Git, GitHub, and CI

A Tutorial for Collaborative Programming

Matteo Dalle Vedove

November 20, 2024

Overview

Overview

In these lectures:

- initialise and populate a (local) git repository;
- connect to a remote GitHub repository;
- collaborate on the repository with other collegues;
- conflict resolution;
- create a simple CI pipeline for our project.

Git

Git

A brief review.

History

Git was created by Linus Torvalds in 2005 to manage the Linux kernel.

It has been designed to be fast and efficient, and serve as a **distributed version control system** that enables **multiple developers** to **work** on the **same project**.

The working principle

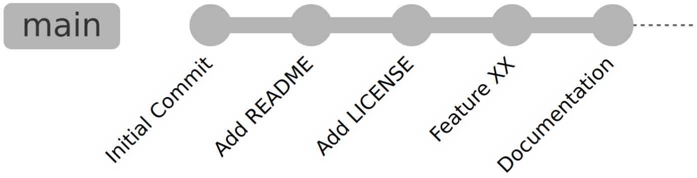
At the core of git for code versioning, there are **commits**.

A commit is a **snapshot** of the project at a certain point in time.

The snapshot includes the **changes** made to the files since the last commit. It means that git saves the **differences** between the files, not the files themselves.

We can regard a commit as a **node** in a **tree graph** where the **edges** describes the relationship between the commits.

An example


main
Initial Commit
Add README
Add LICENSE
Feature XX
Documentation

Key concepts

- **Repository**: a *git* repository is any directory that contains a *.git* directory. It manages the project's files and history.
- **Commit**: as said, a snapshot of the project at a certain point in time. We might refer to it as a *node* in the *tree graph*.
- **Branch**: a branch is a pointer to a commit. It allows us to work on different features or versions of the project.

Creating a repository

If we want to initialise locally a git repository, use the `git init` command.

If we have already the URL of another repository, we can clone it with the `git clone <url>` command.

The `add` and `status` commands

To put a file into the ***staging area***, we use the `git add` command:

```
$ git add <file1> <file2> ...
```

Calling `git add .` will stage all the files in the directory.

To check the status of the staging area, we can use `git status`.

The `commit` command

To add changes in the staging area to the repository, we use the `git commit` command.

With the `-m` flag, we can provide a message for the commit:

```
$ git commit -m "A meaningful message"
```

The `branch` and `checkout` commands

To create a new branch, we use the `git branch` command:

```
$ git branch <branch-name>
```

To see all available branches, we can use `git branch`.

To *switch* to a branch, we use the `git checkout` command:

```
$ git checkout <other-branch-name>
```

Adding a remote repository

To add a remote repository, we use the `git remote add` command:

```
$ git remote add <remote-name> <remote-url>
```

A common (and default) remote name is *origin*.

Synchronising with a remote

To synchronise the local repository with the remote, we use the `git fetch` command:

```
$ git fetch <remote-name>
```

We can also use `git pull` tu download the changes from the remote repository and merge them into the local branch.

Or use `git push` to upload the changes from the local repository to the remote.

GUIs for git

If you need a simple tool to see the changes in the repository, you can use `git gui`.

To have a clearer vision of the branches and the history, you can use `gitg`.

Note: other alternatives exists.

Git playground

If you want a no-code online environment to test git command and see the results visually, you can use Learn Git Branching.

GitHub

GitHub

What is GitHub?

GitHub

GitHub is a **web-based platform** that provides **hosting** for *git* repositories.

Note

GitHub is not the only solution. Other similar exists, like GitLab or Bitbucket.

These services not only provide a place to store the repositories (i.e., a *.git* folder), but also offer a set of **tools** to **collaborate** on the projects.

Disclaimer

From now on, we will mainly talk about **collaborative features** of GitHub, and how to use git at its best in a team.

However, the same concepts are **extremely useful** even if you **work alone** on a project.

How to collaborate?

Here we’ll show a simple and common, yet effective, workflow: **issue**, **branch**, and **pull request**.

