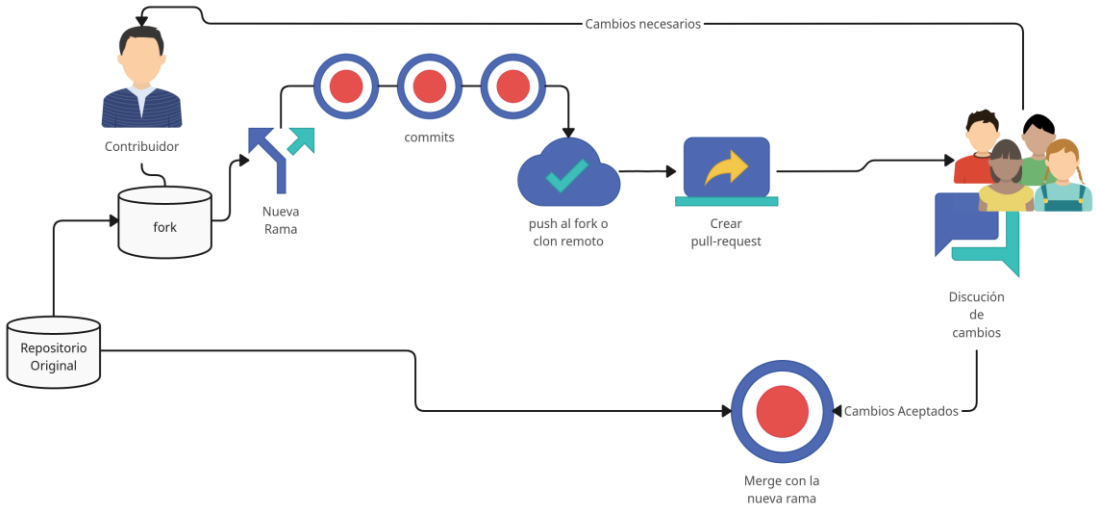
Git Advance Commands

pull request

Los pull request (solicitud de incorporación de cambios) son una forma de contribuir en el código de un proyecto. Esta funcionalidad suele ser comúnmente encontrada en plataformas de alojamiento de repositorios como GitHUb o GitLab, y consiste en que un colaborador puede hacer cambios en un repositorio propio a partir de la copia de un repositorio remoto o principal (fork), para luego enviar una solicitud de extracción (pull request), para así exponer sus cambios y sean evaluados antes de ser fusionados en la rama principal.

Después de recibir la solicitud, el propietario del repositorio revisa y evalúa el nuevo comportamiento, declinando o aprobando los cambios, en cuyo caso, deberá realizar una fusión con la rama principal del repositorio original, incorporando oficialmente los cambios. Los pasos mencionados anteriormente pueden describirse de la siguiente forma:

1. Crear un fork del repositorio original
2. Clonar el repositorio bifurcado de manera local o trabajarlo de la forma que se ajuste a las necesidades.
3. Crear una nueva rama para trabajar en los cambios
4. Realizar los cambios que se consideren adecuados para el proyecto en la nueva rama
5. Hacer el commit con los cambios que se han realizado y subir los cambios en el repositorio bifurcado
6. Debe acceder la página del repositorio original remoto (GitHub, GitLab, etc.) y crear un nuevo “pull request”.
7. Elegir la rama que contiene los cambios que se desan fusionar y la rama a la que se desean fusionar.
8. Se debe agregar también una descripción detallada sobre los cambios realizados.
9. Finalmente, enviar al pull request al adminitrador del repositorio original para su aprobación o rechazo.

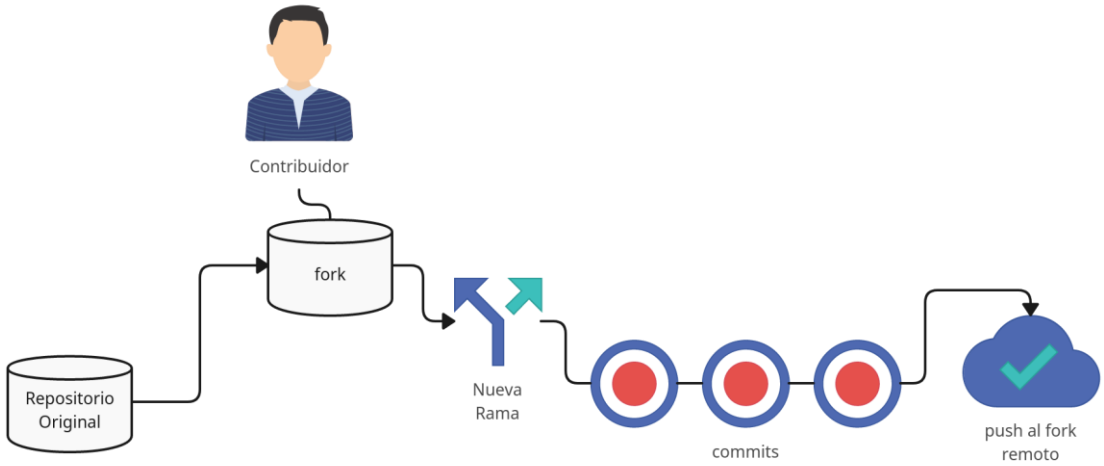


fork

Un fork (bifurcación) en git, es una función consiste en crear un repositorio con una cuenta diferente del proyecto original a partir de la copia del repositorio con código fuente que ya existe. Este nuevo repositorio contendrá los archivos del proyecto original, así como su propia rama principal.

La función principal de esta característica es la de trabajar en una copia de un proyecto original sin afectar esa versión, experimentando y realizando cambios significativos sin alterar la rama principal original, pudiendo contribuir para un futuro pull request o como una forma de tener una versión personalizada de algún software.

