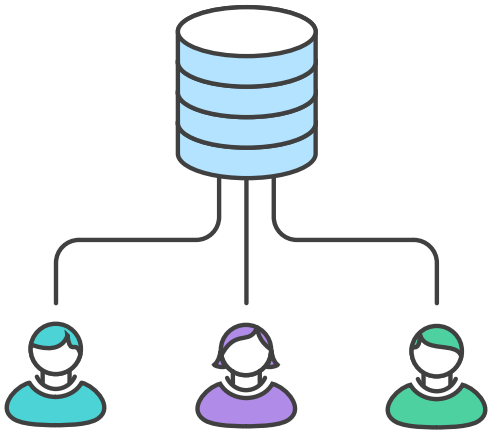
Comparación de flujos de trabajo

Centralizado de flujo de trabajo / de ramas de características de flujo de trabajo / Gitflow de flujo de trabajo / que bifurca de flujo de trabajo

The array of possible workflows can make it hard to know where to begin when implementing Git in the workplace. This page provides a starting point by surveying the most common Git workflows for enterprise teams.

As you read through, remember that these workflows are designed to be guidelines rather than concrete rules. We want to show you what's possible, so you

Centralized Workflow



Transitioning to a distributed version control system may seem like a daunting task, but *you don't have to change your existing workflow* to take advantage of Git. Your team can develop projects in the exact same way as they do with Subversion.

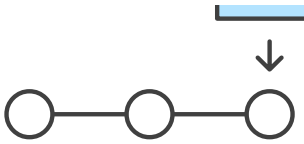
However, using Git to power your development workflow presents a few advantages over SVN. First, it gives every developer their own *local* copy of the entire project. This isolated environment lets each developer work independently of all other changes to a project—they can add commits to their local repository and completely forget about upstream developments until it's convenient for them.

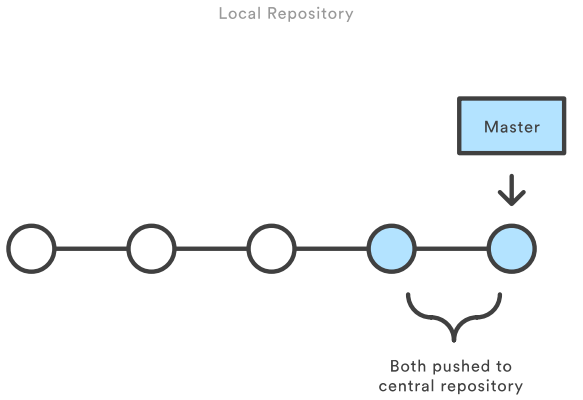
Second, it gives you access to Git's robust branching and merging model. Unlike SVN, Git branches are designed to be a fail-safe mechanism for integrating code and sharing changes between repositories.

Like Subversion, the Centralized Workflow uses a central repository to serve as the single point-of-entry for all changes to the project. Instead of `trunk`, the default development branch is called `master` and all changes are committed into this branch. This workflow doesn't require any other branches besides `master`.

Developers start by cloning the central repository. In their own local copies of the project, they edit files and commit changes as they would with SVN; however, these new commits are stored *locally*—they're completely isolated from the central repository. This lets developers defer synchronizing upstream until they're at a convenient break point.

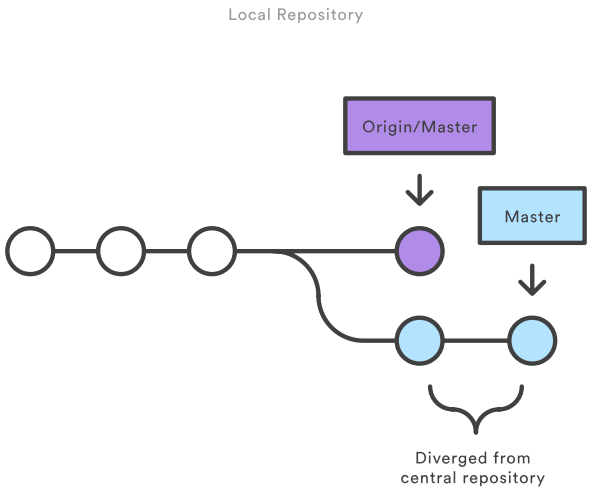
To publish changes to the official project, developers "push" their local `master` branch to the central repository. This is the equivalent of `svn commit`, except that it adds all of the local commits that aren't already in the central `master` branch.





Managing Conflicts

The central repository represents the official project, so its commit history should be treated as sacred and immutable. If a developer's local commits diverge from the central repository, Git will refuse to push their changes because this would overwrite official commits.



Before the developer can publish their feature, they need to fetch the updated central commits and rebase their changes on top of them. This is like saying, "I

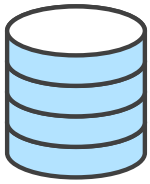
just like in traditional SVN workflows.

If local changes directly conflict with upstream commits, Git will pause the rebasing process and give you a chance to manually resolve the conflicts. The nice thing about Git is that it uses the same `git status` and `git add` commands for both generating commits and resolving merge conflicts. This makes it easy for new developers to manage their own merges. Plus, if they get themselves into trouble, Git makes it very easy to abort the entire rebase and try again (or go find help).

Example

Let's take a step-by-step look at how a typical small team would collaborate using this workflow. We'll see how two developers, John and Mary, can work on separate features and share their contributions via a centralized repository.

Someone initializes the central repository



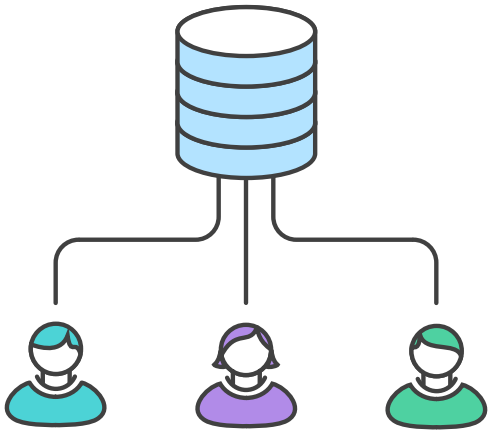
First, someone needs to create the central repository on a server. If it's a new project, you can initialize an empty repository. Otherwise, you'll need to import an existing Git or SVN repository.

which can be created as follows:

```
ssh user@host git init --bare /path/to/repo.git
```

Be sure to use a valid SSH username for `user`, the domain or IP address of your server for `host`, and the location where you'd like to store your repo for `/path/to/repo.git`. Note that the `.git` extension is conventionally appended to the repository name to indicate that it's a bare repository.

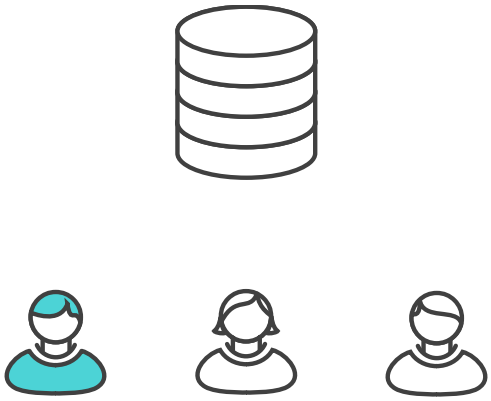
Everybody clones the central repository



Next, each developer creates a local copy of the entire project. This is accomplished via the `git clone` command:

```
git clone ssh://user@host/path/to/repo.git
```

When you clone a repository, Git automatically adds a shortcut called `origin` that points back to the "parent" repository, under the assumption that you'll want to interact with it further on down the road.



