## Project Name: ReconAPI

## 1. Coding Structure:

The following packages will contain all the application sources.

**src/main/java**

All the files will be in java format in the following packages.

- com.dtcc.recon.controller
  This package will contain the Rest controllers i.e. the endpoints of the services and will interact with the business layer.
- com.dtcc.recon.bo
  Business Layer will have the business logic and will interact with the dao layer and return the response back to the controller.
- com.dtcc.recon.dao
  Package will have the classes which will interact with the database and return the response back to the business layer.
- com.dtcc.recon.entities
  This package will contain Java class mapped with database tables.
- com.dtcc.recon.vo
  This package will contain the values objects for data transfer between different layers.
- com.dtcc.recon.util
  This package will contain Utility classes.
- com.dtcc.recon.common
  This package will contain the common functionality classes/ constants classes.
- com.dtcc.recon.exception
  This package will contain the user defined exceptions.
- com.dtcc.recon.logger
  This package will contain Logging related classes for the application.
- com.dtcc.recon.security
  This package will contain classes related to User access and application security.

The following package will store the non-Java artifact files i.e. xml, properties files etc.

**src/main/resources**

The following package will contain all the Junit test classes.

**src/test/java**

The following package will contain all the test resources.

**src/test/resources**

## 2. Java Development Design and Coding Consideration point:

- Java app developers, please have some design session upfront to decide on the common modules, projects layout.
- Ensure to separate business logic into interface and implementation classes
- Provide Each testable java class with a junit or standalone main tester class that can be handed over to ETE functional
- Recommend to use Spring DB libraries for stored procedures and transaction management. (Sample project from Jie's past group to look at the examples of DAO classes etc. \\corp\DATA\ADM\!!! ADM\STL_RC_134318\DAO_SAMPLE_MODULES_SPRING)
- On Maven build step, the CICD document has an out of date keystore file. (Separate note will be shared to explain this). Else developers may get the Nexus certificate error for local maven build.
- Consider Restful API URL to add a parent level consists of consumer and RC sys ID (MRD-RCN) to ensure uniqueness across consumers that may be hosted on the same server.
- Include a validation module that validate the Request object for Restful API. Consider that each validation rule to be configurable so that it can be toggled on/off for consumer customization.
- Consider separate business logic behind each of the Restful API to separate module Logging – follow the standards listed here please

\\corp\DATA\ADM\!!!_ADM\STL_RC_134318\DEV\Logging\ECD Logger v1.1.4 - File Loggers.docx

## BitBucket Configurations

## 3. Bitbucket URL

GIT/Bitbucket : https://code.dtcc.com

## 4. Bitbucket Project & Repository Details:

Project Name: RCN_RECONCILIATION CAPABILITY

**Repositories Details:**

**A. JAVA Project:**

➔ GIT repo Name: **RCN-RECON-WEB-API**

➔ GIT Clone URL: https://code.dtcc.com:8443/scm/rcn/rcn-recon-web-api.git

**B. Database Files:**

➔ GIT repo Name: **RCN-RECON-WEB-API**

➔ GIT Clone URL: https://BMishra3@code.dtcc.com:8443/scm/rcn/rcn_oracle.git

- **ONETIME Folders: for DDL / DML scripts**
- **Root Folder: for Package / Stored Procedures / Function files**

## 5. JIRA URL

JIRA URL: https://jira.dtcc.com:8443/projects/RCN/issues/RCN-1?filter=allopenissues

5.1 Creating feature/bugfix branch from JIRA to link Jira story to branch.

- In any Jira story go to more option on top of story then click on create branch, you will be redirected to bitbucket.
- You can change the source of the branch by default it point to master.
- Make sure repository and branch name are the one you are expecting and click create branch.
- Then you can clone that branch to your local and create local repo to work on.

This way we can link our jira story and branch commits successfully.

# 6. Bitbucket Branching:

Below are the branching model to follow:

