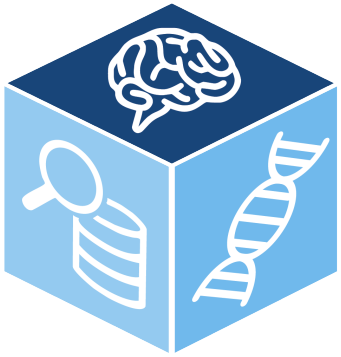


D3b

Center for Data-Driven Discovery in Biomedicine

Developer Handbook

July 31, 2018

Contents

1 Welcome

2 Code Standards

2.1 Languages and Libraries

New projects and code bases should opt to use languages already in use by other projects in the center so as to minimize barriers in on-boarding to the project and allow for better code re-use. In the current case, this means that most data intensive code should focus on Python whenever possible and front-end applications on Javascript and React.

2.2 Formatting

Python code should follow the pep8 standard[10] and use Sphinx styled reST[4] for docstrings. Javascript should follow Airbnb's style guidelines[1].

2.3 Documentation

The format and quantity of documentation necessary is dependent on the purpose and use case of the project. At the very least, the README in the repository should offer a basic overview and elementary steps to get started with development on the repository. See the Section 7.5 for more.

For other forms of documentation, Sphinx[3], Swagger[9], or Slate[7] are good tools to consider.

3 Issues

Github Issues are used to maintain a backlog of work to be done for each repository as well as estimate and schedule work using Zenhub.

3.1 Creating Issues

Issues may be created by users or internal developers for a handful of reasons from questions to feature requests to bugs. Creation of an issue should ideally happen the moment a bug is found or a new feature is desired to promote a shared awareness of that issue and provide a place to track it. Although Zenhub provides an overview of all issues in the organization, issues inherently belong to only one repository. When creating an issue, it is best to target and phrase it towards a single repository. If an issue seemingly spans many repositories, it likely needs to be broken down into more precise tasks.

Issue labels (See section 3.3) should be chosen as needed to easily sort through the backlog. Assignees should be appointed to an issue once they have volunteered to address the issue, or, at the latest, have begun work on it. There currently is no standard across

D3b for formatting an issue body and should be implemented on a repository level. If a repository has an *Issue Template*, make sure to follow it!

3.2 Finding Issues

3.3 Labels

Repositories across the D3b Github organization that have frequent user feedback and bug reports should use our standard set of labels to tag issues shown in Figure 1. Labels act as a means of filtering categories of issues and providing summaries of effort completed in release notes. Users that are not part of the organization may not add labels themselves, so please add labels as you see necessary to new issues. The help wanted label is a special label that is displayed in repository summaries on Github (See the *11 Issues need help* in Figure 8). Use it to mark issues that are self-contained and do not require in-depth knowledge of how the code base works. This tag is useful for guiding new members or external contributors looking to get involved to issues that will be easy for them to accomplish.

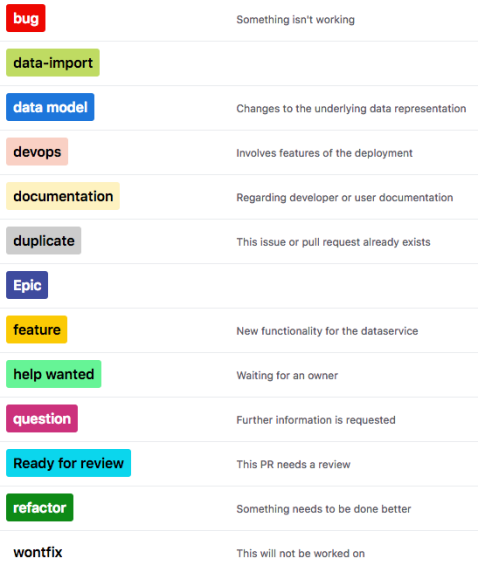
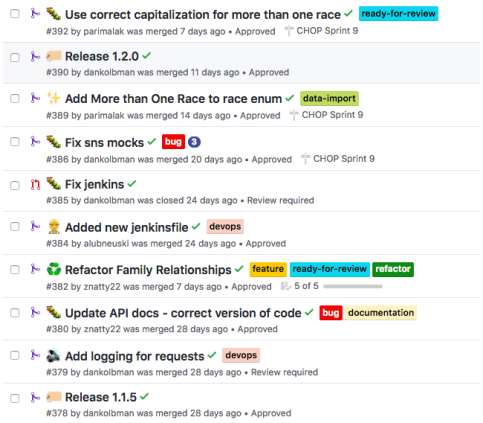