- **Issue**: we write a description of the problem or the feature we want to implement;
- **Branch**: we create a new branch to work on the issue, i.e., to implement the feature or fix the problem;
- **Pull request**: to merge our code changes back into the main branch.

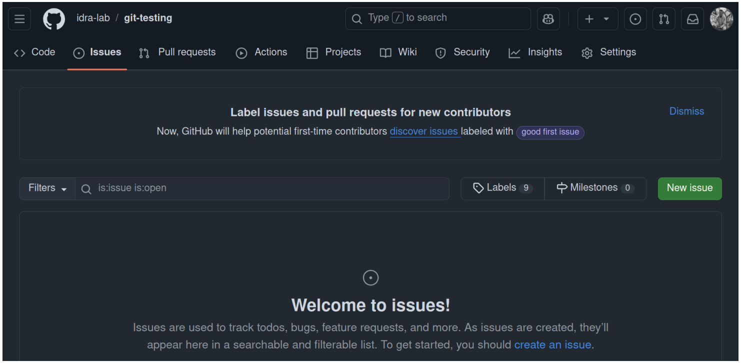
Issues

Issues

A practical guide in GitHub.

Creating an issue

To create an issue on a repository, first navigate to the repository's main page.



Label issues and pull requests for new contributors

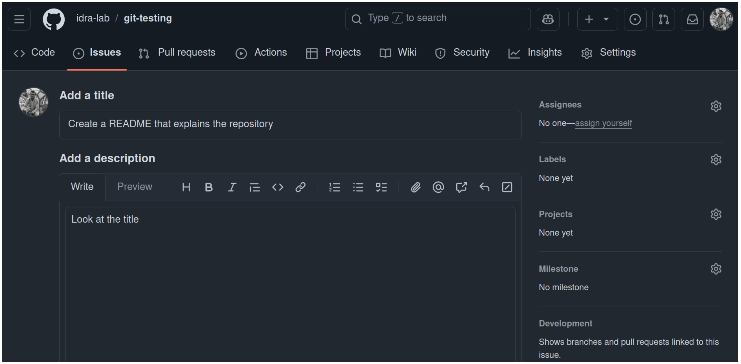
Now, GitHub will help potential first-time contributors discover issues labeled with good first issue

Welcome to issues!

Issues are used to track todos, bugs, feature requests, and more. As issues are created, they'll appear here in a searchable and filterable list. To get started, you should create an issue.

Creating an issue

When creating an issue, provide a **title** and a **description**. You can also provide **assignees**, **labels**, and **milestones**. You can find this issue here.


idra-lab / git-testing

Add a title

Create a README that explains the repository

Add a description

Look at the title

Assignees: No one—assign yourself

Labels: None yet

Projects: None yet

Milestone: No milestone

Development: Shows branches and pull requests linked to this issue.

Tips for issues

If you are working on a specific issue, and needs to share some code you are developing in the issue, you can refer the **issue number** in the **commit message**.

There are also special **keywords** that can trigger special actions, like ***Fixes***, ***Closes***, or ***Resolves***, that automatically mark the issue from open to close.

Why issues?

Issues are a valuable tool to **organize** the work on the project.

They are useful not only to report bugs or problem, but also to **track** the **progress** of the project, and to **discuss** with the other members new features or improvements.

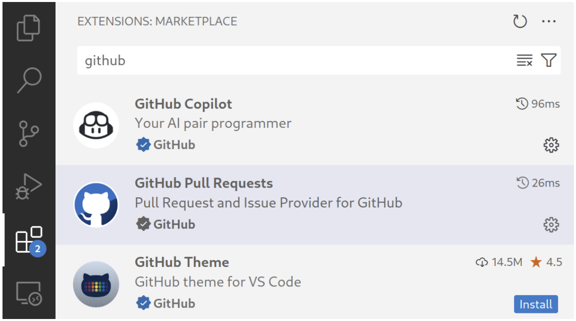
Branches

Branches

We have an issue, now what?

Preliminaries

Even though it is not necessary, you can install the GitHub Pull Request extension for VSCode.


