

MD.120 – MANUAL DE USUARIO GITHUB EN DESPEGAR

GitHub en Despegar

Autor:	Sebastián Piccoli
Fecha de Creación:	05 de Agosto de 2016
Última Actualización:	
Ref. Documento:	MD120_GitHub.doc
Versión:	1.0



Control de Documento

Registro de Cambios

Fecha	Autor	Versión	Referencia de cambio
05-Ago-16	Sebastián Piccoli	1.0	Edición Inicial

Revisores

Nombre	Puesto

Contenidos

Control de Documento	ii
Registro de Cambios.....	ii
Revisores	ii
Resumen.....	1
Primeros pasos	2
Inicializando el repositorio	2
Creación de branches	2
Actualizando el repositorio	3
Subida de cambios al repositorio local	3
Subida de cambios al repositorio remoto	4
Otros comandos útiles.....	4
Bloqueo y desbloqueo de archivos	6
Flujo de trabajo.....	6
Bloqueo de archivos	6
Desbloqueo de archivos	7
Verificación de archivos a subir.....	7
Propagación de novedades al master.....	8
Pull requests	8
Control cruzado	10
Armado de parches	11
Scripts para armado de parches.....	11
Versionado.....	11

Resumen

Este documento describe la metodología de trabajo que se utilizará para trabajar con GitHub en Despegar, detallando procedimientos y flujos de trabajo esperados.

Si bien en principio es una tecnología con pocos antecedentes dentro del mundo Oracle, creemos que hay muchas ventajas que podemos aprovechar al elegir esta herramienta, entre las que podemos mencionar:

- Estandarizar las herramientas con respecto a la que usan el resto de los sectores de Despegar.
- Contar con un sistema de versionado, hoy inexistente.
- Poder trabajar con branches personales (e incluso por desarrollo) que permitan independizar la propagación de novedades al branch productivo.
- Trazabilidad JIRA-Git.
- Redundancia.

Se trabajará acoplando el uso de esta tecnología a un sistema de lockeo de archivos hecho a medida, y a un sistema de creación de parches para realizar los deploys (metodología aceptada por Oracle) que mantenga trazabilidad con el repositorio.



Nota: Se asume que en la instancia de trabajo se tendrá instalado GitBash, que posee un intérprete de bash incorporado (en cualquier SO de Windows o distribución de Unix).

También se asume que el usuario ya ha sido dado de alta en la organización Despegar en Git y en el Team de OracleEBS dentro de la misma.

Primeros pasos

Inicializando el repositorio

Lo primero que deberemos hacer es clonar el repositorio remoto en nuestra computadora de trabajo. Para eso, primero deberemos tener bajado GitBash como intérprete, que se obtiene al instalar [Git](#).

Una vez instalado, abrimos la consola y configuramos nuestro usuario de Git:

- `git config --global user.name "<nombre_usuario>"`
- `git config --global user.email <mail_usuario>`

La autenticación será automática si hemos gestionado las credenciales cuando asociamos nuestro usuario en la organización Despegar en Git.

Luego de esto, nos posicionamos en el directorio en donde querramos bajar [nuestro repositorio](#), y lo clonamos:

- `git clone https://github.com/despegar/OracleEBS.git`

Una vez clonado, debemos acceder a la carpeta del repositorio (`cd OracleEBS`). Veremos algo así:

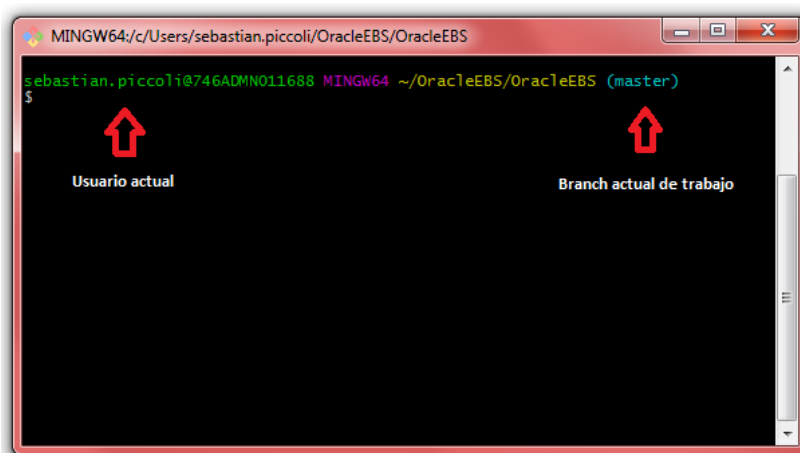


Figura 1. Posicionamiento dentro del repositorio


Creación de branches

Se ha decidido trabajar con un esquema de branches separados por usuario, tanto local como remoto. El flujo de subida/bajada será de 1 a 1. El usuario puede crear branches auxiliares si así lo quiere (para independizar desarrollos). Para crear un nuevo branch, posicionados dentro de la carpeta del repositorio, ejecutamos:

- `git checkout -b <nombre_del_branch>`

Esto nos va a crear un nuevo branch en forma local. Luego de hacer esto, tenemos que definir el flujo de subida del branch en el repositorio remoto:

➤ **git push --set-upstream origin <nombre_del_branch>**

 **Nota:** siempre que utilizamos origin en algún comando, estamos especificando el branch actual del repositorio local (el branch en el que estamos trabajando).

Actualizando el repositorio

Para actualizar las novedades del repositorio remoto al local haciendo:

➤ **git pull**

También podemos actualizar nuestro branch local trayendo las novedades del master (PROD):

➤ **git pull origin master**

Subida de cambios al repositorio local

Para que git sepa que queremos subir los cambios relacionados a determinado archivo (es decir, trackear el archivo), debo indicarlo mediante la sentencia:

➤ **git add <archivo>**

Esto debo hacerlo nuevamente cada vez que hacemos un commit. Puedo agregar todos los archivos que se han modificando especificando git add *. Sin embargo, una vez realizado esto, es recomendable revisar qué cambios se están trackeando, usando la sentencia:

➤ **git status**

Siempre puedo desmarcar un archivo que no quiero subir (antes de hacer el commit):

➤ **git reset <archivo>**

Para propagar los cambios al repositorio remoto, hago:

➤ **git commit -m "<mensaje_del_commit>"**

Subida de cambios al repositorio remoto

Subir los cambios al repositorio local no los propaga al remoto. Para eso es necesario realizar un push. El push envía al repositorio remoto los commits aún no propagados (todos los realizados desde el último push). Debo hacer un push cuando tengo el desarrollo listo para ser probado en TEST (es el momento en el cual voy a armar el parche para el deploy). Ejecutamos:

- **git push**

Otros comandos útiles

Para cambiar de branch de trabajo:

- **git checkout <nombre_branch>**

Para ver el historial del branch:

- **git log**

Para ver las diferencias entre el master y un branch en particular:

- **git diff --summary master <branch>**

Si quiero revertir cambios en mi repositorio local aún no publicados al remoto (commit realizado pero sin push):

- **git reset --hard HEAD~<N>** (siendo N el número de commits a deshacer)

Si quiero pisar la versión de mi archivo local por la del remoto (si aún no hice commit):

- **git checkout -- <archivo>**

Para crear tags:

- **git tag "<nombre del tag>"**

Para publicar tags al servidor remoto:

- **git push origin "<nombre del tag>"**

Para pisar lo que tiene mi branch con lo que tiene el master:

- **git merge master** (Ojo: esto sólo hace el merge de las copias locales, luego hay que realizar el push)

Bloqueo y desbloqueo de archivos

Flujo de trabajo

Se definió un sistema para bloqueo y desbloqueo de archivos, de manera tal que se evite que dos usuarios trabajen sobre el mismo fuente, generando conflictos a la hora de pasar el desarrollo a producción. Esto se tiene en cuenta principalmente por los objetos binarios (forms y reports, sobre todo), en los que no tenemos la posibilidad de hacer un merge. Dado que este es un desarrollo adicional, externo a git, no va a imposibilitar absolutamente nada, sino que servirá como herramienta de control. El desarrollo se compone de 3 scripts de bash, que serán descriptos a continuación.