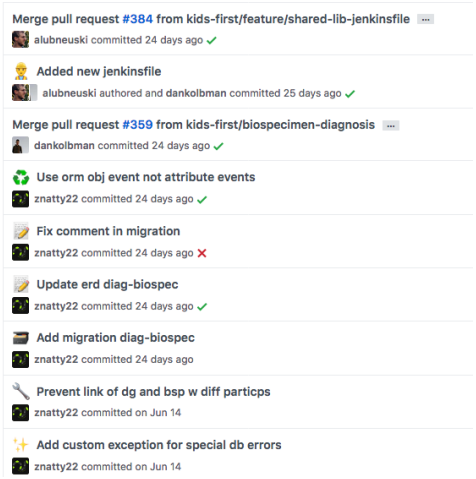
Figure 1: Standard Repository labels with descriptions and colors

4 Creating a Pull Request

Creating a pull request is the first step to getting new code merged back into the code base. Opening of a new pull request may signify that changes are ready for review, or that the author wishes to give visibility into their current status. This section outlines the common standards and practices around the pull request and review process.



(a) Closed pull requests in Github



(b) A slice of the dataservice commit history in Github.

Figure 2: Use of emojis in the dataservice.

4.1 Commit Messages

Commit messages are vital in being able to navigate a code base's history. There are many resources providing best practices for commit messages [2, 8].

We require commits to be prefixed with a single emoji that appropriately summarizes the the gist of the commit. This is helpful in allowing one to scan through the commit log visually inside of Github (Figure 2b) or be able to perform summary analysis inside of release notes or when analyzing git activity as a whole. When adding an emoji to the commit message, use the colon-ated version as opposed to the actual unicode symbol, eg: *:sparkles:*. For inspiration of what emoji may be most appropriate, see the gitmoji guide [5].

4.2 Pull Request Titles

Pull request titles should begin with an emoji, similar to commit messages. This provides similar benefits to commit message emojis where past pull requests may quickly be scanned and summarized (Figure 2a).

4.3 Keeping the Commit Log Tidy

In addition to best practices outlined in Section 4.1, care should be taken to avoid introducing unhelpful commits such as merges of master into a feature branch. Often, a feature branch will become stale during development as other features are merged into master. Instead of using the *Update* button in Github to merge the latest master into the feature branch, one should opt to rebase the branch on the latest master. This will avoid adding many *Merged branch master* commits into a feature branches

4.4 Labeling Pull Requests

The labels on pull requests are shared with those in issues (Section 3.3, Figure 1). Pull requests should be labeled appropriately so that they may be filtered and analyzed easily.

One special label that applies only to pull requests is the *ready-for-review* label. Marking an open pull request with this label signals to other developers that it is ready for a code review. This label may also be used by automated tooling to send notifications on un-reviewed pull requests

4.5 Requesting Reviews

Reviewers may be requested in Github inside of a pull request (Figure 3). This will notify the user via Github notifications and email, if the user has them enabled. Github notifications can be easily missed, so it is advised to use direct messing in Slack if the request is urgent.

5 Review Process

5.1 Code Review

5.2 Testing

Commit hooks should trigger testing integrations upon any push to master or a branch with an open pull request, at the least. Among these integrations should be at least on test runner that will run the repositories unit tests and report status back to Github. Having tests is critical to ensuring that new features are hardened for future development. Tests for new features in a pull request are expected to have tests included at the time of review. Existence and functionality of these tests should be checked as part of the review process.

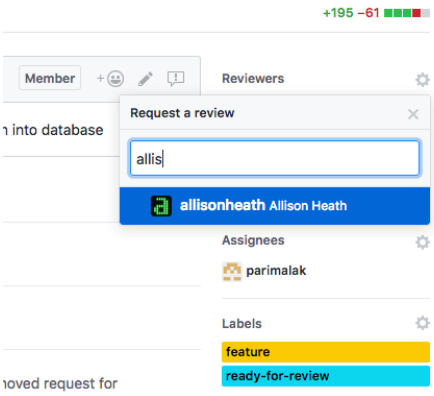


Figure 3: Requesting a reviewer on a pull request.

