Source Code Management Standard and Guidelines

This goal of this document is to explain Apigee’s model for managing source code developed on the Apigee platform. Git via Github has been chosen due to the ease of use and popularity among the developer community. There are many different Git branching and development models that can be used for managing source code. In this document, only one branching model will be described as it has proven to be successful in multiple Apigee implementations, executed by small and large development teams.

The branching model that Apigee uses is based from a model authored by Vincent Driessen, an independent Python software engineer and consultant. His work is referenced throughout this document, print and diagrams, and its original form is located at http://nvie.com/posts/a-successful-git-branching-model. Only the Fork & Pull collaborative development model, using Pull Requests, is used for Apigee source code development. This will also be discussed later in this document.

Branching

Centralized Repository

Let’s start by describing the overall design of the branching model. Logically, there exists only 1 central repository containing trusted code. This is referenced as origin throughout this document. It is currently apigeecs/example-repo hosted on github.com under Apigee’s CS organization. Only a handful of team members will have access to this repo and will be responsible for managing what is pushed into the repo.

Since the Fork & Pull development model is used, developers will only push to the forked repo, which is local to the developer. Pull requests will then be used to initiate a request to merge the code into origin. More about pull requests and the Fork & Pull method is discussed later in this document.

Main Repository Branches

There are 2 main branches, master and develop. These branches will live forever. The origin/master branch contains production-ready code. The origin/develop branch always contains the latest development changes.

When source code in develop has been qualified and is release-ready for production, it is merged back into master and tagged with a release number. More information will be provided later in this document about when and how to tag.

There are additional branches that can be used for sprint development, feature development, and hotfixes.

Additional Repository Branches
In addition to master and develop, more branches can be created to allow for easier parallel development, better tracking of production bug fixes, and isolated feature development. These branches are removed once the code changes are merged back into the main repository branches.

There are 3 different types of branches that can be used:

- Feature branches
- Release branches
- Hotfix branches

These branches are the same 'type' of branches from a Git perspective and only differ on how they are used.

**Feature Branches**

<table>
<thead>
<tr>
<th>Branch from</th>
<th>develop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge to</td>
<td>develop</td>
</tr>
<tr>
<td>Naming convention</td>
<td>{project}{JiraIssue#}. Cannot be master, develop, release-*, hotfix-*. See below information for Apigee specific nomenclature.</td>
</tr>
</tbody>
</table>

Feature branches are used for development of a specific feature and sometimes are referred to as 'topic branches'. These branches should only be created in origin if a Jira user story or task exists for reference. The branch name should follow this nomenclature: \{JiraProject-IssueNumber\}. For example, an APIGEE Jira project with issue 648 means APIGEE-648 becomes the feature branch name. This is the same as the actual Jira issue name.

A targeted release for the feature or topic may not be known at the time. The branch will live until development is complete for the feature and merged back into develop or the feature is dropped. This branch will usually exist within a developer's forked repo and may not exist in origin. If a feature branch is used in a developer's forked repo, it is recommended to use the Apigee nomenclature, but is left to the developer's discretion.

To create a feature branch from the develop branch:

```
Switched to a new branch 'APIGEE-648'
$ git checkout -b APIGEE-648 develop
```

Merging feature branch back to develop:

```
Switched to branch 'develop'
$ git checkout develop
```
The no-ff option is used to disable fast-forward during the merge. This allows the history of a feature branch to be seen historically. If fast-forward was used (default in a plain merge), the develop branch would look as if commits continuously happened on develop and no actual merge of a 'feature' branch took place.

**Release Branches**

<table>
<thead>
<tr>
<th>Branch from</th>
<th>develop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge to</td>
<td>develop, master</td>
</tr>
<tr>
<td>Naming convention</td>
<td>release-{sprint or release #} - An example for release 35, release-35.0.0</td>
</tr>
</tbody>
</table>

Based on Driessen's model, release branches are used in preparation for a production release. Most all development work has been completed. Minor changes are left in order for the code to become 'production ready', such as quick bug fixes, adding version numbers, etc. Using the release branch for this phase of work, the develop branch is opened up for new development on the next release.

The release branch should be opened up when develop is almost at the desired state for production use. An example time for this would be when all code is ready to be deployed in QA for testing.

The version number for the release is set at the start of the release branch, which could be the start of the sprint.

To create a release branch:
This branch may exist for awhile, until it is merged into master for the production release. If sprint work is completed early, develop branch can be used for the next release development. The current release changes will get merged back into develop after the release is completed.