El flujo de trabajo esperado es:

1. Al empezar un nuevo desarrollo, bajo las últimas novedades del master. Para que el master no pise los cambios de mi branch de trabajo (si estoy realizando desarrollos en paralelo), debo tener todos mis cambios commiteados. En caso de no poder hacerlo, se sugiere crear un nuevo branch de trabajo para el nuevo desarrollo. ¡Ojo! El nuevo branch se debe crear **desde master**:
 - **git checkout master**
 - **git checkout -b <nuevo_branch>**
 - **git push --set-upstream <nuevo_branch>**
2. Bloqueo el/los archivo/s que voy a estar modificando en mi desarrollo.
3. Subo el nuevo archivo .gitlocks a **master**. Esto lo realizaremos directamente desde el sitio del repositorio (ver figura 2).
4. Desarrollo.
5. Cuando quiero realizar un commit, verifico los archivos a subir.
6. En cuanto mi desarrollo está listo para ser pasado a PROD, o si el mismo es desestimado, **bajo nuevamente** el archivo .gitlocks de master. Es importante bajarlo nuevamente para contemplar cambios que hubiesen realizado otros usuarios.
7. Desbloqueo el archivo.
8. Subo el nuevo archivo .gitlocks a master, de la misma manera que se hizo en el punto 3.

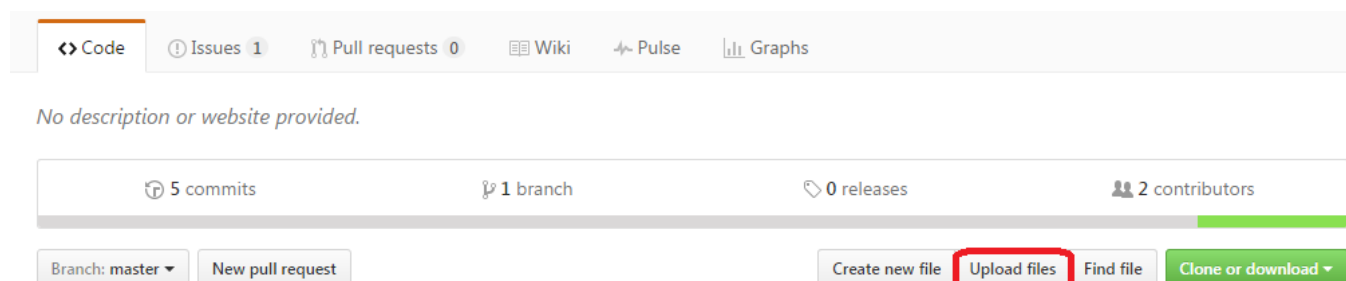


Figura 2. Subida de archivos desde el site del repositorio.

Bloqueo de archivos

Para eso debemos utilizar el script lock.sh. El uso del mismo es el siguiente (posicionados en la carpeta base del repositorio):

- **./lock.sh <archivo_a_bloquear>**

El archivo debe indicarse con la ruta completa desde el directorio en donde estamos parado. La salida puede ser:

- **Se ha agregado el lock sobre el archivo <archivo_a_bloquear>.**
- **Usted ya tiene el archivo lockeado.**
- **El archivo ya se encuentra lockeado por <otro_usuario>.**

En los primeros dos casos, podré trabajar con el archivo, ya que lo tendré bloqueado (en el segundo caso ya lo tenía bloqueado desde antes). Una vez bloqueado debo actualizar el archivo .gitlocks en el master.

En caso de obtener el tercer mensaje, debo ponerme en contacto con el usuario indicado.

Desbloqueo de archivos

Para desbloquear archivos uso el script unloc.sh, de la forma:

- **./unlock.sh <archivo_a_desbloquear>**

El archivo debe indicarse con la ruta completa desde el directorio en donde estamos parado. La salida puede ser:

- **Usted no tiene el archivo lockeado.**
- **Se ha liberado el lock.**

En el primer caso, no tengo el archivo bloqueado con mi usuario, y por tanto no puedo liberar el bloqueo. En el segundo, el archivo se desbloquea. Una vez desbloqueado debo actualizar el archivo .gitlocks en el master.

Verificación de archivos a subir

Se realiza con el script check_commit.sh:

- **./check_commit.sh**

Este script verifica todos los archivos que tengo marcados para subir en mi próximo commit, y me informa si hay alguno que no tengo bloqueado. La salida puede ser:

- **Se puede subir el archivo <archivo_a_commitear>.**
- **Debe lockear el archivo <archivo_a_commitear> antes de realizar un commit.**
- **El archivo <archivo_a_commitear> se encuentra lockeado por <otro_usuario>. No debe subirlo al repositorio.**

En el primer caso, puedo subir el archivo sin problemas, ya que lo tengo bloqueado previamente. En el segundo, debo actualizar el .gitlocks de master e intentar bloquearlo, en caso de poder hacerlo puedo subir el archivo, si descubro que otro usuario lo tiene bloqueado, deberé ponerme en contacto con él. En el tercer caso, debo ponerme en contacto con el usuario indicado.