5.3 Status Checks

Status checks is a feature in Github that listens to services that run in response to git hooks (Figure 4). Pull requests will be prevented from merging until each of these services reports back positively on the state of the code. In addition to services, reviews are may also be included as status checks, requiring one or more approvals from reviewers before being marked as ok.

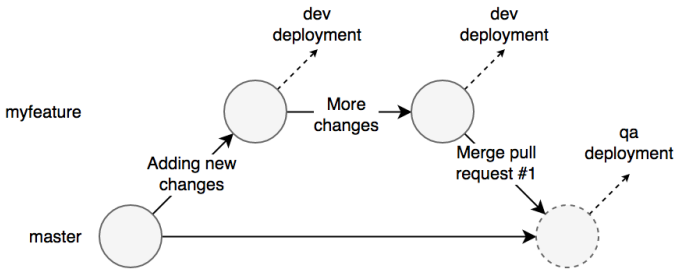


Figure 5: Deployment steps during development of a new feature.

6 Continuous Deployment Process

Service deployments are automatically triggered by new commits to a repository.

All commits branches that are not *master* build and deploy to the *dev* environment (Section 8.1).

Any commit to *master* will trigger a deployment to *qa* (Section 8.2). As a result, once the review process is complete (Section 4) and the feature branch is merged into *master*, a deployment to *qa* will occur.

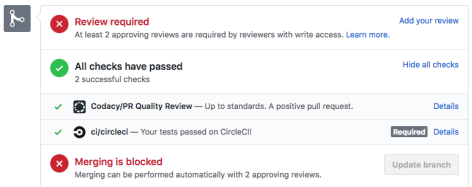


Figure 4: Status checks on a pull request.

A deployment to production, or a release, occurs when the master branch is tagged with a semantic version (Figure 6). There are a couple other steps that should happen before the tag is added, however. The process is outlined here:

First, a feature branch titled *release-x.y.z* is created. A commit bumping all version strings and an updated *CHANGELOG* is created with a message *:label: Release x.y.z*. A pull request for this release branch is created similar to the process in Section 4. During this review, it is important to obtain signoffs from anyone who may be impacted by the new release. This ensures there is a shared awareness that a new version of the software is about to be released. After these signoffs have been acquired, the release branch is merged and a release is created on Github. This will trigger a production deployment and will require one last signoff by an administrator before it is deployed. Once the administrator has allowed the deployment to proceed, the new release should be available in the production environment.

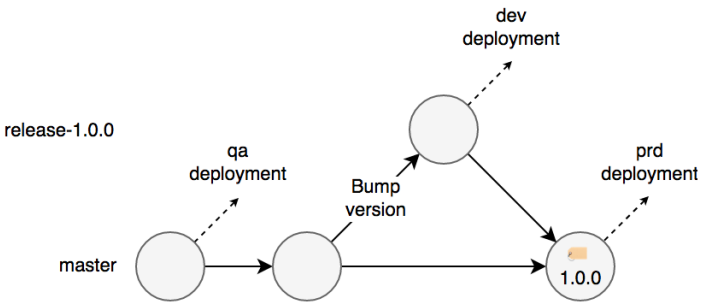


Figure 6: Deployment steps during releases.

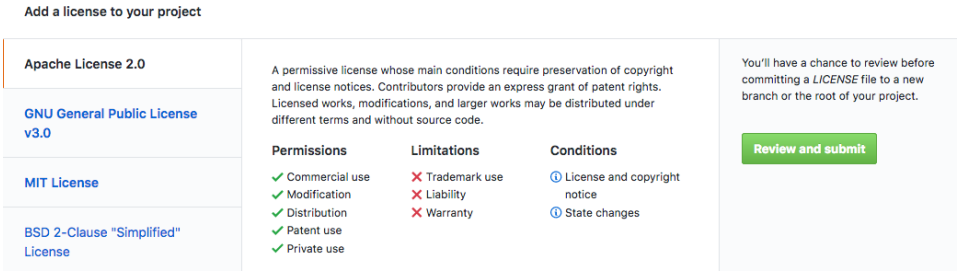


Figure 7: The Apache 2.0 license is available by default from the Github web interface.