EXTENSIONS: MARKETPLACE

github

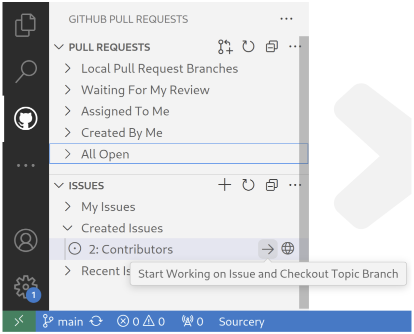
GitHub Copilot
Your AI pair programmer
GitHub
96ms

GitHub Pull Requests
Pull Request and Issue Provider for GitHub
GitHub
26ms

GitHub Theme
GitHub theme for VS Code
GitHub
14.5M ★ 4.5

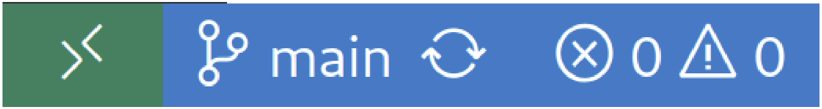
Creating a branch

From the extension, you can see active issues. From here, we create a new branch to work on the issue.

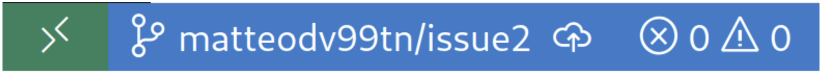


Working on a branch

We start from the *main* branch:



and create a new branch, *username/issue2*, to work on the issue:



Here we work on the feature, or we fix the bug. Doing so, we create 1 or more commits.

Pushing the branch

Once we are done, we push the branch to the repository.

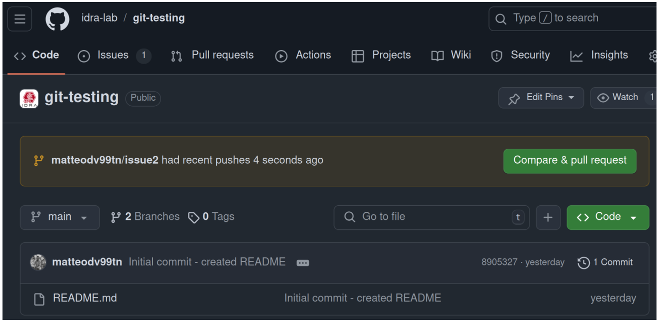
Pull Requests

Pull Requests

Having your contribution merged.

Creating a pull request

After the push, by navigating to the repository's main page, we can see the new branch, as well as a banner that asks wether to create a pull-request.



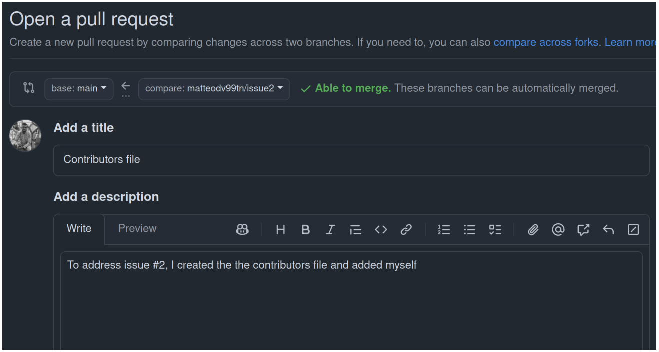
matteodv99tn/issue2 had recent pushes 4 seconds ago

matteodv99tn Initial commit - created README 8905327 · yesterday 1 Commit

README.md Initial commit - created README yesterday

Creating a pull request

When creating a pull request, we must provide a **title** and, possibly, a **description**.



Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks. Learn more

base: main ← compare: matteodv99tn/issue2 ✓ Able to merge. These branches can be automatically merged.

