CAKEDC GIT WORKFLOW

CakeDC Git Workflow is a project development and release work flow which provides a development and release cycle based on key phases:

- **Development**: All active development is driven by **milestones**, and contains the unstable code base in a bleeding edge state.
- **QA**: Quality assurance testing figures directly as part of the development cycle, assessing both requirements compliance and acceptance criteria.
- **Review**: Clients or reviewers experience a stable code base, passed by a QA process backed up by specific requirements and acceptance criteria.
- **Release**: Release are generated after both a QA and a review process.

The initial design of the work flow is based loosely on the gitflow design, by Vincent Driessen. Although there are similarities between each other, they do not infer compatibility.

ORGANIZATION

The principal design of the CakeDC Git Workflow is oriented towards teams or companies which integrate a QA process as part of their development cycle, as well as

providing a stable stage server for clients to review and approve the pending **release**. However, these steps can easily be skipped for those who don't actively provide this as part of their work flow.

The phases through which the full development and **releases** cycle transpires are broken into objectives, which we refer to as **milestones**. It's important to note that a **milestone** doesn't necessarily equate to a **release**. A version of a project could be composed of multiple **milestones**, depending on the planning of the project and the resources available.

These phases are represented by "permanent" and "temporary" branches in the **git** repository, which facilitate the development process through to **release**. The main reason for the existence of persistent branches is to provision multifaceted deployment, allowing anyone to view the status of an application at different stages of it's development. These **2** types of branches contain the following **git** branches, which directly form part for the work flow.

Permanent Branches

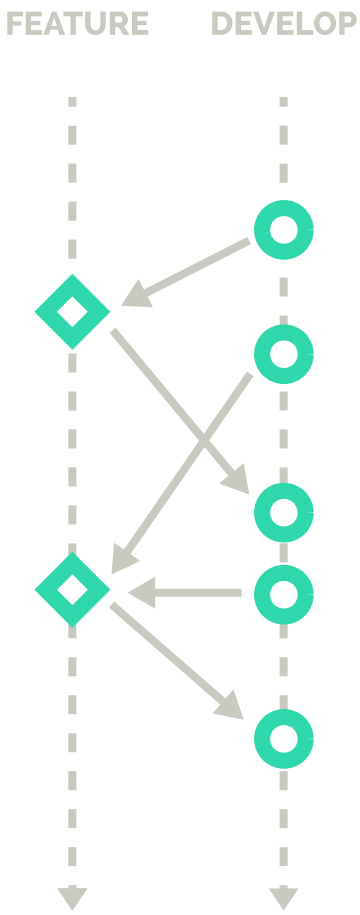
- **develop** Also referred to as bleeding edge, this branch contains the completed **features** for the current **milestone**. This is an alpha code base, and considered very unstable.
- **qa** All active development for a **milestone** is eventually tested on this branch. This is a beta code base, and also considered unstable.
- **stage** Once development for a **milestone** has been tested and passed by QA this branch then hosts the now stable c ode base for review.
- **master:** After approval has been given by the client or reviewer the staged code base is merged to master, which holds the current version of the project in the production environment.

Temporary Branches

- **feature:** These branches are created from the **develop** branch for the isolated development of a particular task. The definition of **feature:** is largely dependent on the management style applied to the project, and whether the **feature:** are in long or short sprints.
- **issue:** These branches are created from the **qa** branch to fix an issue reported by QA while testing a completed **milestone**.
- **hot-fix:** These branches are created to attend to serious and urgent problems detected in the production environment. At best, these branches should never need to be created if the QA process is efficient enough to pass a stable code base to stage for review.

