Search

Introduction to OpenShift Applications

Tell us what you think!

Congratulations on completing at least 25% of the course. We would appreciate feedback on your learning experience. Click below to fill out a brief survey.

TAKE ME TO THE SURVEY (HTTPS://SURVEY.OLE.REDHAT.COM/SURVEY?COURSE=DO101&MODALITY=COURSE&OFFERING=101)

NOT RIGHT NOW

Don't ask again

TRANSLATIONS

PREVIOUS (/ROL/APP/COURSES/DO101-4.2/PAGES/CH01S04) NEXT (/ROL/APP/COURSES/DO101-4.2/PAGES/CH01S06)

Managing Application Source Code with Git

Objectives

After completing this section, you should be able to use version control to collaborate and manage application source code.

Overview of Git Branching

Git version control features a branching model to track code changes. A **branch** is a named reference to a particular sequence of commits.

All Git repositories have a base branch named `master`. By default, when you create a commit in your repository, the `master` branch is updated with the new commit.

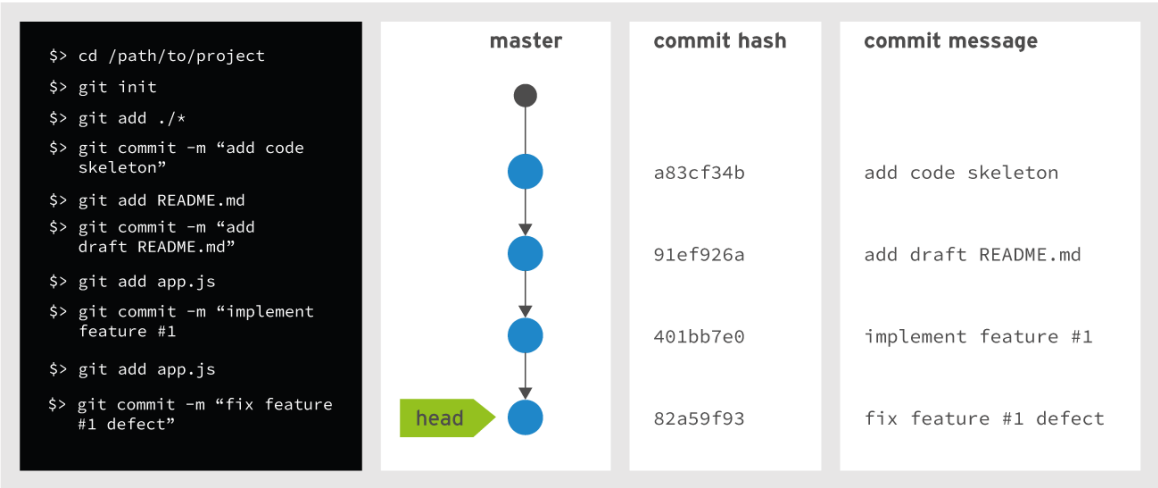

Figure 1.34: Commits to the master branch of a Git repository.

By convention, the `master` branch in a Git repository contains the latest, stable version of the application source code. To implement a new feature or functionality, create a new branch from the `master` branch. This new branch, called a **feature branch**, contains commits corresponding to code changes for the new feature. The `master` branch is not affected by commits to the feature branch.