In his local repository, John can develop features using the standard Git commit process: edit, stage, and commit. If you're not familiar with the staging area, it's a way to prepare a commit without having to include every change in the working directory. This lets you create highly focused commits, even if you've made a lot of local changes.

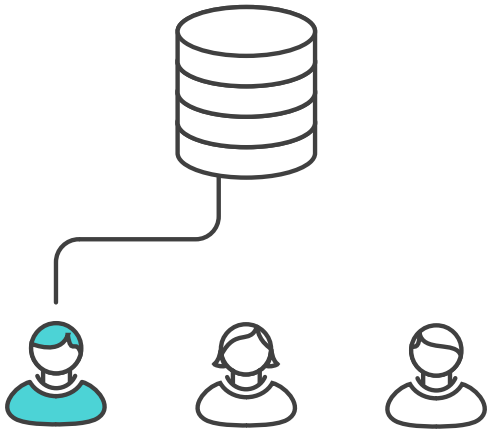
```
git status # View the state of the repo
git add <some-file> # Stage a file
git commit # Commit a file</some-file>
```

Remember that since these commands create local commits, John can repeat this process as many times as he wants without worrying about what's going on in the central repository. This can be very useful for large features that need to be broken down into simpler, more atomic chunks.

Mary works on her feature

Meanwhile, Mary is working on her own feature in her own local repository using the same edit/stage/commit process. Like John, she doesn't care what's going on in the central repository, and she *really* doesn't care what John is doing in his local repository, since all local repositories are *private*.

John publishes his feature



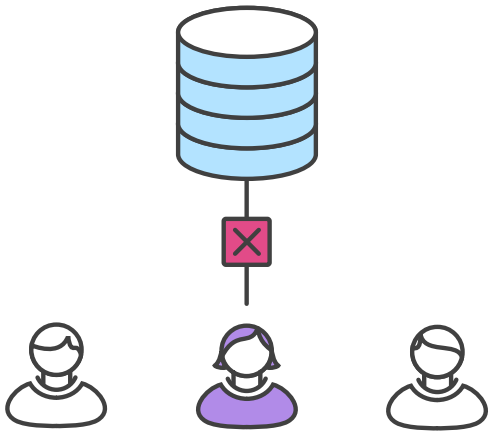
Once John finishes his feature, he should publish his local commits to the central repository so other team members can access it. He can do this with the `git push` command, like so:

```
git push origin master
```

Remember that `origin` is the remote connection to the central repository that Git created when John cloned it. The `master` argument tells Git to try to make the `origin`'s `master` branch look like his local `master`

conflicts and the push will work as expected.

Mary tries to publish her feature



Let's see what happens if Mary tries to push her feature after John has successfully published his changes to the central repository. She can use the exact same push command:

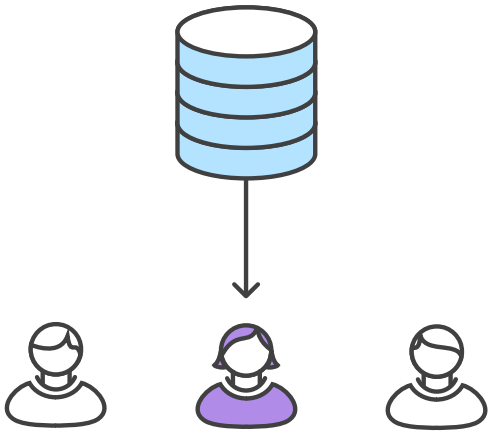
```
git push origin master
```

But, since her local history has diverged from the central repository, Git will refuse the request with a rather verbose error message:

```
error: failed to push some refs to '/path/to/rep
hint: Updates were rejected because the tip of y
hint: its remote counterpart. Merge the remote c
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git
```

This prevents Mary from overwriting official commits. She needs to pull John's updates into her repository, integrate them with her local changes, and then try again.

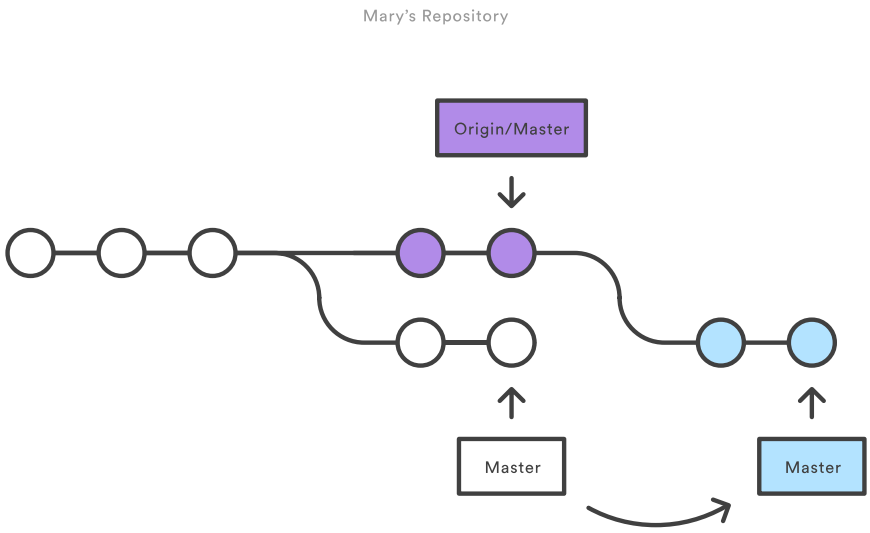
commit(s)



Mary can use `git pull` to incorporate upstream changes into her repository. This command is sort of like `svn update`—it pulls the entire upstream commit history into Mary's local repository and tries to integrate it with her local commits:

```
git pull --rebase origin master
```

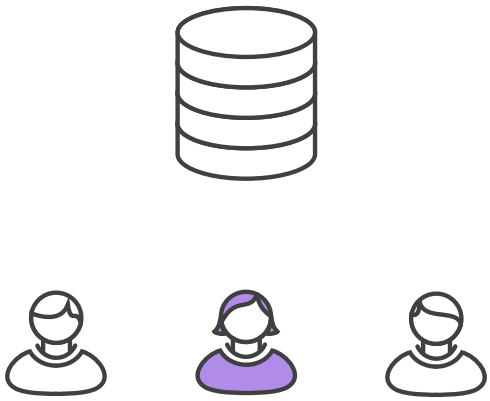
The `--rebase` option tells Git to move all of Mary's commits to the tip of the `master` branch after synchronising it with the changes from the central repository, as shown below:



The pull would still work if you forgot this option, but you would wind up with a superfluous "merge commit"

to rebase instead of generating a merge commit.

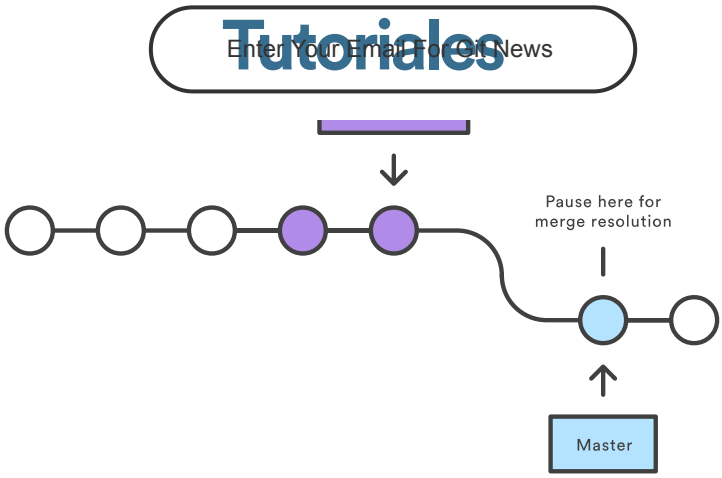
Mary resolves a merge conflict



Rebasing works by transferring each local commit to the updated `master` branch one at a time. This means that you catch merge conflicts on a commit-by-commit basis rather than resolving all of them in one massive merge commit. This keeps your commits as focused as possible and makes for a clean project history. In turn, this makes it much easier to figure out where bugs were introduced and, if necessary, to roll back changes with minimal impact on the project.