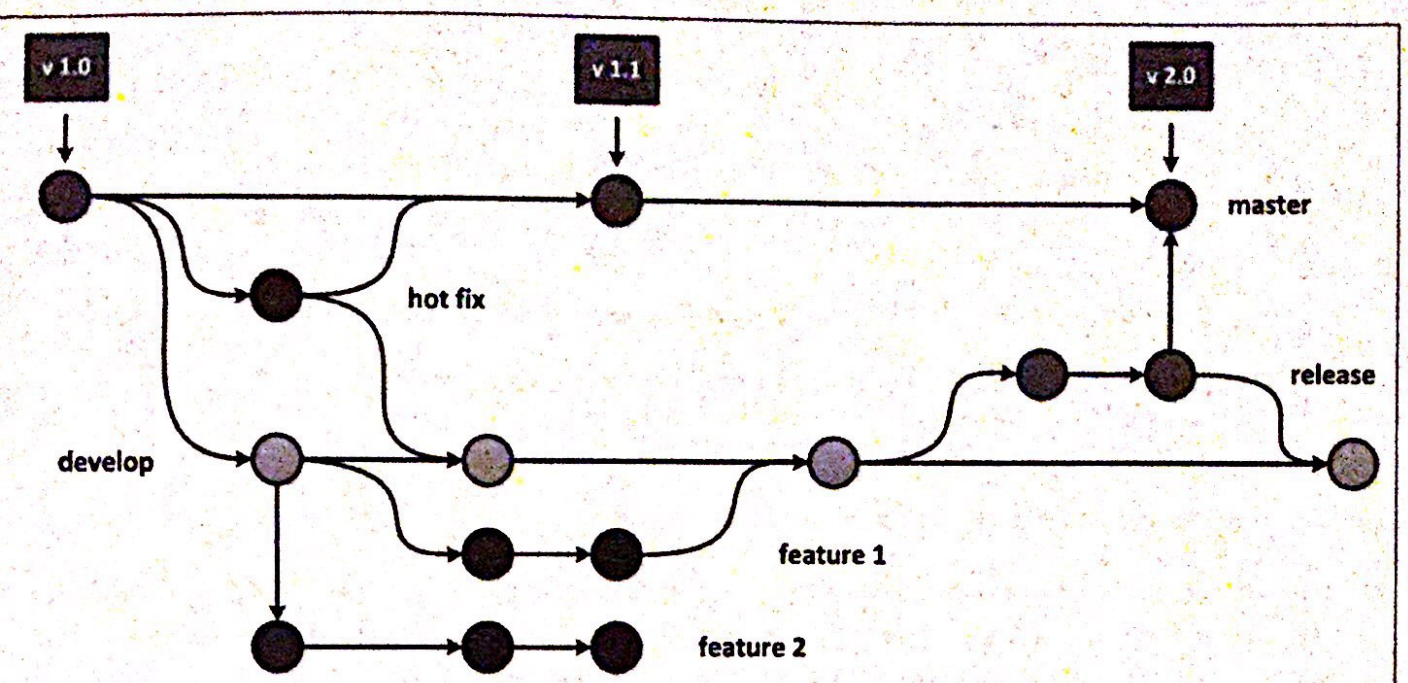
- **Develop branch** – all the developers will clone this branch for local development on their VPODs. All JAVA resources will use eclipse Mars version and will commit and pull the code on daily basis. This branch will be used to deploy the code in DEV environment for Unit/Integration testing.
- **Release/readytoQA** – This branch would be used to merge the code manually from develop to this branch for whatever code is ready to go to QA using beyond compare tool.
- **Release/v1.0** – this branch would be used to deploy the code in QA/PSE/PROD environment. Developer will raise a code merge request from GIT to merge the code from readyToQA to this branch. Approver will approve the request and will click on Merge button from GIT. Then GIT will automatically merge the code. Developer will submit a SBM form and code would be deployed from this branch. Another option for code auto-deployment is JENKIN. After setting the pipeline we can use Jenkin also.
- **Master** – This branch is used to map with PROD environment. After every PROD deployment developer will merge the code from release/v.0.1 or v1.1 or v1.2. ..... to this branch.

Steps to follow Bitbucket setup and integrate with Eclipse IDE: -

- Need to create the project on the Eclipse
- Need to clone the repo URL to the eclipse and then add the project to cloned path
- Finally push and commit the project to GIT, which will create **Master** branch automatically.
- After this we will create other branches like develop (for local development) and release/v1.0(for QA\PSE\PROD) or whatever branch is required as per project need
- Once this is done all the team members will clone the repos on the eclipse and good to go to commit/pull the code.

## 6.1. Understanding Gitflow Workflow Concepts

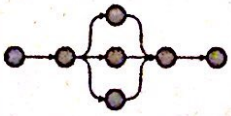
**Gitflow Workflow**




**Legend**

**Commit** (or "revision") – A change to a file (or set of files) that is stored in a Git repository. Each commit is assigned a unique identifier and marked with metadata including the person who made the change, the timestamp for when the change was made, and the previous revision(s) from which the change was based.

**Edge** – An edge is used by Git to track version history of files. An edge is used to connect one commit to the previous commit from which the change was based.



**Commit Graph** – A network of commits and edges that is used by Git to track version history and to otherwise manage changes.

**Branch** – A branch is a logical set of commits that is used to encapsulate a set of changes. Branches in this document are depicted by using edges to connect a series of commits in a horizontal line. The commit graph shown in the *Gitflow Workflow* above includes the following branches:

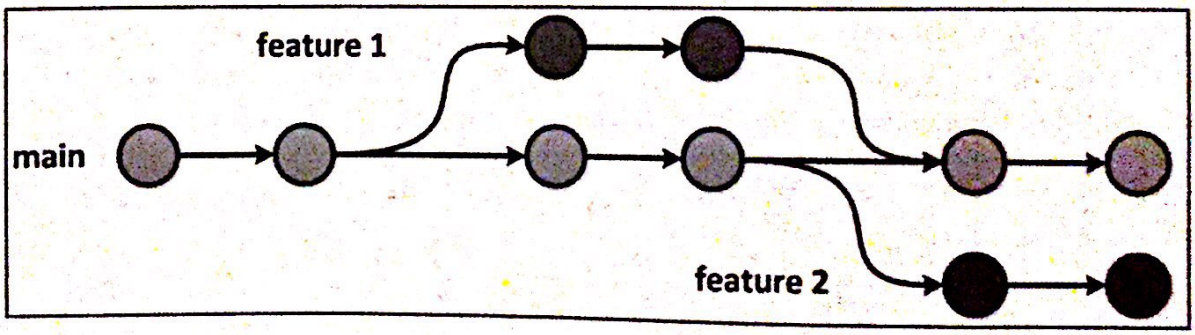
- master
- develop
- feature 1
- feature 2
- release
- bugfix
- hotfix

v 1.0

**Tag** – A reference or marker that is typically used to mark a particular point in the ***commit graph***. The tag can later be used to reference the specific commit that it points to.

## 6.2. Feature Branching

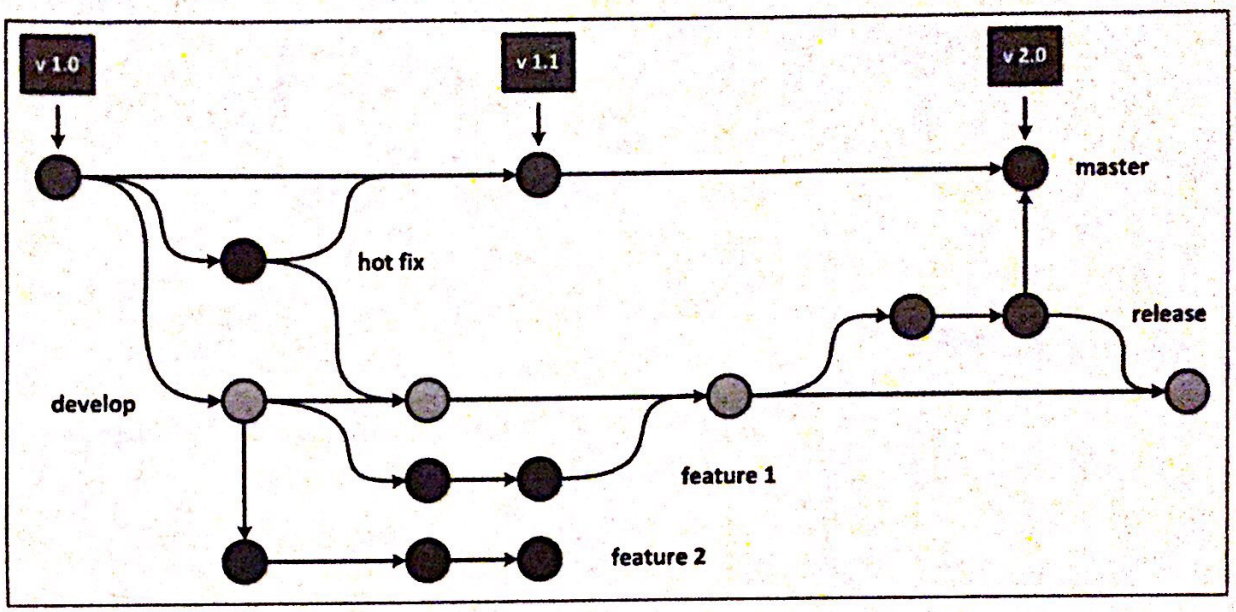
The following diagram illustrates an example of feature branching:




The diagram shows three branches: ***main***, ***feature 1*** and ***feature 2***. The general idea is that all features should be derived from the main branch, developed on an independent branch, and merged into the main branch after completion. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. By dedicating an isolated branch to each feature, it's also possible to initiate code reviews around changes before integrating them into the main

branch. Note that each feature branch typically represents a particular user story being developed, and that two or more developers may be working together on the same feature.

