Guia de entregas de código fuente en la DGSIC

Dirección General de Sistemas de Información y Comunicaciones

Áreas de Gobierno Tecnológico de SSII y del Dato

Servicio Andaluz de Salud
Consejería de Sanidad, Presidencia y Emergencias

Contenido

Resumen

- **Versión:** v01r57
- **Fecha publicación:** 4 oct 2017
- **Entrada en vigor desde:** 4 oct 2017

Guia de desarrollo

El documento actual es una guía. Aunque no es de obligado cumplimiento, enfatizamos la necesidad de seguir las recomendaciones dadas para un correcto encaje entre sus métodos de trabajo y las normativas a las que afecta.

Si desea realizar algún aporte a esta guía o ha encontrado alguna errata, por favor remítala al siguiente contacto.

Contacto Arquitectura: l-arquitectura.stic@juntadeandalucia.es

Histórico de cambios

Versión	v01r56	Fecha publicación	4 oct 2017	Fecha entrada en vigor	4 oct 2017

Alcance
• Versión inicial sobre normativa de git

Versión	v01r57	Fecha publicación	3 may 2021	Fecha entrada en vigor	4 oct 2017
Alcance					
• Añadir comandos gitflows para la creación de HU dentro mejoras					

2. Guía de trabajo

2.1 Preparación del entorno de trabajo

Herramientas

Configuración en el equipo

Git for Windows: https://git-scm.com/download/win

Git for Windows ya incluye todos los comandos necesarios para trabajar con el workflow de GitFlow desde la versión 2.5.3.

Establecer la opción feature.finish.no-ff a true, lo que permitirá que gitflow cree un commit nuevo cada vez que se realice un merge independientemente de si se puede o no aplicar fast-forward. Esto facilita la revisión del histórico, y agiliza tareas de mantenimiento que requieran excluir features completas.

Para activar el parámetro no-ff basta con ejecutar el siguiente comando por consola y se aplicaría a todos los proyectos:

git config --global gitflow.feature.finish.no-ff TRUE

Si se trabaja con proxys será necesario realizar configuración adicional creando las variables de entorno "HTTP_PROXY" y "HTTPS_PROXY", también se podrán excluir direcciones con la variable global NO_PROXY

Prerequisitos

- Se debe estar en la red corporativa.
- Se debe de tener acceso al repositorio del SAS.
- El usuario que se use para la entrega tiene que tener los permisos correspondientes en el repositorio.

El flujo para realizar las entregas en el repositorio del SAS es el siguiente:

a. El usuario realizara una copia local del repositorio de desarrollo con la opción --mirror

```
$ git clone --mirror {url del repositorio de desarrollo}
```

- Si ya se tenia anteriormente creada, para actualizarla se usara:

```
$ git remote update
```

a. Se añadirá el repositorio del sas al repositorio local.

```
$ git remote add sas http://git.sas.junta-andalucia.es/...
```

a. Se subirán las siguientes ramas:

```
$ git push sas {rama}
                    #master
                    #develop
                    #support/*
```

- Solo se subirán las ramas descritas anteriormente.

2. Se subirán todos los tags

```
$ git push --tags sas
```

Se adjunta un script que permite realizar todos estos pasos automáticamente, indicando el repositorio de origen y el de destino, o desde un repositorio local existente solo indicando el destino.

- Linux

2.2 Git

Git fue creado pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente, es decir Git nos proporciona las herramientas para desarrollar un trabajo en equipo de manera inteligente y rápida.

Algunas de las características más importantes de Git son:

- Rapidez en la gestión de ramas.
- Gestión distribuida.
- Gestión eficiente de proyectos grandes.
- Realmacenamiento periódico en paquetes.

2.2.1 Ordenes básicas

Iniciar un repositorio vacío en la carpeta actual.

```
$ git init
```

Añadir un archivo especifico.

```
$ git add "nombre_de_archivo"
```

Añadir todos los archivos del directorio act

```
$ git add .
```

Confirmar los cambios realizados. El "mensaje" generalmente se usa para asociar al commit una breve descripción de los cambios realizados.

```
$ git commit -m "mensaje"
```

Revertir el commit identificado por un hash.

```
$ git revert "hash_commit"
```

Subir una rama(branch) al servidor remoto.

```
$ git push origin "nombre rama"
```

Mostrar el estado actual de la rama(branch).

```
$ git status
```

2.3. SVN vs GIT

La principal diferencia con Subversion es que éste es un Sistema Centralizado mientras que GIT es un sistema Distribuido:

Subversion = Sistema centralizado

En estos sistemas hay un servidor que mantiene el repositorio y en el que cada programador mantiene en local únicamente aquellos archivos con los que está trabajando en un momento dado. Se necesita conectarse al servidor para poder enviar los cambios realizados. Es un sistema donde todo el código del proyecto esta en el servidor únicamente.