If Mary and John are working on unrelated features, it's unlikely that the rebasing process will generate conflicts. But if it does, Git will pause the rebase at the current commit and output the following message, along with some relevant instructions:

```
CONFLICT (content): Merge conflict in <some-file>
```



The great thing about Git is that *anyone* can resolve their own merge conflicts. In our example, Mary would simply run a `git status` to see where the problem is. Conflicted files will appear in the Unmerged paths section:

```
# Unmerged paths:
# (use "git reset HEAD <some-file>..." to unstage)
# (use "git add/rm <some-file>..." as appropriate
#
# both modified: <some-file>
```

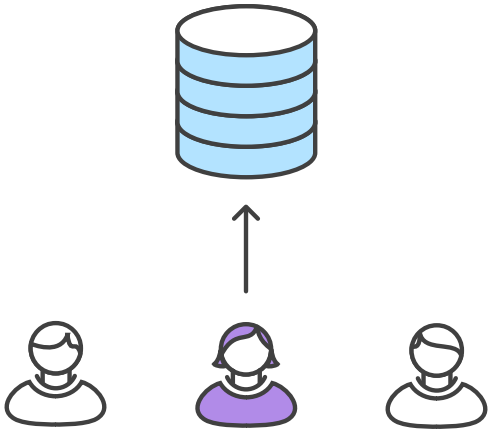
Then, she'll edit the file(s) to her liking. Once she's happy with the result, she can stage the file(s) in the usual fashion and let `git rebase` do the rest:

```
git add <some-file>
git rebase --continue
```

And that's all there is to it. Git will move on to the next commit and repeat the process for any other commits that generate conflicts.

If you get to this point and realize and you have no idea what's going on, don't panic. Just execute the following command and you'll be right back to where you started before you ran [`git pull --rebase`](/tutorials/syncing/git-pull):

Mary successfully publishes her feature



After she's done synchronizing with the central repository, Mary will be able to publish her changes successfully:

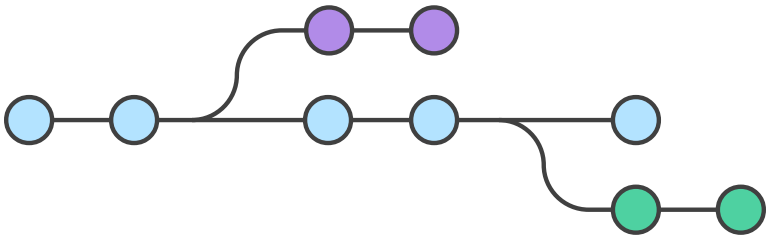
```
git push origin master
```

Where To Go From Here

As you can see, it's possible to replicate a traditional Subversion development environment using only a handful of Git commands. This is great for transitioning teams off of SVN, but it doesn't leverage the distributed nature of Git.

If your team is comfortable with the Centralized Workflow but wants to streamline its collaboration efforts, it's definitely worth exploring the benefits of the Feature Branch Workflow. By dedicating an isolated branch to each feature, it's possible to initiate in-depth

Feature Branch Workflow



Once you've got the hang of the Centralized Workflow, adding feature branches to your development process is an easy way to encourage collaboration and streamline communication between developers.

The core idea behind the Feature Branch Workflow is that all feature development should take place in a dedicated branch instead of the `master` branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the `master` branch will never contain broken code, which is a huge advantage for continuous integration environments.

Encapsulating feature development also makes it possible to leverage pull requests, which are a way to initiate discussions around a branch. They give other developers the opportunity to sign off on a feature before it gets integrated into the official project. Or, if you get stuck in the middle of a feature, you can open a pull request asking for suggestions from your colleagues. The point is, pull requests make it incredibly easy for your team to comment on each other's work.

The Feature Branch Workflow still uses a central repository, and `master` still represents the official project history. But, instead of committing directly on their local `master` branch, developers create a new branch every time they start work on a new feature. Feature branches should have descriptive names, like `animated-menu-items` or `issue-#1061`. The idea is to give a clear, highly-focused purpose to each branch.

Git makes no technical distinction between the `master` branch and feature branches, so developers can edit, stage, and commit changes to a feature branch just as they did in the Centralized Workflow.

In addition, feature branches can (and should) be pushed to the central repository. This makes it possible to share a feature with other developers without touching any official code. Since `master` is the only "special" branch, storing several feature branches on the central repository doesn't pose any problems. Of course, this is also a convenient way to back up everybody's local commits.

Pull Requests

Aside from isolating feature development, branches make it possible to discuss changes via pull requests. Once someone completes a feature, they don't immediately merge it into `master`. Instead, they push the feature branch to the central server and file a pull request asking to merge their additions into `master`. This gives other developers an opportunity to review

Code review is a major benefit of pull requests, but they're actually designed to be a generic way to talk about code. You can think of pull requests as a discussion dedicated to a particular branch. This means that they can also be used much earlier in the development process. For example, if a developer needs help with a particular feature, all they have to do is file a pull request. Interested parties will be notified automatically, and they'll be able to see the question right next to the relevant commits.

Once a pull request is accepted, the actual act of publishing a feature is much the same as in the Centralized Workflow. First, you need to make sure your local `master` is synchronized with the upstream `master`. Then, you merge the feature branch into `master` and push the updated `master` back to the central repository.

Pull requests can be facilitated by product repository management solutions like Bitbucket Cloud or Bitbucket Server. View the Bitbucket Server pull requests documentation for an example.

Example

The example included below demonstrates a pull request as a form of code review, but remember that they can serve many other purposes.

Mary begins a new feature

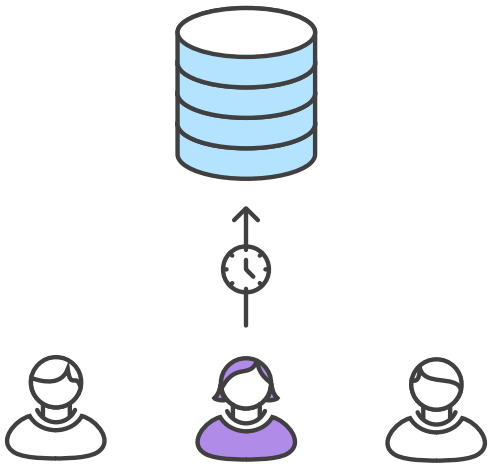
Before she starts developing a feature, Mary needs an isolated branch to work on. She can request a new branch with the following command:

```
git checkout -b marys-feature master
```

This checks out a branch called `marys-feature` based on `master`, and the `-b` flag tells Git to create the branch if it doesn't already exist. On this branch, Mary edits, stages, and commits changes in the usual fashion, building up her feature with as many commits as necessary:

```
git status
git add <some-file>
git commit
```

