Examen 2. Academia Java

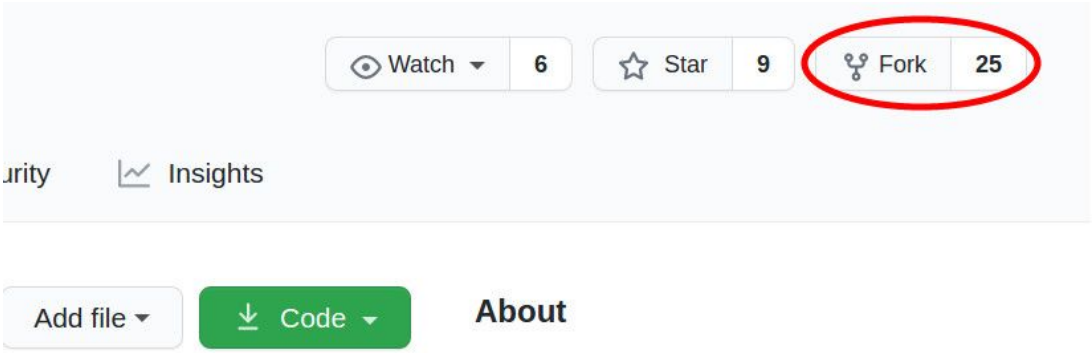
Martha Paola Peña Sotelo

▼ Comandos Git

Fork

Cuando nos gusta el repositorio de alguien y nos gustaría tenerlo en nuestra cuenta de GitHub, hacemos un **fork** o bifurcación para poder trabajar con él en forma separada. Cuando hacemos un fork de un repositorio, obtenemos una instancia de todo el repositorio con todo su historial. Luego, podemos hacer lo que queramos sin afectar la versión original.

💡 Haciendo click en el botón de **fork** se crea una instancia del repositorio completo en tu cuenta.

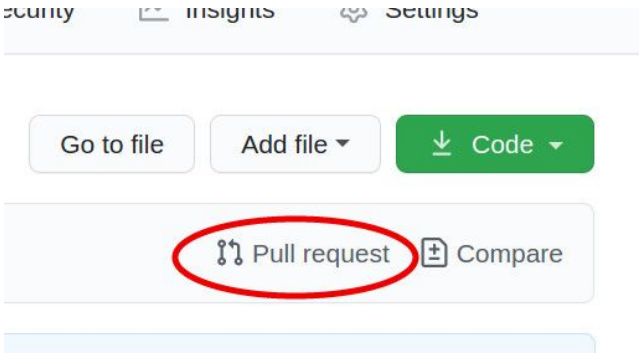


pull request

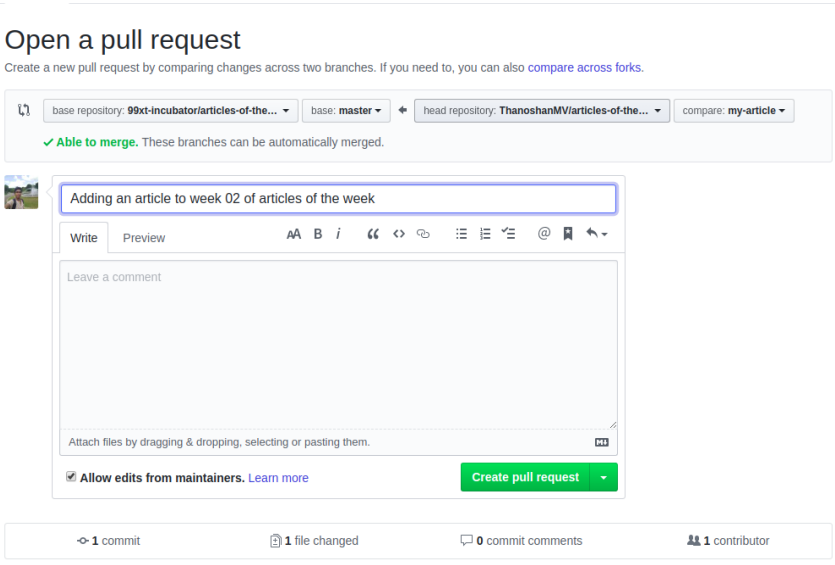
Los **pull requests** son la forma de contribuir a un proyecto grupal o de código abierto. Para realizar esto primero se debe hacer un fork y clonar el repositorio al que se quiere contribuir.

Una vez realizados los cambios que queremos agregar al proyecto nos vamos a GitHub para hacer el pull request.