7 New Repositories

7.1 Naming

7.2 Description

7.3 Tags

7.4 Protected Branches

7.5 README

7.6 License

Kids First uses the Apache 2.0 [6] license exclusively for all public repositories. The file should be placed in the root of the directory with the name *LICENSE*. The Apache 2.0 license may be easily added from the root repository view on Github using *add new file* and naming it LICENSE. This will present a *Choose license template* button to the right of the file name where the Apache 2.0 License may be chosen to automatically populate the file (See Fig 7).

7.7 Config Repository

In addition to creating the main code repository, each code repository that is to be deployed must have a second, *private* repository that contains the infrastructure and CI/CD strategy for that code base. The repository for the config *must match the name of the code repository suffixed with -config*. This is a rule of thumb that is followed throughout so that the configuration may be discovered automatically by other scripts. In addition to the naming standard, the repository should be *private*. It is best practice to keep the infrastructure obscured and prevent external contributions as the config contains code that is often run at elevated privileges in our environments.

8 Environments

Often, entire software stacks are deployed concurrently in many different *environments*. This provides us with a handful of benefits:

- Assurance that there is always a stable deployment for end-users
- Provides a safe sandbox to develop new features in isolation
- Insures the boilerplate to re-deploy a new environment exists

We divide our environments into three primary stages that are tightly couple to our Continuous Integration

8.1 dev

The development environment (dev) is intended as sandbox for new features and code bases. It is the most unstable environment with no guarantees on data availability. When new code is pushed to a feature branch, it automatically triggers its CI pipeline to test, build, and, deploy to the development environment. The branch must pass through testing and deployment successfully to before it is allowed to be merged to insure that it will successfully deploy to the next stage.

8.2 qa

The QA environment provides a mostly-stable environment where new features may be used, evaluated, and tested before being released. This is is where integration test-suites may be run to ensure that all services cooperate nicely. The data in QA should also be close to production quality to fully flex all features. New features enter QA after merging a feature branch into the master branch of a code base.

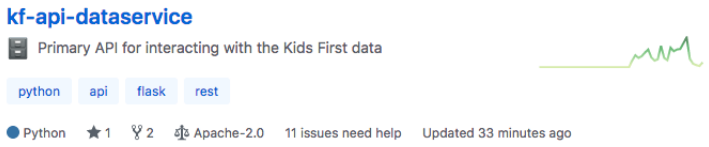


Figure 8: An exemplary repository summary including: proper naming scheme, emoji followed by short description, labels, the *Apache 2.0* license, and issues tagged with *help wanted*

8.3 prd

The production environment (prd) is the end-user environment. This is critically stable as it must be exposed to the public. To introduce features into prd, a repository must go through the release process. This includes creating a pull request for the version change, signoff on it from any potential stakeholders, merging, and finally acquiring final approval from an administrator to deploy the infrastructure and code into the environment.

References

[1] Airbnb. Airbnb javascript style guide. `https://github.com/airbnb/javascript`.

[2] Chris Beams. Git commits. `https://chris.beams.io/posts/git-commit/`.

[3] Georg Brandl and the Sphinx team. Sphinx. `https://pythonhosted.org/an_example_pypi_project/sphinx.html`.

[4] Andrew Carter. Documenting your project using sphinx. `https://pythonhosted.org/an_example_pypi_project/sphinx.html`.

[5] Carlos Cuesta. gitmoji. `https://gitmoji.carloscuesta.me`.

[6] The Apache Software Foundation. Apache license version 2.0. `https://www.apache.org/licenses/LICENSE-2.0`, 2004.

[7] Robert Lord. Beautiful static documentation for your api. `https://github.com/lord/slatel`.

[8] Stack Overflow. Standard to follow when writing git commit messages. `https://stackoverflow.com/questions/15324900/standard-to-follow-when-writing-git-commit-messages`.

[9] SmartBear. Swagger. `https://swagger.io`.

[10] Guido van Rossum. Pep 8 – style guide for python code. `https://www.python.org/dev/peps/pep-0008`.