Software Development Guidelines

Virtual Environment

A **virtual environment** is an isolated space to manage dependencies and configurations independently for a project, preventing conflicts with system-wide settings.

Why Use a Virtual Environment?

- **Avoid conflicts** – Different projects can use different versions of dependencies.
- **Reproducibility** – Ensures consistent setup across development, testing, and deployment.
- **Better collaboration** – Team members get the same environment setup.
- **Cleaner system** – Prevents cluttering global installations.

Popular Virtual Environments by Technology

- **Python** → venv, conda, pyenv
- **Node.js** → nvm (Node Version Manager)
- **Java** → sdkman, jenv
- **Ruby** → rbenv, rvm
- **Docker** → Containerized environments for complete isolation

```
● (base) harihararajjayabalan@Harihararajs-MacBook-Pro CI-CD-pipeline-guidelines % python -m venv provide_environment_name
● (base) harihararajjayabalan@Harihararajs-MacBook-Pro CI-CD-pipeline-guidelines % source provide_environment_name/bin/activate
● (provide_environment_name) (base) harihararajjayabalan@Harihararajs-MacBook-Pro CI-CD-pipeline-guidelines % pip install -r requirements.txt
  Collecting fastapi
    Downloading fastapi-0.115.12-py3-none-any.whl (95 kB)
       ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 95.2/95.2 KB 2.7 MB/s eta 0:00:00
  Collecting uvicorn
    Downloading uvicorn-0.33.0-py3-none-any.whl (62 kB)
       ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.3/62.3 KB 7.7 MB/s eta 0:00:00
  Collecting httpx
    Using cached httpx-0.28.1-py3-none-any.whl (73 kB)
  Collecting flake8
```

Dependency Management

Dependency management is the process of handling external libraries or packages that a project relies on to function properly. These dependencies can range from utility libraries (e.g., logging, HTTP requests) to full-fledged frameworks (e.g., web frameworks, data analysis libraries).

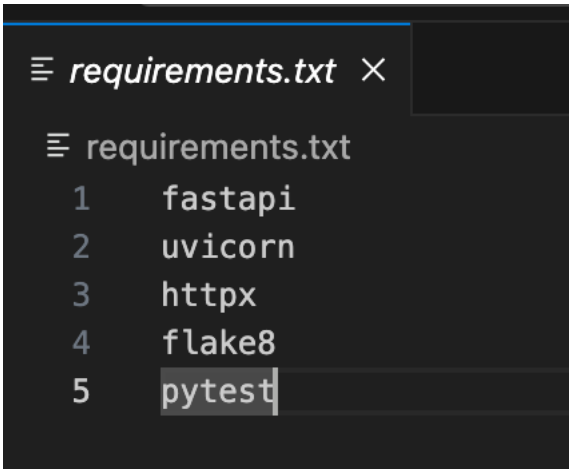
Why is Dependency Management Important?

- **Reproducibility** – Ensures that your project behaves the same way across different environments by locking dependencies to specific versions.
- **Compatibility** – Helps avoid conflicts between different versions of dependencies.
- **Collaboration** – Makes it easier for teams to install and work with the same set of dependencies.

- **Security** – Helps track and manage the security vulnerabilities of dependencies by keeping them updated.
- **Efficiency** – Reduces errors and bugs caused by missing or incompatible dependencies, leading to faster development cycles.

Types of Dependency Management Tools

- **Package Managers** – Tools that automate the process of installing, updating, and removing dependencies (e.g., npm for Node.js, pip for Python, gem for Ruby).
- **Dependency Definition Files** – Files where the list of dependencies and their versions are specified (e.g., package.json in Node.js, requirements.txt in Python, Gemfile in Ruby).
- **Lock Files** – Files that freeze the specific versions of dependencies to avoid discrepancies across different environments (e.g., package-lock.json, Pipfile.lock, Gemfile.lock).



Code Quality

Code quality refers to how well-written, maintainable, and efficient a codebase is. High-quality code is **readable, consistent, efficient, secure, and well-documented**.

Why is Code Quality Important?