GIT = Sistema distribuido

En este tipo de sistemas cada uno de los integrantes del equipo mantiene una copia local del repositorio completo. Al disponer de un repositorio local, se pueden hacer commit (enviar cambios al sistema de control de versiones) en local, sin necesidad de estar conectado a Internet o cualquier otra red. En el momento que lo quieras compartir con los demás integrantes del equipo se realiza el registro a nivel de servidor.

Migracion SVN --> GIT

La migración de los repositorios actuales que se encuentran en el SVN se realizara de forma progresiva e individualizada por producto a lo largo del tiempo.

Diferencias

En el día a día, el desarrollador trabaja sobre la copia de trabajo en un ciclo de editar ficheros, actualizar la copia con las modificaciones realizadas por otros desarrolladores, y enviar sus modificaciones al repositorio.

En subversion, esto se realiza con los comandos:

```
            #editar ficheros en la copia local
$ svn update
$ svn commit -m "descripción de los cambios realizados"
```

`El flujo de información y los comandos asociados en svn se pueden representar como:`

blocked URL

Los comandos equivalentes en git son:

```
            # editar ficheros en la copia local ...
$ git commit -m "descripción de los cambios realizados" -a
$ git pull
$ git push
```

y su representación gráfica es:

blocked URL

Como vemos, en el ordenador local existen tres elementos: los ficheros con los que se está trabajando, una "staging area" y un repositorio local, mientras que en svn sólo existen dos, los ficheros con los que se está trabajando, y la "working copy".

Por otra parte, en git lo normal es que existan varios repositorios remotos, y habitualmente uno de ellos es el repositorio compartido en el que se consolidan los cambios realizados por un grupo de desarrolladores. Por lo tanto, para enviar unas modificaciones realizadas localmente al repositorio compartido, en git hay que realizar más pasos que en svn: llevar la modificación a la "staging area" (con un comando add, mv o rm), desde allí al repositorio local (con un comando commit), y desde el repositorio local al repositorio remoto. (con un comando push). Por comodidad, los dos primeros pasos se pueden ejecutar con un sólo comando "commit -a".

Para actualizar la copia local con los cambios realizados en el repositorio remoto, en svn se ejecuta el comando "update". En git, primero hay que actualizar el repositorio local con un comando "fetch", y después actualizar la staging area y los ficheros de trabajo con el comando "merge". Por comodidad, los dos pasos se pueden ejecutar con un único comando "pull".

Añadir, eliminar y cambiar de nombre ficheros

En ocasiones, se deben añadir ficheros para que queden bajo el sistema de control de versiones, o bien eliminar ficheros que ya no son necesarios. Esto se realiza de una manera muy similar en ambos casos.

En svn:

```
            $ svn add fichero1
$ svn delete fichero2
$ svn move fichero3 fichero4
```

En git:

```
            $ git add fichero1
$ git rm fichero2
$ git mv fichero3 fichero4
```

Para deshacer las modificaciones realizadas en un fichero, basta con ejecutar el comando revert: Deshacer los cambios realizados

```
            $ svn revert fichero
```

En git, el equivalente a "svn revert fichero" es:

```
$ git checkout fichero
```

Crear un repositorio

En git, para crear un repositorio basta con ejecutar el comando "git init" en el directorio en donde va a estar la copia de trabajo:

```
$ mkdir git_repo
$ cd git_repo
$ git init
Initialized empty Git repository in /home/usuario/git_repo/.git/
```

Esto crea e inicializa el subdirectorio ".git". A continuación, se pueden comenzar a ejecutar comandos "git add", "git commit -a", etc.¿

Establecer la identificación de usuario

En git, cada commit se guarda junto con el nombre y el email del usuario que lo realizó. Para establecer un valor de estos parámetros para todos los repositorios, se utilizan los comandos:

```
$ git config --global user.name "Nombre de Usuario"
$ git config --global user.email email@dominio.com
```

Para comprobar el valor asignado, se utilizan los mismos comandos sin especificar un nuevo valor. Ejemplo:

```
$ git config --global user.name
Gonzalo Rodríguez
$ git config --global user.email
gonzalo@ejemplo.com
```

Para cada repositorio, se puede configurar un valor distinto para estos parámetros, utilizando los comandos sin el modificador "¿global" Crear una copia local de un repositorio remoto En svn, se obtiene una copia de un repositorio en el área de trabajo mediante un "checkout":

```
$ svn checkout svn+ssh://usuario@servidor:/ruta/al/repositorio
```

En git, se realiza una copia del repositorio con el comando "git clone":

```
$ git clone ssh://usuario@servidor/ruta/al/repositorio
```

2.4. GitFlow

Gitflow es uno de los muchos workflows que existen para trabajar con GIT. En el SAS se debe de usar este workflow por parte de los proveedores.