Add a title

Contributors file

Add a description

To address issue #2, I created the the contributors file and added myself

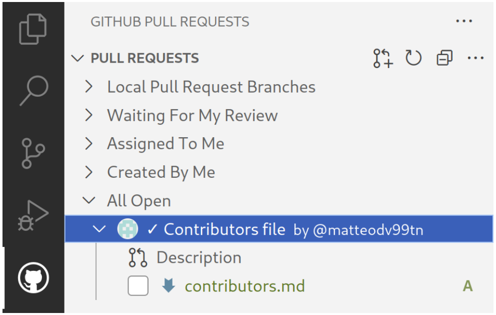
Managing a pull request

During (or after) the creation of a pull request, as per issues, we can set **reviewers**, **assignees**, **labels**, and **milestones**.

Within the pull request page, it is possible to **comment** the code, **review** or **suggest** changes, and finally **approve** the pull request to get it merged on the target branch.

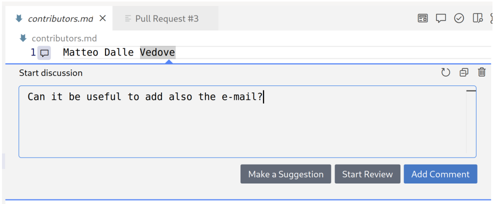
Reviewing a pull request

Using the VSCode extension, it is possible to review from the editor the changes made in the pull request.



Reviewing a pull request: comments

A powerful feature to use in pull requests are **comments**, to discuss the code changes with the other members.

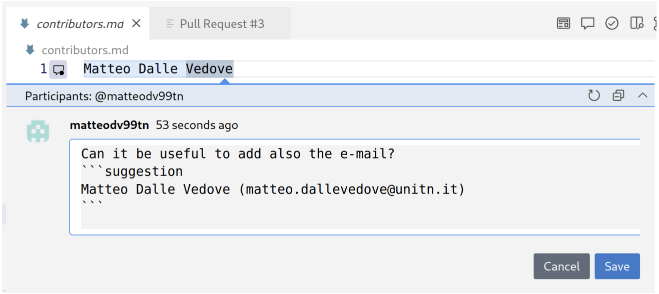


```
1 Matteo Dalle Vedove
```

Can it be useful to add also the e-mail?

Reviewing a pull request: suggestions

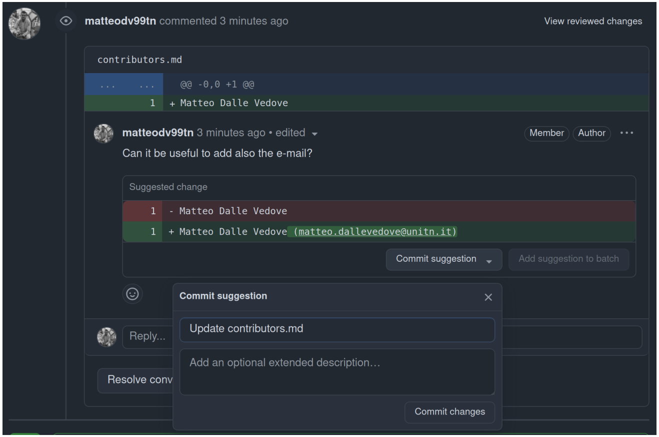
Along with the comments, it is possible provide code **suggestions** that might solve the problem, while not actually committing to the branch.



```
Matteo Dalle Vedove
```

~~~
Can it be useful to add also the e-mail?
```suggestion
Matteo Dalle Vedove (matteo.dallevedove@unitn.it)
```
~~~

Reviewing a pull request: accepting suggestions


matteodv99tn commented 3 minutes ago

View reviewed changes

contributors.md

```
... ... @@ -0,0 +1 @@
      1 + Matteo Dalle Vedove
```

matteodv99tn 3 minutes ago • edited

Member Author

Can it be useful to add also the e-mail?

Suggested change

```
1 - Matteo Dalle Vedove
1 + Matteo Dalle Vedove (matteo.dallevedove@unitn.it)
```

Pull request

The final goal of a pull request is to get the developed feature or the bug fix merged into the main branch.

Pull requests can also be closed without merging, maybe to indicate that the feature is not needed anymore, or that the code is not ready to be merged.