- **Maintainability** – Easier to modify and extend.
- **Readability** – Improves collaboration and understanding.
- **Performance** – Ensures efficient execution.
- **Security** – Reduces vulnerabilities.
- **Scalability** – Supports future growth.

Key Aspects of Code Quality

- **Readability** – Clear variable names and structure.
- **Consistency** – Follow a coding style guide.
- **Efficiency** – Optimize algorithms and performance.
- **Security** – Prevent vulnerabilities like SQL injection.

- **Testability** – Ensure functionality with unit tests.
- **Documentation** – Write meaningful comments.

Popular Code Quality Tools by Technology

- **Python** → flake8, black, pylint, pytest
- **javaScript** → ESLint, Prettier, Jest
- **Java** → Checkstyle, FindBugs, Junit
- **C++** → cpplint, clang-format, Google Test
- **Go** → golint, gofmt, Go test

Git

GitHub is a cloud-based platform for version control using **Git**. It allows developers to store code, track changes, collaborate on projects, and manage repositories.

Why Use Git?

- **Version Control** – Track changes in your code and revert to previous versions.
- **Collaboration** – Work with others through features like pull requests and code reviews.
- **Backup** – Store code in the cloud for easy access and disaster recovery.
- **Integration** – Integrate with CI/CD tools and other services.

Key GitHub Concepts

- **Repositories** – A place to store your project files and history.
- **Branches** – Isolate new features or fixes from the main codebase.
- **Commits** – Record changes to files, linked to specific versions.
- **Pull Requests** – Propose changes and initiate code reviews before merging.

Sample commands:

git init – Initialize git.

git status – Status of the git.

git add . – adds all the uncommitted changes files to the commit.

git commit -m ‘Comment Message’ – commits the changes to the repository.

git push – pushes local server changes to the remote server.

Docker

Docker is a platform for developing, shipping, and running applications in lightweight, portable containers. Containers encapsulate all dependencies, libraries, and configurations needed for an application to run, ensuring consistency across environments.

Why Use Docker?

- **Portability** – Run applications anywhere: from a developer's machine to production, without configuration issues.
- **Isolation** – Run multiple applications with different dependencies on the same system without conflict.
- **Consistency** – Eliminate the "it works on my machine" problem by standardizing the environment.
- **Scalability** – Easily scale applications by running multiple containers or orchestrating with tools like Kubernetes.

Key Docker Concepts

- **Containers** – Lightweight, isolated environments where applications run.
- **Images** – Read-only templates to create containers (e.g., python:3.9, nginx).
- **Dockerfile** – A script to build custom Docker images, defining the environment setup.
- **Volumes** – Persistent storage used by containers to retain data outside of the container's lifecycle.
- **Docker Compose** – A tool for defining and running multi-container applications with a single configuration file.

Unit Testing

Unit testing is the practice of testing individual components (or units) of code, such as functions or methods, in isolation from the rest of the application. It ensures that each unit of code performs as expected.

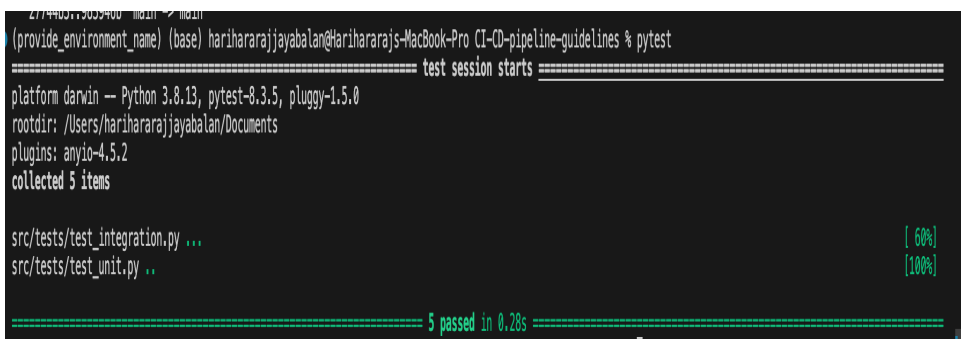
Integration Testing

Integration testing verifies that different components or systems of an application work together as expected. It tests the interaction between modules, services, or systems to ensure they function correctly when integrated.

