Pip Technical Guidelines

R. Andres Castenada and Ronak Shah

2025-08-03

Table of contents

Preface **3**

1 Introduction **4**

2 Add code coverage badge to your GitHub Repository. **5**
2.1 Codecov.io 5
2.2 GitHub 6

3 Setting up Github Actions for Auto Deployment of Quarto book **10**
3.1 Introduction 10
3.2 Workflow 10
3.2.1 Triggering the Workflow 11
3.2.2 Naming the workflow 11
3.2.3 Defining the job 11
3.2.4 The Steps 12
3.3 Don't ignore the `.gitignore` file 13
3.4 Conclusion : 13

4 Summary **14**

References **15**

Preface

This is a Quarto book. In this book we intend to write technical information about PIP projects.

1 Introduction

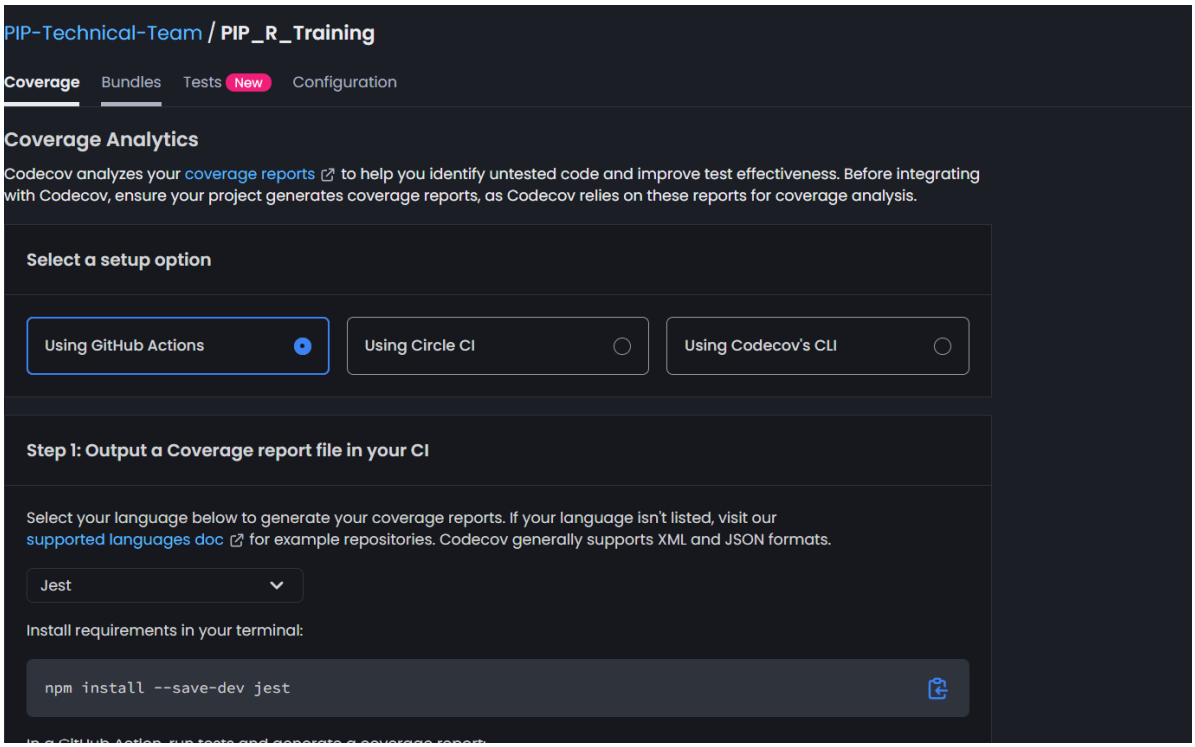
The purpose of this book is to gather all the technical knowledge specific to PIP in one place.

2 Add code coverage badge to your GitHub Repository.

In this article we will learn how to add code coverage badge to your GitHub repository.