Mary goes to lunch

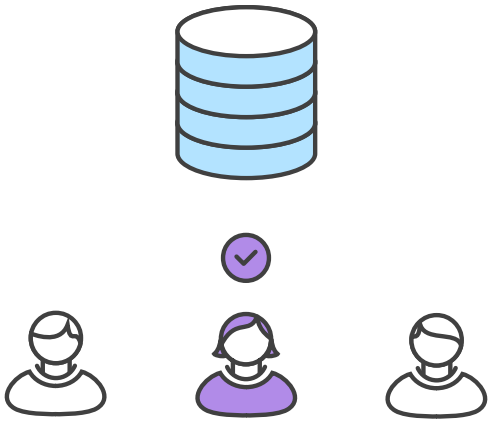


María añade algunos compromisos a su característica en el transcurso de la mañana. Antes de que ella salga para el almuerzo, es una buena idea empujar su rama característica hasta el repositorio central . Esto sirve como una copia de seguridad conveniente, pero si María estaba colaborando con otros

```
git push -u origin marys-feature
```

Este comando empuja `marys-feature` al repositorio central (`origin`), y el `-u`indicador lo añade como una rama de seguimiento remoto. Después de configurar la rama de seguimiento, Mary puede llamar `git push` sin ningún parámetro para empujar su función.

Mary termina su presentación



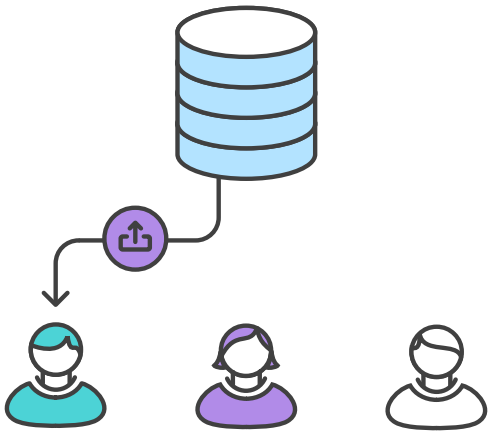
Cuando Mary regresa del almuerzo, completa su presentación. Antes de fusionarla`master` , necesita presentar una solicitud de extracción dejando que el resto del equipo sepa que ha terminado. Pero primero, ella debe asegurarse de que el repositorio central tenga sus compromisos más recientes:

```
git push
```

A continuación, se presenta la solicitud de extracción en su Git interfaz gráfica de usuario que pide a fusionarse `marys-feature` en `master`, y los miembros del equipo se notificará automáticamente. La gran cosa acerca de las solicitudes de extracción es que muestran comentarios justo al lado de sus

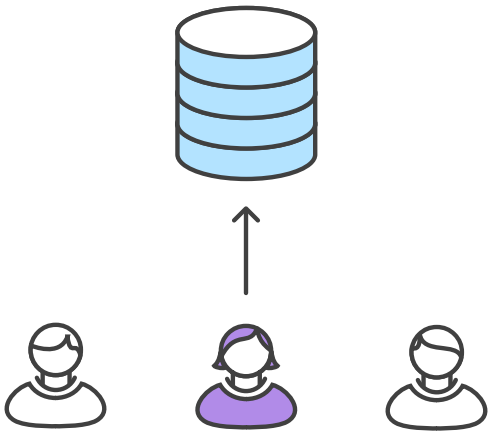
específicos.

Bill recibe la solicitud de extracción



Bill obtiene la solicitud de extracción y echa un vistazo `marys-feature` . Él decide que quiere hacer algunos cambios antes de integrarla en el proyecto oficial, y él y Mary tienen algo de ida y vuelta a través de la solicitud de pull.

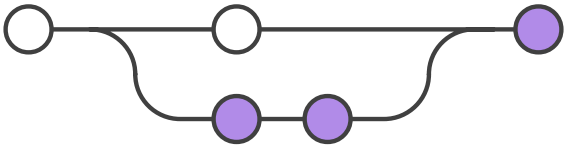
María hace los cambios



Para hacer los cambios, Mary utiliza exactamente el mismo proceso que ella hizo para crear la primera iteración de su característica. Ella edita, pone en escena, comete y empuja las actualizaciones al repositorio central. Toda su actividad aparece en la

Si lo deseaba, Bill podía `marys-feature` entrar en su repositorio local y trabajar en él por su cuenta. Cualquier commit que agregara también aparecería en la petición pull.

María publica su artículo



Una vez que Bill está listo para aceptar la solicitud de extracción, alguien necesita fusionar la función en el proyecto estable (esto puede hacerse por Bill o Mary):

```
git checkout master
git pull
git pull origin marys-feature
git push
```

Primero, quien esté realizando la fusión necesita revisar su `master` sucursal y asegurarse de que esté actualizada. A continuación, se `git pull origin marys-feature` fusiona la copia del repositorio central de `marys-feature` . También puede utilizar un `git merge marys-feature` comando simple , pero el comando mostrado anteriormente le asegura que siempre está tirando de la versión más actualizada de la rama de funciones. Por último, las `master` necesidades actualizadas para obtener empujado de nuevo a `origin`.

Este proceso a menudo resulta en un commit de combinación. Algunos desarrolladores les gusta esto porque es como una unión simbólica de la función con

característica en la extremidad de master antes de ejecutar la fusión, dando por resultado una fusión de avance rápido.

Algunas GUI automatizan el proceso de aceptación de la solicitud de extracción ejecutando todos estos comandos simplemente haciendo clic en el botón "Aceptar". Si el tuyo no lo hace, al menos debería ser capaz de cerrar automáticamente la solicitud de extracción cuando la rama característica se funde en `master`

Mientras tanto, John está haciendo exactamente lo mismo

Mientras Mary y Bill están trabajando `marys-feature` y discutiendo esto en su petición de atracción, John está haciendo exactamente lo mismo con su propia rama característica. Al aislar las funciones en ramas separadas, todo el mundo puede trabajar de forma independiente, pero aún así es trivial compartir los cambios con otros desarrolladores cuando sea necesario.

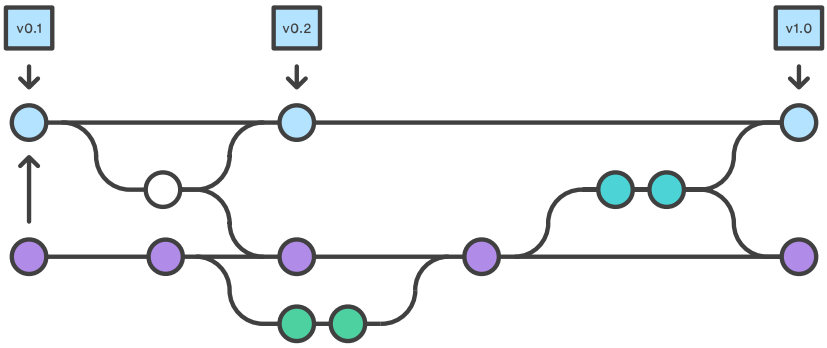
A dónde ir desde aquí

Para obtener una explicación detallada de la función de ramificación en Bitbucket, consulte la documentación Utilizando Git Branches . Por ahora, es de esperar que se vea cómo las ramas de funciones son una manera de multiplicar literalmente la funcionalidad de la `master` rama única utilizada en el flujo de trabajo centralizado . Además, las ramas de

dentro de la GUI de control de versiones.

