms like Kubernetes.

6. Integrate with GitHub Actions

- You can automate the entire package lifecycle (build, test, publish) using GitHub Actions.
- **CI/CD Pipelines**: Use GitHub Actions to automatically publish packages whenever code is merged, tests pass, or new releases are created.

7. Security and Auditing

- **Signed Packages**: GitHub Packages can be configured to require packages to be signed, adding an extra layer of security.
- **Package Scanning**: GitHub also scans packages for known vulnerabilities, helping you avoid security risks in your dependencies.

8. Search and Discovery

- **Discover Public Packages**: Search for public packages published by other users or organizations.
- **Tagging and Labeling**: Use tags and labels to categorize your packages, making them easier to find and manage.

9. Package Insights and Usage Metrics

- View download statistics, such as how many times a package has been downloaded, and track overall usage trends.
- **Repository Insights**: See which repositories are consuming or contributing to your packages.

10. Custom Registries

- GitHub allows you to host your own private or public package registries for your team or organization.

Use Cases for GitHub Packages:

- **Publishing Libraries**: Share reusable libraries and code modules (e.g., npm packages, Python libraries) with teams or the public.
- **Managing Internal Tools**: Host internal dependencies or tools that are not meant to be shared publicly.
- **Docker Image Management**: Manage and distribute Docker images for deploying applications.
- **Automated Releases**: Build automated pipelines to publish, test, and deploy code to production environments.

GitHub Packages is particularly useful for teams or individuals who want a fully integrated solution to manage both their source code and dependencies in one place.

Question: What is sementic versioning and how Github handles this?

Semantic versioning (SemVer) is a versioning scheme that helps communicate changes in a software project in a clear and predictable manner. The idea behind **semantic** is that each part of the version number conveys a specific meaning about the state of the software.

The format for semantic versioning is typically:

```
MAJOR.MINOR.PATCH
```

Each of these components signifies the type of change in the software:

1. MAJOR version (X)

- Increment this when you make incompatible or breaking changes.
- **Example**: If you change the API or make updates that are not backward-compatible, you increase the major version.
- **Example**: Going from `2.5.1` to `3.0.0`.

2. MINOR version (Y)

- Increment this when you add new features or functionality in a backward-compatible way.
- **Example**: Adding a new feature without changing existing functionality increases the minor version.
- **Example**: Going from `2.5.1` to `2.6.0`.

3. PATCH version (Z)

- Increment this when you make backward-compatible bug fixes or improvements.
- **Example**: Fixing a bug that does not affect the API or features increments the patch version.
- **Example**: Going from `2.5.1` to `2.5.2`.

Optional Components:

- **Pre-release version**: Can be appended to indicate that a release is in a pre-release state (e.g., alpha, beta):

  ```
  2.0.0-alpha.1
  ```

- **Build metadata**: Additional information (e.g., build information) can be appended:

  ```
  1.0.0+20130313144700
  ```

Examples:

- **Version 1.0.0**
  - This is the initial stable release. It suggests that breaking changes could occur in future `2.0.0` or higher versions.
- **Version 1.1.0**
  - This means a new feature has been added, but it is backward-compatible.
- **Version 1.1.1**
  - A bug or small issue was fixed without affecting any existing features or breaking anything.

Why Semantic Versioning Matters:

- **Predictability**: Developers can easily understand the nature of changes (breaking changes, new features, bug fixes) just by looking at the version number.
- **Dependency Management**: When using a package, you can safely update PATCH versions, cautiously update MINOR versions, and avoid MAJOR version changes unless you're ready for potential breaking changes.

Summary:

- **MAJOR** for incompatible changes.
- **MINOR** for new, backward-compatible features.
- **PATCH** for backward-compatible bug fixes.

This standardized system helps everyone (users, contributors, package managers) maintain consistency and avoid unintended issues when updating software.

Question: How github action takes care of versioning?