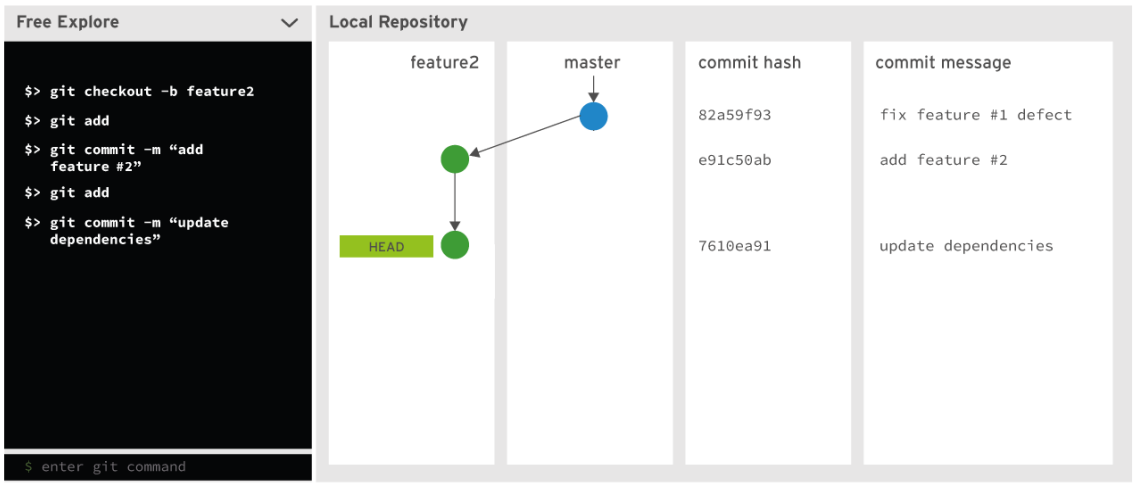

Figure 1.35: Commits to a feature branch of a Git repository.

When you use a branch for feature development, you can commit and share your code frequently without impacting the stability of code in the `master` branch. After ensuring the code in the feature branch is complete, tested, and reviewed, you are ready to merge the branch into another branch, such as the `master` branch. **Merging** is the process of combining the commit histories from two separate branches into a single branch.

Branching and Merging

This animation uses a feature branch workflow to demonstrate a sequence of code changes.

Merge Conflicts

Git has sophisticated mechanisms to merge code changes from one branch into another branch. However, if there are changes to the same file in both branches, then a **merge conflict** can occur.

A merge conflict indicates that Git is not able to automatically determine how to integrate changes from both branches. When this happens, Git labels each affected file as a conflict. VS Code displays an entry for each file with a merge conflict in the Source Control view, beneath the `MERGE CHANGES` heading. Each file with a merge conflict entry contains a `C` to the right of the entry.

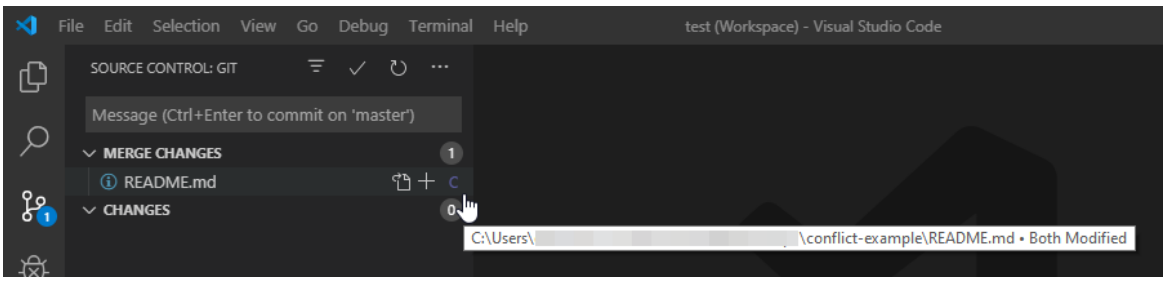

Figure 1.36: A merge conflict in the Source Control view of VS Code.

Git also inserts markers in each affected file to indicate the sections that contain content conflicts from both branches. If you click on the merge conflict entry in the VS Code Source Control view, then an editor tab displays and highlights the sections of the file that conflict.

