Committing Code Guidelines

Introducing GitHub

The EdgeX has select 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 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:

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

EdgeX GitHub Getting Started

The EdgeX GitHub repository has been setup with branch protection in place. This requires new contributors to fork the repository and work on your own copy before submitting a Pull Request to merge any changes back into the main project repository. 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.

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

3. Configure Git to sync your fork with the original 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 (using the convention naming convention *feature/my-feature*).

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

2. Update your code and commit.

```
$ git commit -as
```

3. Add a commit message: See Commit Messages section below.
4. Push work to your repository as a new branch.

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

5. Create a Pull Request
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. Merge 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 merge --ff-only upstream/master
```

9. Push the updated master back to your fork.

```
$ git push
```

10. Start new feature.

```
$ git checkout feature/new-feature2
```

11. Repeat the steps above as appropriate

12. If a change was requested on the new-feature PR then:

```
$ git checkout master
$ git fetch upstream
$ git merge --ff-only upstream/master
$ git checkout feature/new-feature
$ git rebase 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 feature/new-feature --force
```

Bu 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 deployable
- 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
- When you need feedback or help, or you think the branch is ready for merging, open a Pull Request
- After someone else has reviewed and signed off on the feature, you can merge it into master
- Once it is merged and pushed to ‘master’, you can and *should* deploy immediately

The main rule of GitHub Flow is that master should always be deployable. 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.
- Feature branches should have the naming convention *feature/my-feature*
- 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 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.
- 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

EdgeX will follow will well-established conventions for creating consistent, well written Git commit messages. Just 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 here

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

- **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:
    i. 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)