Chapter 3: Open Source Development in Practice

3. 1. Introduction

Understanding the philosophy and tools of open source (Chapter 2) is the first step; applying them is the next. This chapter transitions from theory to practice, guiding you through the end-to-end process of modern open-source development. You will learn how to structure and manage a project for collaboration, ensure its quality through automated testing and integration, package it for deployment across diverse environments, and navigate the broader ecosystem that sustains it. The practices covered here—DevOps, Continuous Integration/Continuous Deployment (CI/CD), and containerization—are not just open-source standards but are fundamental industry skills. This chapter will provide the practical knowledge to not only contribute to projects but to also confidently initiate and maintain your own, transforming you from a passive user into an active, effective open-source practitioner. Besides, this chapter includes, whenever applicable, examples applied to Git and Github platforms.

3. 2. Project Setup and Sustainable Management

Launching an open-source project requires more than just code; it requires creating a welcoming and well-structured environment for potential collaborators. Sustainable management is key to a project's longevity.

3. ***2. 1. Initializing a Project:*** The journey begins with `git init`. A logical, conventional project structure is crucial for clarity. A typical structure includes directories for source code (`src/`), documentation (`docs/`), tests (`tests/`), and configuration files. A `.gitignore` file must be created immediately to exclude temporary files, local configuration, and secrets from version control, preventing accidental exposure.

- The *src* directory contains different source files of the project, including sub-directories. The nature of the file depends on the programming and development languages and environments. Examples of files are python files (extension .py), images, datasets, etc.

- The *docs* directory contains the documentation for the project. It may include user guides, API references, installation steps, contribution instructions, and other resources that assist users and developers in understanding and working with the project. On GitHub, a docs folder is often used as the publishing source for project websites via GitHub Pages.

- The *tests* directory contains the project's automated tests, such as unit tests, integration tests, and other scripts used to validate the functionality and reliability of the source code. These tests are essential for maintaining code quality and avoiding regressions. In

some projects, this folder may be called spec or tests.

Application to GitHub: In GitHub, it is possible to create a new repository that will contain the project files. By default, the repository is empty, and the folders src, docs and tests need to be created. An example of a repository containing these directories can be found following this link: https://github.com/psf/requests

3. ***2. 2. Essential Documentation:*** Documentation is the interface to your community. Four files are non-negotiable for a healthy project:

- README.md: The project's homepage. It must offer a clear description, installation instructions, a quickstart guide, and links to further resources. A bad README is a major barrier to adoption.

- CONTRIBUTING.md: This file outlines how to contribute. It should specify the process for submitting pull requests, the coding standards to follow, and how to report bugs effectively. It lowers the barrier for new contributors.

- CODE_OF_CONDUCT.md: This document sets the ground rules for respectful and inclusive participation. Adopting a standard like the Contributor Covenant is a best practice to foster a healthy community and protect maintainers from toxic behavior.

- LICENSE: Without an explicit open-source license, the project is de facto proprietary. The license text must be included in a file simply named `LICENSE` in the project's root directory.

Application to GitHub: in the repository, the above files can be added manually using the Add File option, and the user can add free content to the file. Particularly, the readme.md file can be added using the Add Readme option within the repository.

Example:

- The content of a readme file can be found following this link: https://github.com/psf/requests/blob/main/README.md
- A template of a contributing file and code of conduct can be found following this link: https://gist.github.com/PurpleBooth/b24679402957c63ec426#file-good-contributing-md-template-md

3. ***2. 3. Versioning and Release Management:*** As a project evolves, a clear versioning scheme is essential for users and developers. Semantic Versioning (SemVer) is the industry standard. A version number is formatted as `MAJOR.MINOR.PATCH`:

- MAJOR version increment indicates incompatible API changes.
- MINOR version increment indicates backwards-compatible functionality additions.
- PATCH version increment indicates backwards-compatible bug fixes.

Application to GitHub: Releases are formally marked in Git using tags (`git tag v1.0.0`) and can be packaged with release notes on platforms like GitHub, providing users with stable points of reference.

3. ***2. 4. Collaboration Management:*** Platforms like GitHub and GitLab offer integrated tools to manage work. Issues are used for bug reports, feature requests, and task tracking. Labels (e.g., `bug`, `enhancement`, `good first issue`) and Milestones help categorize and schedule work. Project boards provide a Kanban-style view (e.g., "To Do," "In Progress," "Done") of the project's workflow, offering transparency into the development process.

Application to GitHub: You can create a project directly from the main menu. Projects can be built using predefined templates or started from scratch. They can be viewed in different formats, such as a table, a board, or a roadmap, and can also be linked to an existing repository. Items can be added to a project as issues or selected from the repository's list of pull requests and issues. Collaborators can then update the project by modifying individual items like issues and pull requests.

3.3. Ensuring Quality: Testing and Continuous Integration

Quality assurance in open source is largely automated, a necessity for projects with distributed, asynchronous contributors.

3. 3. 1. The Imperative of Automated Testing: Manual testing does not scale. Automated tests provide a safety net that allows developers to make changes with confidence.

There are two primary levels of testing:

- Unit Tests: Verify the correctness of individual functions or modules in isolation. They are fast, numerous, and form the foundation of a test suite.

- Integration Tests: Verify that different modules or services work together as expected. They are broader and more complex than unit tests.

3. 3. 2. Continuous Integration (CI): CI is the practice of automatically building and testing every change pushed to a shared repository. Its benefits are profound:

- Early Bug Detection: Bugs are identified within minutes of a change being proposed, making them far easier and cheaper to fix.
- Enforced Consistency: The CI system ensures the codebase always compiles and passes all tests, preventing "it works on my machine" problems.
- Automated Quality Gates: A Pull Request cannot be merged until the CI system's checks pass, enforcing a baseline of quality.

3. 3. 3. Implementing CI with GitHub Actions: GitHub Actions is a powerful CI/CD (continuous integration/continuous delivery) platform integrated directly into GitHub. Workflows are defined in a YAML file (`.github/workflows/main.yml`) within the repository. A typical CI workflow includes steps to:

1. Checkout the code.
2. Set up the programming language environment (e.g., Python, Node.js).
3. Install project dependencies.
4. Run linters and formatters to enforce code style.
5. Execute the test suite.

The results are displayed directly on the commit and Pull Request, providing immediate feedback to the contributor.

Example: the following code implements building and testing when changes are pushed from a local repository to a shared one:

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repo
        uses: actions/checkout@v4
      - name: Set up Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '18.x'
      - run: npm install
      - run: npm run build

  test:
    needs: build # This job depends on the successful completion of the 'build' job
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repo
        uses: actions/checkout@v4
      - name: Set up Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '18.x'
      - run: npm install
      - run: npm test
```

To be continued

References & Further Reading

1. Books
   * Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The Science of Lean Software and DevOps*. IT Revolution Press.
   * Freeman, S., & Pryce, N. (2009). *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley.

2. Websites
   * Docker Documentation: https://docs.docker.com/
   * GitHub Actions Documentation: https://docs.github.com/en/actions
   * Semantic Versioning (SemVer): https://semver.org/
   * Contributor Covenant (Code of Conduct): https://www.contributor-covenant.org/

3. Articles
   * The DevOps Roadmap: https://roadmap.sh/devops
   * Nadareishvili, I. (2016). *How to Write a Good README*.