2.1 Codecov.io

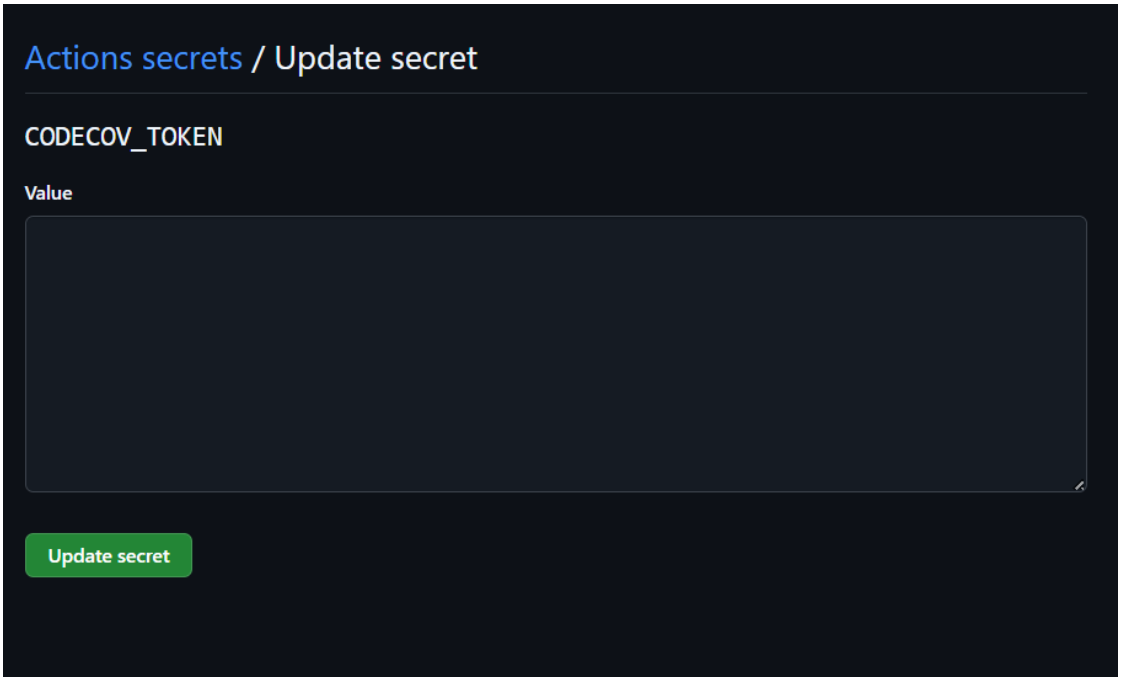
- Create an account at https://about.codecov.io/ , sign up with your GitHub account if you don't have an account already. **Codecov** is a popular tool for measuring and visualizing **code coverage** in software projects. It integrates with GitHub, GitLab, Bitbucket, and other CI/CD systems to provide insights into how much of your code is tested by your test suite.
- You can sync your private Github repositories on codecov platform to get started. If you want to add code coverage badge to a repository which is part of an organization (like PIP-Technical-Team, GPID-WB etc) then you need to be an admin of that organization. Admin role is needed because to sync the communication between Codecov.io with GitHub we need to generate a token which can only be done by admins.
- Once your repo is synced with codecov and you can see it there click on Configure to start the process. As an example it should give you this screen



- If you scroll below it will ask you to generate a repository secret, click on that to get a unique token for your repository and copy it.
- You can ignore rest of the steps mentioned on that page since those are very generic language agnostic steps and since we want to setup this for R packages, we have a better option which I will share below.

2.2 GitHub

- Now, moving to GitHub go to your repository. Click on Settings -> Secrets and Variables -> Actions -> Repository Secrets add the new token with name `CODECOV_TOKEN` and copy the token value which was generated in the previous step.