An important difference between "permanent" and "temporary" branches is that no changes can be made directly to a "permanent" branch, they may only be inherited via the merge of a "temporary" branch. The complete process is represented in the diagram below:

FEATURE
DEVELOP
ISSUE
QA
STAGE
HOTFIX
MASTER
DEVELOPMENT
QA
RELEASE
TIMELINE
milestone/1
release/
PERMANENT BRANCH
TEMPORARY BRANCH
TAG



These branches are named "feature/", and followed by the identifier of the task. This would typically be the ID of the ticket in your project management system, for example:

```
$ git checkout -b feature/1234 develop
$ git push -u origin feature/1234
```

By separating the task off onto a separate branch developers avoid destabilizing the **develop** branch to the point where they may be interfering with the work done by others. Additionally, branches can be updated (rebase) with other **features** already committed to the **develop** branch, in case some {3} are based on or integrate with others.

By If you're working on the same **feature** with another developer you can checkout their branch and work on it in conjunction.

```
$ git checkout -t origin/feature/1234
```

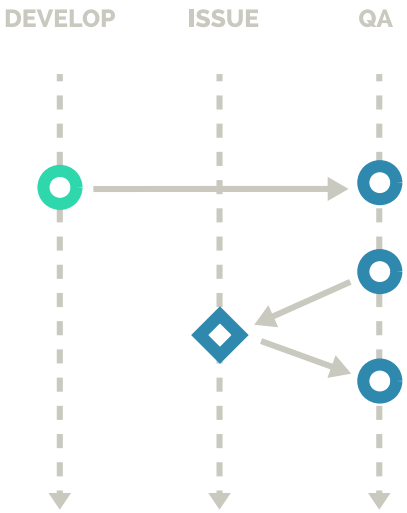
When the development of a **feature** is complete, it's merged back into the **develop** branch, and the **feature** branch is then deleted.

```
$ git checkout develop
$ git merge --no-ff feature/1234
$ git branch -d feature/1234
$ git push origin : feature/1234
```

While the {0} branch is considered unstable, it can be of great benefit to developers to have a server hosting this branch, so they can easily revise the bleeding edge version of the project, aiding in discussion or allowing review of the current progress. This also helps project managers unfamiliar with the development process to get an impression of the real status of the upcoming {1}.

Testing

When a **milestone** is considered complete, the **develop** branch is merged into the **qa** branch, and the QA process begins.



It's important to note that, when **develop** is merged into qa, any new **features** constitute the next **milestone**. This allows for testing of the current **milestone** as well as development of the next to run in parallel. Additionally, as the QA process is handled on a dedicated branch, this allows the testing phase to be scheduled without impeding the development to continue.

```
$ git checkout qa
$ git merge --no-ff develop
```

During the testing phase it's possible that QA may discover issues with the **milestone** development, and therefore fail some **features**. When this occurs, the modifications required to rectify these bugs are created as issue branches, based on the code base from the **qa** branch. These branches are named "issue/", and followed by the identifier of the task. This would typically be the ID of the ticket in your project management or bug tracking system, for example:

```
$ git checkout -b issue/1234 qa
$ git push -u origin issue/1234
```

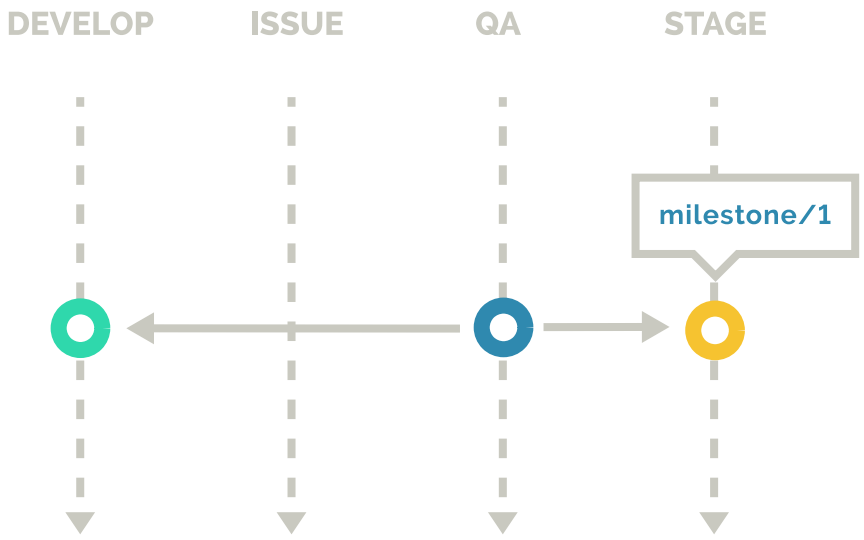
While active, issue branches can also be updated (rebase) with other issues committed to the **qa** branch, in case one issue depends upon the completion of another. When an issue is resolved, it's merged back into the **qa** branch, and the issue branch is then deleted.

```
$ git checkout qa
$ git merge --no-ff issue/1234
$ git branch -d issue/1234
$ git push origin :issue/1234
```