## Gitflow Workflow



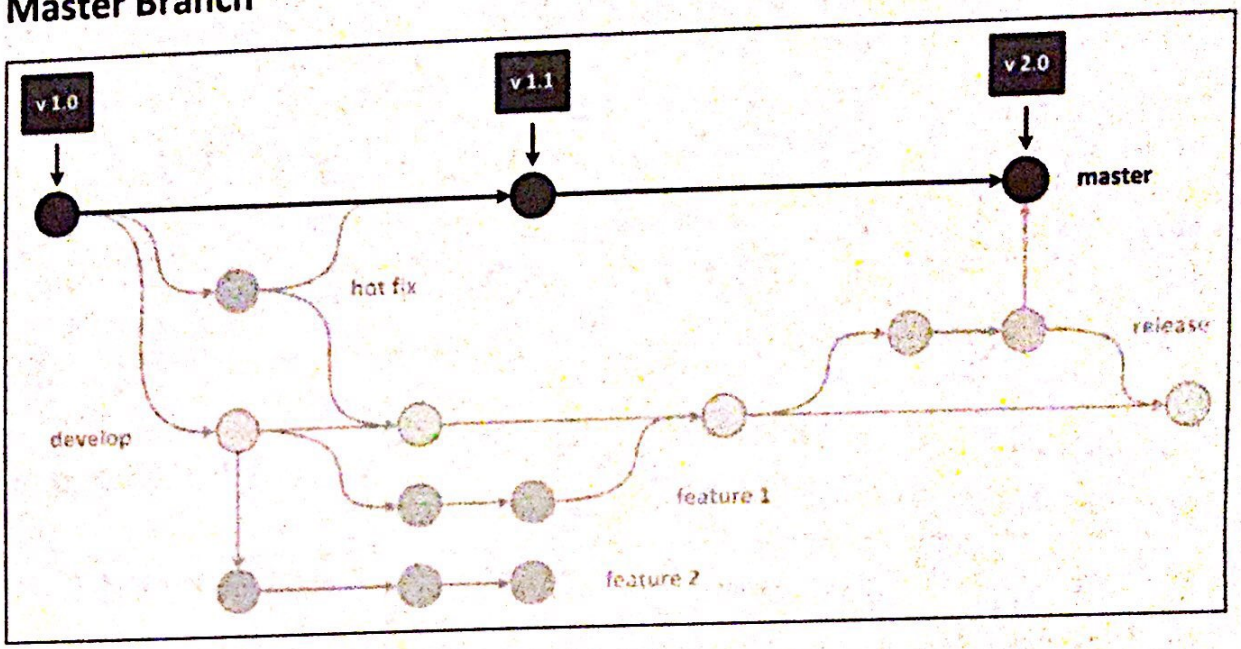

Gitflow Workflow utilizes feature branching. The workflow assigns very specific roles to different branches and defines how and when these branches should interact. It defines common patterns for managing feature development, release preparation, and maintenance. It defines a strict branching model designed around project release and is especially suited for large projects. It also works really well for small projects. Finally, it is ideal for a Continuous Integration environment where automated builds can be triggered by making commits to specific branches.

The different types of branches used by Gitflow are as follows:

- a **master** branch representing the code deployed for production use. The master branch always represents the latest version of the project code deployed to production. Support branches should be forked from master for previous production releases that need ongoing code maintainence (e.g., previous versions of library modules).
- a **develop** branch representing the most recent version of the project code under development.
- **multiple feature** branches
- **multiple release** branches
- **multiple hotfix** branches branching from the ***master*** branch
- **multiple bugfix** branches branching from a ***develop*** or ***release*** branch
- **multiple support** branches

Each type of branch has a specific purpose and is bound to strict rules as to where they originate as and to where they must be merged.

## Master Branch



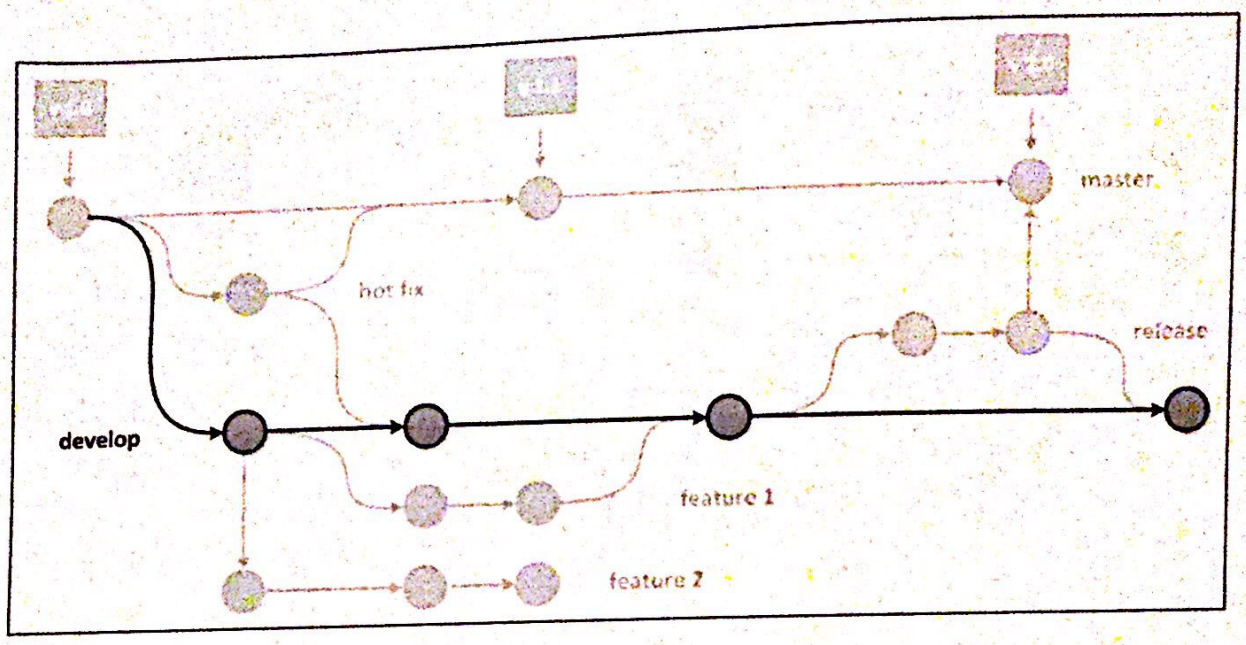

The following key points should be considered for the **master** branch.