- Next, we are going to setup GitHub Action to run and calculate code coverage after every push. The calculated coverage report would be uploaded on codecov.io and would be visible on their dashboard.
- Additionally, I also added a possibility to run `R CMD CHECK` after every push. `R CMD check` is a tool that runs a series of automated checks on an R package to ensure it's correctly structured, documented, and error-free. It helps catch issues in code, tests, and documentation before sharing or submitting to CRAN. So it is like an additional validation that we have on our code.
- The new workflow file looks like below

```yaml
name: R-CMD-check and Codecov

on:
  push:
    branches: [master]
  pull_request:
    branches: [master]

jobs:
  R-CMD-check:
    runs-on: ubuntu-latest

    steps:
```

```
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up R
        uses: r-lib/actions/setup-r@v2

      - name: Set up pandoc
        uses: r-lib/actions/setup-pandoc@v2

      - name: Install dependencies
        run: |
          install.packages(c("remotes", "rcmdcheck", "covr"))
          remotes::install_deps(dependencies = TRUE)
        shell: Rscript {0}

      - name: Run R CMD check
        run: |
          rcmdcheck::rcmdcheck(args = "--no-manual", error_on = "warning")
        shell: Rscript {0}

      - name: Run test coverage
        run: |
          covr::codecov()
        shell: Rscript {0}
        env:
          CODECOV_TOKEN: ${{ secrets.CODECOV_TOKEN }}
```

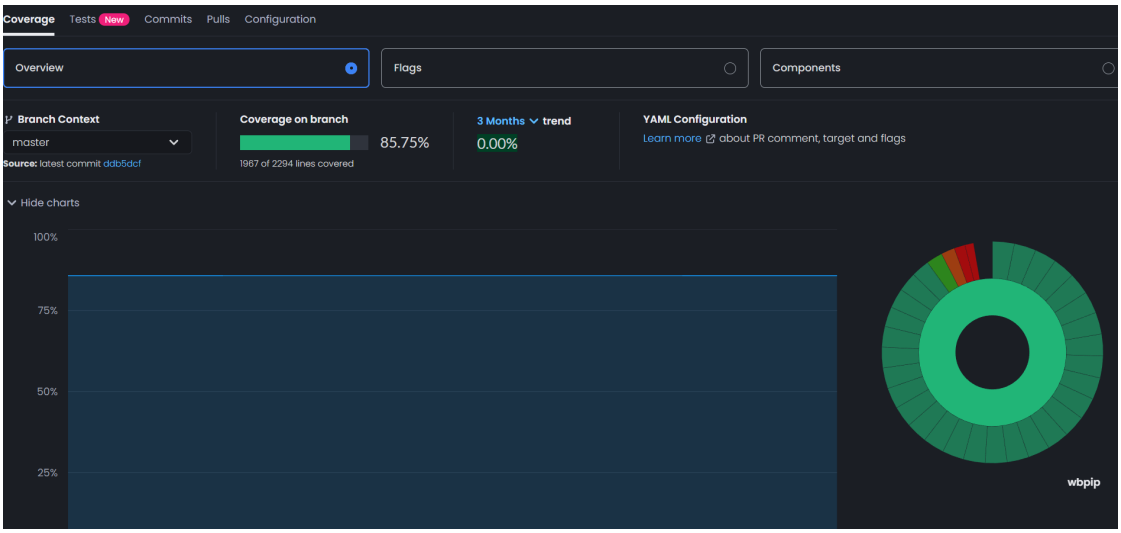
- This file is self explanatory but briefly, it checks out the repository that we want to run our action on, sets up R to run `R CMD CHECK` and finally generate code coverage report and upload it to codecov.io .
- One tip that I can share is to check if this workflow file works on your local branch before running on `master` branch. To do that you should temporarily enable the workflow file to run on your local branch. This can be done as below -

```
on:
  push:
    branches: [master, your-branch]
```

  where `your-branch` is the name of the local branch that you want to run the workflow for. Once you have verified that everything works as expected in the local branch, you can remove `your-branch` from the list again.

- Once the workflow runs successfully the dashboard on codecov.io should be updated and

you should see something like this



- Every time a push or PR is made to `master` the dashboard will be updated with latest data.