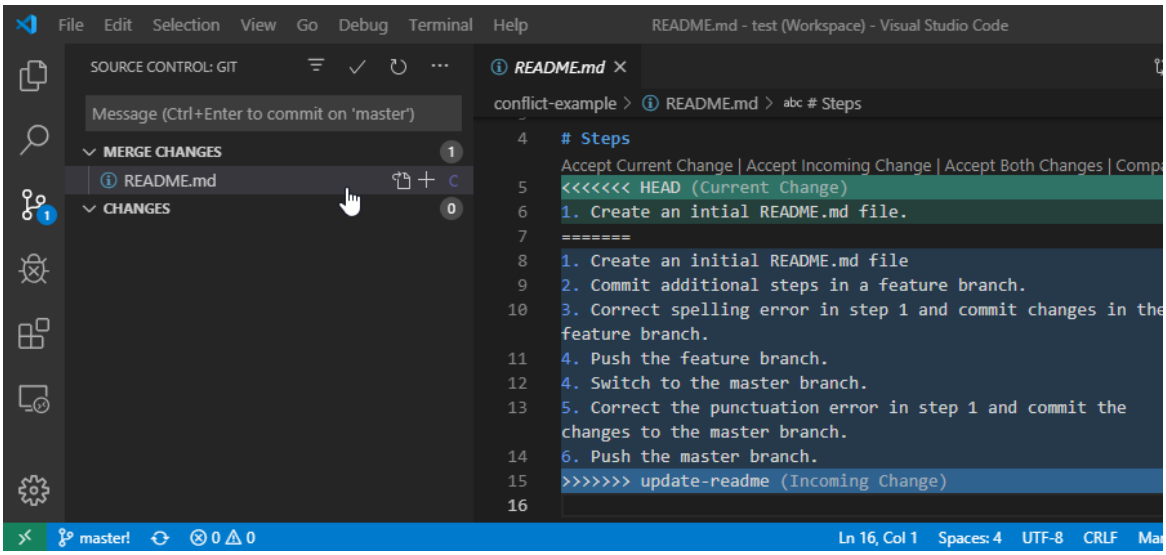

Figure 1.37: VS Code highlights the conflicts in a merge conflict file.

In the preceding figure, the green highlighted section contains content from the current branch, in this case the `master` branch. The blue highlighted text contains changes from the `update-readme` branch, which is being merged into the `master` branch.

There are many options to resolve merge conflicts that happen as a result of the normal merging process.

For each conflict in a conflicted file, replace all of the content between the merge conflict markers (<<<<<<<< and >>>>>>>> inclusive) with the correct content from one branch or the other, or an appropriate combination of content from both branches.

For example, the following figure shows that the conflict section from the preceding example is replaced with content from both the `master` and `update-readme` branches.

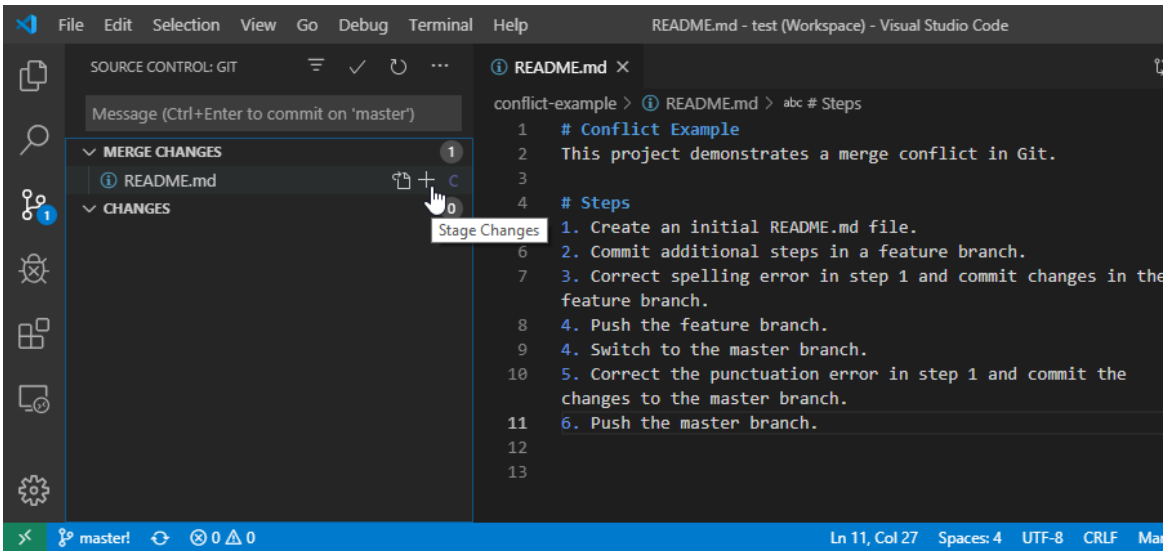

Figure 1.38: Resolving a merge conflict in VS Code.