- The ***master*** branch stores the official release history.
- All commits to the ***master*** branch should be tagged with a version number.
- ***feature*** branches should never interact directly with the ***master*** branch.

The following rules also apply:

- Branch Creation Rule:
  - The ***master*** branch should be created based on the most recent release of the application that exists at the time of creation.
- Branch Merging Rule:
  - All commits from the ***release*** branch must be merged into ***master*** or ***support*** branch after the release has been deployed successfully to production
  - All commits should be merged into the ***develop*** branch.
- Branch Naming Conventions:
  - The branch should be called ***master.***

## Develop Branch



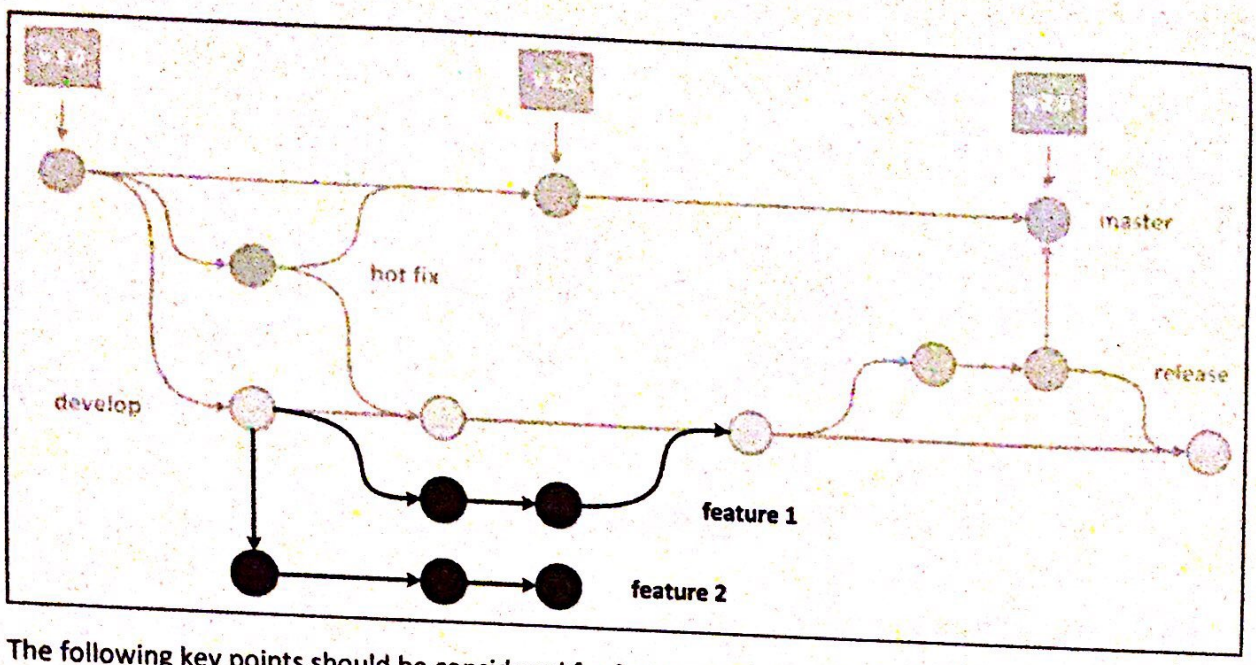

The following key points should be considered for the **develop** branch.

- The ***develop*** branch is considered to be the main line of development for the most recent version of the project.
- The ***develop*** branch serves as an integration branch for features.

The following rules also apply:

- Branch Creation Rule:
  - The ***develop*** branch should originate from current release at the time of initiation or created from scratch if there are no existing releases.
- Branch Merging Rule:
  - As this is the main line of development, no commits are merged into any other branches from this branch.
- Branch Naming Conventions:
  - The branch should be called ***develop.***