Propagación de novedades al master

Pull requests

La propagación de novedades de un branch de trabajo hacia el master se debe hacer en el momento en el que el desarrollo será impactado en PROD. Este pasaje se hará utilizando pull requests. Para eso, debemos ingresar al site del repositorio, y seguir los pasos que se indican a continuación:

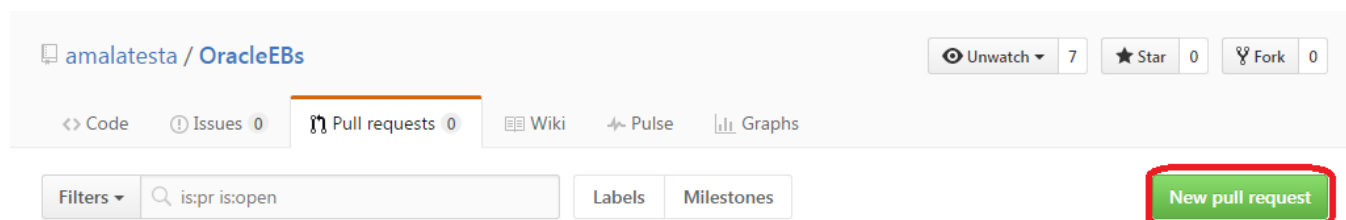


Figura 3. Creo un nuevo pull request.