During the QA phase it's likely that a dedicated testing server would be provisioned. This is where the **qa** branch would be deployed for the QA team or member to process. The QA phase of the project is ambiguous as to the nature of the testing, as this process can vary significantly depending on the requirements and type of project at hand, so the factors which determine that a **milestone** is complete or the code base is stable may differ.

Review

Once a **milestone** has gone through active development, and passed the QA process, the new functionality is now ready to be added to the stage code base for review.



Here, the **qa** branch is merged into stage, and also merged back into **develop** for future **milestones**, for example:

```
$ git checkout develop
$ git merge --no-ff qa
$ git checkout stage
$ git merge --no-ff qa
```

Additionally, in order to mark the completion of the **milestone**, a tag is created from the stage branch. These tags are named "milestone/", and followed by the **milestone** identifier, which would normally be a sequential number. The tag may also be signed cryptographically with the -s or -u options.

```
$ git tag -a milestone/1
```

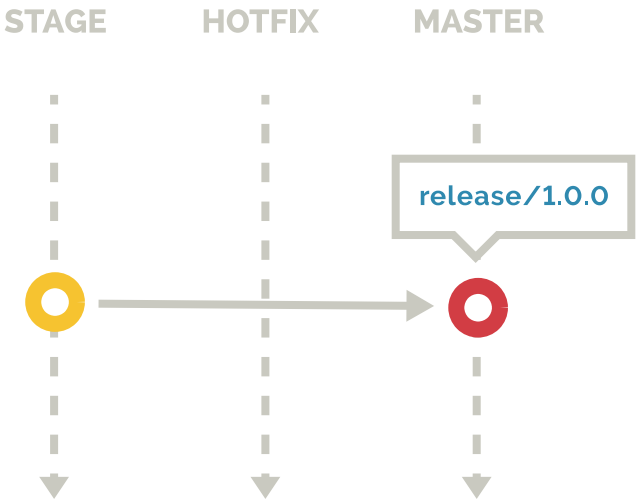
The stage branch may now be deployed to a staging server for the client or reviewer to approve. If any problems are detected or new functionality defined at this time, they

would become part of the currently active **milestone** on the **develop** branch, or scheduled for future development.

No changes may be made to the **qa** or stage branches directly, only through inheritance from a **feature** branch merged to develop, and then passing through the QA process again. It's extremely important to respect this process, however easy it may seem to just patch the stage branch, as it can compromise the quality assurance process.

Release

When the stage branch has been reviewed, after completing one or many **milestones**, a **release** can be created. This is when the code base in the production environment is updated to reflect a new version of the application.



To create the **release**, the stage branch is merged into master, for example.

```
$ git checkout master
$ git merge --no-ff stage
```