Hacer click en **pull request** mandará a una ventana de descripción de los cambios



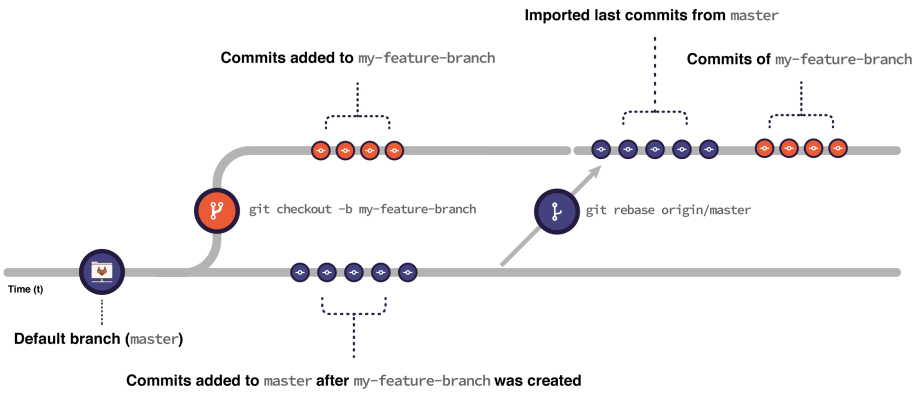
Se debe de hacer una descripción de los cambios realizados antes de mandar el pull request. Al mandar el pull request se envía una solicitud al creador del repositorio, quien decidirá si acepta o no los cambios efectuados



rebase

Cuando tenemos una rama master y una rama secundaria podemos unirlas mediante un merge, sin embargo el merge aunque une ambas ramas almacena los comits hechos en

la rama secundaria, lo cual a la larga vuelve engorroso el control de versiones, para evitar eso podemos utilizar el rebase. Lo que hace **rebase** es igualar la rama master con la rama secundaria y volver a poner los commits encima para proceder con un merge fast forward, de esta forma tendremos una sola rama con los commits almacenados en ella, mientras que la rama secundaria desaparecerá.



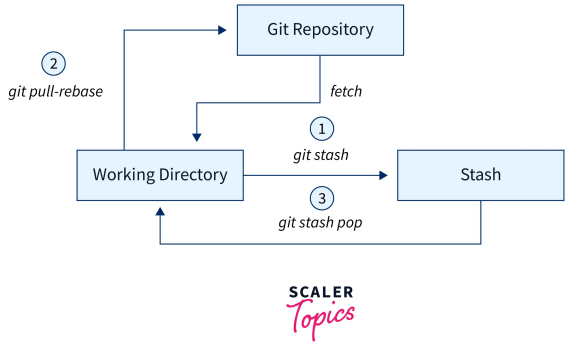
clean

El comando **git clean** opera en archivos sin seguimiento. Los archivos sin seguimiento son archivos que se han creado en el directorio de trabajo del repositorio, pero que no se han añadido al índice de seguimiento del repositorio con git add. Cuando se ejecuta el comando git clean, no se puede deshacer. Cuando se ejecuta por completo, git clean hará una eliminación permanente del sistema de archivos,

stash

El comando `git stash` almacena temporalmente (o guarda en un *stash*) los cambios que hayas efectuado en el código en el que estás trabajando para que puedas trabajar en otra cosa y, más tarde, regresar y volver a aplicar los cambios más tarde. Guardar los cambios en stashes resulta práctico si tienes que cambiar rápidamente de contexto y ponerte con otra cosa, pero estás en medio de un cambio en el código y no lo tienes todo listo para confirmar los cambios.

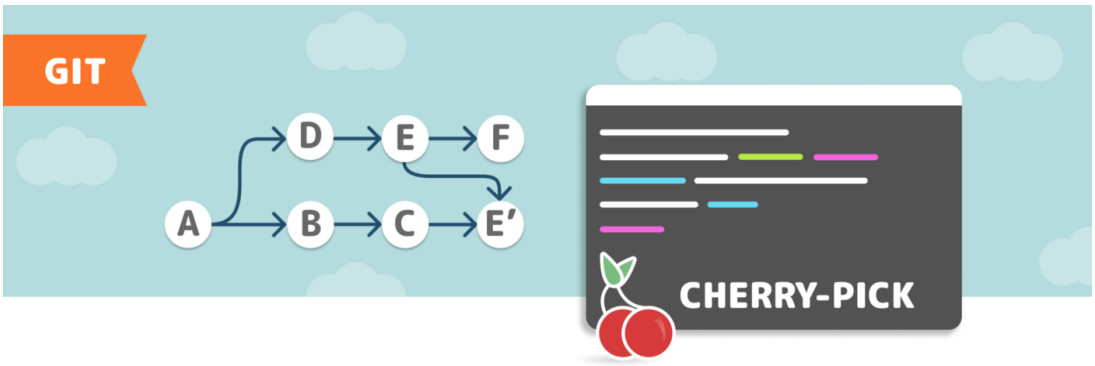
> 💡 El comando `git stash` toma los cambios sin confirmar, los guarda aparte para usarlos más adelante y, acto seguido, los deshace en el código en el que estás trabajando