Una vez que se realiza un fork, el usuario puede hacer cambios en la copia del repositorio y crear nuevas ramas, solicitudes de extracción y nuevas versiones del proyecto. También puede mantener la copia sincronizada con el repositorio original, utilizando la funcionalidad de pull request para enviar cambios desde su fork hacia el proyecto original. Los cambios que se hacen al repositorio original no se transmiten de forma automática al fork ni viceversa.



rebase

Un rebase (reproduce) en git, es una operación que permite que los cambios de una rama se integren en otra de forma limpia y ordenada. Es decir, que no fusiona los cambios de dos ramas en un solo commit, si no que se encarga de mover los cambios de forma que parezca que los cambios fueron realizados en la rama de destino directamente.

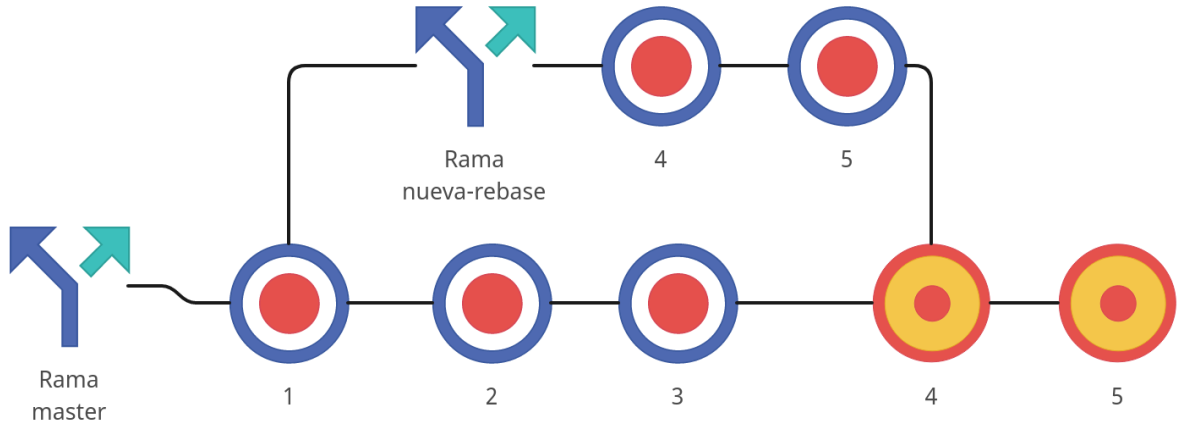
La principal razón para llevar a cabo una fusión mediante rebase es para mantener el historial del proyecto de forma lineal. Esto con la finalidad de mantener un historial que ayuda en la identificación de errores más fácil y permite tener una visión más clara.

Para realizar rebase, asegúrate de tener todos los commits que quieras en el rebase en tu rama master. Revisar la rama en la que quieres hacer el rebase y escribe

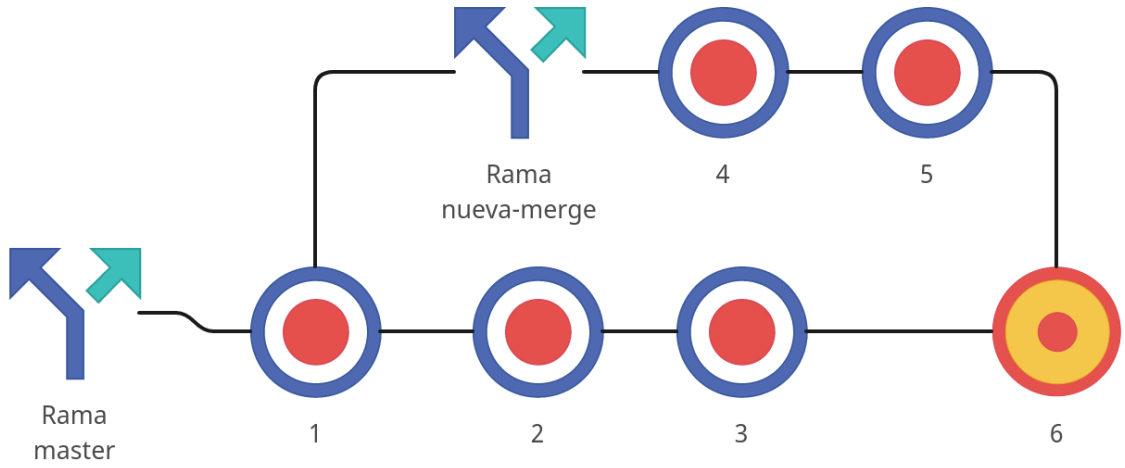
```
//master es la rama en la que quieres hacer el rebase

$ git rebase master
```

También es posible hacer rebase en una rama diferente, de modo que, por ejemplo, una rama que se basaba en otra rama.

Git rebase generará una estructura como lo muestra el siguiente diagrama, donde la rama master se ha fusionado con la rama nueva-rebase, añadiendo al historial de forma lineal los commits realizados en esta rama.



A diferencia de merge, que en lugar de colocar los commits de la rama nueva-rebase, coloca los cambios en un nuevo merge commit extra.



Rebase es una característica que puede ser peligrosa en un repositorio remoto, ya que se afecta el historial de commits y pude crear problemas cunado otros desarrolladores intentan sacar los últimos cambios de código del repositorio remoto.

Para usar git rebase en la consola con una lista de commits, puedes elegir, editar o soltar en el rebase:

```
// con el último número que sea cualquier número de commits del más reciente hacia atrás que quieras revisar

$ git rebase -i HEAD~5
```

En vim, presiona esc, luego i para empezar a editar la prueba.

A la izquierda puedes sobreescribir *pick* con uno de los comandos de abajo. Si quieres aplastar (*squash*) un commit a uno anterior y descartar el mensaje de commit, introduce *f* en lugar de *pick* en el commit.

Guarda y sal del editor de texto. Cuando se detiene a rebase, haz los ajustes necesarios, y luego usa **git** rebase --continue hasta que el rebase sea exitoso.

Si el rebase es exitoso, entonces necesitas forzar el *push* de tus cambios con *git push -f* para agregar la versión con rebase a tu repositorio remoto.