After you reconcile the content for each conflict in a file, then save, stage, and commit the file changes.

> **NOTE**
>
> Managing merge conflicts is beyond the scope of this course. For more information on merge conflicts, see **Basic Merge Conflicts** at https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging (https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging)

Collaboration Strategies Using Git

Git provides development teams with the flexibility to implement code collaboration processes and branching strategies that fit the size and dynamics of each team. When a team agrees on a Git workflow, then code changes are managed consistently and team productivity increases.

Centralized Workflow

A centralized Git workflow uses a central Git repository as the single source of record for application code. The central repository is typically hosted on a repository hosting platform, such as GitHub. In this workflow, the development team agrees that the central repository will only contain a single branch, `master`. Developers push code changes directly to the `master` branch, and do not push commits in other branches to the central repository.

Developers clone the central repository, creating a local copy of the repository for each developer. Developers make code changes and commit to the master branch of their local repository. Developers merge in the latest changes from the remote repository, before pushing new changes to the remote repository.

Because the workflow results in commits to a single branch only, team members are prone to merge conflicts. Merge conflicts are mitigated when teams clearly identify separate code changes for each team member that can be implemented in distinct files.

In addition to merge conflicts, this workflow allows you to commit partial or incomplete code changes to the `master` branch. When incomplete code changes are committed, the stability of code in the `master` branch is compromised. Teams that use a centralized workflow must adopt additional processes and communication strategies to mitigate risks to the stability of project code on the `master` branch.

One way teams mitigate code stability issues with a centralized worfklow is to implement Git tags in the code repository. A **tag** is a human-readable reference to a particular commit in a Git repository, independent of any branch. Teams can implement a tag nomenclature to identify commits corresponding to stable versions of the project source code.

> **NOTE**
>
> Git tags are beyond the scope of this course. See **Tagging** in the Git documentation at https://git-scm.com/book/en/v2/Git-Basics-Tagging (https://git-scm.com/book/en/v2/Git-Basics-Tagging).

The centralized Git workflow works well for small teams collaborating on a small project with infrequent code changes, but becomes difficult to manage with larger teams and large code projects.

Feature Branch Workflow

A feature branch workflow implements safety mechanisms to protect the stability of code on the `master` branch. The aim of this workflow is to always have deployable and stable code for every commit on the `master` branch, but still allow team members to develop and contribute new features to the project.

In a feature branch workflow, each new feature is implemented in a dedicated branch. Multiple contributors collaborate on the feature by committing code to the feature branch. After a feature is complete, tested, and reviewed in the feature branch, then the feature is merged into the `master` branch.

The feature branch workflow is an extension of the centralized workflow. A central repository is the source of record for all project files, including feature branches.

When a developer is ready to merge a feature branch into the master branch, the developer pushes the local feature branch to the remote repository. Then, the developer submits a request to the team, such as a pull request or merge request, to have the code changes in the feature branch reviewed by a team member. After a team member approves the request, the repository hosting platform merges the feature branch into the master branch.

> **NOTE**
>
> A **pull request** is a feature of several repository hosting platforms, including GitHub and BitBucket.
>
> A pull request lets you submit code for inclusion to a project code base. Pull requests often provide a way for team members to provide comments, questions, and suggestions about submitted code changes. Pull requests also provide a mechanism to approve the merging of the code changes into another branch, such as `master`.
>
> On other platforms, such as GitLab and SourceForge, this code review feature is called a **merge request**.

For example, the following figure shows the GitHub user interface after pushing the `feature1` branch. GitHub displays a notification that you recently pushed the `feature1` branch. To submit a pull request for the `feature1` branch, click **Compare & pull request** in the notification.