Para volver a aplicar los cambios de un stash se usa el comando `git stash pop`

cherry pick

Permite aplicar commits de una rama en otra, también puede ser útil para deshacer cambios. Por ejemplo, supongamos que una confirmación se realiza accidentalmente en la rama incorrecta. Puede cambiar a la rama correcta y seleccionar la confirmación donde debería pertenecer.



▼ *Aplicaciones monolíticas vs M*icroservicios

Microservicios:

Todos los componentes de la aplicación es parte de una única unidad, es decir, todas las funcionalidades de una aplicación se encuentran un solo código base: una sola **aplicación monolítica.**

En una aplicación monolítica todo se desarrolla, implementa y escala como 1 unidad. Lo cuál significa que la aplicación debe ser escrita en un solo lenguaje con una sola tecnología en un solo tiempo de ejecución.

Si se trabaja en equipo en esa aplicación se debe tener cuidado de que el trabajo de unos no afecte el trabajo de otros.

▼ Desventajas

La coordinación entre el equipo es difícil

La escalabilidad es para toda la aplicación lo cual resulta más caro.

El tiempo de liberación es más largo

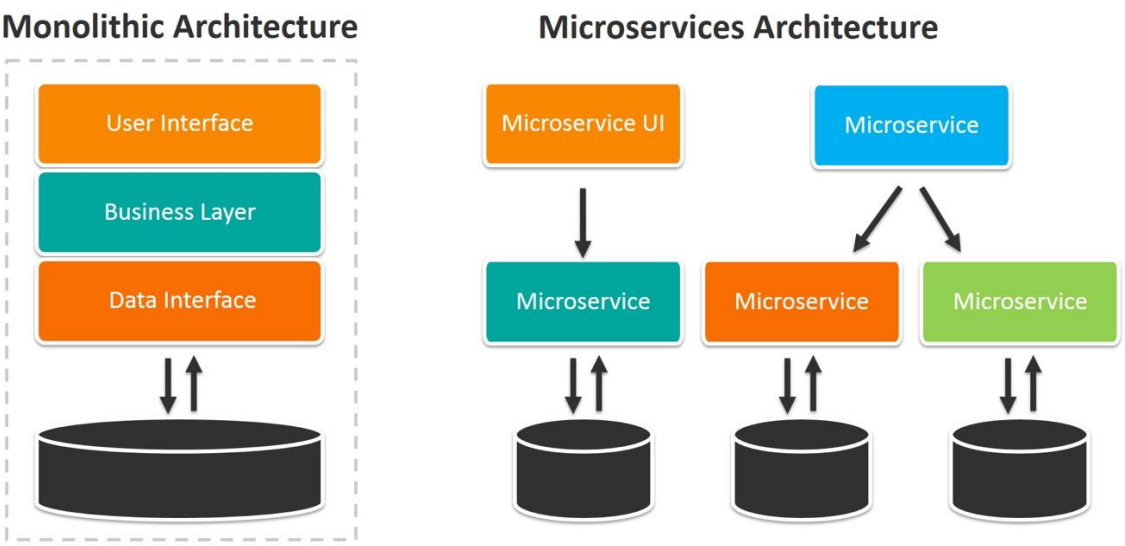
Se tiene que testear cada cambio para aplicarlo

Microservicios

Se divide la aplicación en múltiples y pequeñas aplicaciones. Se trata de una gran aplicación compuesta de muchas micro aplicaciones basadas en las funcionalidades del negocio y no en las funcionalidades técnicas. Cada microservicio tiene una única funcionalidad.

