# Git y Git-Flow

apacin@oesia.com

# Índice

# ¿Qué es Git?

- Se pronuncia “Guit”
- Software de control de versiones
- Recomendado para aplicaciones con una gran cantidad de código gestionado por mucha gente.
- Inicialmente diseñado por Linus Torvalds para la gestión del código del kernel de GNU/Linux

# ¿Qué es Git?

- Características:
  - Fuerte apoyo al **desarrollo no lineal**, por ende rapidez en la gestión de ramas y mezclado de diferentes versiones
    - Git incluye herramientas específicas para navegar y visualizar un historial de desarrollo no lineal
    - Una presunción fundamental en Git es que un cambio será fusionado mucho más frecuentemente de lo que se escribe originalmente, conforme se pasa entre varios programadores que lo revisan
  - Gestión distribuida
    - Git le da a cada programador una copia local del historial del desarrollo entero, y los cambios se propagan entre los repositorios locales
    - Los cambios se importan como ramas adicionales y pueden ser fusionados en la misma manera que se hace con la rama local

# ¿Qué es Git?

- Características:
  - Los repositorios pueden comunicarse por múltiples protocolos (HTTP, FTP, rsync, protocolo nativo Git) a través de una conexión TCP/IP, existiendo la posibilidad de que la transmisión sea cifrada por SSH
  - Es compatible tanto con CVS (emulador) como con Subversion (git-svn)
  - Gestión eficiente de proyectos grandes
    - Rapidez de gestión de diffs entre archivos

# Git vs Subversion

| | Git | Subversion |
|---|---|---|
| Control de versiones | Distribuida | Centralizada |
| Repositorio | Copias locales del repositorio en las que se trabaja directamente | Un repositorio central donde se generan copias de trabajo |
| Autorización de acceso | Para la totalidad del directorio | Dependiendo de la ruta de acceso |
| Seguimiento de cambios | Basado en contenido | Basado en archivos |
| Historial de cambios | Tanto el repositorio como las copias de trabajo individuales incluyen el historial completo | Completo sólo en el repositorio; las copias de trabajo incluyen únicamente la versión más reciente |
| Conectividad de red | Sólo necesario para la sincronización | Con cada acceso |

# ¿Por qué Git?

- Ventajas para SGA:
  - Soporte a desarrollo no lineal
    - Casuística:
      - Muchos proyectos (XWMS > 30, WMS > 20)
      - Mucha gente (40 < x < 120)
      - Muchas versiones (Silo > 5 versiones en producción)
    - Ventajas:
      - La gestión de ramas es mucho más sencilla
      - Los merge son mucho más sencillos