Why Use Unit and Integration Testing?

- **Improved Code Quality** – Catch issues early and ensure code behaves as expected.
- **Reduced Bugs** – Prevent bugs from reaching production by testing individual units and integrations.
- **Faster Development** – With automated tests, developers can quickly identify problems and fix them.
- **Documentation** – Tests serve as documentation for how components should behave.

Aspect	Unit Testing	Integration Testing
Scope	Individual functions, methods, or classes.	Interaction between components or systems.
Goal	Verify the correctness of isolated units.	Ensure components work together as expected.
Complexity	Simpler, involves fewer dependencies.	More complex, involves multiple modules.
Execution Speed	Fast, isolated tests.	Slower, tests multiple components together.



GitHub Actions

GitHub Actions is a CI/CD tool built into GitHub that automates workflows like building, testing, and deploying applications. It lets you define actions and workflows in YAML files to automate repetitive tasks whenever events like push, pull_request, or release occur.

Why Use GitHub Actions?

- **Integrated with GitHub** – No external tools needed, everything happens within GitHub.
- **Customizable** – Create workflows for any process like testing, deployment, or code formatting.
- **Scalable** – Easily scale for any project size, from small scripts to large enterprise applications.
- **Free Tier** – GitHub offers free usage for public repositories and limited free minutes for private repositories.

Key Concepts in GitHub Actions

- **Workflows** – A series of jobs that run based on specific events (e.g., a push to a branch).
- **Jobs** – A set of steps executed in a specific runner (e.g., Linux, Windows).
- **Steps** – Individual tasks that are run within a job (e.g., checking out code, installing dependencies).
- **Actions** – Reusable units of code within steps that handle tasks like checking out code or setting up languages.
- **Runners** – Virtual environments where your jobs are executed, available for Linux, Windows, and macOS.

Kubernetes

Kubernetes (K8s) is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It helps manage clusters of containers, providing tools for scaling, networking, and load balancing.

Why Use Kubernetes?

- **Scalability** – Automatically scale applications up or down based on demand.
- **High Availability** – Ensures application uptime by redistributing resources during failures.
- **Efficient Resource Management** – Efficiently allocate compute resources to containers.
- **Automated Deployment** – Simplify rollouts, rollbacks, and upgrades with minimal downtime.
- **Platform Agnostic** – Works across public, private, and hybrid cloud environments.

Key Kubernetes Concepts

- **Pod** – The smallest deployable unit in Kubernetes, typically running a single container or multiple containers in a shared environment.
- **Node** – A physical or virtual machine in the cluster running one or more Pods.
- **Cluster** – A set of Nodes managed by Kubernetes that run containerized applications.
- **Deployment** – Manages a set of identical Pods, ensuring they are always running the desired number of replicas.
- **Service** – Exposes Pods to other services or external traffic, providing load balancing.
- **ReplicaSet** – Ensures a specified number of identical Pods are running.
- **Namespace** – A way to divide resources within a cluster into virtual clusters for better organization.

CI/CD

CI (Continuous Integration) and **CD (Continuous Delivery/Deployment)** are practices for automating the process of integrating code changes and deploying them to production.

- **CI** automates testing and integrating code.
- **CD** automates deployment to production or staging.