GitHub Actions itself doesn't automatically determine whether a build should be marked as a major, minor, or patch version. However, there are **ways** you can **configure GitHub Actions** to automate the process of versioning, based on the changes being made to your code.

Typically, GitHub Actions relies on:

1. **Manual Triggers**: You or your team define the version (major, minor, or patch) based on the nature of the changes.
2. **Automated Tools or Conventions**: You can use third-party tools or commit message conventions to automate version bumps.

Here's how this works in practice:

1. Manual Versioning

- **When to bump a version** is decided manually by the developer or maintainer.
- You can manually specify in your `package.json` (for JavaScript projects) or other configuration files which version to bump.
- Example in a workflow: ```yaml steps:
    - name: Update version to 1.2.0 run: npm version minor ```

2. Automated Semantic Versioning (using tools)

You can automate versioning using tools like **semantic-release** or **commit message conventions**. These tools help automate version bumps based on the types of changes that are made to the code.

Example: semantic-release

semantic-release is a tool that analyzes commit messages and decides if the version should be bumped (major, minor, or patch).

- **How it works**:
  - You follow a **conventional commit message** format, where each commit message clearly defines the type of change (e.g., feature, fix, breaking change).
  - Based on the commit types, `semantic-release` will automatically bump the correct version:
    - `fix:` → bumps **patch** version.
    - `feat:` → bumps **minor** version.
    - `BREAKING CHANGE:` in the commit body → bumps **major** version.
- **GitHub Actions setup with semantic-release**: Here's a basic workflow to use `semantic-release` with GitHub Actions:

```yaml
name: Release

on:
  push:
    branches:
      - main

jobs:
  release:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Setup Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '14'

      - name: Install dependencies
        run: npm install

      - name: Run semantic-release
        env:
          GITHUB_TOKEN: $
        run: npx semantic-release
```

- In this example, `semantic-release` automatically analyzes your commit history and pushes a new tag or release based on the changes in your commits.

3. Using Conventional Commit Messages

You can also enforce a **conventional commit** style in your project, where each commit message is structured in a way that signals what kind of version bump should happen:

- `fix: ...` → Patch
- `feat: ...` → Minor
- `BREAKING CHANGE: ...` → Major

Then, based on these messages, a tool like `standard-version` or `semantic-release` can automatically decide which version bump to apply.

4. GitHub Releases and Tags

You can also integrate GitHub releases into this process:

- When you tag a release (manually or via automation), you can specify the version based on the changes.
- GitHub Actions can automate the process of tagging new versions during the CI/CD pipeline.

Conclusion

GitHub Actions doesn't automatically determine version bumps (major, minor, patch) on its own, but it can be configured to use tools like **semantic-release** or conventions like **Conventional Commits** to automate the process. You can manually or automatically decide when to change versions based on the types of changes in your codebase.

Question: What is the lifecycle of the code between writing to deployment?

From writing code for some software features in visual studio to making a release built what are the different stages code go through within git? You can tell all the steps with oneline description?

1. **Write Code (Feature Development)**: Developers write new code or modify existing code to implement features or fix bugs.
2. **Stage Changes**: Selected files are marked for inclusion in the next commit using `git add`.
3. **Commit Changes**: A snapshot of the staged changes is saved to the local repository with a message using `git commit`.
4. **Push to Remote Repository (Feature Branch)**: The committed changes are pushed to the remote repository in a feature-specific branch using `git push`.
5. **Create a Pull Request (PR)**: A request is made to merge the feature branch into the main branch, typically initiating a review process.
6. **Code Review**: Team members review the code, providing feedback and suggesting improvements or fixes.
7. **Make Requested Changes (if any)**: Developers make any necessary changes based on feedback and push updated commits to the feature branch.
8. **Approve and Merge PR (to Main Branch)**: Once approved, the feature branch is merged into the main branch, often using a GitHub or GitLab interface.
9. **Run CI/CD Pipelines**: Automated pipelines start, running tests and building the software to ensure everything works correctly.
10. **Run Tests**: Automated tests are executed to verify that the new code doesn't break existing functionality.
11. **Build the Application**: The code is compiled or packaged into a build artifact, ready for deployment or distribution.
12. **Tag a Release (Versioning)**: A new version number is assigned using semantic versioning, and a tag is created in Git for this release.
13. **Generate Release Notes**: A summary of changes included in the release is prepared, often automatically based on commits or PR descriptions.
14. **Push Release Tag**: The release tag is pushed to the remote repository, making the release official.
15. **Deploy to Staging/Production**: The built application is deployed to a staging or production environment for users to access.
16. **Release Build Available for Users**: The final release is published, and users can download or use the new version of the software.