El flujo de trabajo de la rama de funciones es una forma increíblemente flexible de desarrollar un proyecto. El problema es que a veces es demasiado flexible. Para equipos más grandes, a menudo es beneficioso asignar roles más específicos a diferentes ramas. El flujo de trabajo de Gitflow es un patrón común para gestionar el desarrollo de funciones, la preparación de lanzamientos y el mantenimiento.

Flujo de trabajo de Gitflow



La sección de flujo de trabajo de Gitflow se deriva de Vincent Driessen en nvie .

El flujo de trabajo de Gitflow define un modelo de ramificación estricto diseñado alrededor de la versión de proyecto. Aunque es un poco más complicado que el flujo de trabajo de la rama de funciones , proporciona un marco robusto para administrar proyectos más grandes.

Este flujo de trabajo no añade ningún nuevo concepto o comando más allá de lo requerido para el Flujo de

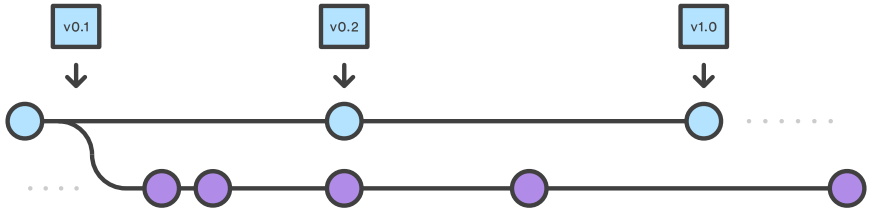
define cómo y cuándo deben interactuar. Además de las ramas de características, utiliza ramas individuales para preparar, mantener y grabar las versiones. Por supuesto, también obtendrá aprovechar todos los beneficios del flujo de trabajo de la rama de funciones: solicitudes de extracción, experimentos aislados y colaboración más eficiente.

Cómo funciona

El flujo de trabajo de Gitflow todavía utiliza un repositorio central como centro de comunicaciones para todos los desarrolladores. Y, al igual que en los otros flujos de trabajo , los desarrolladores trabajan localmente y empujan las sucursales al repo central. La única diferencia es la estructura de la sucursal del proyecto.

Sucursales históricas

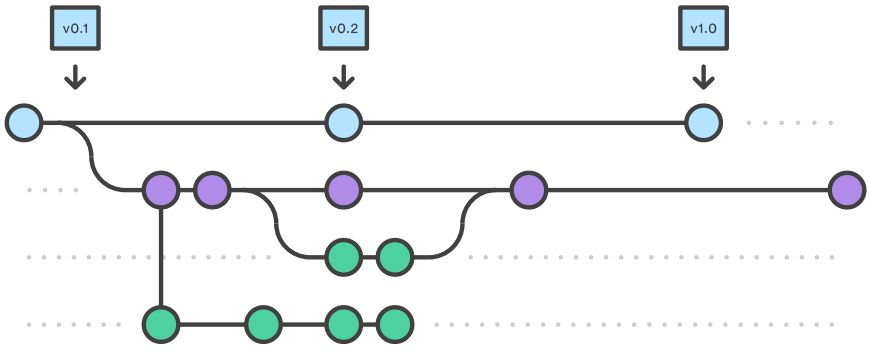
En lugar de una sola `master`rama, este flujo de trabajo utiliza dos ramas para registrar el historial del proyecto. La `master`sucursal almacena el historial oficial de lanzamientos, y la `develop`sucursal sirve como rama de integración para funciones. También es conveniente marcar todos los compromisos en la `master`sucursal con un número de versión.



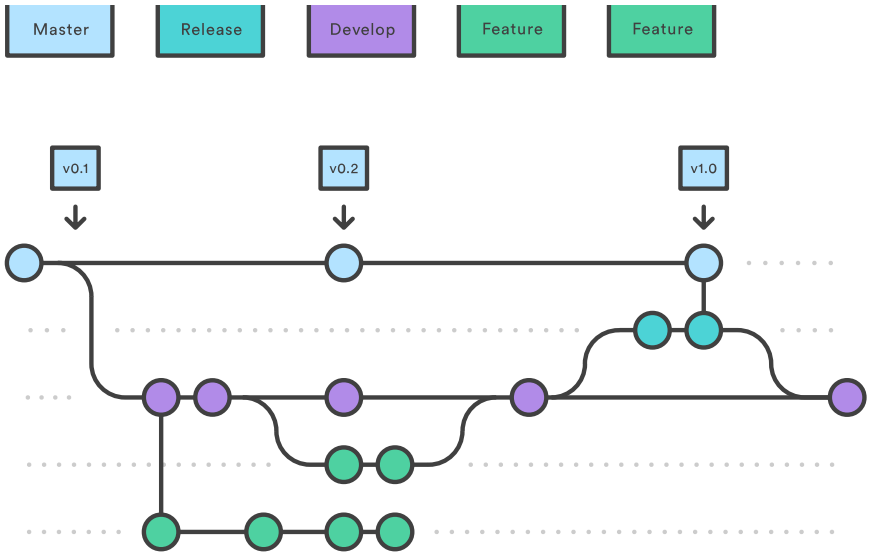
El resto de este flujo de trabajo gira en torno a la distinción entre estas dos ramas.

Rama de funciones

Cada nueva característica debe residir en su propia rama, que se puede enviar al repositorio central para la copia de seguridad / colaboración . Sin embargo, en lugar de derivarse `master`, las ramas funcionales se desarrollan como su rama principal. Cuando una característica está completa, se vuelve a fusionar de nuevo `develop` . Las funciones nunca deben interactuar directamente con `master`.



Tenga en cuenta que las ramas de entidades combinadas con la `develop`sucursal son, a todos los efectos, el flujo de trabajo de la rama principal. Sin embargo, el flujo de trabajo de Gitflow no se detiene allí.



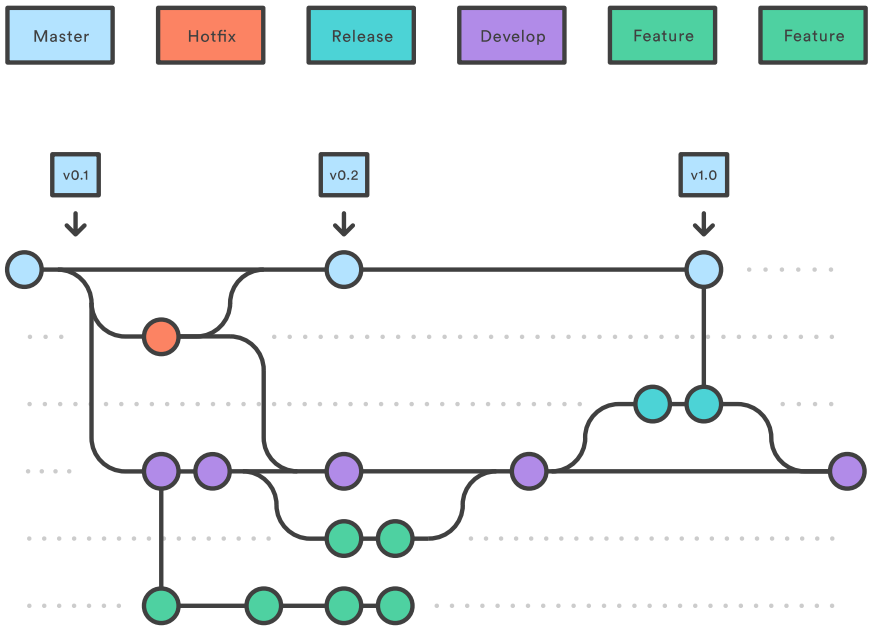
Una vez que `develop` ha adquirido suficientes características para una liberación (o una fecha de lanzamiento predeterminada se acerca), se bifurca una rama de liberación de `develop`. La creación de esta rama inicia el ciclo de lanzamiento siguiente, por lo que no se pueden agregar nuevas características después de que sólo se solucionen errores de punta, generación de documentación y otras tareas orientadas a la liberación. Una vez que esté listo para enviar, la versión se fusionará `master` y se marcará con un número de versión. Además, debería fusionarse nuevamente `develop`, lo que puede haber progresado desde que se inició el lanzamiento.