Si hay un *merge conflict*, hay varias maneras de arreglarlo. Una forma es abrir los archivos en un editor de texto y eliminar las partes del código que no quieras. Luego usa **git** add <file name> seguido de **git** rebase --continue. Puedes saltarte el commit de conflicto ejecutando **git** rebase --skip, para el rebase ejecutando **git** rebase --abort.

```
pick 452b159 <message for this commit>
pick 7fd4192 <message for this commit>
pick c1af3e5 <message for this commit>
pick 5f5e8d3 <message for this commit>
pick 5186a9f <message for this commit>

# Rebase 0617e63..5186a9f onto 0617e63 (30 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but stop to edit the commit message.
# e, edit = use commit, but stop to amend or add commit.
# s, squash = use commit, meld into previous commit and stop to edit the commit message.
# f, fixup = like "squash", but discard this commit's log message thus doesn't stop.
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

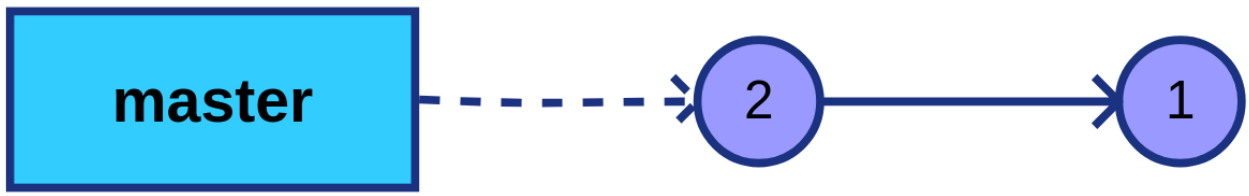
stash

En Git, un stash (apilamiento) es una característica que permite guardar temporalmente los cambios que se han realizado en un repositorio sin realizar un commit, lo que puede ser útil cuando se necesita cambiar de rama o de tarea sin terminar completamente lo que se está haciendo en ese momento. Está separada del directorio de trabajo (working directory), del área de preparación (staging area), y del repositorio.

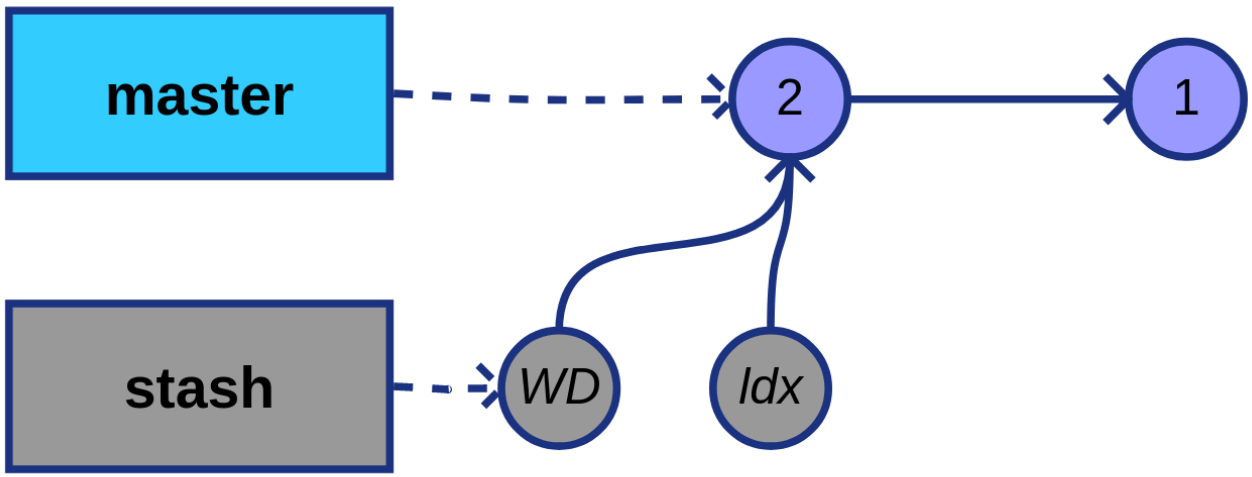
Un stash guarda los cambios que se han realizado en el directorio de trabajo y en el índice (stage) del repositorio en un área de almacenamiento temporal. Para guardar los cambios en el stash se debe ejecutar el comando:

```
$ git stash save "mensaje opcional"
```

Si se aplica el stash a una rama, como la que se ve a continuación:



Se obtiene una estructura como la que se ve a continuación, donde se han almacenado los cambios y se ha logrado guardar el trabajo que aún no se ha versionado mediante commit:



Mediante este comando se guardan los cambios y se revierte también el directorio de trabajo a como se veía en el último commit. Además, los cambios guardados están también disponibles desde cualquier rama del repositorio.

Solo los archivos rastreados por git pueden ser guardados en el stash, los nuevos archivos no pueden ser almacenados de esta forma. Para ver lo que hay dentro del stash, se debe ejecutar el comando:

```
$ git stash list
```

Esto devuelve una lista de tus capturas guardadas en el formato stash@{0}: RAMA-STASHED-CAMBIOS-SON-PARA: MESSAGE. La parte de stash@{0} es el nombre del stash, y el número en las llaves ({ }) es el índice (index) del stash. Si tienes múltiples conjuntos de cambios guardados en stash, cada uno tendrá un índice diferente.

Se puede ver un resumen de los cambios realizados con el comando:

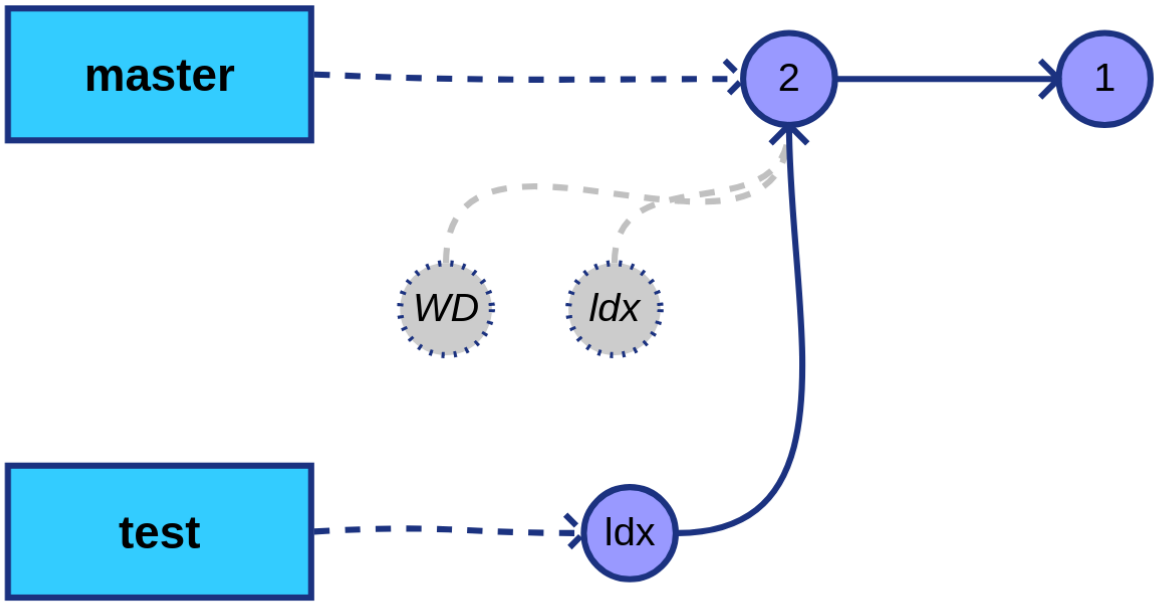
```
$ git stash show NOMBRE-DEL-STASH


//Si se desea ver con el diseño diff se puede incluir la opción -p (para parche o patch)

$ git stash show -p stash@{0}
```

Po ejemplo:

```
diff --git a/PathToFile/fileA b/PathToFile/fileA
index 2417dd9..b2c9092 100644
--- a/PathToFile/fileA
+++ b/PathToFile/fileA
@@ -1,4 +1,4 @@
-What this line looks like on branch
+What this line looks like with stashed changes
```

Para recuperar los cambios del stash y aplicarlos a la rama actual en la que estás, tienes dos opciones:

```
//aplica los cambios y deja una copia en el stash

$ git stash apply NOMBRE-DEL-STASH

//aplica los cambios y elimina los archivos del stash

$ git stash pop NOMBRE-DEL-STASH
```



Puede haber conflictos cuando se aplican los cambios. Puedes resolver los conflictos de forma similar a un merge.

Si quieres remover los cambios guardados en stash sin aplicarlos, ejecuta el comando:

```
$ git stash drop NOMBRE-DEL-STASH

//para limpiar todo del stash

$ git stash clear
```

clean

En Git, el comando "git clean" es una herramienta que permite eliminar los archivos no rastreados del directorio de trabajo, es decir, los archivos que no han sido agregados al control de versiones de Git.

Este comando es útil cuando se desea eliminar archivos temporales, archivos generados por el sistema operativo o por la compilación, o archivos que se han incluido accidentalmente en el directorio de trabajo y que no se deben incluir en el repositorio. Para utilizar el comando, se puede elegir entre las siguientes opciones:

```
//para correr una prueba con los posibles archivos que se eliminarían

$ git clean -n

//para forzar la eliminación de archivos

$ git clean -f
```

```
//para la eliminación de archivos .gitignore

$ git clean -f -x

//para la eliminación de directorios no rastreados

$ git clean -f -d
```

Es importante tener en cuenta que "git clean" borra los archivos de forma permanente, por lo que se deben tener precauciones al usarlo para evitar la pérdida de datos importantes.

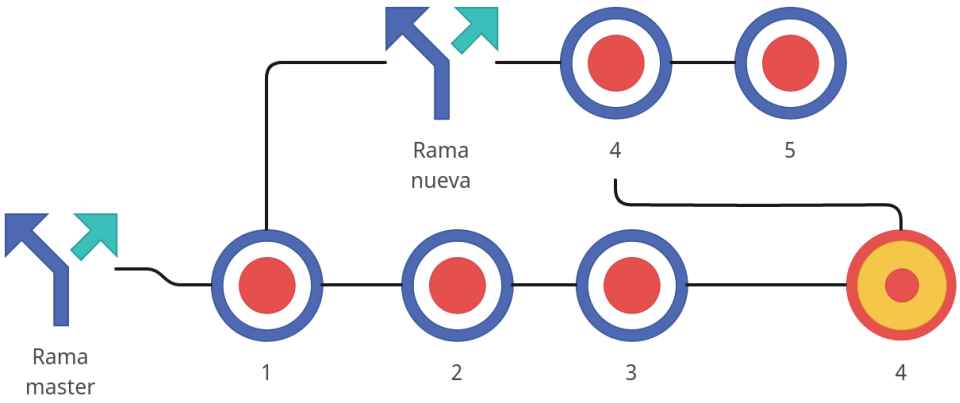
cherry-pick

Cherry-pick es una operación que permite aplicar un conjunto específico de commits desde una rama a otra rama diferente. Es decir, cherry-pick selecciona un conjunto de cambios (commits) específicos de una rama y los aplica a otra rama sin fusionar toda la rama de origen.

La operación de cherry-pick es útil cuando se desea aplicar un conjunto de cambios específicos a una rama diferente, sin necesidad de fusionar toda la rama de origen. Por ejemplo, puede ser útil para aplicar cambios específicos a una rama de producción sin tener que incluir todo el historial de la rama de desarrollo.

Suponiendo que un desarrollador no reconoce en qué rama se encuentra y que por error el realiza un commit en otra rama que no es la indicada. Para solucionarlo, se debe ejecutar git show, después guardar el commit, pasar a la rama principal, aplicar le parche en la rama principal y hacer el commit con el mismo mensaje de confirmación. Sin embargo, todos estos pasos pueden resumirse ejecutando únicamente un cherry-pick.

Toma uno o varios commits de una rama y los aplica a la rama actual. El comando crea nuevos commits en la rama de destino que contienen los cambios seleccionados del commit original.



Es importante tener en cuenta que, aunque cherry-pick puede ser una herramienta útil, no se recomienda su uso para fusionar cambios grandes o para fusionar ramas completas, ya que puede generar conflictos y problemas en el historial de Git. Se recomienda utilizar la fusión (merge) para esos casos.

En un caso de uso real, suponiendo que hay una rama principal llamada main y una rama de características llamada feature. En la rama feature, se han realizado dos commits que contienen cambios importantes, y se requiere solo uno de ellos en la rama main sin fusionar toda la rama feature.

Para hacer esto, puedes usar el comando git cherry-pick. Por ejemplo, si quieres aplicar el segundo commit de la rama feature en la rama main, puedes hacer lo siguiente:

```
//posicionarse en la rama principal

$ git checkout main
```

Usa el comando git log para encontrar el hash del commit que se desea aplicar en la rama main. Por ejemplo, el hash es abc123.

```
//aplicar el commit en la rama main

$ git cherry-pick abc123
```

Este comando aplicará los cambios realizados en el commit abc123 en la rama main. Si el commit aplica cambios en archivos que ya existen en la rama main, Git intentará fusionar los cambios automáticamente. Si hay conflictos, deberás resolverlos manualmente.

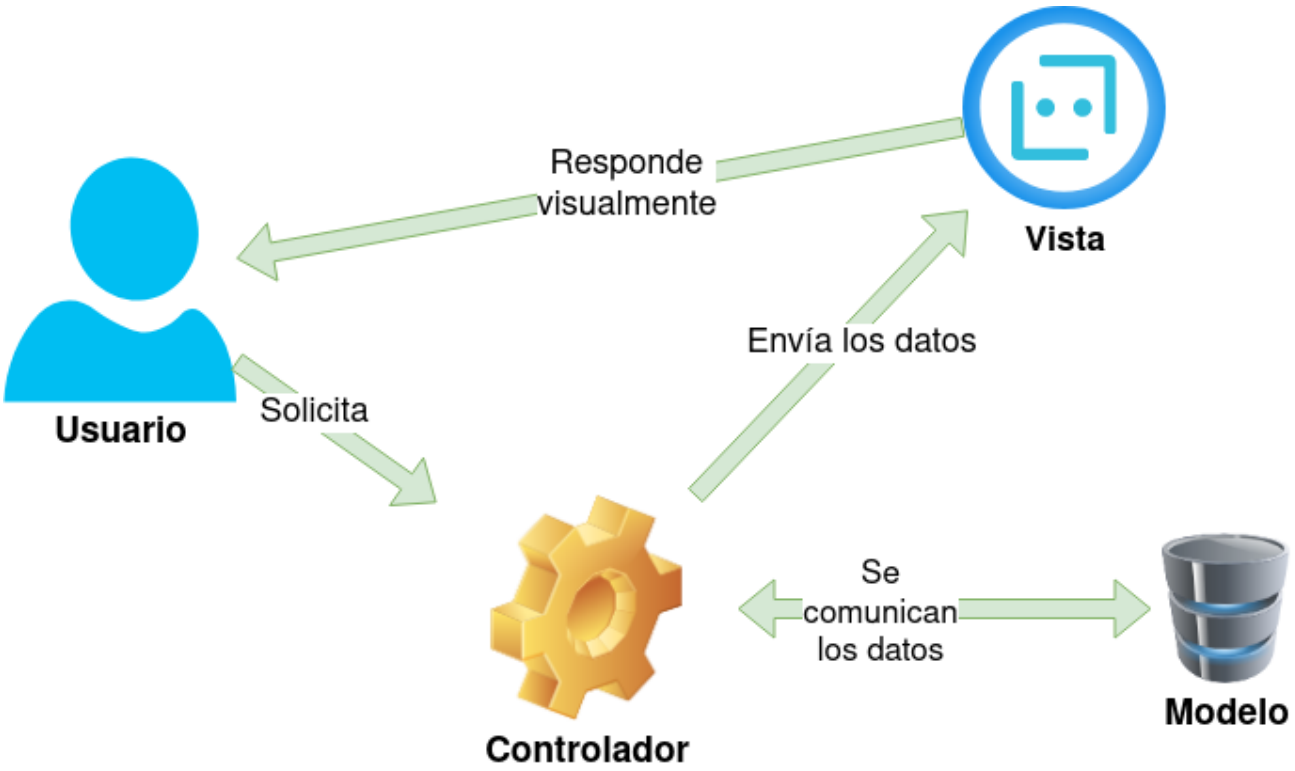
Una vez que hayas aplicado el commit, puedes confirmar los cambios y hacer un nuevo commit en la rama main para registrar los cambios.

Es importante tener en cuenta que git cherry-pick copia el contenido de un commit de una rama y lo aplica en otra rama, por lo que no se trasladan las referencias o el historial del commit original.

Modelo Vista Controlador (MVC)

El patrón de arquitectura MVC (Modelo-Vista-Controlador) es un patrón de diseño comúnmente utilizado en el desarrollo de software para separar la lógica de negocio, la presentación de la interfaz de usuario y el manejo de eventos en componentes independientes.

Esta modularidad permite que diferentes desarrolladores trabajen de forma simultánea en una aplicación sin tener tantos conflictos entre códigos y funciones que choquen entre sí. En general, en el modelo, se contiene todo el backend que contiene la lógica de los datos, la vista contiene el frontend o interfaz gráfica de usuario (GUI), y el controlador es el encargado de controlar como se muestran los datos comunicando la vista con el modelo.



El concepto de MVC fue introducido por primera vez por Trygve Reenskaug, como una forma de desarrollo de aplicaciones GUI de aplicaciones de escritorio, sin embargo, es también posible usarlo hoy en día en aplicaciones web modernas, lo que le permite ser escalable, mantenible y fácil de expandir.

Modelo (model): El modelo es responsable de almacenar y manejar los datos de la aplicación. No contiene ninguna lógica que describa cómo presentar los datos al usuario. El modelo simplemente administra los datos, ya sea que provengan de una base de datos una API o de un JSON.

Vista (view): La vista es responsable de presentar los datos del modelo al usuario en una interfaz gráfica de usuario (GUI). La vista sabe cómo acceder a los datos del modelo, pero no sabe qué significan estos datos o qué puede hacer el usuario con ellos.

Controlador (controller): El controlador es responsable de manejar las acciones del usuario y de actualizar el modelo y la vista según sea necesario. Existe entre la vista y el modelo, escuchando los eventos que se desencadenas por la vista o por fuentes externas y ejecutando las acciones adecuadas como respuesta, regularmente, llamar a un método en el modelo.

Utilizar este patrón de arquitectura, tiene sus ventajas, entre las que destaca:

- Permitir a varios desarrolladores trabajando de forma simultánea en el modelo, el controlador y la vista.
- Permitir la agrupación de la lógica de las acciones relacionadas con el controlador.
