Committing Code Guidelines

- Introducing GitHub
- Contribution Guidance - Top 4 things to do (or not do)
- Conventional Commits
- EdgeX GitHub Getting Started
- GitHub Flow
    - Branching Conventions
    - Commits
    - Commit Messages
    - Merging
    - Miscellaneous

Introducing GitHub

The EdgeX has selected GitHub as the code management repository for the project. GitHub is a web-based Git or version control repository and Internet hosting service. It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features. It provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project.

If you are completely new to GitHub then the Hello World project is a time-honored tradition in computer programming. It is a simple exercise that gets you started when learning something new. By following the GitHub hello world example you will learn how to:

- Create or clone a repository
- Start and manage a new **branch**
- Make changes to a file and push them to GitHub as **commits**
- Open and merge a **pull request**

Contribution Guidance - Top 4 things to do (or not do)

While this document and the others on our Wiki site provide a lot of great guidance to developers planning to make contributions to the project, we would ask that developers (and the maintainers/committers) of the project pay particular attention to the following "top 4" practices and adhere to them closely when working on project repositories.

1. Commit subject line messages should explain the problem that the commit is solving in 50 characters or less. “Fixing bug” or “adding comments” doesn’t explain the problem the commit is solving!.
2. Provide a message body to the commit to provide more detail explanation. When you do, keep lines in the body to 72 characters or less.

   You can add a commit subject line and multiple message bodies like this:

   git commit –s –m “This is the subject; keep to 50 char” –m “This is the first line of the message body"$'\n'"This is the second line of the message body."$'\n'"Keep all message lines to 72 char.”
3. Commit your changes in logical chunks. Make it easier for the community to review, comment on, and accept your contribution. For example, create a separate commit for a big fix and an enhancement.
4. Rebase the upstream development branch into your topic branch. Do not merge master into your local PR branch (this is the lazy approach to dealing with committed code locally in a branch, and then realizing that your PR isn’t merge-able because HEAD has changed). See https://git-scm.com/book/en/v2/Git-Branching-Rebasing if you need more help.

Conventional Commits

EdgeX Foundry developers follow the Conventional Commits specification for commit messages when submitting pull requests. The specification aims to make commit messages more human and machine readable. Conventional commit messages can be used to automatically generate release notes. They also help developers focus PRs into smaller "chunks" of related functionality.

When you submit your pull request (PR) against an EdgeX repository, you must provide a Conventional Commit type and scope in the subject line of your commit. The type specifies the nature of the pull request. The scope provides the service or component being addressed or context of the pull request. In addition to type and scope, you must describe your commit. You can optionally provide additional comments (or commit body) and footers.

The types and scope will vary per repository. The following is the current standard as of 2020-09-24

From: EdgeX-TSC-Core@lists.edgexfoundry.org <EdgeX-TSC-Core@lists.edgexfoundry.org> **On Behalf Of** Jim White
Sent: Thursday, September 24, 2020 12:12 PM
To: Core WG <edgex-tsc-core@lists.edgexfoundry.org>
Subject: [Edgex-tsc-core] Conventional Commits

Hello core working group members. Per the Core WG meeting today, I am looking for feedback on our upcoming adoption of conventional commits specification across repositories under Core WG purview. Thanks to Mike Johanson, the types and scopes for edgex-go are to be set at these:

edgex-go

types: feat (for feature), fix, docs, style, refactor, perf (for performance), test, build, ci, revert

scopes: core-data (or data), core-metadata (or metadata or meta), core-command (or command or cmd), snap, docker, security, scheduler, notifications, sma, deps (for dependencies), all

For the other repositories, I propose the following.