El uso de una rama dedicada a la preparación de versiones hace posible que un equipo pueda pulir la versión actual mientras otro equipo continúa trabajando en las funciones de la próxima versión. También crea fases bien definidas de desarrollo (por ejemplo, es fácil decir, "esta semana estamos preparando para la versión 4.0" y verlo en realidad en la estructura del repositorio).

Convenciones comunes:

- convenio de denominación:
  `release-* or release/*`

Ramas de mantenimiento

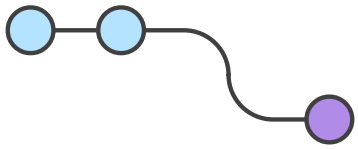


Las ramas de mantenimiento o "hotfix" se usan para rápidamente parches de lanzamientos de producción. Esta es la única rama que debe bifurcar directamente de `master`. Tan pronto como la corrección se haya completado, debería fusionarse en ambos `master`y `develop`(o en la rama actual de la versión) y `master` debería estar marcada con un número de versión actualizado.

Tener una línea dedicada de desarrollo para correcciones de errores le permite a su equipo resolver problemas sin interrumpir el resto del flujo de trabajo o esperar el próximo ciclo de lanzamiento. Puede pensar en ramas de mantenimiento como ramas de lanzamiento ad hoc que trabajan directamente con `master`.

El ejemplo siguiente muestra cómo se puede utilizar este flujo de trabajo para gestionar un solo ciclo de lanzamiento. Asumiremos que ya ha creado un repositorio central.

Crear una sucursal de desarrollo



El primer paso es complementar el valor predeterminado `master` con una `develop`rama. Una forma sencilla de hacerlo es que un desarrollador cree una `develop`rama vacía localmente y la empuje al servidor:

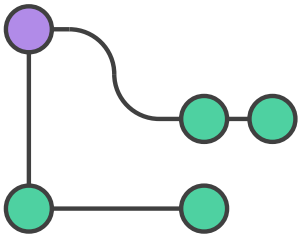
```
git branch develop
git push -u origin develop
```

Esta sucursal contendrá la historia completa del proyecto, mientras `master` que contendrá una versión abreviada. Otros desarrolladores ahora deben clonar el repositorio central y crear una rama de seguimiento para desarrollar:

```
git clone ssh://user@host/path/to/repo.git
git checkout -b develop origin/develop
```

Ahora todo el mundo tiene una copia local de las ramas históricas establecidas.

características



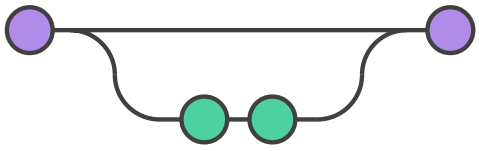
Nuestro ejemplo comienza con Juan y María trabajando en características separadas. Ambos necesitan crear ramas separadas para sus características respectivas. En lugar de basarla `master`, deberían basar sus ramas de características en `develop`:

```
git checkout -b some-feature develop
```

Ambos agregan compromisos a la rama de la característica de la manera generalmente: corrija, la etapa, confía:

```
git status
git add <some-file>
git commit
```

Mary termina su presentación

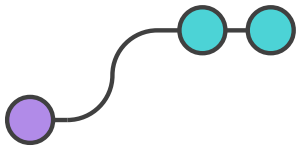


Después de agregar algunos commits, Mary decide que su característica está lista. Si su equipo está utilizando solicitudes de extracción, este sería el momento adecuado para abrir una solicitud para fusionar su función `develop`. De lo contrario, puede fusionarlo en su local `develop`y empujarlo al repositorio central, así:

```
git push
git branch -d some-feature
```

El primer comando asegura que la `develop`rama esté actualizada antes de intentar combinar la función. Tenga en cuenta que las funciones nunca deben fusionarse directamente en `master`. Los conflictos se pueden resolver de la misma manera que en el flujo de trabajo centralizado .

María comienza a preparar una liberación



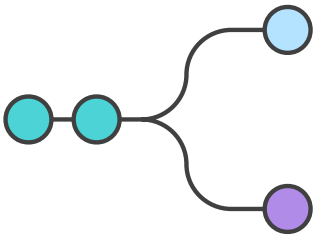
Mientras John sigue trabajando en su tema, Mary comienza a preparar el primer lanzamiento oficial del proyecto. Al igual que el desarrollo de características, utiliza una nueva rama para encapsular las preparaciones de liberación. Este paso es también donde se establece el número de versión de la versión:

```
git checkout -b release-0.1 develop
```

Esta sucursal es un lugar para limpiar el lanzamiento, probar todo, actualizar la documentación y hacer cualquier otro tipo de preparación para la próxima versión. Es como una rama característica dedicada a pulir el lanzamiento.

Tan pronto como Mary crea esta rama y la empuja al repositorio central, la liberación está congelada.

Mary termina el lanzamiento



Una vez que el lanzamiento está listo para enviar, Mary lo fusiona `master`y `develop`luego elimina la rama de liberación. Es importante volver a fusionarse `develop`porque las actualizaciones críticas pueden haber sido agregadas a la rama de lanzamiento y deben ser accesibles para las nuevas características. Una vez más, si la organización de Mary hace hincapié en la revisión de código, este sería un lugar ideal para una solicitud de tracción.

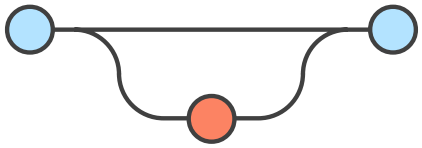
```
git checkout master
git merge release-0.1
git push
git checkout develop
git merge release-0.1
git push
git branch -d release-0.1
```

Las ramas de lanzamiento actúan como un buffer entre el desarrollo de características (`develop`) y las versiones públicas (`master`). Cada vez que se fusiona algo `master`, debe etiquetar el commit para facilitar la referencia:

```
git tag -a 0.1 -m "Initial public release" maste
git push --tags
```

dentro de un repositorio. Es posible configurar un gancho para crear automáticamente una versión pública cada vez que se empuja la `master` sucursal al repositorio central o se empuja una etiqueta.