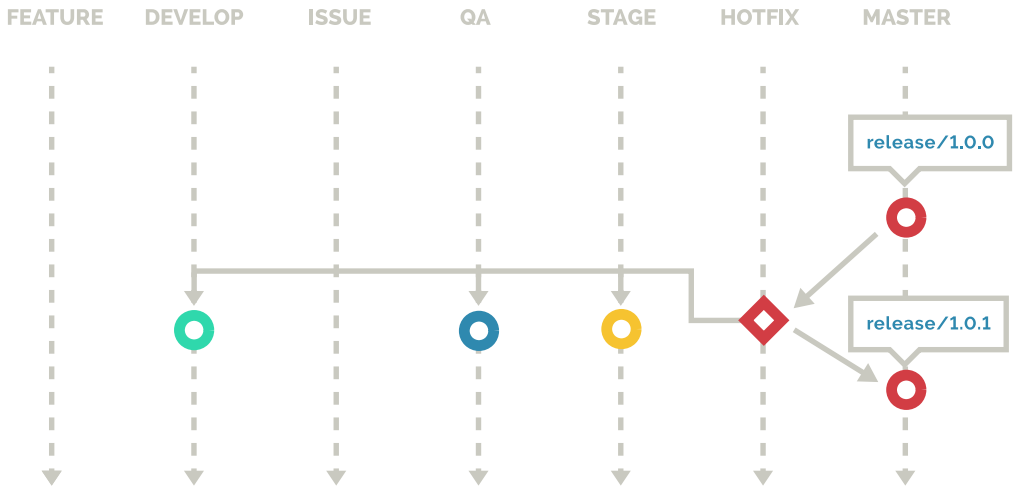
Additionally, in order to mark the creation of the **release**, a tag is created from the master branch. These tags are named "release/", and followed by the version number, which vary between projects depending upon the versioning strategy used. The tag may also be signed cryptographically with the -s or -u options.

```
$ git tag -a release/1.1.0
```

All code merged from the **qa** branch into stage after the new **release** will now constitute the next version of the application.

Hot Fixes

While the situation should rarely occur when effectively implementing this work flow, there are sometimes urgent bugs detected in the production environment which cannot wait until a pending **release** is created.



In these cases, a **hot-fix** branch is created, based on the code base from the master branch. These branches are named "hot-fix/", followed by the identifier of the task. This would typically be the ID of the ticket in your project management system, for example:

```
$ git checkout -b hot-fix/1234 master
$ git push -u origin hot-fix/1234
```

Depending on the requirements imposed by QA, it may be necessary to test the resulting patch in the **hot-fix** after the problem has been resolved. There are 3 ways to approach this issue:

- **Kamakazee:** Here the **hot-fix** is merged directly to master and QA is performed in the production environment. This is highly discouraged, as data inconsistency can result from the bug.
- **Copycat:** Here the **hot-fix** is merged directly to master and the master branch itself is staged on a dedicated server. This may not be possible if the production environment is deployed automatically, based on a pushes to the repository or a scheduled build.
- **Paranoid:** Here the **hot-fix** branch is staged on a dedicated server, for QA to review the patch before it's merged to master. The staged server may need to replicate the data used in the production environment, which may not be an option based on legal agreements or obligations, an alternative being to stage on the production server itself.

Once the patch is successful, the **hot-fix** branch must also be propagated and merged back into the stage, **qa** and **develop** branches.

```
$ git checkout master
$ git merge --no-ff hot-fix/1234
$ git checkout stage
$ git merge --no-ff hot-fix/1234
$ git checkout qa
$ git merge --no-ff hot-fix/1234
$ git checkout develop
$ git merge --no-ff hot-fix/1234
```

However, depending on the length of the **milestones**, it's possible that sufficient changes have been made in the pending **release** that the problem found in production has already been rectified, or the functionality surrounding the issue has been modified to a point where the problem no longer exists, or has been altered completely.

Finally, once the **hot-fix** has been applied to all the relevant branches it can now be removed.

```
$ git branch -d hot-fix/1234
$ git push origin :hot-fix/1234
```

If you find that there are numerous bugs in your production environment this can be attributed to insufficient details at the requirements stage of the project, or an inefficient QA process. Keep in mind that the QA process is only as good as the initial criteria, so validating the specification for a project is key to it's success.

It's also worth noting that any developer who creates a "temporary" branch should remain responsible for it, as they are the most likely candidates to know the status of the branch.