all mods (bootstrap, core-contracts, messaging, registry, configuration)
types: feat (for feature), fix, docs, style, refactor, perf (for performance), test, build, ci, revert
scopes: [no scope for general code or scope does not apply], models, errors, clients, config, docker

edgex-docs
types: feat (for feature), fix, style, build, ci, revert
scopes: [no scope for general docs or scope does not apply], intro (and getting started), api, adr, design, examples, microservices, ref (for reference)

edgex-cli
types: feat (for feature), fix, docs, style, refactor, perf (for performance), test, build, ci, revert
scopes: [no scope for general code or scope does not apply], res, samples, config

edgex-ui-go
types: feat (for feature), fix, docs, style, refactor, perf (for performance), test, build, ci, revert
scopes: [no scope for general code or scope does not apply], image

docker-edgex-consul
types: feat (for feature), fix, docs, refactor, perf (for performance), ci, revert
scopes: [no scope for general code or scope does not apply], config, scripts (or bin)

docker-edgex-mongo
types: feat (for feature), fix, docs, refactor, perf (for performance), ci, revert
scopes: [no scope for general code or scope does not apply], config, scripts (or bin)

developer-scripts
types: feat (for feature), fix, docs, style, refactor, perf (for performance), test, build, ci, revert
scopes: no scope or compose or source (for compose source)

edgex-examples
types: feat (for feature), fix, docs, style, refactor, perf (for performance), test, build, ci, revert
scopes: app (for app services), device (for device services), temp (for templates), security, all, or no scope if it does not apply to these

Please weigh in with any other changes or additions. We'll resolve and adopt with next week's meeting.

Jim White

Examples of proper messages (in the context of sample Git CLI commands) are:

- git commit -s -m "fix (core-data): repair the incorrect status code returned by the get all query"
- git commit -s -m "build (all): upgrade the services to Go 1.15"

If you do not provide type in your commit message, an automated GitHub bot will flag your pull request and it will fail its checks. When your pull request has failed checks, it cannot be merged until the problem is corrected. If you do not provide scope, the EdgeX reviewers are expected to flag your pull request and ask that you make a change to provide a scope. Without reviewers approval, your pull request does not pass the reviewer check and it cannot be merged.

The body of the commit should contain a reference to issue that is addressed (closes and/or fixes). Example:

- git commit -s -m "fix(core-data): fixed some big error description" -m "fixes: #123"
- git commit -s -m "fix(core-data): fixed some big error description" -m "closes: #123"

EdgeX GitHub Getting Started

The EdgeX GitHub organization has been configured such that all repositories it hosts have been setup with branch protection in place. This requires new contributors to fork a project repository and work on a local copy before submitting a Pull Request so that any new changes get merged back into the master project repository on GitHub. It is not an option for a committer to just clone the repository and push changes directly back to it.

1. Create a Fork on GitHub (see fork example).

2. Create a local clone of your fork.

```
$ git clone git@github.com:$MYACCOUNT/$REPO
```

3. Configure Git to sync your fork with the original repository.

4. Add a remote reference so that Git can sync your fork with the master project repository.

```
$ cd $REPO

$ git remote add upstream git@github.com:$UPSTREAM/$REPO
```

A detailed example illustrating these steps is provided here.

From this point the basic workflow is as follows:

1. Start new work on a new feature branch.

```
$ git checkout -b new-feature
```

2. Update your code and commit. You can either:

a. explicitly add individual file changes in your working directory to you index.

```
$ git add file1.txt

# or to add all files

$ git add .

# review changes

$ git diff –cached

$ git commit -s
```

b. use the git commit -a to automatically stage all tracked, modified files before the commit If you think the git add stage of the workflow is too cumbersome. This basically tells Git to run git add on any file that is "tracked" - that is, any file that was in your last commit and has been modified. Please use caution when using this approach as it's easy to accidently commit files unintentionally (eg. build artifacts, temp files from editors, ...).

```
git commit -as
```

The **-s** option used for both alternatives causes a committer **signed-off-by** line to be appended to the end of the commit message body. It certifies that committer has the rights to submit this work under the same license and agrees to our Developer Certificate of Origin (see Contributor's Guide for more details about our DCO). E.g.

```
signed-off by: John Doe johndoe@example.com
```

In order to use the **-s** option, you need to make sure you configure your git name (user.name) and email address (user.email):

```
$ git config --global user.name “John Doe”

$ git config --global user.email johndoe@example.com


# To check

$ git config --list
```

3. Add a commit message: See Commit Messages section below.

4. Push work to your repository as a new branch.

```
$ git push origin new-feature
```

5. Create a Pull Request. This is a formal request for the project's maintainers to review the diff and approve/merge, or possibly request that changes be made.