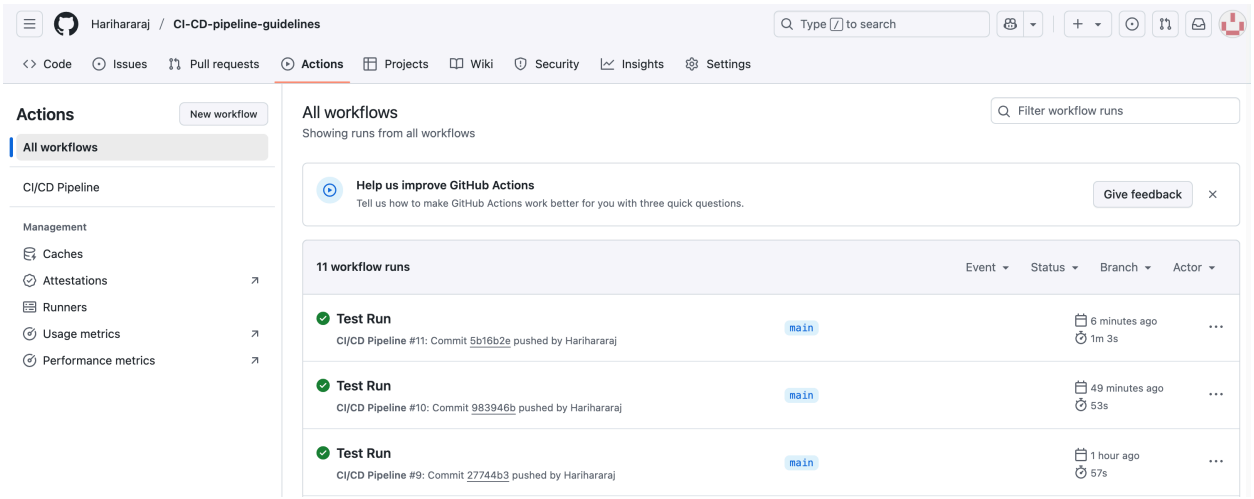
Why Use CI/CD?

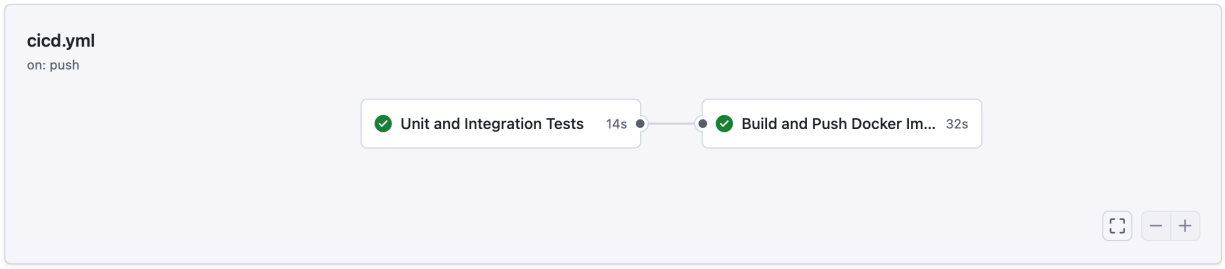
- **Automation** – Automates build, test, and deployment processes to reduce manual intervention.
- **Faster Development** – Frequent integrations and deployments ensure quicker delivery of features and bug fixes.
- **Consistency** – Ensures code is always in a deployable state, reducing errors and inconsistencies.
- **Collaboration** – Team members can collaborate more efficiently with automated workflows.

