The main branches

In simple terms, the development model described here follows a structure with two main branches: "master" and "develop."

The "master" branch is considered the main branch where the source code always represents a stable and production-ready state. It's the version of the code that is ready for use by customers or users.

On the other hand, the "develop" branch is also a main branch but serves as an integration branch. The source code in this branch reflects the latest changes and developments that are being worked on for the next release. Nightly builds (automatic builds that happen every night) are typically created from this branch to test and validate the changes made during development.

When the source code in the "develop" branch reaches a stable point and is ready for release, all the changes made in that branch are merged back into the "master" branch. This merging process combines the new developments with the stable code in "master." The merged code is then tagged with a release number, indicating that it is a new production release.

The idea behind this approach is to ensure that the "master" branch always contains reliable and working code, while the "develop" branch allows for continuous development and integration of new features. By following this model, it becomes possible to automatically build and deploy the software to production servers whenever there is a commit (change) in the "master" branch, ensuring a streamlined and reliable release process.

Supporting Branches

In addition to the main branches' "master" and "develop," this development model uses supporting branches to facilitate parallel development, track features, prepare for releases, and quickly address production issues. These supporting branches have a limited lifespan and are eventually removed.

There are three types of supporting branches:

1. Feature branches: These branches are created to work on specific features or enhancements. They allow team members to work on different features simultaneously without interfering with each other. Feature branches originate from the "develop" branch and are merged back into it once the feature is completed.
2. Release branches: Release branches are used to prepare for a new production release. They allow for finalizing and stabilizing the codebase before it goes live. Release branches originate from the "develop" branch and are merged into both "develop" and "master" branches. Once the release is complete, the branch is removed.
3. Hotfix branches: Hotfix branches are created to quickly address live production problems or critical issues that require immediate attention. They branch off from the "master" branch, allowing for a fix to be implemented separately from ongoing development. Once the hotfix is applied, it is merged back into both the "develop" and "master" branches.

It's important to note that these branches are not technically different from regular Git branches. They are categorized based on their purpose and how they are utilized in the development workflow.

Feature Branch

In simple terms, a feature branch is like a separate workspace where developers can work on specific features for future releases. When they start working on a new feature, they create a branch dedicated to that feature. This allows them to make

changes, write code, and test the new feature without affecting the main development branch.
The feature branch exists if the feature is being developed. During this time, the developers can experiment, make improvements, and collaborate on the new feature. They don't have to worry about interfering with other features or the stability of the main branch.
Once the feature is complete and ready to be included in an upcoming release, the changes from the feature branch are merged back into the main development branch. This ensures that the new feature becomes a part of the software for the upcoming release. It will be thoroughly tested and prepared for release alongside other features.
However, if the feature doesn't meet expectations or turns out to be unsuccessful, it can be discarded. In such cases, the feature branch is not merged into the main development branch. This allows developers to explore ideas and experiment without negatively impacting the stability of the software.
In summary, feature branches provide a dedicated space for developing new features. They allow developers to work on features independently, merge successful features into the main branch for future releases, and discard unsuccessful features without affecting the main development process.

When creating a feature branch, you start by branching off from the "develop" branch. This allows you to have a separate branch where you can work on your new feature without affecting the main development branch.
To create a feature branch using Git, you can use the following command:

```
$ git checkout -b myfeature develop
```

This command creates a new branch called "myfeature" and switches to it. The branch is based on the latest state of the "develop" branch.

Once you have finished developing your feature and it's ready to be added to the upcoming release, you can merge it back into the "develop" branch.

To merge your feature branch into the "develop" branch, follow these steps:

1. Switch to the "develop" branch:
   ```
   $ git checkout develop
   ```
2. Perform the merge, using the "--no-ff" flag to ensure a new commit object is always created:
   ```
   $ git merge --no-ff myfeature
   ```

This command incorporates the changes from your feature branch into the "develop" branch. It creates a new commit object that summarizes the changes made.

3. Delete the feature branch after it has been merged:
   ```
   $ git branch -d myfeature
   ```
   This command deletes the local feature branch since it's no longer needed.
4. Push the changes to the remote repository
   ```
   $ git push origin develop
   ```

This command pushes the changes, including the merged feature, to the remote repository.

The "--no-ff" flag used during the merge ensures that a new commit object is created, even if a fast-forward merge is possible. This preserves the history of the feature branch and groups together all the commits related to that feature. This approach allows for easier management, reverting, and understanding of the changes made for that feature.

Although using "--no-ff" creates additional commit objects, the benefits of preserving the feature history outweigh the minor increase in commit count.