Cómo funciona

Gitflow utiliza un repositorio central como centro de comunicación para los diferentes desarrolladores. Su punto fuerte es la estructura de ramas (branches) que posee para controlar la evolución del proyecto.

blocked URL

Gitflow se caracteriza por el uso de cuatro tipos de ramas diferentes:

- Historical Branches
- Feature Branches
- Release Branches
- Maintenance Branches

CheatSheet: https://danielkummer.github.io/git-flow-cheatsheet/index.es_ES.html

Historical Branches

En lugar de trabajar todo desde la rama master, aquí se utilizan dos ramas para registrar el historial de proyecto:

- **Master** se registran los diferentes lanzamientos a nivel de producto,
- **Develop** se utiliza como rama de integración de los diferentes evolutivos a desarrollar.

Es recomendable etiquetar todos los commits en la rama master con un número de versión.

En resumen en la rama master no se desarrolla, solamente se registran los diferentes lanzamientos. Todas las integraciones se realizan desde la rama de Develop.

blocked URL

El resto de ramas trabajan alrededor de estas dos.

Feature Branches

Cada nueva funcionalidad debe residir en su propia rama, de esta forma en caso de existir problemas es muy rápido volver a versiones anteriores.

Las ramas de nuevas funcionalidades, features, siempre se crean a partir de la rama de develop. Una vez finalizado el desarrollo sobre la feature, se fusiona con la rama de develop , nunca con la rama master.

blocked URL

Release Branches

Una vez el evolutivo se ha finalizado y se acerca la fecha de lanzamiento se crea una rama para éste, release. Las ramas releases son utilizadas para marcar o etiquetar lanzamientos de producto, (evolutivos) y poder tener identificados de forma rápida los cambios de las diferentes releases.

A la rama release no se le pueden añadir nuevos evolutivos de ningún tipo, solamente la documentación asociada al lanzamiento de la release o errores puntuales.

Una vez se finalizan la rama release se tiene que fusionar tanto la rama master como la rama develop. Estas dos, siempre que realicemos una publicación, tienen que estar sincronizadas con el mismo contenido.

blocked URL

Maintenance Branches

La rama maintenance se utiliza cuando se detectan errores críticos de la aplicación.

La forma de trabajar sería crear una rama proveniente de la master, solventar el correctivo y fusionar con la rama master.

Maintenance es la única rama que interactúa directamente con la rama master, todas las demás siempre se trabaja desde la rama de desarrollo.

Una vez se ha solventado el correctivo se tiene también que fusionar con la rama develop, para que el correctivo se mantenga en los diferentes evolutivos.

blocked URL

2.5 Procedimiento de trabajo en GIT con GIT FLOW

A continuación se describe el procedimiento de subida a GIT haciendo uso de GIT FLOW tal y como está indicado en la presente normativa. Para ello, el procedimiento será el siguiente:

1- En el entorno de desarrollo local del proveedor, deberá estar clonado el repositorio GIT remoto del SAS con la misma estructura de ramas obligatoria (DEVELOP y MASTER). Siendo la r**ama de trabajo** siempre, la **rama DEVELOP**.

2- Se deberá iniciar GIT FLOW. Esta sentencia solo es necesario ejecutarla una vez en el repositorio local, mientras no se elimine y se cree un nuevo repositorio local.

```
$ git flow init
```

3- Para cada una de las funcionales a desarrollar, se deberá crear una rama FEATURE desde la rama DEVELOP con GIT FLOW. Esta rama FEATURE deberá seguir la nomenclatura especificada en la normativa vigente (Normativa)

```
$ git flow feature start "FEATUREID_DESCRIPCION"
```

3.1 - Para los proyectos agile en los que se necesitan crear features dentro de otros features (HUs dentro de Mejoras) se tendrá la siguiente estructura

- feature/mejora_descripcion
- feature/mejora/hu_descripcion

```
$ git flow feature start "MEJORAID/HUID_DESCRIPCION" "feature/MEJORAID_DESCRIPCION"
```