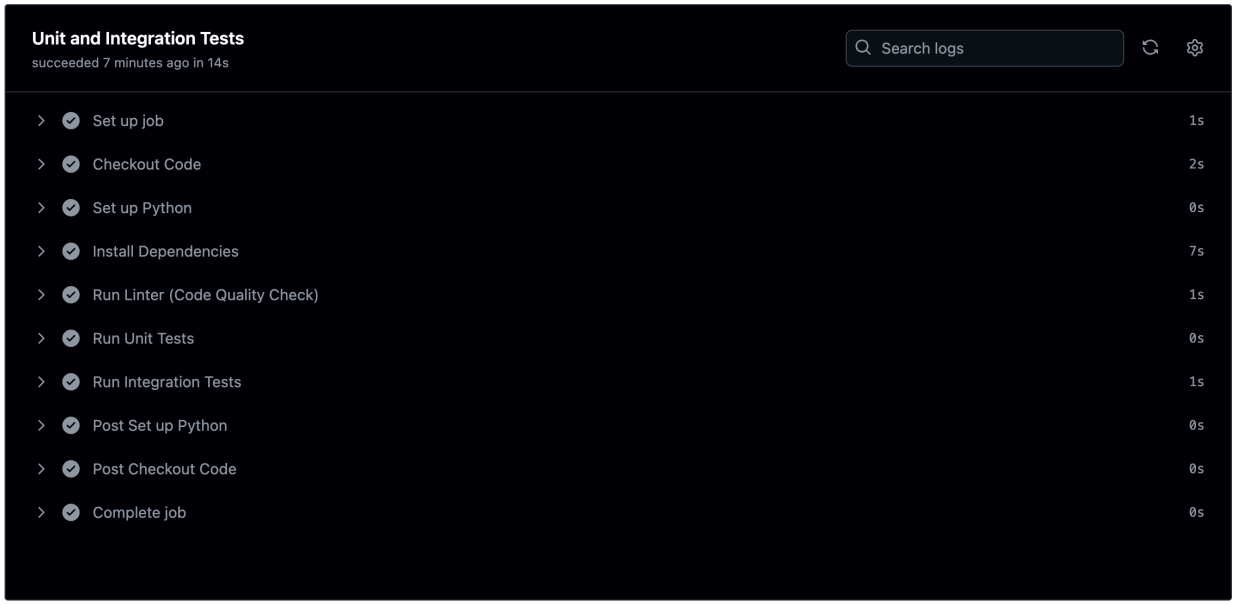
Key CI/CD Concepts

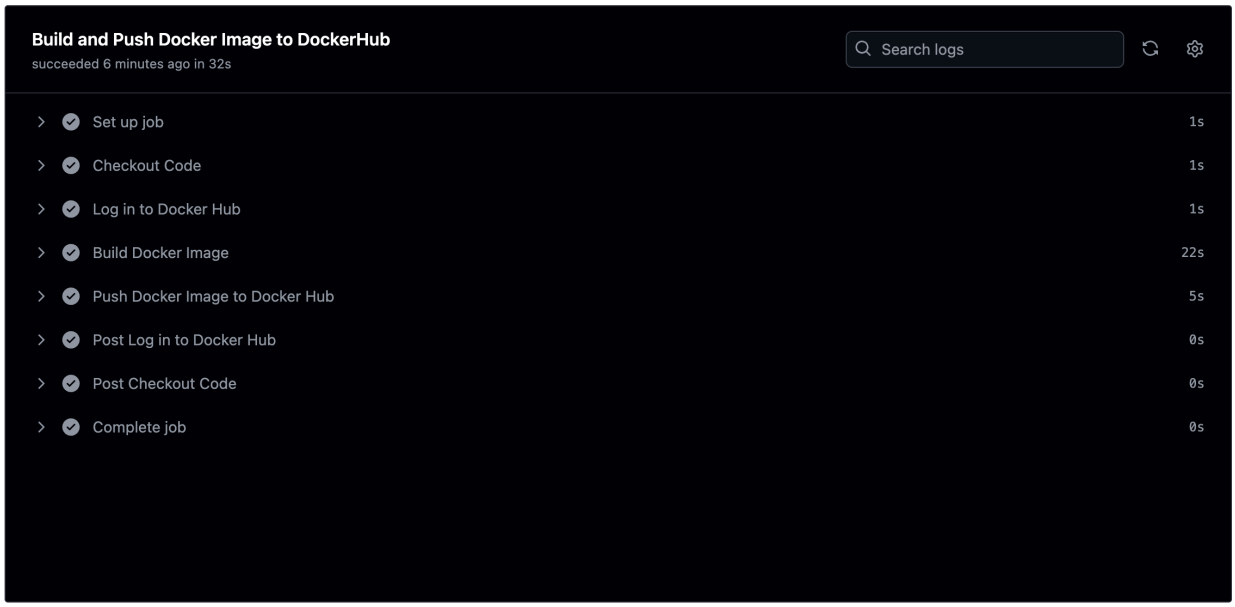
- **Pipelines** – A series of automated steps (build, test, deploy).
- **Builds** – Automate compiling, packaging, or preparing code.
- **Tests** – Automated unit, integration, or acceptance tests to validate code.
- **Deployments** – Automate moving code to production, staging, or test environments.

Pipeline Demo









Code: https://github.com/Harihararaj/CI-CD-pipeline-guidelines

Clone the repo and play around with the code.

Happy Learning

😎