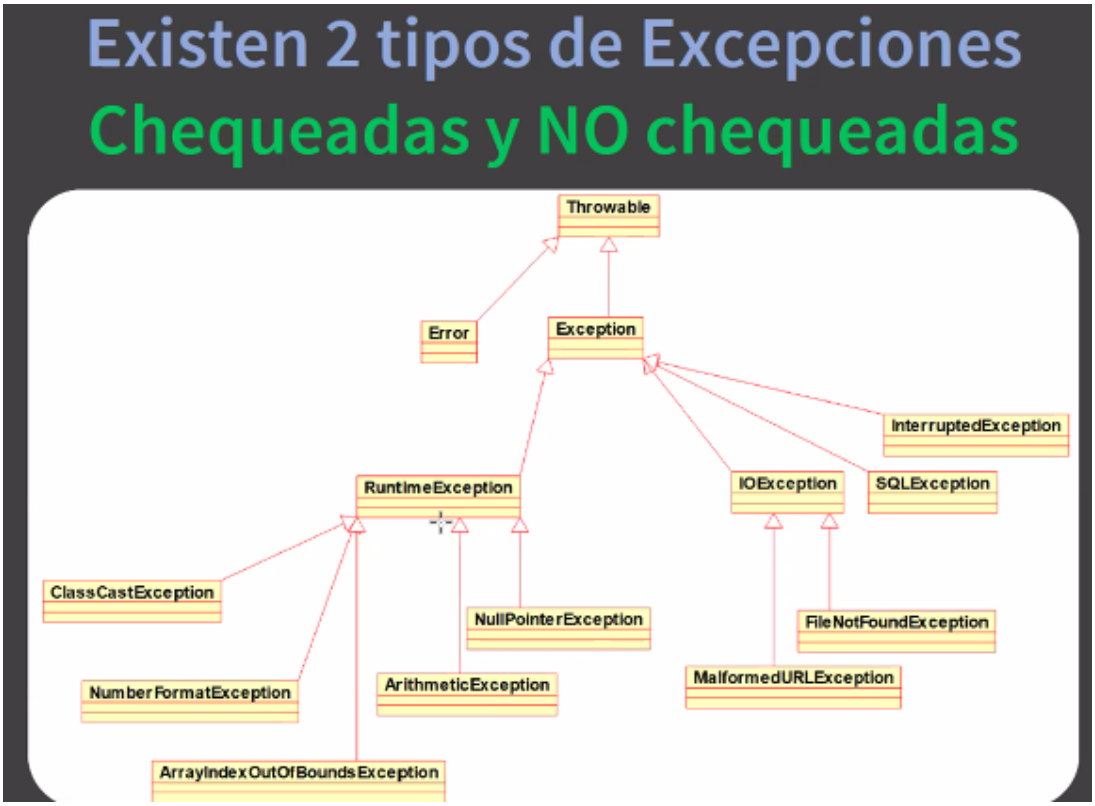
Los **microservicios** tienen que independientes uno de otro, significa que cada uno se desarrolla, implementa y escala por separado y sin depender de los otros. Así los cambios implementados en uno, no afectan a los demás. Cada uno puede ser desarrollado en diferente lenguaje y con distintas tecnologías de manera independiente sin afectar a otros.

La forma en la que se comunican estos microservicios es por medio de llamadas a la API, o bien por medio de kubernetes en el cual un servicio externo se encargue únicamente de la comunicación entre microservicios.



▼ Excepciones

Las excepciones son problemas o eventos que ocurren durante la ejecución de un programa que interrumpen el flujo normal.



- Error: Se refiere a errores graves en la máquina virtual de Java, como por ejemplo fallos al enlazar con alguna librería. Normalmente en los programas Java no se tratarán este tipo de errores.
- **Checked exception:** este tipo de excepciones heredan directamente de la clase Exception, no dependen directamente del programador, tienen que ver con las entradas y salidas de los programas así que el compilador siempre nos obliga a manejarlas
    - **IOException:** significa que se ha producido un error en la entrada/salida. Es la clase base para excepciones que se producen mientras se tiene acceso a la información mediante secuencias, archivos y directorios.
    - SQLException: se producen mientras se está accediendo a un origen de datos, se lanza cuando hay algún problema entre la base de datos y el programa Java JDBC.
    - InterruptedException: Se lanza cuando un subproceso está esperando, durmiendo u ocupado de otro modo, y el subproceso se interrumpe, ya sea antes o durante la actividad.
- **Unchecked exception:** se trata de errores en la programación que se producen al momento de ejecución y el compilador no obliga a manejar. Este tipo de excepciones heredan de la clase RuntimeException.
    - NullPointerException: ocurre al declarar una variable y no crear el objeto. Se obtiene si se intenta referenciar la variable antes de crear el objeto.
    - ArithmeticException: se produce para los errores de una operación aritmética o de conversión.
    - NumberFormatException: Lanzado para indicar que la aplicación ha intentado convertir una cadena a uno de los tipos numéricos, pero que la cadena no tiene el formato adecuado.