- Permite que los modelos puedan tener varias vistas diferentes entre sí, pero respondiendo a los mismos datos.

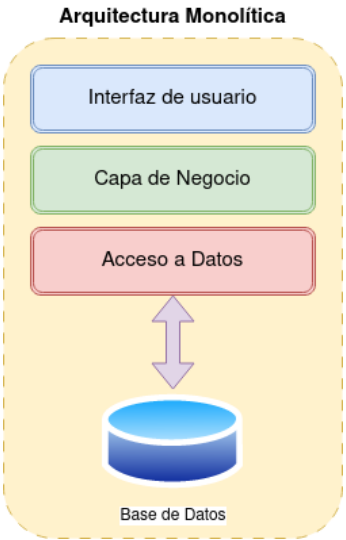
También hay algunas desventajas que es importante tomar en cuenta:

- La navegación puede ser compleja, debido a las capas de abstracción y requiere qye los usuarios se adapten a los criterios de descomposición de MVC.
- El conocimiento en múltiples tecnologías debe ser obligatorio para desarrolladores que utilizan MVC.

Aplicación Monolítica

Una aplicación monolítica es un tipo de aplicación que se desarrolla como una sola unidad de software, en la que todas las funcionalidades y componentes de la aplicación están integrados en un solo programa o paquete de software.

Esta aplicación es autosuficiente, y contiene absolutamente toda la funcionalidad necesaria para realizar la tarea para la que fue diseñada. En una aplicación monolítica, el código fuente se escribe en un solo lenguaje de programación y se ejecuta en una sola instancia de proceso en el sistema operativo. Esto significa que todas las funcionalidades de la aplicación, desde la capa de presentación hasta la capa de base de datos, se ejecutan en la misma instancia de proceso.



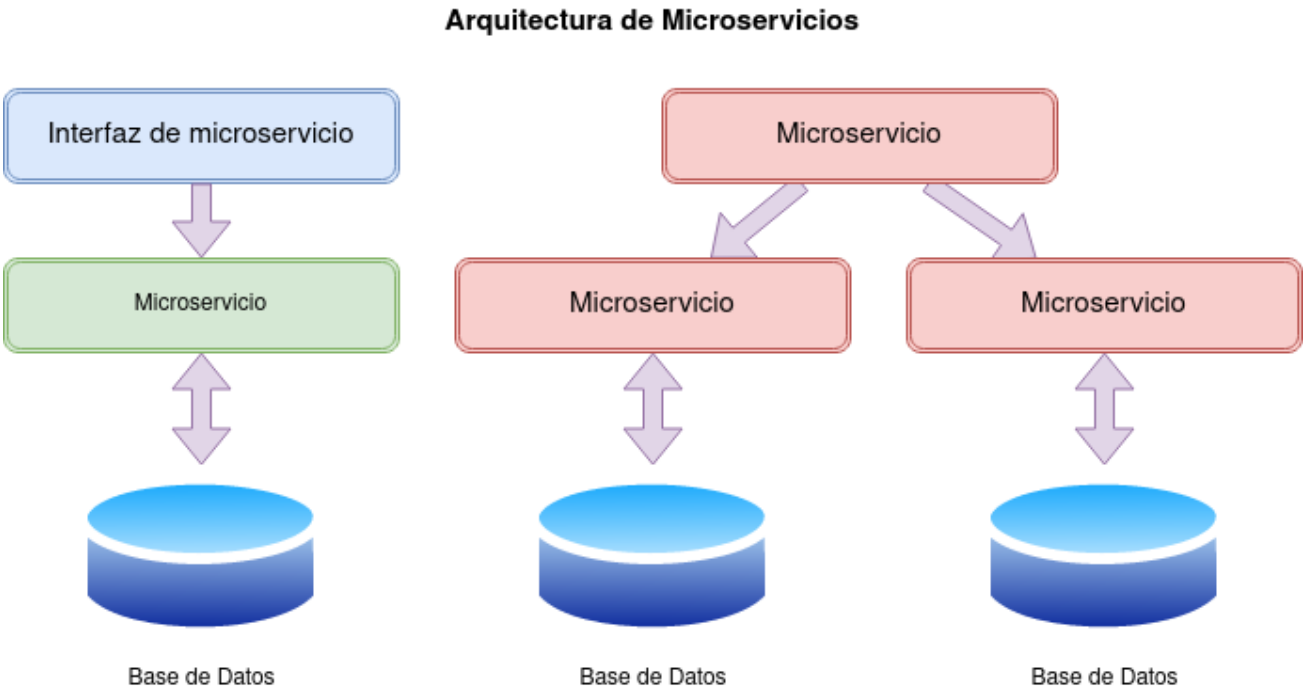
Cada componente de la aplicación está estrechamente acoplado y depende de otros componentes dentro de la misma aplicación. Esto hace que el proceso de mantenimiento y actualización de la aplicación sea más complejo y costoso, ya que cualquier cambio en un componente puede afectar a otros componentes.

Las aplicaciones monolíticas son comunes en el desarrollo de software tradicional, especialmente en aplicaciones empresariales. Sin embargo, con el auge de la arquitectura de microservicios y la necesidad de escalabilidad y flexibilidad en el desarrollo de software, cada vez más organizaciones están optando por arquitecturas más distribuidas y modularizadas en lugar de las aplicaciones monolíticas tradicionales.

Aplicación con Microservicios

Una aplicación con microservicios es una aplicación diseñada y construida como un conjunto de servicios independientes y autónomos, cada uno de los cuales se ejecuta en su propio proceso y se comunica con los demás mediante una API.

Cada microservicio tiene su propia responsabilidad y realiza una tarea específica en la aplicación. Esto significa que cada servicio puede ser desarrollado, probado y desplegado de forma independiente sin afectar a otros servicios en la aplicación.



Cada servicio se puede desarrollar en diferentes lenguajes de programación y tecnologías, lo que permite a los equipos de desarrollo utilizar las herramientas y tecnologías más adecuadas para cada tarea. Además, los microservicios pueden escalar horizontalmente, lo que significa que se pueden agregar más instancias de un servicio para manejar una mayor carga de trabajo.

Los microservicios han creado infraestructuras IT más adaptables y flexibles. Porque si se quiere modificar solamente un servicio, no es necesario alterar el resto de la infraestructura.

Aplicaciones Monolíticas vs Aplicaciones con Microservicios