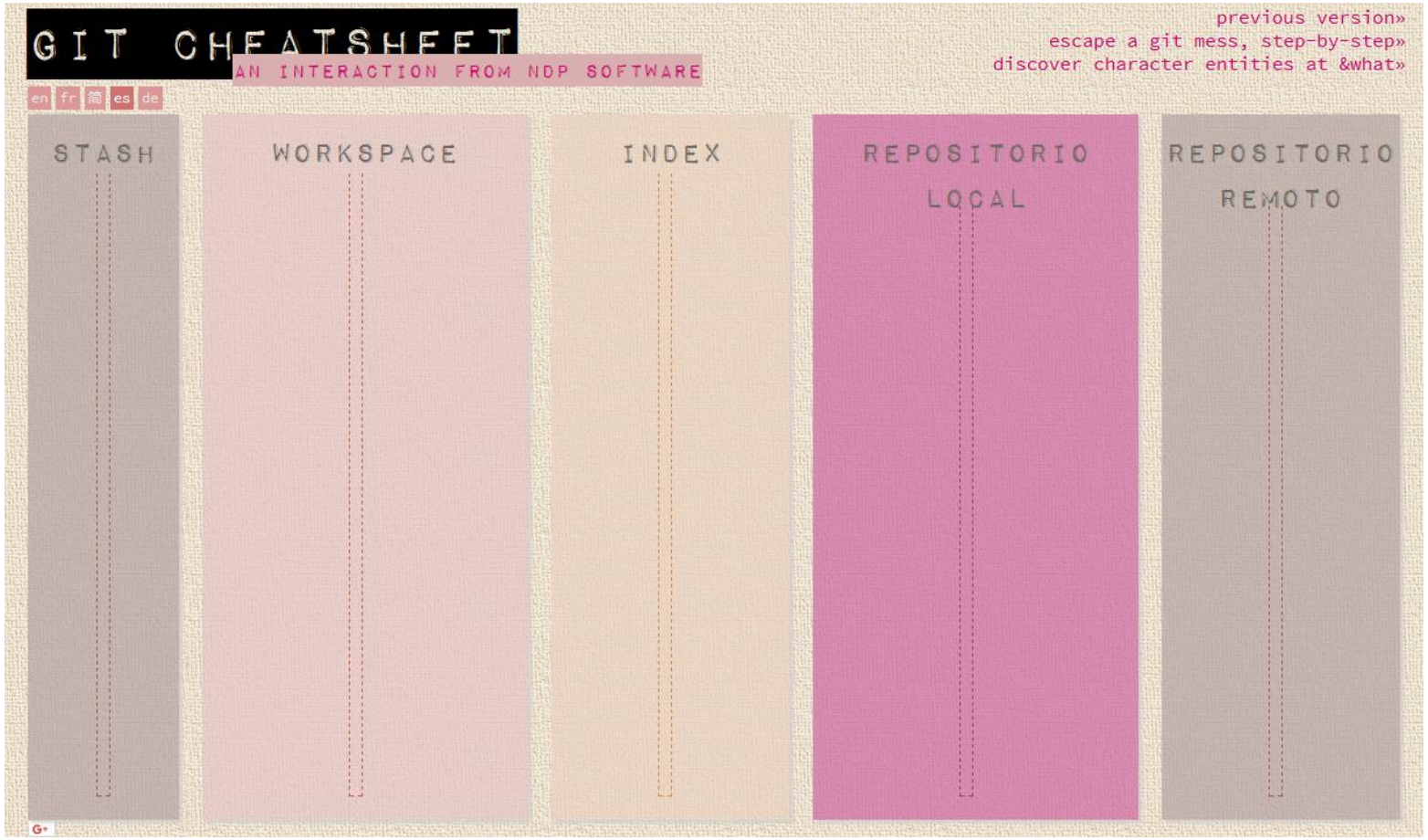
# ¿Por qué Git?

- Ventajas para SGA:
  - Gestión distribuida
    - Casuística:
      - Pérdida de conexión
      - Subidas a trunk de cambios sucios intermedios
      - Pérdida del servidor
    - Ventajas:
      - Posibilidad de commitear en local
      - Recuperación ante desastres

# ¿Por qué Git?

- Ventajas para SGA:
  - Gestión eficiente de proyectos grandes
    - Casuística:
      - Lentitud del servidor
      - Peso de las copias locales
      - Gestión de ramas
    - Ventajas:
      - Descarga rápida de las copias locales
      - Creación rápida de ramas
      - Merge rápido entre ramas
      - Subida rápida de los cambios al servidor

# Nociones básicas



# Nociones básicas

- Stash
  - Un lugar para ocultar cambios mientras trabajas en otra cosa
- Workspace
  - Espacio de trabajo: archivos locales posicionados en una rama
- Index
  - El index (o “staging area”) contiene una captura del contenido del árbol de trabajo
  - Esta captura representa a los contenidos del próximo commit
- Repositorio local
  - Un subdirectorio llamado .git que contiene todos los archivos necesarios
    - Un esqueleto del repositorio Git
    - Ramas típicas: master, feature-x, bugfix-x
- Repositorio remoto
  - Version(es) del proyecto que están alojadas en Internet o una red, asegurando que todos los cambios están disponibles para otros desarrolladores
    - Por defecto es “origin”
    - Ramas típicas aquí son: master, shared-feature-x, release-y

# Nociones básicas

- Operaciones básicas:
  - https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pd
  - Nota: si no funciona el enlace, pegadlo en un buscador.