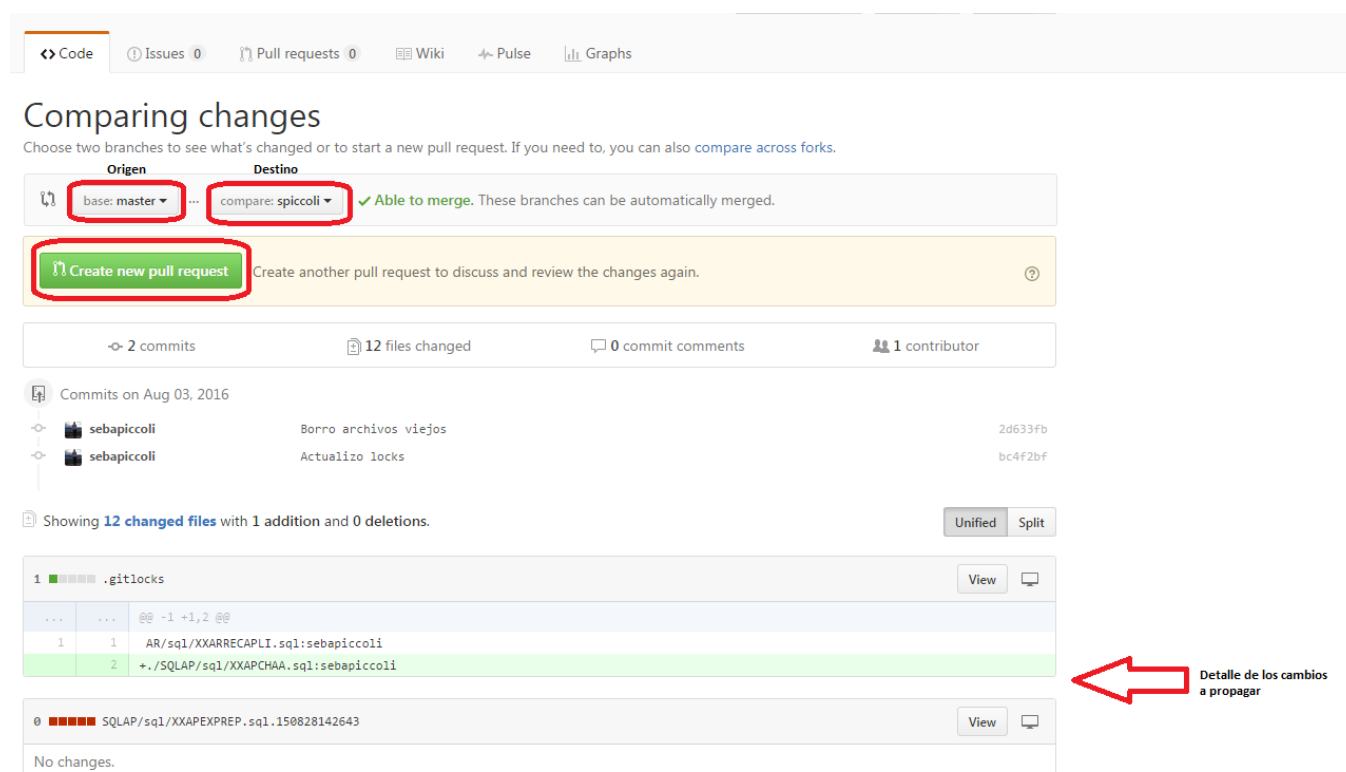


Figura 4. Indico el origen y el destino del merge a realizar.

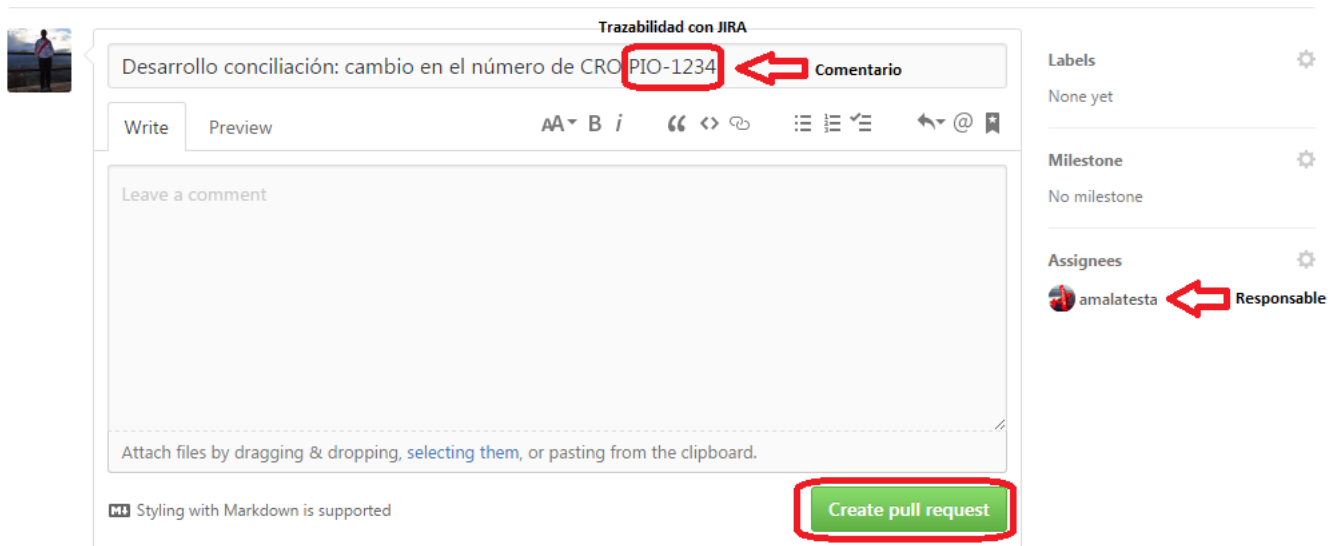


Figura 5. Confirmando la creación del pull request.

Una vez creado el pull request, se debe confirmar su ejecución:

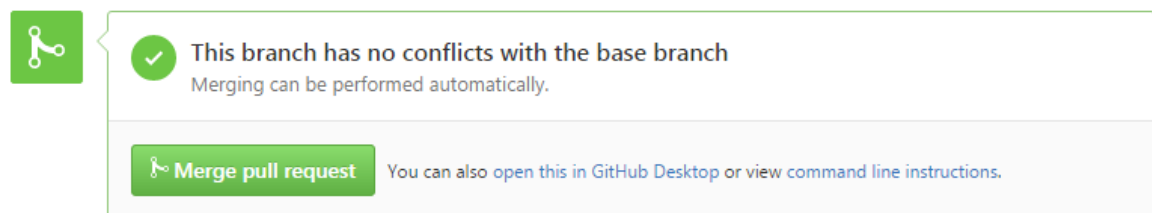


Figura 6. No hay conflictos. Puedo realizar el merge automáticamente.