El usuario final descubre un error



Después de enviar el lanzamiento, Mary vuelve a desarrollar características para el próximo lanzamiento con John. Es decir, hasta que un usuario final abra un ticket quejándose de un error en la versión actual. Para abordar el error, Mary (o John) crea una rama de mantenimiento de `master`, corrige el problema con tantos commits como sea necesario, y luego se fusiona directamente en `master`.

```
git checkout -b issue-#001 master
# Fix the bug
git checkout master
git merge issue-#001
git push
```

Al igual que las ramas de lanzamiento, las ramas de mantenimiento contienen actualizaciones importantes que deben incluirse `develop`, por lo que Mary necesita realizar esa combinación también. Entonces, ella es libre de eliminar la rama :

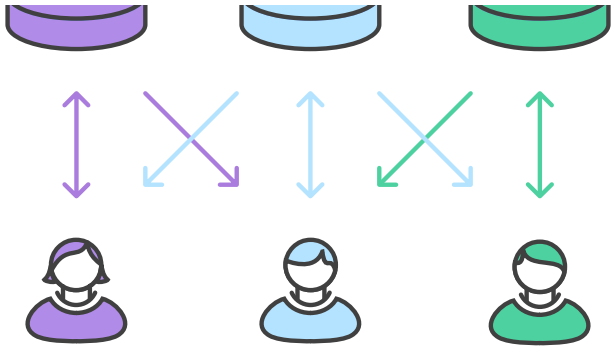
```
git checkout develop
git merge issue-#001
git push
git branch -d issue-#001
```

Por ahora, esperamos estar bastante a gusto con el flujo de trabajo centralizado , el flujo de trabajo de la rama de funciones y el flujo de trabajo de Gitflow. También debe tener una comprensión sólida sobre el potencial de los repositorios locales, el patrón push / pull y el robusto modelo de ramificación y fusión de Git.

Recuerde que los flujos de trabajo presentados aquí son meramente ejemplos de lo que es posible, no son reglas duras y rápidas para usar Git en el lugar de trabajo. Por lo tanto, no tenga miedo de adoptar algunos aspectos de un flujo de trabajo y hacer caso omiso de los demás. El objetivo siempre debe ser hacer que Git funcione para usted, no al revés.

Flujo de trabajo de bifurcación

El flujo de trabajo de Bifurcación es fundamentalmente diferente de los otros flujos de trabajo discutidos en este tutorial. En lugar de usar un solo repositorio del lado del servidor para actuar como base de código "central", proporciona a *cada* desarrollador un repositorio del lado del servidor. Esto significa que cada colaborador no tiene uno, sino dos repositorios de Git: uno local privado y uno público del servidor.



La principal ventaja del flujo de trabajo de bifurcación es que las contribuciones se pueden integrar sin la necesidad de que todo el mundo empuje a un único repositorio central. Los desarrolladores empujan a *sus propios* repositorios del lado del servidor, y sólo el responsable del proyecto puede pasar al repositorio oficial. Esto permite al mantenedor aceptar compromisos de cualquier desarrollador sin darles acceso de escritura a la base de código oficial.

El resultado es un flujo de trabajo distribuido que proporciona una forma flexible para que los grandes equipos orgánicos (incluidos los terceros no confiables) colaboren de forma segura. Esto también lo convierte en un flujo de trabajo ideal para proyectos de código abierto.

Cómo funciona

Al igual que en los otros flujos de trabajo de Git, el flujo de trabajo de Forking comienza con un repositorio público oficial almacenado en un servidor. Pero cuando un nuevo desarrollador quiere comenzar a trabajar en el proyecto, no clonan directamente el repositorio oficial.

nueva copia sirve como su repositorio público personal; no se permite a otros desarrolladores empujar hacia él, pero pueden extraer cambios de él (veremos por qué esto es importante en un momento). Después de haber creado su copia del servidor, el desarrollador realiza `git clone`una copia para obtener una copia de la misma en su máquina local. Esto sirve como su entorno de desarrollo privado, al igual que en los otros flujos de trabajo.

Cuando están listos para publicar un commit local, empujan el commit a su propio repositorio público, no el oficial. A continuación, archivan una solicitud de extracción con el repositorio principal, lo que permite al responsable del proyecto saber que una actualización está lista para ser integrada. La solicitud de extracción también sirve como un hilo de discusión conveniente si hay problemas con el código aportado.

Para integrar la característica en la base de código oficial, el mantenedor extrae los cambios del contribuyente en su repositorio local, comprueba que no rompe el proyecto, lo fusiona en su `master`sucursal local y , a continuación, empuja la `master`sucursal al repositorio oficial en el servidor . La contribución es ahora parte del proyecto, y otros desarrolladores deben retirarse del repositorio oficial para sincronizar sus repositorios locales.

El Repositorio Oficial

Es importante entender que la noción de un repositorio "oficial" en el flujo de trabajo de Bifurcación es simplemente una convención. Desde un punto de

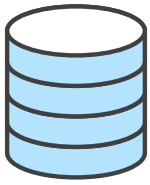
De hecho, lo único que hace que el repositorio oficial sea tan oficial es que es el repositorio público del mantenedor del proyecto.

Ramificación en el flujo de trabajo de bifurcación

Todos estos repositorios públicos personales son realmente una manera conveniente de compartir ramas con otros desarrolladores. Todo el mundo debería seguir utilizando ramas para aislar las características individuales, al igual que en el Flujo de trabajo de la rama principal y el flujo de trabajo de Gitflow . La única diferencia es cómo se comparten esas ramas. En el flujo de trabajo de bifurcación, se insertan en el repositorio local de otro desarrollador, mientras que en la rama de características y los flujos de trabajo Gitflow se envían al repositorio oficial.

Ejemplo

El responsable del proyecto inicializa el repositorio oficial



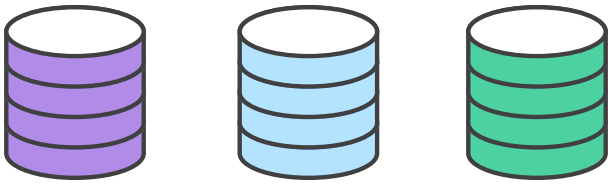
Como con cualquier proyecto basado en Git, el primer paso es crear un repositorio oficial en un servidor accesible a todos los miembros del equipo.

Los repositorios públicos siempre deben estar desnudos , independientemente de si representan o no la base de código oficial. Por lo tanto, el responsable del proyecto debe ejecutar algo como lo siguiente para configurar el repositorio oficial:

```
ssh user@host
git init --bare /path/to/repo.git
```

Bitbucket también proporciona una alternativa GUI conveniente a los comandos anteriores. Este es exactamente el mismo proceso que configurar un repositorio central para los otros flujos de trabajo en este tutorial. El mantenedor también debe empujar la base de código existente a este repositorio, si es necesario.

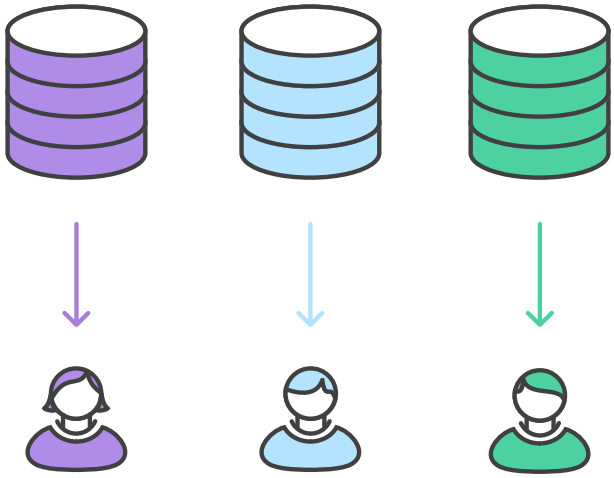
Los desarrolladores forjan el repositorio oficial



A continuación, todos los demás desarrolladores necesitan forjar este repositorio oficial. Es posible hacerlo por SSH 'ing en el servidor y en ejecución `git clone`para copiar a otra ubicación en el servidor, sí, el bifurcación es básicamente sólo un clon del lado del servidor. Pero de nuevo, Bitbucket permite a los desarrolladores forjar un repositorio con el clic de un botón.

el repositorio oficial, todos estos deben ser repositorios desnudos.