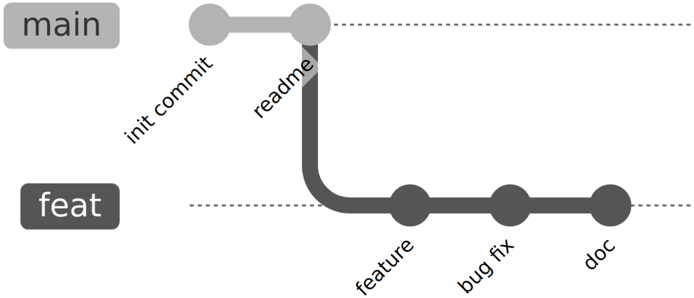
Conflicts

Conflicts

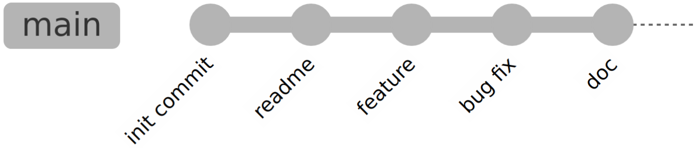
The big elephant in the room.

A one developer story

If you work alone it is easy to go from this



to



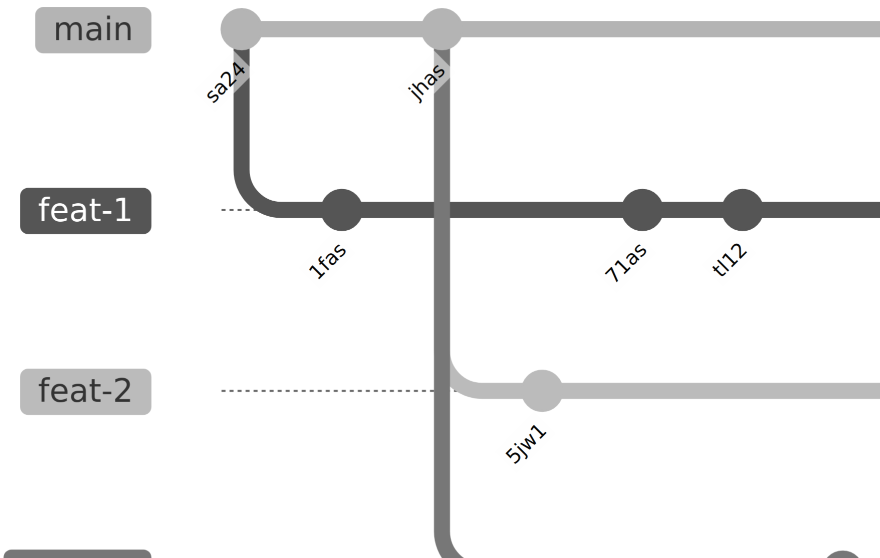
A linear history

In most single-developer projects, the commit history of a repository is **linear**.

This means that to merge the changes from a branch to another, git can simply **fast-forward** the commits, i.e., update the commit to which *main* is pointing to.

Working in a team

When working in a team, chances are that that the commit tree is something like this:



Working in a team

In this case, trying to merge *feat-1* into *main* is not trivial, since the commit history is not linear. It is likely that the same file has been modified from 2 branches, giving rise to **conflicts**.

These conflicts must be resolved manually, but luckily there are tools that can make this process easier, like the **merge tool** in VSCode. It allows to see the differences between the files, and to choose which version to keep.

Merging changes

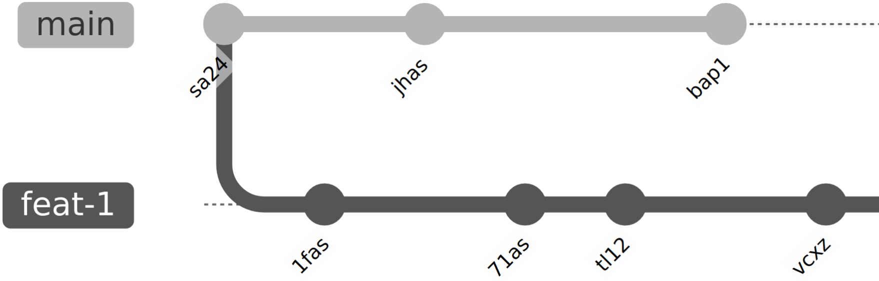
To merge changes from a branch to another there are 2 main strategies:

- **merge**: it creates a new commit that merges the changes from the source branch to the target branch;
- **rebase**: it moves the commits from the source branch to the target branch, creating a linear history.