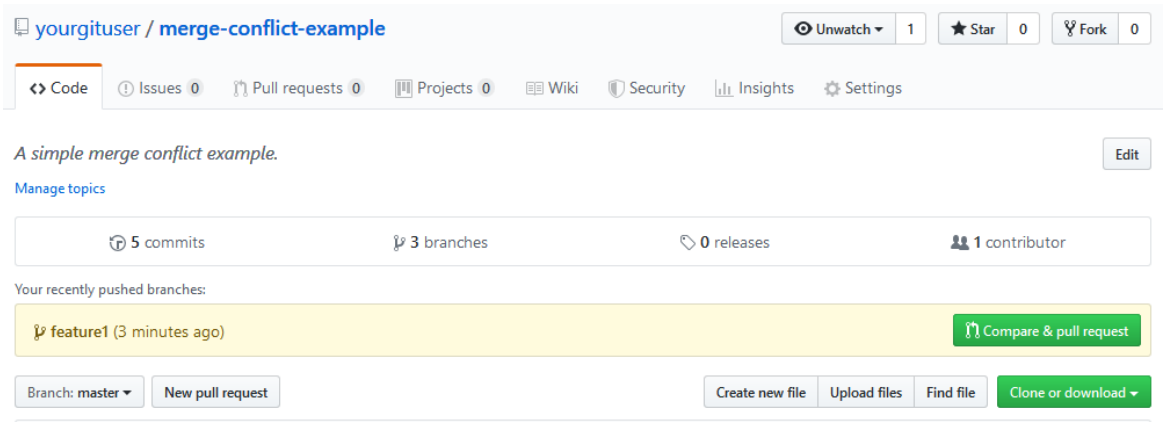

Figure 1.39: GitHub user inteface after you push a feature branch.

GitHub displays a form that you must complete to submit the pull request.

By default, you request that the `feature1` branch be merged (or pulled) into the `master` branch. If you want to merge the feature branch into a different branch, then you select the correct base branch from the list of options. GitHub displays a message to indicate if the merge operation will cause conflicts. In the figure that follows, GitHub indicates that the two branches will merge without any conflicts and can be automatically merged.

The GitHub pull request form allows you to provide a title and description for the pull request. The form also allows you to assign the pull request to a GitHub user, and also specify a set of GitHub users to review the code changes in the pull request.

After completing the form, click **Create pull request** to create the pull request.

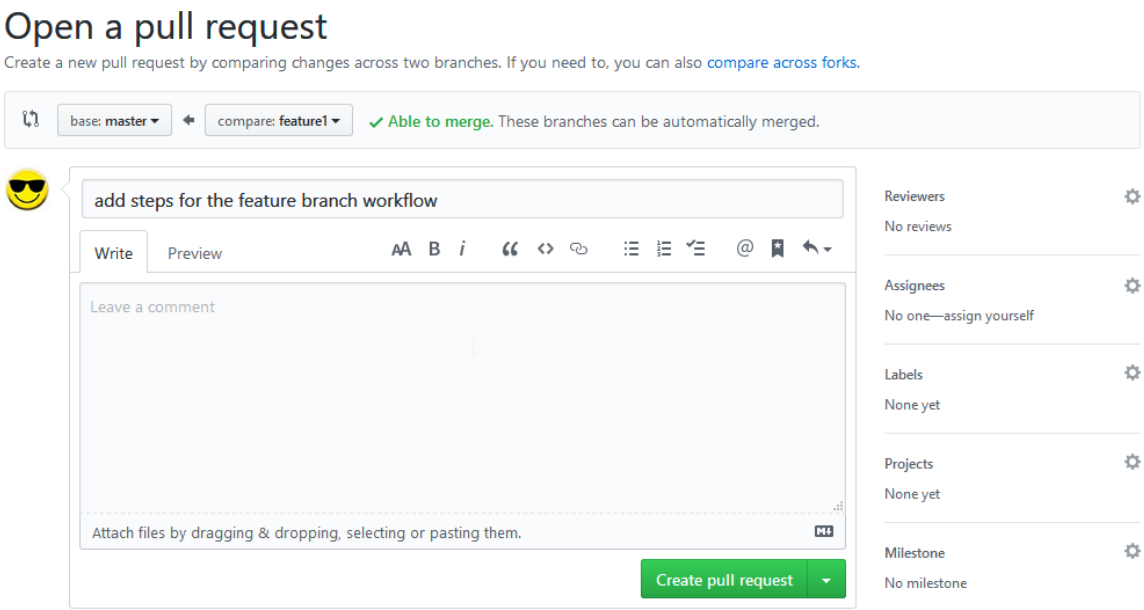

Figure 1.40: The GitHub form to submit a pull request.