Release Branches

Release branches are used to prepare for a new production release. They serve as a dedicated space for finalizing details, fixing minor bugs, and preparing metadata for the release, such as version numbers and build dates. By using release branches, the main development branch (usually "develop") is freed up to receive new features for future releases.

The release branch is typically created when the main development branch (e.g., "develop") closely represents the desired state of the new release. All the features intended for the upcoming release should be merged into the main branch at this point. Features targeted for future releases should wait until after the release branch is created.

When the release branch is created, it marks the assignment of a version number to the upcoming release. Until this moment, the main branch represents changes for the "next release," but it is uncertain whether it will be version 0.3 or 1.0. The decision on the version number is determined when the release branch is started, following the project's rules for version number increments.

In summary, release branches provide a focused environment for finalizing a production release. They allow for last-minute adjustments, bug fixes, and versioning decisions before the release is made available to users. By separating this work from ongoing development, release branches ensure that the main development branch remains clear for future feature development.

To create a release branch, you start by branching off from the main development branch, typically "develop." Let's say the current production release is version 1.1.5, and you're preparing for a significant release. You decide that the next release will be version 1.2. Here's a generalized overview of the process:

1. Create and switch to the release branch:

```
$ git checkout -b release-1.2 develop
```

This command creates a new branch named "release-1.2" based on the latest state of the "develop" branch and switches to it.

2. Update the version number: Update the version number in the necessary files to reflect the new release. This can be done manually or using a script specific to your project.
3. Commit the version number change:

```
$ git commit -a -m "Bumped version number to 1.2"
```

The release branch will exist for a while as you work on finalizing the release. Bug fixes specific to this release can be applied to this branch, but adding significant new features is not allowed. New features should be merged into the main development branch ("develop") for the next major release.

When the release branch is ready to become a real release, you need to perform several actions:

1. Merge the release branch into the master branch:

```
$ git checkout master
$ git merge --no-ff release-1.2
```

2. Tag the release commit:

```
$ git tag -a 1.2
```

3. Merge the changes back into the develop branch:

```
$ git checkout develo
$ git merge --no-ff release-1.2
```

After completing these steps, the release process is done, and the release branch can be deleted since it's no longer needed:

```
$ git branch -d release-1.2
```

In a generalized form, the release branch allows for finalizing a production release, merging it into the master branch, tagging it for reference, and merging the changes back into the develop branch for future releases. The release branch serves as a temporary workspace for preparing the release and ensuring that important bug fixes are included.

Hotfix branches

Hotfix branches are used to address critical issues or bugs in a live production version that require immediate resolution. They are similar to release branches in their purpose of preparing for a new production release, but hotfix branches are created in response to unplanned situations.

When a critical bug is discovered in a production version that needs to be fixed urgently, a hotfix branch is created. The hotfix branch is typically based on the specific tag in the master branch that corresponds to the production version affected by the issue.

The key idea behind hotfix branches is to allow team members to continue their work on the main development branch (e.g., "develop"), while another person focuses on quickly resolving the critical issue in the hotfix branch.

Hotfix branches are created from the master branch when there is a critical bug or issue in the current production release that requires an immediate fix. The hotfix branch allows for the necessary changes to be made without disrupting the ongoing development work on the main branch (develop).

Here is a generalized explanation of the hotfix branch workflow:

1. Create a hotfix branch:

```
$ git checkout -b hotfix-<version> master
```

2. Make the necessary fixes: Fix the critical bug or issue in the hotfix branch. This may involve modifying code, applying patches, or making configuration changes specific to the problem.
3. Commit the fixes:

```
$ git commit -m "Fixed severe production problem"
```

   Commit the changes made in the hotfix branch to record the fixes.
4. Merge the hotfix into the master branch:

```
$ git checkout master
$ git merge --no-ff hotfix-<version>
```

Switch to the master branch and merge the hotfix branch into it. The "--no-ff" flag ensures a new commit object is created, even if a fast-forward merge is possible.

5. Tag the hotfix commit:

```
$ git tag -a <version>
```

Tag the hotfix commit for future reference. This helps to identify the version with the applied hotfix.

6. Merge the hotfix into the develop branch:

```
$ git checkout develop
$ git merge --no-ff hotfix-<version>
```

Switch to the develop branch and merge the hotfix branch into it. This ensures that the hotfix is also incorporated into future releases.

7. Remove the hotfix branch:

```
$ git branch -d hotfix-<version>
```

Delete the hotfix branch from the local repository once it has been merged and is no longer needed.

By following this workflow, critical issues in the production version can be addressed promptly without disrupting the main development branch. The hotfix is applied to both the master and develop branches to ensure that future releases incorporate the fix.