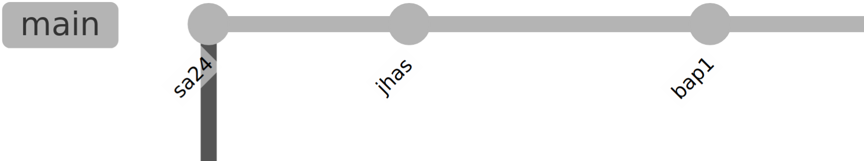
From the CLI, once you are on the target branch, we can call `git <merge/rebase> <source-branch>` command.

The merge strategy

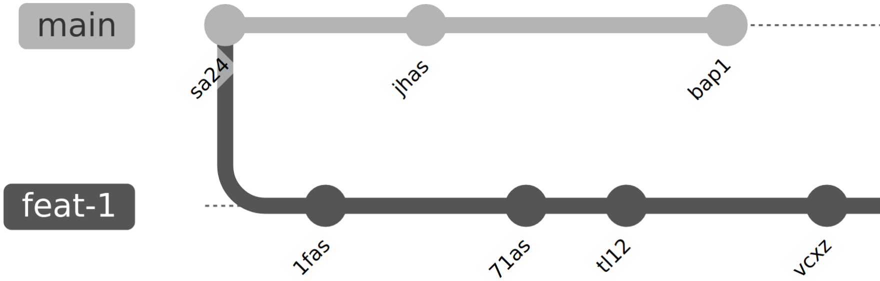
We start from this:



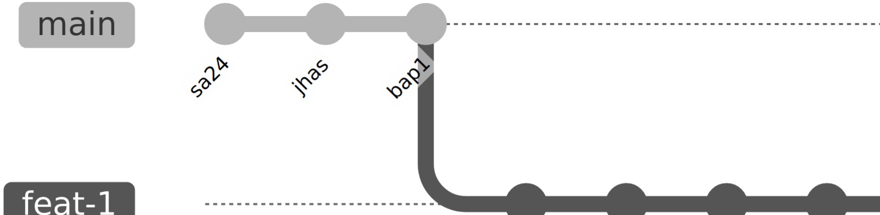
And we merge *feat-1* into *main*, creating a new commit that includes the changes from both branches:



The rebase strategy



When rebasing, all commits from *feat-1* are iteratively applied to the current *main* branch, creating a linear history:



The rebase strategy

Rebase creates a linear history, **but**:

- it rewrites the commit history, making it difficult to track the changes in time;
- each commit applied might have conflicts that must be resolved.

⚠ Warning

The rebase process actually changes the commit hash, so **don't rebase if you have already pushed the branch to the repository**. If you do so, other people working on the same branch will have a hard time, and a `--force-push` might be required...

Bonus: squash

One cool feature of git is the ability to **squash** commits, i.e., to merge multiple commits into a single one.

This is particularly useful when working on a complex feature, or when fixing bugs. As we develop, we create multiple small commits, that might be squashed into a single one before merging the branch into the main branch.

How to squash

- check the commit history: `git log (--oneline)`.
- perform an interactive rebase: `git rebase -i HEAD~<n>`, where `<n>` is the number of commits you want to squash.
- change the `pick` keyword to `squash` for the commits you want to squash. Make sure to keep the first commit as `pick`.
- save and close the editor;
- call `git rebase --continue` to finish the rebase, and update commit message at wil

Continuous Integration

Continuous Integration

Fixing the *it works on my machine* problem.