After a reviewer grants approval, you can merge the pull request.

Click **Pull requests** for the GitHub repository to display a list of pull requests. Select your pull request from the list. GitHub displays information about the pull request, including comments, questions, or suggestions by other reviewers. When you are ready to merge the changes to the `master` branch, click **Merge pull request**.

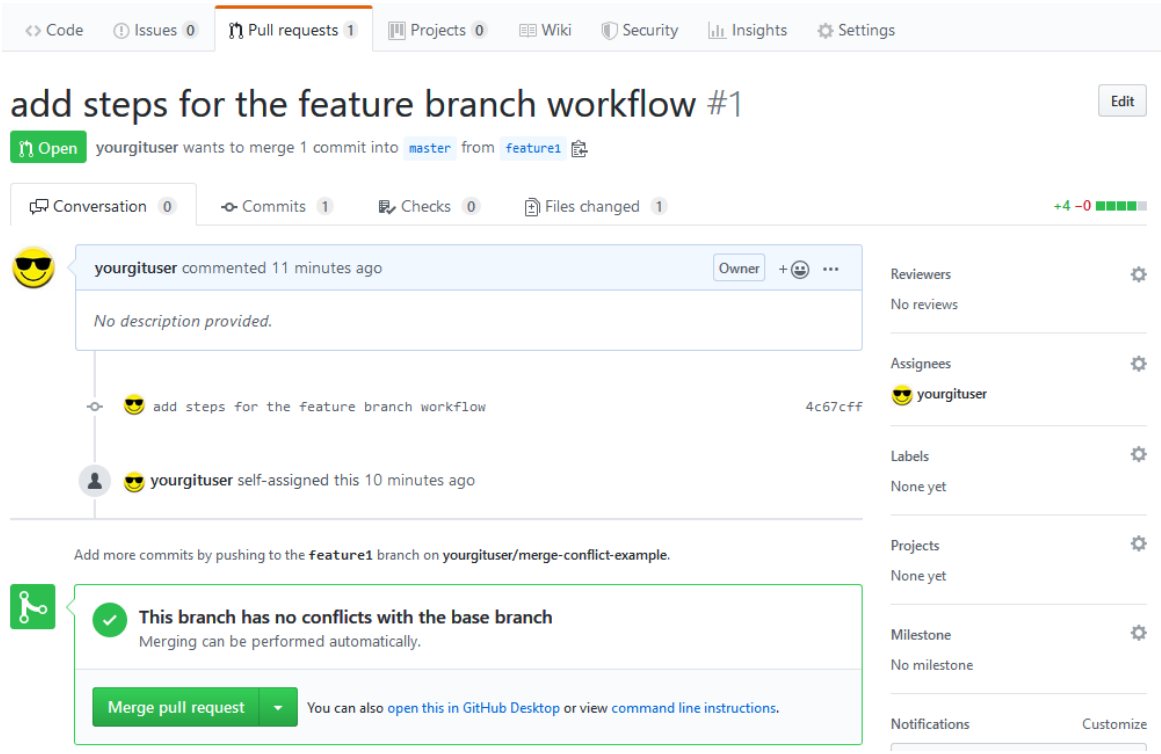

Figure 1.41: A GitHub pull request.

GitHub displays a message for the merge operation. Click **Confirm merge** to merge the `feature1` branch into the `master` branch. GitHub then displays a **Delete branch** button.

Finally, in the feature branch workflow you delete a feature branch after it is merged into the `master` branch. Click **Delete branch** to delete the branch.