Los desarrolladores clonan sus repositorios bifurcados



A continuación, cada desarrollador necesita clonar su propio repositorio público. Pueden hacer con lo familiar `git clone command`.

Nuestro ejemplo asume el uso de Bitbucket para alojar estos repositorios. Recuerde, en esta situación, cada desarrollador debe tener su propia cuenta de Bitbucket y deben clonar su repositorio del lado del servidor usando:

```
git clone https://user@bitbucket.org/user/repo.g
```

Mientras que los otros flujos de trabajo de este tutorial utilizan un único `origin`control remoto que apunta al repositorio central, el flujo de trabajo de Forking requiere dos controles remotos, uno para el repositorio oficial y otro para el repositorio personal del desarrollador. Aunque se puede llamar a estos

repositorio bifurcado (esto se creará automáticamente al ejecutarlo `git clone`) y `upstream` para el repositorio

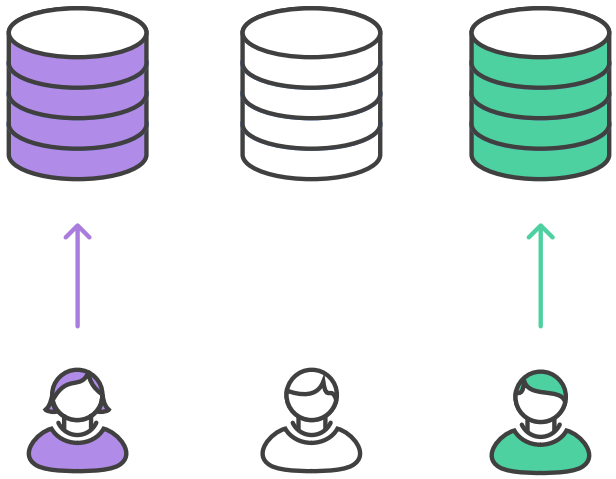
```
git checkout -b some-feature
# Edit some code
git commit -a -m "Add first draft of some feature
```

Todos sus cambios serán enteramente privados hasta que lo empujen a su repositorio público. Y, si el proyecto oficial ha avanzado, pueden acceder a nuevos compromisos con `git pull`:

```
git pull upstream master
```

Dado que los desarrolladores deben trabajar en una rama dedicada, esto generalmente debería dar lugar a una fusión de avance rápido .

Los desarrolladores publican sus características



Una vez que un desarrollador está listo para compartir su nueva función, tienen que hacer dos cosas. En primer lugar, tienen que hacer su contribución accesible a otros desarrolladores empujándolo a su

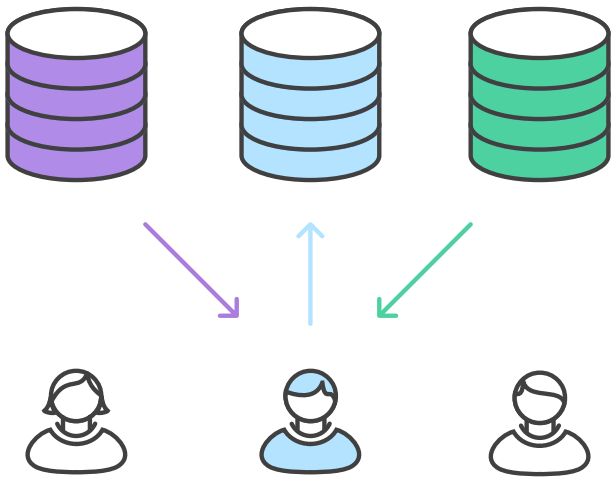
deberían hacer es el siguiente:

```
git push origin feature-branch
```

Esto diverge de los otros flujos de trabajo en que los `origin` puntos remotos al repositorio personal del desarrollador del servidor, no la base de código principal.

En segundo lugar, necesitan notificar al responsable del proyecto que desean fusionar su característica en la base de código oficial. Bitbucket proporciona un botón " Pedir solicitud " que lleva a un formulario que le pide que especifique qué rama desea fusionar en el repositorio oficial. Normalmente, deseará integrar su rama de funciones en la sucursal del control remoto ascendente `master`.

El responsable del proyecto integra sus características



Cuando el responsable del proyecto recibe la solicitud de extracción, su tarea es decidir si se debe integrar o no en la base de código oficial. Pueden hacer esto de dos maneras:

solicitud de extracción

2. Tire del código en su repositorio local y fusionarlo manualmente

La primera opción es más simple, ya que permite al mantenedor ver una diferencia de los cambios, comentarlo y realizar la combinación a través de una interfaz gráfica de usuario. Sin embargo, la segunda opción es necesaria si la solicitud de extracción produce un conflicto de combinación. En este caso, el mantenedor necesita buscar la rama de características del repositorio del servidor del desarrollador, fusionarla en su `master` sucursal local y resolver cualquier conflicto:

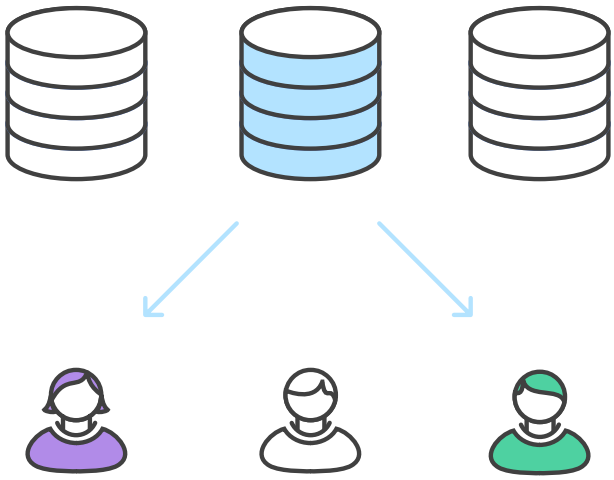
```
git fetch https://bitbucket.org/user/repo feature
# Inspect the changes
git checkout master
git merge FETCH_HEAD
```

Una vez que los cambios se integran en su local `master`, el mantenedor debe empujarlo al repositorio oficial del servidor para que otros desarrolladores puedan acceder a él:

```
git push origin master
```

Recuerde que el mantenedor `origin` apunta a su repositorio público, que también sirve como base de código oficial para el proyecto. La contribución del desarrollador ahora está totalmente integrada en el proyecto.

repositorio oficial



Dado que la base de código principal ha avanzado, otros desarrolladores deben sincronizarse con el repositorio oficial:

```
git pull upstream master
```

A dónde ir desde aquí

Si usted viene de un fondo SVN, el flujo de trabajo de bifurcación puede parecer un cambio de paradigma radical. Pero no tengas miedo, todo lo que realmente está haciendo es introducir otro nivel de abstracción en la parte superior del flujo de trabajo de la rama principal . En lugar de compartir las sucursales directamente a través de un único repositorio central, las contribuciones se publican en un repositorio del servidor dedicado al desarrollador de origen.

Este artículo explica cómo una contribución fluye de un desarrollador a la `master`sucursal oficial , pero la misma metodología puede utilizarse para integrar una contribución en cualquier repositorio. Por ejemplo, si

cambios entre ellos de la misma manera, sin tocar el repositorio principal.

Esto hace que el flujo de trabajo de bifurcación sea una herramienta muy potente para los equipos de punto suelto. Cualquier desarrollador puede compartir fácilmente los cambios con cualquier otro desarrollador, y cualquier sucursal se puede fusionar eficientemente en la base de código oficial.

Tutoriales

Enter Your Email For Git News

Enter Your Email For Git News