3 Setting up Github Actions for Auto Deployment of Quarto book

3.1 Introduction

One of the best parts of using Quarto for websites, blogs, or reports is how easily it integrates with GitHub Pages. With a simple GitHub Actions workflow, you can automatically render and publish your site every time you update your repository. In this post we are going to learn how we have enabled auto deployment for this quarto book.

3.2 Workflow

This is the workflow that we are using in Github Actions . Let's look at it one by one.

```yaml
on:
  workflow_dispatch:
  push:
    branches: main

name: Quarto Publish

jobs:
  build-deploy:
    runs-on: ubuntu-latest
    permissions:
      contents: write
    steps:
      - name: Check out repository
        uses: actions/checkout@v4

      - name: Set up Quarto
        uses: quarto-dev/quarto-actions/setup@v2
        env:
          GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

```
        with:
          tinytex: true
      - name: Render and Publish
        uses: quarto-dev/quarto-actions/publish@v2
        with:
          target: gh-pages
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

3.2.1 Triggering the Workflow

```
on:
  workflow_dispatch:
  push:
    branches: main
```

This tells GitHub Actions when to run the workflow. There are two triggers here:

- **`push to main`** – Any time you commit or merge changes into the `main` branch, the workflow runs
- **`workflow_dispatch`** – Allows you to manually trigger the workflow from the GitHub Actions tab in your repository. This is useful when you want to force a rebuild and republish without committing new changes.

3.2.2 Naming the workflow

```
name: Quarto Publish
```

This gives the workflow a friendly name that will appear in the Actions tab.

3.2.3 Defining the job

```
jobs:
  build-deploy:
    runs-on: ubuntu-latest
    permissions:
      contents: write
```

Here we're defining a single job called `build-deploy`.

- **`runs-on: ubuntu-latest`** – The job will run inside an Ubuntu-based virtual machine provided by GitHub.
- **`permissions: contents: write`** – The workflow needs permission to write to the repository (required for publishing to the `gh-pages` branch).

3.2.4 The Steps

3.2.4.1 1. Check out the repository

```yaml
- name: Check out repository
  uses: actions/checkout@v4
```

This makes your repository's files available in the workflow environment so Quarto can render your project.

3.2.4.2 2. Set up Quarto

```yaml
- name: Set up Quarto
  uses: quarto-dev/quarto-actions/setup@v2
  env:
    GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
  with:
    tinytex: true
```

This installs Quarto in the workflow environment. The `tinytex: true` option ensures LaTeX support is available for rendering PDFs. The `GH_TOKEN` is github token repository secret that is added in Repo settings -> Secrets and Variables -> Actions . It is used for authentication when publishing.

3.2.4.3 3. Render and Publish

```yaml
- name: Render and Publish
  uses: quarto-dev/quarto-actions/publish@v2
  with:
    target: gh-pages
  env:
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

This step does two things:

1. **Renders** your Quarto project (turns `.qmd` files into HTML, PDF, or other output formats).
2. **Publishes** the output to the `gh-pages` branch, which GitHub Pages uses to serve your site. The `target: gh-pages` option ensures everything is pushed to the right branch.

3.3 Don't ignore the `.gitignore` file

Make sure that your `.gitignore` file excludes `_book`, `_site` folders. These are the folders where Quarto renders HTML/PDF files when testing them locally. These files should not be tracked since Github Actions will build them with our auto deployment process.

3.4 Conclusion :

With this workflow in place, the Quarto book will automatically rebuild and deploy whenever a push is made to the `main` branch or whenever we manually trigger the workflow. No more running commands locally or remembering to push generated files.

This is a clean, reproducible, and automated way to publish your Quarto projects using GitHub Pages. As a side not `usethis` package has a lot of good utility functions that helps you to set up similar workflow. You may explore using them. A good starting point is `usethis::use_github_action("render-quarto")`.

For reference the Quarto book is published here.

4 Summary

References