💡 Para arreglar los programas de errores de ejecución, necesitamos especificar una lista de excepciones usando **throws**, o necesitamos usar un bloque **try-catch.**

Declaración de las excepciones

Para tratar con excepciones de tipo **io exception** tenemos dos opciones: o declaramos la excepción que se puede dar en el método o la capturamos con un try catch

1. Declarar la excepción

Utilizamos la palabra **throws** para indicarle a JAVA que en el método es muy probable que exista una excepción de tal tipo

```
public class PruebaExcepciones {
    //Declarado con la clase inmediata
    public void leerArchivo() throws FileNotFoundException {
        File archivo = new File("C:\\Users\\LENOVO\\Desktop\\empleabilidad.txt");
        FileReader fr = new FileReader(archivo);
    } //Utilizamos este primer método dentro del siguiente:

    //declarado con la superclase
    public void leerArchivo2() throws IOException {
        leerArchivo();
    }


    public static void main(String[] args) {
    }
}
```

2. Utilizar try catch

Sintaxis:

```
try{
    [bloque que lanza la excepcion...Se pone todo aquello que produce la excepción]
} catch (Exeption ex) {
    [acá manejamos el error... se puede poner un dialogo que indique el error y cómo proceder]
} catch (Exeption ex) {
    [Se pueden poner varias excepciones en este caso la clase Exception se pone al final]
} finally {
    [bloque opcional, siempre se ejecuta sin importar que los catch se ejecuten o no]
}
```

▼ Try-with-resource y multicatch

1. Try with resource: se utiliza con el objetivo de cerrar los recursos de forma automática en la sentencia *try-catch-finally* y hacer más simple el código. Aquellas variables cuyas clases implementan la interfaz AutoCloseable pueden declararse en

el bloque de inicialización de la sentencia *try-with-resources* y sus métodos close() serán llamados después del bloque *finally* como si su código estuviese de forma explícita.

Ejemplo:

```
//método para leer la primera linea de un archivo, puede lanzar una IOException
public static String readFirstLineFromFile(String path) throws IOException {
    //se crea un objeto BufferedReader utilizando un objeto FileReader,
    //que se utiliza para leer el archivo especificado por la ruta "path".
    //El objeto BufferedReader se crea como parámetro del "try-with-resources",
    //lo que significa que se cerrará automáticamente al final del bloque "try".
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine(); //ejecuta el método de lectura
    }
}
//Al final del bloque "try", el objeto BufferedReader se cerrará
//automáticamente debido al "try-with-resources"
```

2. Multicatch: Sirve para meter las excepciones en el mismo bloque de código

   Condiciones:

   - Solo se pueden emplear cuando todas las excepciones les vamos a dar el mismo tratamiento
   - Las excepciones no deben ser padres e hijas entre sí

```
try {
    resultado = calculaDiv(x,y);
  //Multicatch:
    //lanza CeroException, NegativoException y UnsopportedOperationException
    } catch (CeroException | NegativoException | UnsupportedOperationException e) {

        e.printStackTrace(); //esta linea es para pintar la excepcion
    }
```

▼ MVC

MVC (Modelo-Vista-Controlador) es un patrón en el diseño de software utilizado para implementar interfaces de usuario, datos y lógica de control. Enfatiza una separación entre la lógica de negocios y su visualización.

Las tres partes del patrón de diseño de software MVC se pueden describir de la siguiente manera:

1. Modelo: Maneja datos y lógica de negocios. El modelo define qué datos debe contener la aplicación. Si el estado de estos datos cambia, el modelo generalmente notificará a la vista (para que la pantalla pueda cambiar según sea necesario) y, a veces, el controlador (si se necesita una lógica diferente para controlar la vista actualizada).
2. Vista: Se encarga del diseño y presentación. La vista define cómo se deben mostrar los datos de la aplicación.
3. Controlador: Enruta comandos a los modelos y vistas. El controlador contiene una lógica que actualiza el modelo y/o vista en respuesta a las entradas de los usuarios de la aplicación.