Continuous Integration

Continuous Integration (CI) is a practice that aims to improve the quality of the code by **automating** the **process** of building, testing, and deploying the code **whenever a change is made**.

Whenever code is pushed to a repository, we can run some scripts on a server that will validate the code.

GitHub Actions

For sake of simplicity, we will use **GitHub Actions** to create a simple CI pipeline.

Still other solutions exists, like Jenkins, Travis, CircleCI...

Creating a workflow

To associate a CI pipeline to a repository (the **workflows**), we must create a `.yml` file in the `.github/workflows` directory.

```
name: <workflow-name>

on:
   <triggers>

jobs:
   <job descriptions>
```

Workflow triggers

```yaml
on:
  push:
    branches: devel
  pull_request:
    branches: main
```

We can **trigger the workflow** on different events:

- when a commit is pushed on a specific branch;
- when a pull request is opened;
- at a specific time / with a given frequency;
- ...

Workflow jobs

In each workflow we can define one or more **jobs**.

Each job is independent from the others, and can run sequentially or in parallel (based on their dependencies).

```yaml
jobs:
  sphinx-build:
    runs-on: ubuntu-22.04
    permissions:
      contents: write
      pages: write
      id-token: write
    environment:
      name: github-pages
      url: ${{ steps.deployment.outputs.page_url }}
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
      - name: Installs
```

Workflow job configuration

For each job, we can specify

- the base image to use, with the `runs-on` key;
- some `environment` variables;
- the dependency from another job successfull execution, with the `needs` key;
- a sequence of `steps` to accomplish.

Workflow steps

Each step is a (list of) commands that are called in the job environment.

They can optionally have a `name`.

It is also possible to use off-the-shelp actions, i.e., pre-built scripts that can be used in the workflow to accomplish standard behaviours (e.g., setup a specific python version).

Testing locally

If you want to test locally the *correctness* of the workflow as you implement them, you can use Act.

It requires Docker to be installed on the host machine.

CI for ROS2

CI for ROS2

Building and testing ROS2 packages.

GitHub actions

Even though we could specify commands to install and configure ROS2, and build and test our packages, we can use already available GitHub actions.

Some are:

- ichiro-its/ros2-ws-action;
- ROS tooling;
- ROS-industrial;
- ...

Preliminaries

First, let's decide the events that will trigger the workflow:

```yaml
name: ros_ws_build

on:
  push:
    branches:
      - main
      - devel
  pull_request:
    branches:
      - devel
```

Jobs

We must pull the (updated) code, setup the ROS2 installation, and build our workspace.

```yaml
jobs:
  workspace-build:
    runs-on: ubuntu-22.04
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
        with:
          path: workspace
      - name: Setup workspace
        uses: ichiro-its/ros2-ws-action/setup@v1.0.0
        with:
          distro: humble
      - name: Build workspace
        uses: ichiro-its/ros2-ws-action/build@v1.0.0
```

Unit testing

ROS2 also provides a set of tools to test packages, and perform static analysis (linting).

Here we see how to setup unit tests for CPP packages using the *Google test* suite.

Unit test: source code

First let's prepare the source code file for the unit tests:

```cpp
#include <gtest/gtest.h>

TEST(cpp_publisher_subscriber, useless_assertion) {
    ASSERT_EQ(4, 2 + 2);
}

int main(int argc, char** argv) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Package setup

Update the `package.xml` to include the test dependency:

```xml
<test_depend>ament_cmake_gtest</test_depend>
```

Update the `CMakeLists.txt` to include the test:

```cmake
if(BUILD_TESTING)
    find_package(ament_cmake_gtest REQUIRED)
    ament_add_gtest(
        ${PROJECT_NAME}_test
        src/tests.cpp
    )
endif()
```

Running the tests

One we build the workspace (`colcon build`), we can run the tests with the `colcon test` command.

We can also automate this process in the CI pipeline:

```
jobs:
  workspace-build:
    runs-on: ubuntu-22.04
    steps:
      # ...
      - name: Test packageworkspace
        run: |
          source install/setup.bash
          colcon test --return-code-on-test-failure
```