Forked Repository Workflow

The Forked repository workflow is often used with large open source projects. With a large number of contributors, managing feature branches in a central repository is difficult. Additionally, the project owner may not want to allow contributors to create branches in the code repository. In this scenario, branch creation on the central repository is limited to a small number of team members. The forked repository workflow allows a large number of developers to contribute to the project while maintaining the stability of the project code.

In a forked repository workflow, you create a **fork** of the central repository, which becomes your personal copy of the repository on the same hosting platform.

After creating a fork of the repository, you clone the fork. Then, use the feature branch workflow to implement code changes and push a new feature branch to your fork of the official repository.

Next, open a pull request for the new feature branch in your fork. After a representative of the official repository approves the pull request, the feature branch from your fork is merged into the original repository.

For example, consider the workflow of the OpenShift Origin GitHub repository.

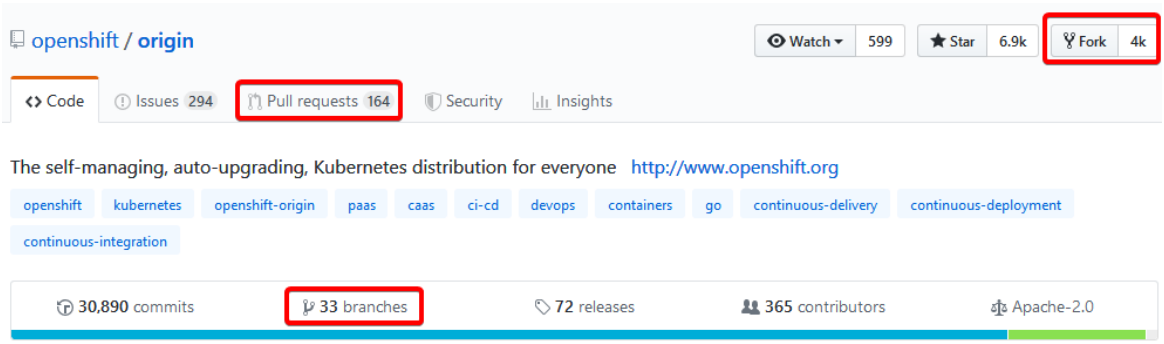

Figure 1.42: The OpenShift Origin GitHub repository.

There are more pull requests than branches in the OpenShift Origin GitHub repository. Many (if not all) of the pull requests contain code from one of the forks of the repository, instead of from a repository branch.

Git Branching in VS Code

In VS Code, use the Source Control view (**View** → **SCM**) to access the branching features in Git. The `SOURCE CONTROL PROVIDERS` section contains an entry for each Git repository in your VS Code workspace. Each entry displays the current branch for the repository.

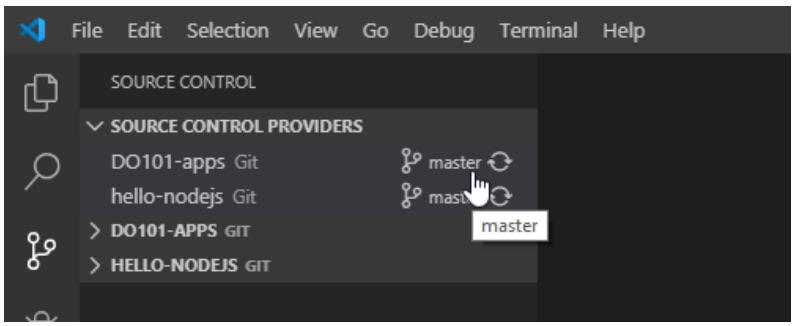

Figure 1.43: The Source Control view shows the current branch for each workspace Git repository.