6. While that is happening you can work on something else! Sync your fork of the repository to keep it up-to-date with the upstream repository. Fetch the branches and their respective commits from the upstream repository.

```
$ git fetch upstream
```

7. Check out your fork's local master branch.

```
$ git checkout master
```

8. Rebase the changes from upstream/master into your local master branch. This brings your fork's master branch into sync with the upstream repository, without losing your local changes.

```
$ git pull --rebase upstream master
```

9. Push the updated master back to your fork.

```
$ git push origin master
```

Note: it was suggested to use git checkout master, then git pull origin master in preference to above (to be discussed).

10. Start new feature.

```
$ git checkout -b new-feature2
```

11. Repeat the steps above as appropriate

12. If a change was requested on the new-feature PR then simply rebase your working branch from upstream:

```
$ git pull --rebase upstream master
```

13. Make changes.

14. Update the PR killing off the older changes.

```
$ git commit -as --amend
```

15. Force push the updated PR back up.

```
$ git push origin new-feature2 –-force
```

By using the --amend and --force options means that any of the commits that you produce should be clean, non-breaking changes at all times that get merged in, instead of having a set of patches in a PR that may have one or two 'fixup' changes.

GitHub Flow

The EdgeX project has adopted the GitHub Flow workflow, a lightweight, branch-based workflow that supports teams and projects where deployments are made regularly.

GitHub Flow is in a nutshell:

- Anything in the master branch is tested, functioning code that is usable.
- To work on something new, create a descriptively named branch off master (i.e. new-oauth2-scopes).
- Commit to that branch locally and regularly push your work to the same named branch on the server. Rebase regularly from upstream or face the pain of merge conflicts.
- When you need feedback or help, or you think the branch is ready for merging, open a Pull Request.
- A Pull Request is a request that someone else (e.g. an appropriate reviewer such as a project Maintainer or Committer) review the work that you've done and merge your changes in. When you create a pull request, you need to select 2 branches on GitHub, the branch that you've made your changes on, and the branch where you would want to merge your changes into. Because Pull Requests occur in GitHub, you would need to push your branch to GitHub before you create the request. If they review the changes on your branch and everything looks good, they can just merge

the branch (and often delete the branch to clean up). However, if they have questions or comments, they can leave a comment on the pull request, i.e. with changes that you might need to make before they merge in. Then if they needed you to make another change, you can make the change to your code, then commit and push to your existing branch. Because a Pull Request is just a request to merge to branches in GitHub, since you've updated your branch, you've updated the Pull Request as well. They can then review your latest update and merge in.
- The main rule of GitHub Flow is that master should always be in deployable (usable operational state). GitHub Flow allows and encourages continuous delivery.
- Changes are submitted from developer feature branches as pull-requests against master, then merged using merge commits (which is the default for GitHub merges via their UI).
- When a new version is released, a tag is created, and development continues as before, via pull-requests submitted against master.
- If/when the need for supporting a maintenance release of a specific microservice arises, a branch is created from the required release tag, which is then used for release-specific bug fixes, potentially cherry-picked from master and/or other release branches (if they exist).

Branching Conventions

- Choose *short* and *descriptive* names

```
# good

$ git checkout -b oauth-migration

# bad - too vague

$ git checkout -b login-fix
```

Identifiers from corresponding tickets in an external service (e.g. a GitHub issue) are also good candidates for use in branch names. For example:

```
# GitHub issue #15

$ git checkout -b issue-15
```

Use *hyphens* to separate words.

- When naming Feature branches common practice is to use the naming convention *feature/my-feature.* This allows feature specific branches to be easily identified in the repository.
- When several people are working on the *same* feature, it might be convenient to have *personal* feature branches and a *team-wide* feature branch. Use of a team-wide branch will require the co-operation of the project's Maintainers/Committers to setup the branch in the main repository. Use the following naming convention:

```
$ git checkout -b feature/master # team-wide branch

$ git checkout -b feature/maria  # Maria's personal branch

$ git checkout -b feature/nick   # Nick's personal branch
```

Merge at will the personal branches to the team-wide branch. Eventually, the team-wide branch will be merged to master.

Commits

- Each commit should be a single *logical change*. Don't make several *logical changes* in one commit. For example, if a patch fixes a bug and optimizes the performance of a feature, split it into two separate commits.

Tip: Use git add -p to interactively stage specific portions of the modified files.