Por un lado, las aplicaciones monolíticas y las aplicaciones con microservicios tienen diferentes enfoques de diseño y tienen sus propias ventajas y desventajas.

Las aplicaciones monolíticas ofrecen las siguientes ventajas:

- Son más fáciles de desarrollar y de desplegar.
- Tienen un menor costo inicial y son más fáciles de mantener, ya que solo hay un conjunto de código y una base de datos.
- Pueden tener un mejor rendimiento ya que no hay necesidad de comunicarse a través de la red u otros medios.

Por su parte, las aplicaciones con microservicios, cuentan con las siguientes ventajas:

- Son altamente escalables y pueden ser escalados horizontalmente agregando más instancias de un servicio determinado.
- Permite a los equipos de desarrollo trabajar de forma más autónoma, lo que significa que pueden elegir las tecnologías y herramientas que mejor se adapten a cada servicio.
- Los servicios pueden ser actualizados y desplegados de forma independiente sin afectar al resto de la aplicación.

Las aplicaciones monolíticas tienen entre sus principales desventajas:

- Son más difíciles de escalar verticalmente, lo que significa que se necesitan recursos adicionales para aumentar la capacidad.
- El desarrollo de nuevas características puede ser más difícil ya que los diferentes componentes están interconectados.
- La aplicación completa puede verse afectada si se produce un fallo en cualquier componente.

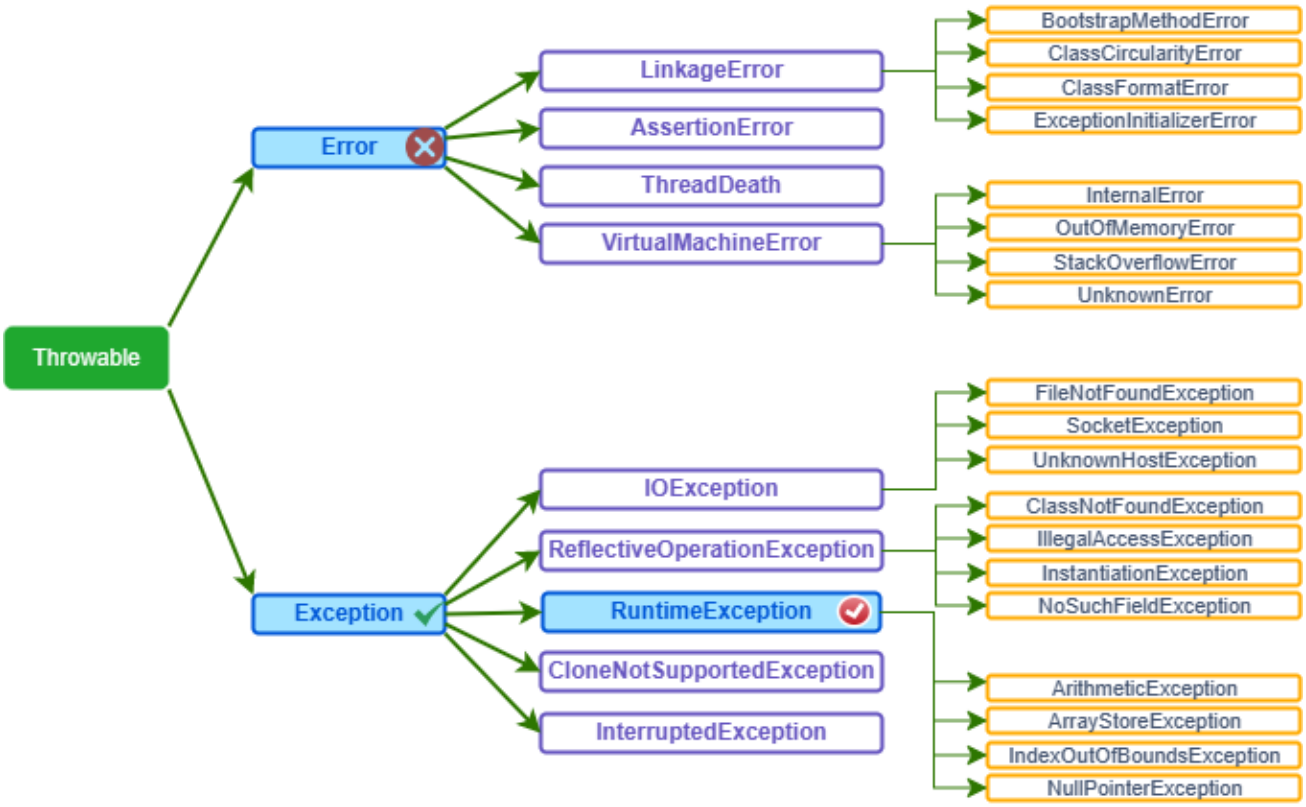
Sin embargo, las aplicaciones monolíticas tienen como desventaja:

- La gestión de la comunicación entre los diferentes servicios puede ser más compleja y puede requerir una mayor coordinación entre los equipos de desarrollo.
- Puede ser más difícil de depurar y monitorear ya que la aplicación completa es una combinación de muchos servicios diferentes.
- El rendimiento puede ser más lento debido a la necesidad de comunicarse a través de la red.

Tipos de Excepciones (Exceptions)

Por defecto Java nos permite tener diferentes excepciones para los diferentes tipos de errores en los cuales pueda caer nuestra aplicación. La clase Throwable es la clase que agrupa todas las excepciones que podemos utilizar en nuestra aplicación hecha en Java e inclusive si queremos crear nuestro propio tipo de excepción podemos hacerlo al heredar de la clase Exception o de alguna de sus sub-clases.

En java existen 3 grandes tipos de excepciones para manejar los errores en un programa. Estos 3 tipos se denominan: errores (errors), las excepciones verificadas (checked exceptions) y las excepciones no verificadas (unchecked exceptions).



Errores (errors): Los errores son excepciones que ocurren durante la ejecución de una aplicación y que no pueden ser manejados por la aplicación. Estos errores son de naturaleza grave y suelen estar relacionados con problemas en el entorno de ejecución de la aplicación, como falta de memoria o problemas de hardware. Este tipo de excepciones son arrojadas cuando ocurren por la Java Virtual Machine o JVM, y están comprendidas dentro de la clase Error, este tipo de errores ocurre con muy poca frecuencia y no podemos hacer mucho más que informar al usuario y terminar el programa. Por ejemplo:

```java
public class EjemploError {
  public static void main(String[] args) {
    // crea un arreglo de tamaño máximo para causar un OutOfMemoryError
    int[] arr = new int[Integer.MAX_VALUE];
  }
}
```

Excepciones verificadas (Checked Exceptions): Las excepciones verificadas son aquellas que el compilador de Java obliga a ser capturadas o declaradas en el método que las lanza, lo que significa que el programador debe tomar medidas explícitas para manejar la excepción. Estas excepciones suelen estar relacionadas con condiciones externas a la aplicación que pueden afectar su funcionamiento normal. Regularmente estas excepciones heredan de la clase Exception o de aluna de sus clases hijas. Por ejemplo:

```java
public class EjemploChecked {
  public static void main(String[] args) {
    // intenta abrir un archivo que no existe para causar un FileNotFoundException
    BufferedReader br = new BufferedReader(new FileReader("archivo_inexistente.txt"));
  }
}
```

Excepciones no verificadas (Unchecked Exceptions): Las excepciones no verificadas son excepciones que no están obligadas a ser capturadas o declaradas por el compilador de Java. Estas excepciones suelen estar relacionadas con errores de programación o violaciones de las reglas de Java. Estas excepciones heredan de la clase RuntimeException o de alguna de sus clases hijas. Por ejemplo:

```java
public class EjemploUnchecked {
  public static void main(String[] args) {
    // divide un número por cero para causar un ArithmeticException
    int num = 10/0;
  }
}
```

Try-with-resources

Try-with-resources es una construcción de lenguaje introducida en Java 7 que facilita el manejo de recursos que deben cerrarse después de su uso, como flujos de entrada/salida, conexiones a bases de datos, etc. Esta construcción proporciona una forma concisa y segura de escribir código que gestiona recursos que deben cerrarse.

Si no cerramos los recursos, puede constituir una fuga de recursos y también el programa podría agotar los recursos disponibles para él. En lugar de utilizar bloques try-catch-finally para cerrar los recursos, un bloque try-with-resources cierra automáticamente los recursos declarados en la cláusula "try" cuando el bloque finaliza. El recurso que se declara en el bloque try-with-resources debe implementar la interfaz AutoCloseable.

Cuando se trata de excepciones, hay una diferencia en el bloque try-catch-finally y el bloque try-with-resources. Si se produce una excepción tanto en try como en finally, el método devuelve la excepción lanzada en finally.

Para try-with-resources, si se produce una excepción en un bloque try y en una instrucción try-with-resources, el método devuelve la excepción lanzada en el bloque try. Las excepciones lanzadas por try-with-resources se suprimen, es decir, podemos decir que try-with-resources block arroja excepciones suprimidas.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class EjemploTryWithResources {
   public static void main(String[] args) {
      try (BufferedReader br = new BufferedReader(new FileReader("archivo.txt"))) {
         String linea;
         while ((linea = br.readLine()) != null) {
            System.out.println(linea);
         }
      } catch (IOException e) {
         System.out.println("Error al leer el archivo: " + e.getMessage());
      }
   }
}
```

Multi-catch

El multicatch (múltiples capturas) es una construcción de lenguaje introducida en Java 7 que permite capturar múltiples excepciones en un solo bloque catch. Esto simplifica el código al reducir la cantidad de bloques catch necesarios.

Para utilizar el multicatch, se especifican las excepciones que se desean capturar separándolas con una barra vertical (|) en la cláusula catch. Es importante tomar en cuenta que, aunque el multicatch puede simplificar el código, también puede hacerlo menos específico. Por lo tanto, se recomienda utilizar el multicatch solo cuando las excepciones capturadas tienen un manejo similar o cuando no es posible manejar las excepciones de manera diferente.

```
public class EjemploMultiCatch {
  public static void main(String[] args) {
    try {
      // código que puede lanzar dos excepciones diferentes
      String cadena = null;
      System.out.println(cadena.length());
      Class.forName("com.mysql.jdbc.Driver");
    } catch (NullPointerException | ClassNotFoundException e) {
      // manejo de excepciones para ambos casos
      System.out.println("Ocurrió una excepción: " + e.getMessage());
    }
  }
}
```