## Feature Branches



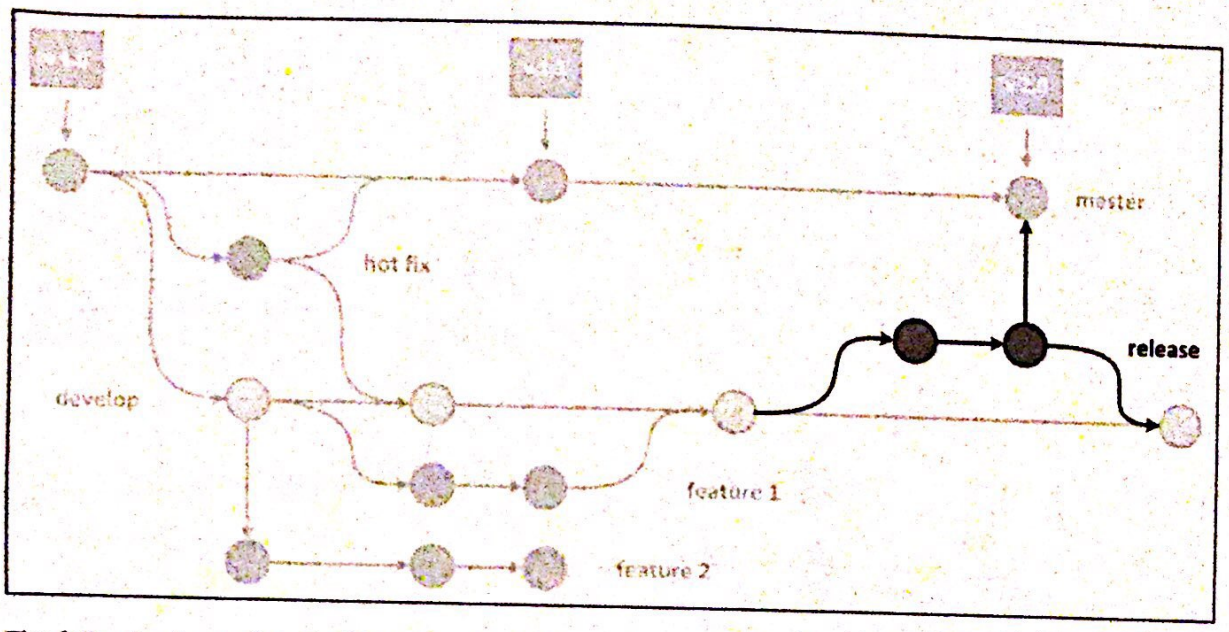

The following key points should be considered for **feature** branches.

- Each feature (typically a single user story) should be developed on its own branch.
- ***feature*** branches should never interact directly with the ***master*** branch.
- ***feature*** branches, like all other branches, should be pushed to the central repository for backup and collaboration purposes, as necessary.

The following rules also apply:

- Branch Creation Rule:
  - ***feature*** branches should be created by branching off the ***develop*** branch.
- Branch Merging Rule:
  - All completed features should be merged into the ***develop*** branchon completion.
- Branch Naming Conventions:
  - ***feature*** branches should be named anything except **master, develop, release-*,** or **hotfix-*.**
  - ***feature*** branches should have descriptive names; for example, ***automate-order-resubmission.***
  - If you are using an issue tracking system (i.e. Jira), ***feature*** branches should include the issue number (for example: RCN-86: Implement Send OPS Restful API SEND_OPS_COMPLETENESS)
  - It should use a prefix of feature/

## Release Branches



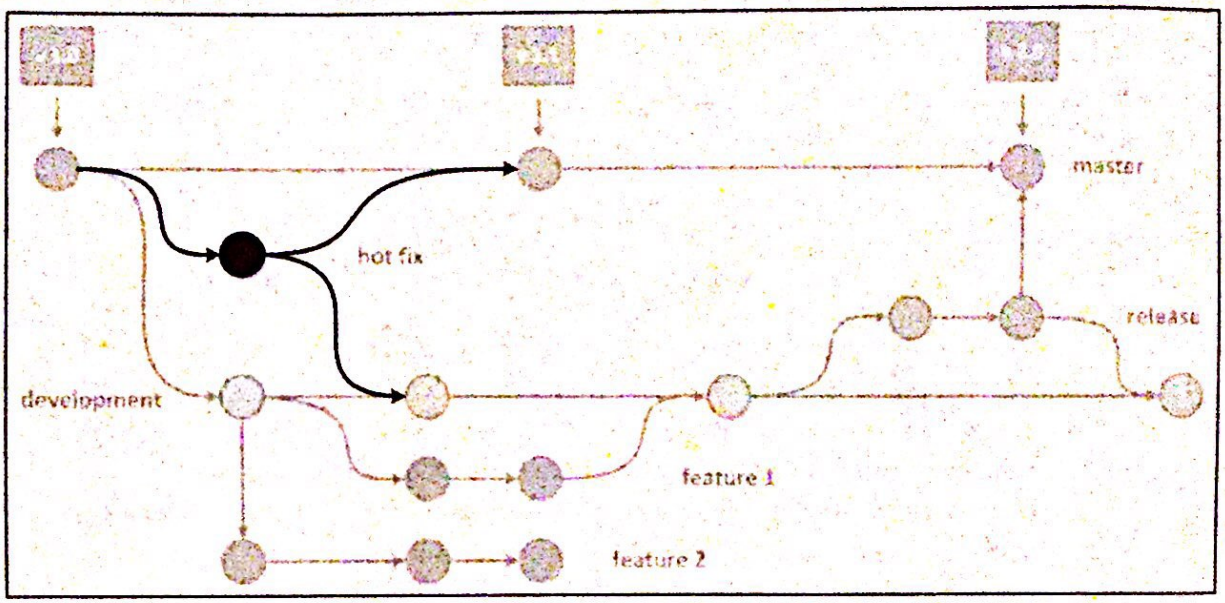

The following key points should be considered for **release** branches.

- ***release*** branches can be considered as a buffer between the ***develop*** and ***master*** branches.
- ***release*** branches are created for the testing of and to otherwise solidify and polish an upcoming release.
- ***release*** branches are "feature-frozen" and no additional features should be added to the release after creation.