This is the typical flow in a modern Git-based development environment with CI/CD pipelines and release management tools.

Question: How to organize diverse technology projects, business projects, POC in GitHub?

To organize and manage your diverse projects effectively on GitHub, you can adopt a structured approach for configuration management and repository organization. Here's a strategy tailored to your situation:

1. Categorize by Industry and Technology

- **Create distinct GitHub repositories for each project** based on industry or technology. For instance, you can categorize them as:
  - **Machine Learning** (ML)
  - **Web Development** (WebDev)
  - **Mobile Development** (Mobile)
  - **POC** (for Proof-of-Concept projects)
  - **Industry-Specific Projects** (like Healthcare, Finance, etc.)
- Within each industry/technology, create repositories for individual projects. Each repository should be specific to a project or a technology framework (e.g., PyTorch, TensorFlow).

2. Create Organization or Grouped Repos

- If you have multiple repositories related to a specific technology or framework, **use GitHub Organizations** to manage them. For example, an organization could be created for all your machine learning projects.
- You can also group your projects using **GitHub Topics** (e.g., `machine-learning`, `web-development`, `mobile-apps`, etc.) to help others easily navigate through similar projects.

3. Fork Repositories and Use Templates

- For projects where you extend existing repositories, **fork them** but maintain a clear structure and provide **detailed documentation** on how your work builds on the original repo.
- Consider creating **template repositories** for common setups like TensorFlow, PyTorch, or mobile development projects, which you can use as a starting point for similar projects.

4. POCs and Experimental Projects

- Since some POCs are industry-agnostic or tech-agnostic, you could maintain a **separate repository for POCs**, categorizing them by type or technology inside the repo using subfolders (e.g., `poc/ml`, `poc/web`, etc.). Provide detailed README files to explain each POC's objective.

5. Version Control and Branching Strategy

- Implement a strong **branching strategy** for each repository:
  - **Main** for stable, demo-ready code.
  - **Development** for ongoing work.
  - Feature-specific branches for major changes or technology integrations (e.g., `feature-pytorch-2.0`).
- Use **tags and releases** to mark stable versions of projects or POCs, especially when they are ready for demonstrations.

6. Automated CI/CD and Demos

- For demonstration purposes, configure **CI/CD pipelines** using GitHub Actions to automatically build, test, and deploy demos of your projects.
- Use GitHub Pages or other hosting services (like Netlify for web projects) to make POC and demo deployments easy for sharing.

7. Documentation and Configuration Management

- Create well-documented `README.md` files for each repository, describing:
  - The project's purpose.
  - Technologies used.
  - Installation and setup instructions.
  - Demos or how to replicate the results.
- Use `.env` files for configuration management of environment variables and secure secrets like API keys (as you're already using .env for API keys).
- For large or more complex projects, include **project-specific wiki** or **GitHub Projects** to manage tasks and progress.

8. GitHub Stars and Contributions

- Engage with the GitHub community by **starring relevant repositories** that align with your work and contributing to them. This will also drive traffic to your own projects.

This way, you'll have a well-organized GitHub portfolio that others can benefit from, and you can easily track and maintain various projects.