Completing a release branch:
When the state of the release branch has been signed-off for production readiness, the release branch is merged into master, again with no fast-forward (--no-ff). Each commit in master, by definition, is a release, and you must tag the new commit from the merge with a release number. Lastly, the release branch also needs merged back into develop so future releases will contain all changes since the branch creation.

Merging and tagging in master:

```
$ git checkout master
```

```
Switched to branch 'master'
```

```
$ git merge --no-ff release-35.0.0
```

```
Merge made by recursive
```

```
$ git tag -a 35.0.0
```

```
Summary of Changes
```

```
The release is now completed and tagged for future reference.
```

Merge release-35.0.0 back into develop:

```
$ git checkout develop
```

```
Switched to branch 'develop'
```

```
$ git merge --no-ff release-35.0.0
```

```
Merge made by recursive
```

```
Summary of changes:
This step would likely lead to a merge conflict if development has occurred on develop since the release-35.0.0 was opened up and used. If so, fix the merge conflicts and commit. Now that the release branch is merged into master and develop, we can drop the branch.
```
Hotfix Branches

Hotfix branches are similar to release branches as they will contain production-ready code, but these contain unplanned changes. When a critical bug exists in production and needs resolved immediately, a hotfix branch can be branched off the corresponding tag of the master branch containing the production version. Using a hotfix branch allows the team to work on develop or release-* without disruption.

How to handle a hotfix when a release branch is still active?
Merge the hotfix branch into master and the release-* branch. The release branch will get merged back into develop before the next release so the hotfix will exist for future releases if this process is followed.

To create a hotfix branch:

Switched to a new branch "hotfix-35.0.1"

$ git checkout -b hotfix-35.0.1 master

Here a new branch was created from master using the naming convention mentioned above. You then fix the bug and commit the changes in one or more commits:

Commit changes

$ git commit -m "Fixed APIGEE-1254, APIGEE-1255"

[hotfix-35.0.1 abbe5d6] Fixed APIGEE-1254, APIGEE-1255
5 files changed, 32 insertions(plus), 17 deletions(minus)

Completing the hotfix branch:

If there is no active release-* branch, the bugfix needs to be merged back into master and develop. This hotfix branch is also merged into
develop to ensure the fix exists in future releases as well. Essentially the same process as completing a release branch is followed for merging a hotfix branch to master:

- Switched to branch 'master'
  - `$ git checkout master`

- Merge made by recursive
  - `$ git merge --no-ff hotfix-35.0.1`

- Summary of changes
  - `$ git tag -a 35.0.1`

Merge back to develop:

- Switched to branch 'develop'
  - `$ git checkout develop`

- Merge made by recursive
  - `$ git merge --no-ff hotfix-35.0.1`

- Summary of changes:

As mentioned before, there is one exception to the above process. If a release branch still exists and hasn't been merged to master, the change should be merged into release-. These changes will eventually carry to develop before the next release once release- is merged to master and develop as part of the normal process. If the hotfix changes are urgently needed in develop, they can be merged into develop as well when merging into release- and master.

After the hotfix branch is merged back into master, release-, and/or develop, the temporary hotfix branch can be removed:

- Deleted branch hotfix-35.0.1 (was abbe5d6)
  - `$ git branch -d hotfix-35.0.1`

### Pull Requests

Pull requests provide granular control of the base repository (origin). Using pull requests is required. Only a few key team members have administrator access, including push access, to origin or release-* and master branches in origin. Developers typically only have pull access to origin, but a Shared Repository development model will allow developers to have push access to develop or feature branches. More information about pull requests is located and referenced from: [https://help.github.com/articles/using-pull-requests](https://help.github.com/articles/using-pull-requests). When creating a pull request, a developer must provide the Jira user story or task in the comments along with any additional information. This helps to provide proper context for the admin team when reviewing the request. Either the Shared Repository or Fork & Pull development model can be used with in conjunction with pull requests.

**Pull Requests with Fork & Pull Model**
Using Fork & Pull model for pull requests provides another layer of control for the repository. Developers fork a repo from origin (apigeecs/example-repo) and this will create a copy of the repo in their personal github account, creating michaelarusso/example-repo for example. This allows developers to make any changes to their personal fork, without requiring push access to origin. Once the developed code in the local, forked, repo is ready to be brought into origin, a developer will create a pull request using the information above. The admin team of origin will be notified and are required to review the request. This triggers any discussion, if required, to ensure the code requested to be merged is up to standard. More details on creating a fork are located here: [+]https://help.github.com/articles/fork-a-repo+

References

- GitHub instructions how to setup GitHub see in https://help.github.com/articles/set-up-git
- The following GitHub resources could be used for additional support:
  - https://github.com/blog/1557-github-flow-in-the-browser
  - http://pcottle.github.io/learnGitBranching/