- Don't split a single *logical change* into several commits. For example, the implementation of a feature and the corresponding tests should be in the same commit.
- If you are adding or changing functionality, your commit should include new or modified tests to cover the change.
- Commit *early* and *often*. Small, self-contained commits are easier to understand and revert when something goes wrong.
- Commits should be ordered *logically*. For example, if *commit X* depends on changes done in *commit Y*, then *commit Y* should come before *commit X*.

Note: While working alone on a local branch that *has not yet been pushed*, it's fine to use commits as temporary snapshots of your work. However, it still holds true that you should apply all of the above *before* pushing it.

Commit Messages

Follow the seven rules below and you shouldn't run into any problems.

The seven rules of a good Git commit message are:

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line

6. Wrap the body at 72 characters
7. Use the body to explain *what* and *why* vs. *how*

For example:

> **Summarize changes in around 50 characters or less**
>
> **More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of the commit and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); various tools like `log`, `shortlog` and `rebase` can get confused if you run the two together.**
>
> **Explain the problem that this commit is solving. Focus on why you are making this change as opposed to how (the code explains that).**
>
> **Are there side effects or other unintuitive consequences of this change? Here's the place to explain them.**
>
> **Further paragraphs come after blank lines.**
>
> - Bullet points are okay, too
> - Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary
>
> **If you use an issue tracker, put references to them at the bottom, like this:**
>
> **Resolves: #123**
>
> **See also: #456, #789**

Ultimately, when writing a commit message, think about what you would need to know if you run across the commit in a year from now.

- If a *commit A* depends on *commit B*, the dependency should be stated in the message of *commit A*.
- Similarly, if *commit A* solves a bug introduced by *commit B*, it should also be stated in the message of *commit A*.
- If a commit is going to be squashed to another commit use the --squash and --fixup flags respectively, in order to make the intention clear.

```
$ git commit --squash f387cab2
```

Merging

- **Always prefer a rebase to a merge (see above)**
- **Do not rewrite published history.** The repository's history is valuable in its own right and it is very important to be able to tell *what actually happened*. Altering published history is a common source of problems for anyone working on the project.
- However, there are cases where rewriting history is legitimate. These are when:
    - You are the only one working on the branch and it is not being reviewed.

That said *never rewrite the history of the "master" branch* or any other special branches (i.e. used by production or CI servers).

- Keep the history *clean* and *simple*. *Just before you merge* your branch:
    - Make sure it conforms to the style guide and perform any needed actions if it doesn't (squash/reorder commits, reword messages etc.)
- If your branch includes more than one commit, do not merge with a fast-forward

```
# good - ensures that a merge commit is created

$ git merge --no-ff my-branch

# bad

$ git merge my-branch
```

Miscellaneous

- *Be consistent.* This is related to the workflow but also expands to things like commit messages, branch names and tags. Having a consistent style throughout the repository makes it easy to understand what is going on by looking at the log, a commit message etc.
- *Test before you push.* Do not push half-done work.
- Use annotated tags for marking releases or other important points in the history. Prefer lightweight tags for personal use, such as to bookmark commits for future reference.
- Keep your repositories at a good shape by performing maintenance tasks occasionally:
    - git-gc(1)
    - git-prune(1)
    - git-fsck(1)

Commit subject line messages should explain the problem that the commit is solving in 50 characters or less. “Fixing bug” or “adding comments” doesn’t explain the problem the commit is solving!.

Provide a message body to the commit to provide more detail explanation. When you do, keep lines in the body to 72 characters or less.

You can add a commit subject line and multiple message bodies like this:

git commit –s –m “fix(core-data): This is the subject; keep to 50 char” –m “This is the first line of the message body"$'\n'"This is the second line of the message body."$'\n'"Keep all message lines to 72 char.”

Commit your changes in logical chunks. Make it easier for the community to review, comment on, and accept your contribution. For example, create a separate commit for a big fix and an enhancement.

Rebase the upstream development branch into your topic branch. Do not merge master into your local PR branch (this is the lazy approach to dealing with committed code locally in a branch, and then realizing that your PR isn’t merge-able because HEAD has changed). See https://git-scm.com/book/en/v2/Git-Branching-Rebasing if you need more help.