To switch to a different branch in a repository, click the current branch name for the repository entry under the `SOURCE CONTROL PROVIDERS` heading. VS Code displays two options to create a new branch, and also displays a list of existing branches and tags in the repository:

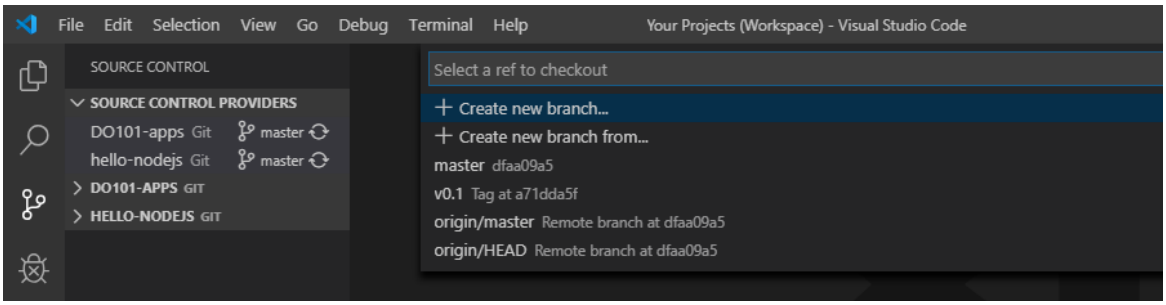

Figure 1.44: Checkout a branch in the Source Control view.

When you select either of the two options to create a new branch, VS Code prompts you for a name to assign to the new branch. If you selected the `Create new branch...` option, VS Code creates a new branch from the current repository branch.

If you selected the `Create a new branch from...` option, then VS Code also provides you list of existing repository tags and branches. After you select an item, VS Code creates a new branch that begins from the tag or branch that you select.

> **NOTE**
>
> Often software projects adopt branch naming conventions or standards. Branch naming standards help you summarize the code changes contained in a branch.
>
> The following are examples of branch name templates for a branch naming standard:
>
> - `feature/`*`feature-id/description`*
> - `hotfix/`*`issue-number/description`*
> - `release/`*`release-string`*
>
> A branch naming standard also defines the set of allowable characters. Branch names are often limited to alphanumeric characters and field seperators (such as `/`, `_`, or `-` characters).

After VS Code creates the new local branch, the Source Control view updates the repository entry with the name of the new branch. When you click the `Publish Changes` icon, VS Code publishes the new local branch to the remote repository.

Any new code you commit is added to the new local branch. When you are ready to push your local commits to the remote repository, click the `Synchronize Changes` icon in the Source Control view. The `Synchronize Changes` icon displays:

- the number of commits in the local repository to upload
- the number of commits in the remote repository to download

To download commits from a remote repository, VS Code first fetches the remote commits and merges them into your local repository. Then, VS Code pushes your local commits to the remote repository.

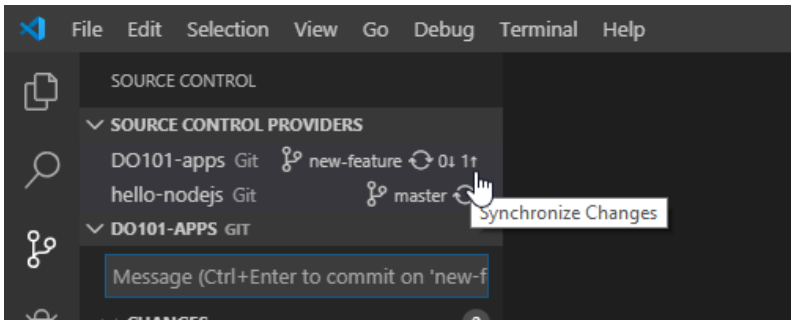

Figure 1.45: Push a local commit to the remote repository.

REFERENCES

For more information about Git branches, refer to the Git documentation at https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell (https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell)

For more information about merging, refer to the Git documentation at https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging (https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging)

For more information about using Git and version control in VS Code, refer to the VS Code documentation at https://code.visualstudio.com/docs/editor/versioncontrol (https://code.visualstudio.com/docs/editor/versioncontrol)

← PREVIOUS (/ROL/APP/COURSES/DO101-4.2/PAGES/CH01S04) → NEXT (/ROL/APP/COURSES/DO101-4.2/PAGES/CH01S06)