The following rules also apply:

- Branch Creation Rule:
  - ***release*** branches should be created by branching off the ***develop*** branch.
- Branch Merging Rule:
  - All completed releases should be merged into the ***master*** branch.
  - All completed releases should also be merged into the ***develop*** branch.
  - Once a ***release*** branch is merged into the ***master*** branch, the commit on the ***master*** branch should be tagged with a release version.
- Branch Naming Conventions:
  - ***release*** branches should be named using the following pattern: ***release/****, where "*" represents the "major.minor" project version

## Hotfix Branches




The following key points should be considered for **hotfix** branches.

- ***hotfix*** branches are maintenance branches used to implement emergency changes or to quickly fix bugs that may be found in a production release.
- ***hotfix*** branches are the only branches that should be created by branching directly off the ***master*** branch
    - The **develop** branch only does this on initial creation.

The following rules also apply:

- Branch Creation Rule:
    - ***hotfix*** branches should be created by branching off the ***master*** branch.
- Branch Merging Rule:
    - ***hotfix*** branches should be merged into the ***master*** branch.
    - ***hotfix*** branches should also be merged into the ***develop*** branch.
    - ***hotfix*** branches should also be merged into any existing ***release*** branches.
    - Once a ***hotfix*** branch is merged into the ***master*** branch, the commit on the ***master*** branch should be tagged with a release version.
- Branch Naming Conventions:
    - ***hotfix*** branches should be named using the following pattern: ***hotfix-****, where "*" represents the "major.minor.patch" project version

### Bugfix Branches

The following key points should be considered for **bugfix** branches.

- ***bugfix*** branches are maintenance branches used to fix bugs that may be found in the development or release process.

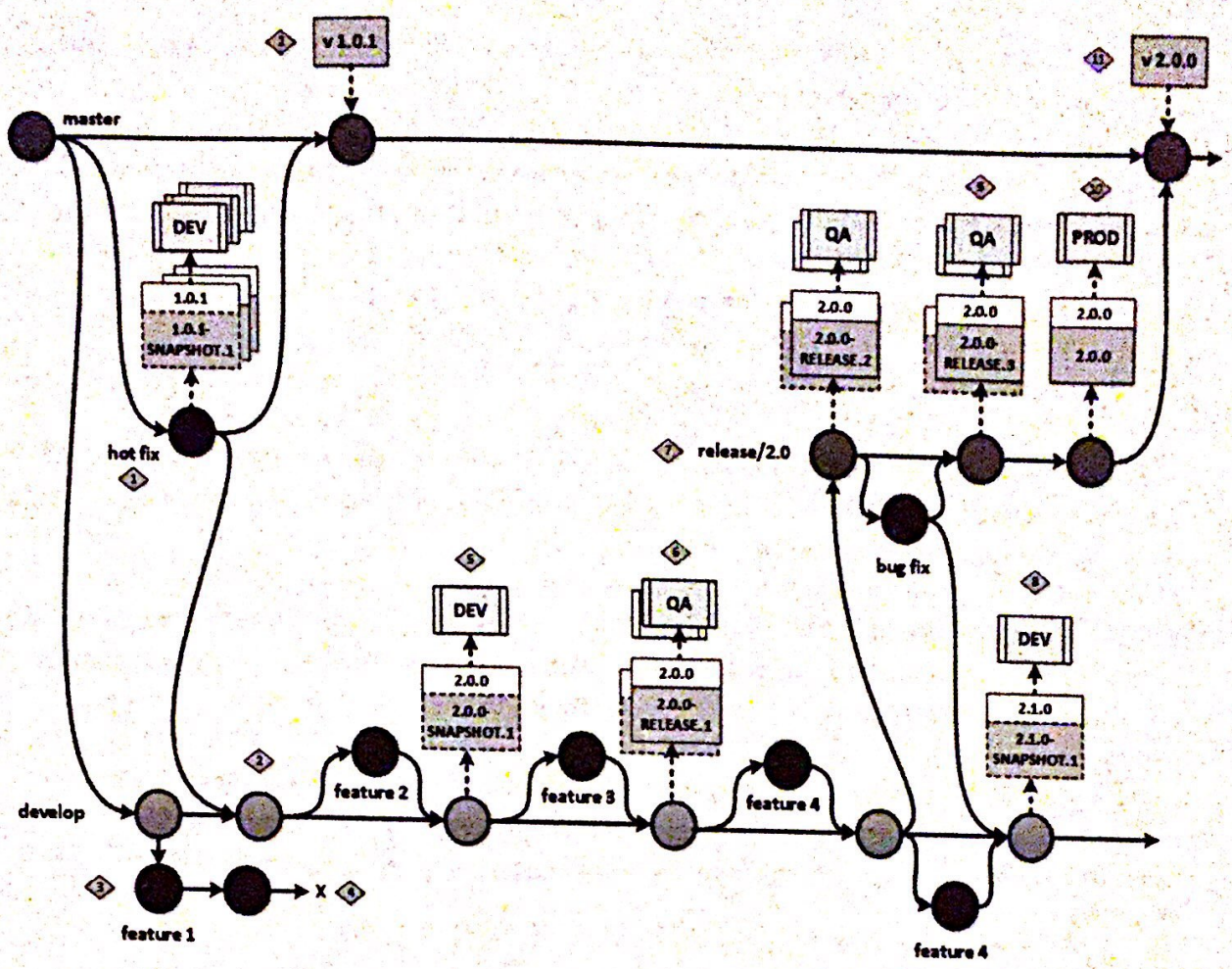
The following rules also apply:

- Branch Creation Rule:
  - ***bugfix*** branches should be created by branching off the ***develop*** or ***release*** branch.
- Branch Merging Rule:
  - ***bugfix*** branches should be merged into the ***develop*** branch.
  - ***bugfix*** branches should also be merged into any existing ***release*** branches.
  - Once a ***bugfix*** branch is merged into the ***develop*** branch, the commit on the ***develop*** branch should be tagged with an updated version number.
- Branch Naming Conventions:
  - ***bugfix*** branches should be named using the following pattern: ***bugfix -****, where "*" represents the "major.minor.patch" project versionMerging Changes into Branches

Whenever changes are to be merged into branches, an opportunity to perform a code review arises. Teams are encouraged to perform a code review at these points. The Bitbucket Server product helps to facilitate code reviews using a Pull Request.

## 6.3. Example Software Development Workflow

The following diagram illustrates a typical software development process. Key points in the process are indicated by a numbered diamond (e.g., ◇1). Following the diagram is commentary for each numbered point.




1. A bug has been identified in the current production code. To fix the bug developers create a ***hotfix*** branch from the ***master*** branch (example, hotfix/1.0.1). After fixing the bug the developers:
    a. update the maven artifact version in the POM file's "patch" number (1.0.1-SNAPSHOT) , tag the code with the "SNAPSHOT" suffix, and deploy (CI/CD) the code to Dev to verify the fix. After several iterations, the patch is deemed good.
    b. change the artifact version in the POM to replace the "-SNAPSHOT" suffix with a new GIT tag labeled (tag/v1.0.1-RELEASE-1). Promote to QA with this tag. This can go through several iterations based on QA testings. Each time it is deployed to QA, a GIT label is increased with a suffix (eg for the second iteration in QA, tag/v1.0.1-RELEASE-2

2. After the code is QA tested and promoted to production the developers merge the fixed code back into the ***master*** branch and tag it with the same version number as that in the POM file (tag/v1.0.1). Finally the ***hotfix*** branch is merged into the current ***develop*** branch and the hotfix branch is deleted.
3. Developers commence development of a new feature by creating a ***feature*** branch (feature/1) from the ***develop*** branch.

4. The feature does not work well and is rejected by the client. Further development of the feature is abandoned in favor of a different approach.
5. Additional feature(s) (feature/2) are created, unit tested, and integrated back into the ***develop*** branch. At this point the developers change the POM file minor number version to indicate that the code base now contains new features. The code is deployed to Dev for integration testing. Note that the minor number is incremented by one regardless of the number of new features added.
6. Feature development continues. At this point enough features have been developed such that EQA can start testing portions of the system while the other features are being developed. Note that this version has been tagged with a "RELEASE" suffix indicating the code is to be deployed to the QA environment.
7. At this point the code is feature-complete for the current release. Developers freeze the release by creating a ***release*** branch. While testing is performed on the ***release*** branch development continues on the ***develop*** branch for the next release.
8. A new feature has been completed for the next release. Developers update the minor release number within the code's POM file to indicate that the next release contains new features.
9. During testing EQA identifies a bug in the current release. Developers create a ***hotfix*** branch to fix the bug and then merge the fix into the ***release*** and ***develop*** branches. The release code is again deployed to QA and verified by EQA.
10. At this point EQA has verified that the current release is free from defects. Developers deploy the code to production and verify proper system operation. Once the system has been verified developers merge the ***release*** branch into ***master*** and tag it with the version number listed in the code's POM file.

## Coding Best Practices:

### 7. Java Coding Best Practices

a. Class names should start with an uppercase.
b. Only one class should be declared in one java file.
c. Ensure variable and method names meaningful. Variables and method names should follow camel case pattern.
d. Proper comments should be provided.
e. Import only necessary classes. Do not use wildcard imports (E.g. java.util.*) unless there are 4 or more classes from that package.
f. Use "Value".equals(s) instead of s.equals("Value") to handle the case of s== null.
g. Avoid resource leaks.
h. Use collection interface references instead of concrete type references. Also specify the type using < > (diamond operator).
i. Catch the most specific exception type.
j. Make error messages as useful as possible.

### 8. Junit Best Practices

a. Provide useful test names.
b. Each test should be independent of other test cases. You may not assume any test execution order.
c. Keep your tests small by limiting the number of failures a test reports. A test failure should indicate one particular problem. Consider refactoring long tests.
d. Properly test exceptions according to your JUnit framework best practices.
e. Don't just test the happy path. Include boundary conditions, etc.
f. JUnit tests should print nothing!