Question: What to do when I have made changes in code on the local and want to pull remote repo code

---

You have two options: either **commit** the local changes or **stash** them for later retrieval. Here's how you can proceed:

Option 1: Commit Your Changes

If you want to keep your local changes, commit them before running `git pull`:

1. **Add your changes**:

   ```
   git add .
   ```

2. **Commit your changes**:

   ```
   git commit -m "Describe the changes you made"
   ```

3. **Pull the latest changes**:

   ```
   git pull origin master
   ```

This will merge the latest changes with your local changes.

Option 2: Stash Your Changes

If you're not ready to commit your changes but want to pull the latest changes, you can stash your changes and apply them later.

1. **Stash your changes**:

```
git stash
```

2. **Pull the latest changes**:

```
git pull origin master
```

3. **Apply your stashed changes** (if you want them back):

```
git stash apply
```

This will temporarily hide your local changes, allow the pull to complete, and then you can reapply your changes afterward.

Option 3: Discard Local Changes (If You Don't Need Them)

If you don't need the local changes and want to discard them, you can reset your local files:

1. **Discard local changes**:

```
git reset --hard
```

2. **Pull the latest changes**:

```
git pull origin master
```

This will remove any local changes and sync your branch with the latest from `master`.

Question: Is it possible to clone only a particular folder from github?

Git by default doesn't support cloning a single folder from a repository. However, there are several workarounds you can use to achieve this goal:

1. Using Git Sparse Checkout (Partial Clone)

With Git's sparse checkout feature, you can clone a repository but only check out specific directories or files. I have tried this on my machine, but unfortunately i couldn't succeed on wsl on windows 11 with mounted drive. But as gone through blogs and github documentation it is mentioned it can be achieved.

Steps:

1. **Clone the Repository Without Checking Out Files:** Run this command to initialize the repository without actually checking out files:

   ```
   git clone --no-checkout https://github.com/username/repository.git
   cd repository
   ```

2. **Enable Sparse Checkout:** In the repository, enable sparse checkout:

   ```
   git sparse-checkout init --cone
   ```

3. **Specify the Folder You Want to Clone:** Add the folder path that you want to clone:

   ```
   git sparse-checkout set path/to/folder
   ```

4. **Pull the Specific Folder:** Now, pull only the specified folder:

   ```
   git pull origin main
   ```

   Replace `main` with the branch you want to pull from (e.g., `master` or another branch).

This way, you'll only download the files from the specific folder.

2. Using GitHub Zip Download (Browser Method)

If you don't need to use Git for the folder and just want the contents, you can download it directly via GitHub's web interface.

Steps:

1. Go to the repository on GitHub.
2. Navigate to the folder you want to download.
3. Click on the folder, and on the top right, you'll see a **"Download ZIP"** option for the entire repository. Unfortunately, this downloads the whole repo, but...
4. Alternatively, use a service like Download Directory (https://download-directory.github.io/), which allows you to download only a particular folder by pasting the URL of the folder you want. This method is the most simple and works smoothly.

3. Using SVN (Subversion) to Clone a Folder

GitHub also supports Subversion, and with it, you can check out a specific folder.

Steps:

1. Install `svn` (Subversion) if you don't have it:
    - On Ubuntu:

      ```
      sudo apt install subversion
      ```

    - On macOS (with Homebrew):

      ```
      brew install svn
      ```

2. Use this command to clone just the specific folder:

   ```
   svn checkout https://github.com/username/repository/trunk/path/to/folder
   ```

   Replace `username`, `repository`, and `path/to/folder` with the actual values. Also note github's "tree/main" (main is the branch name here) should be replaced with "trunk". Keep in mind some github path may not be accessible via svn.

This method checks out only the desired folder, but it won't provide the full Git history.

Updated: August 27, 2024

Categories:

- dsblog

Tags:

- Package Management
- Software Installation
- Linux Software Installation
- Github