> *IMPORTANTE:*
>
> *1: Las features deben tener en su nombre la descripción . Si no, al coincidir el nombre de la carpeta con el nombre de la feature el git flow dará error.*
>
> *2: Si la rama HU se cierra en un equipo local diferente desde donde se creó o bien por un usuario diferente del que la creó, se va a realizar sobre la rama develop, en vez de sobre la mejora. En este caso, hay que hacerlo de forma manual de este modo:*
>
> - *Nos situamos en la rama de mejora*

```
git checkout "feature/MEJORAID_DESCRIPCION"
```

> - *Fusionamos los nuevos cambios de la rama HU a la rama mejora*

```
git merge "MEJORAID/HUID_DESCRIPCION"
```

> - *Eliminamos de local la rama de HU*

```
git branch -delete "MEJORAID/HUID_DESCRIPCION"
```

> - *Eliminamos de origin la rama de HU*

```
git push origin --delete  "MEJORAID/HUID_DESCRIPCION"
```

4- Cuando se finalice el trabajo con la rama FEATURE que corresponda, se procederá a finalizar la rama con GIT FLOW. La finalización de dicha rama con GIT FLOW, realizará automáticamente lo siguiente:

- Fusionar la rama FEATURE en cuestión con la rama DEVELOP.
- Borrar la rama FEATURE en cuestión.
- Cambiar a rama DEVELOP como rama de trabajo actual.

```
$ git flow feature finish "FEATUREID_DESCRIPCION"
```

5- Una vez la rama DEVELOP esté actualizada con todas las FEATURES que corresponda a la versión que se va a entregar, se deberá crear con GIT FLOW una rama RELEASE siguiendo lo especificado en la normativa vigente (Normativa).

```
$ git flow release start "X.Y.Z.A"
```

6- La finalización con GIT FLOW de la rama RELEASE, realizará automáticamente lo siguiente:

- Fusionar la rama RELEASE con la rama MASTER.
- Crea el TAG de la versión que corresponda desde la rama MASTER.
- Fusionar la rama RELEASE con la rama DEVELOP.
- Borrar la rama RELEASE en cuestión.

```
$ git flow release finish "X.Y.Z.A"
```

7- Por último, es muy importante no olvidar **subir todas las rmas (incluidos los TAG)** al repositorio GIT remoto del SAS.

```
$ git push --tags
```

2.6 Gitlab

Como interfaz gráfica de Git, en la que apoyarse para gestionar los repositorios, disponemos de Gitlab en su versión Community Edition.

Desde el SAS se aplica una política de seguridad sobre los repositorios de GIT, con el objetivo de mantener su privacidad y que únicamente los miembros del equipo de proyecto tengan acceso al código fuente.

Los permisos sobre GIT tienen la siguiente configuración:

- Usuarios Master (administradores): Administradores. Tienen permiso de lectura , escritura en el repositorio y acceso total a todas las ramas.
- Usuarios Developer (RW): Tienen permiso de lectura , escritura en el repositorio y restricción rama master.
- Usuarios Guest (R): Tienen únicamente permisos de lectura en el repositorio.

Los permisos más usados serán los de Master y Guest, ya que el repositorio ubicado en el SAS no será usado directamente por los desarrolladores.

La solicitud de permisos se realizara por petición CGES a la OCA, de la misma manera que se realiza actualmente para el SVN

Integración Git - Jira

La plataforma Gitlab nos proporciona una integración sencilla para poder enlazar incidencias (Jira) y nuestro repositorio (Gitlab).

Referencia a nivel de comentarios

Gitlab puede detectar automáticamente enlaces a issues de JIRA. Para ello, sólo tendremos que añadir en los commits, los identificadores de las issues JIRA que deseemos.

De esta forma podremos acceder rápidamente a la issue para ver el detalle.

Ejemplo de comentario: "CHKI-1 Solventado el problema".

2.7 Recomendaciones

Para un correcto uso de GIT recomendamos:

- Realizar los commits necesarios siempre en el intervalo de los diferentes evolutivos.
- Comentar correctamente los diferentes commits indicando a alto nivel las modificaciones realizadas.
- Nunca trabajar en la rama master, se recomienda utilizar ramas para diferentes evolutivos
- Almacenar sólo el código y no otro tipo de documentos o entregables.
- Desarrollar siguiendo los estándares de codificación (code clean).
- Ejecutar pruebas unitarias antes de realizar los commits.
- Usar un flujo de trabajo, por ejemplo gitflow

Referencias

- Normativa GIT
- https://www.atlassian.com/git/tutorials/comparing-workflows#gitflow-workflow
- http://nvie.com/posts/a-successful-git-branching-model/
- https://about.gitlab.com/2014/09/29/gitlab-flow/