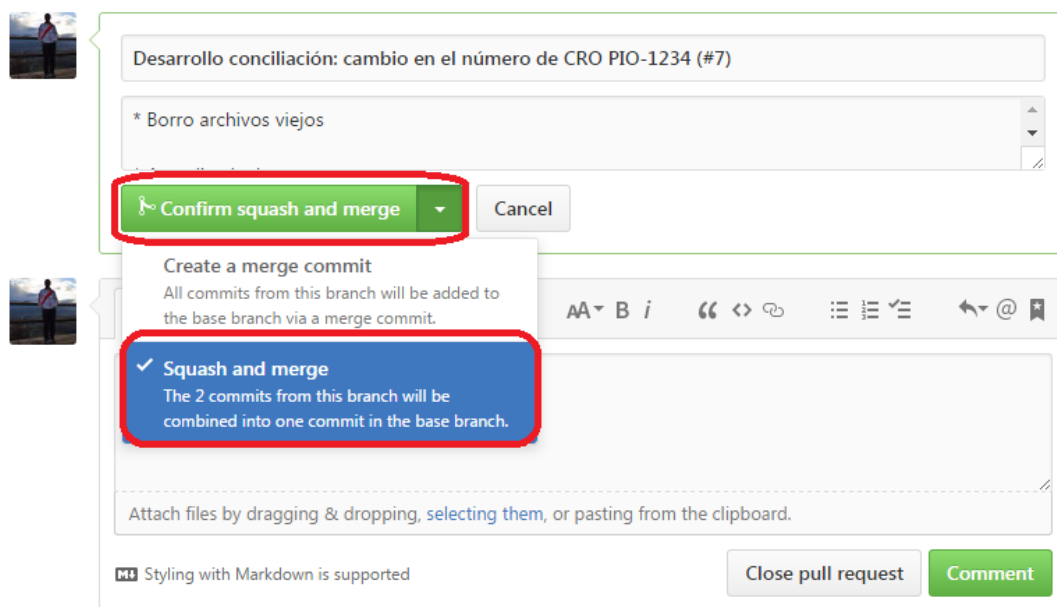


Figura 7. Indico que se agrupen todos los commits en uno.

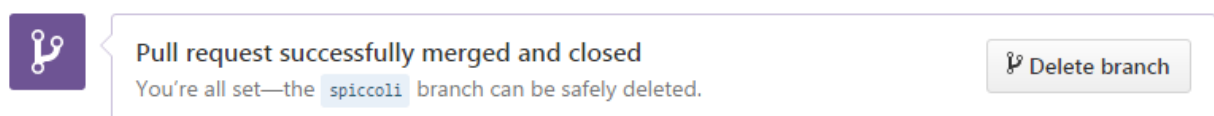


Figura 8. Confirmación del merge.

Control cruzado

Para un mejor control de concurrencia, es deseable asignar mi pull request a otro usuario, y que sea responsable del pasaje al master.


Armado de parches

Scripts para armado de parches

- Script de bajada de objetos y armado del parche:

Este script se encarga de:

1. Ir a la revisión del último commit afectado para la creación del parche (el commit en el que finalicé el desarrollo):
- **git checkout <id_del_commit>**
2. Armar la estructura de directorios.
3. Copiar los objetos del repositorio dentro del parche.
4. Bajar, mediante FNDLOAD, las definiciones no versionadas.
5. Empaquetar el parche.

 **Nota:** Puedo obtener el id de un commit si, una vez realizado, ejecuto el comando:

- **git log -1 --format="%H"**

- Script de instalación del parche:

Es el driver de instalación tal cual se utiliza hoy en día.

Se adjunta un ejemplo de ambos scripts:

Ejemplo de script de bajada de objetos y armado del parche:



buildpatch_XXAPCANT_db.sh

Ejemplo de script de instalación:



XXAPCANT_db.drv

Versionado

No se versionarán los parches, sino sólo los scripts para bajada de objetos y armado del mismo y el script de instalación. Estos estarán en la carpeta scripts del repositorio.

Con el script de bajada de objetos y armado del parche se puede replicar en cualquier momento el parche, respetando las versiones originales de los objetos al momento de creación del mismo.