# Mi primer repositorio

- Instalar Git
  - Alternativa (Windows/MacOs): https://www.sourcetreeapp.com
  - Alternativa (multiplataforma): https://www.syntevo.com/smartgit
- Usando un terminal:
  - Configurar Git
  - Crear repositorio en local
    - $ git init [project-name]
    - Alternativa: crear repositorio remoto y clonarlo
      - $ git clone [url]
      - GitHub (repositorio privados limitados)
      - GitLab (repositorios privados ilimitados)
      - Bitbucket
    - Recomendable hacer ambas cosas
      - El remoto hará falta más tarde
  - Comprobar el estado
    - $ git status

# Mi primer repositorio

- Individual
- Usando un terminal:
  - Crear un fichero HelloWorld.java en el workspace
    - $ touch HelloWorld.java
  - Comprobar el estado
  - Añadir el fichero creado al index
    - $ git add HelloWorld.java
  - Comprobar el estado
  - Commitear los cambios
    - $ git commit –m “[descriptive message]”
  - Comprobar el estado
  - Subir el commit al repositorio remoto (opcional)
    - $ git push
  - Ver el histórico de commits
    - $ git log
  - Ver los cambios del último commit
    - $ git show [commit]

# Mi primer repositorio

- Individual
- Usando un terminal:
  - Hacer otra modificación y otro commit
  - Subir el commit al repositorio remoto
  - Deshacer el último commit conservando todos los cambios
    - $ git revert [commit]
    - Nota: no tiene por qué ser el último
  - Hacer otra modificación y otro commit
  - Deshacer el último commit de manera que no quede rastro del mismo en el repositorio local, pero se conserven los cambios en el entorno de trabajo
    - $ git reset [commit]
  - Hacer otra modificación y otro commit
  - Deshacer el último commit de manera que no quede rastro del mismo en ningún sitio
    - $ git reset –hard [commit]

# Mi primer repositorio

- Individual
- Usando un terminal:
  - Crear un fichero ByeWorld.java en el workspace
  - Añadir el fichero creado al index
  - Commitear los cambios
  - Renombrar el fichero a ByeByeWorld.java
    - $ git mv ByeWorld.java ByeByeWorld.java
  - Commitear los cambios
  - Borrar el fichero conservando los cambios en local
    - $ git rm –cached ByeByeWorld.java
  - Añadir nuevamente el fichero al index
  - Commitear los cambios
  - Borrar el fichero por completo
    - $ git rm ByeByeWorld.java
  - Commitear los cambios

# Mi primer repositorio

- Individual
- Usando un terminal:
  - Ejercicio opcional
  - Crear un fichero HelloWorld.class en el workspace
  - Agregar el fichero a la lista de ficheros ignorados
    - .gitignore
  - Comprobar el estado

# Mi primer repositorio compartido

- Por parejas (o incluso más)
- Usando un terminal:
  - Crear y/o clonar un repositorio remoto compartido
  - Cada uno por separado:
    - Crear una rama desde /master (p.e. /apacin)
      - $ git branch [branch-name]
    - Listar las ramas locales
      - $ git branch
    - Cambiar el entorno de trabajo a la nueva rama
      - $ git checkout [branch-name]
    - Crear un fichero dentro de dicha rama (p.e. apacin.txt) con algo de contenido
    - Commitearlo y subirlo al repositorio remoto

# Mi primer repositorio compartido

- Por parejas (o incluso más)
- Usando un terminal:
  - Cada uno por separado:
    - Obtener el histórico de la rama /master del repositorio remoto
      - $ git fetch origin/master
    - Traer los últimos cambios de /master a nuestra rama
      - $ git checkout master
      - $ git pull master
      - $ git checkout [branch-name]
      - $ git merge master
      - $ git commit –m “Merge de master”
      - $ git push
    - Llevar nuestros cambios a /master
      - $ git checkout master
      - $ git merge [branch-name]
      - $ git commit –m “Merge de [branch-name]
      - $ git push
    - Comprobar el contenido de nuestra rama
    - ¿Hay diferencias entre las ramas personales de cada uno?

# Mi primer repositorio compartido

- Por parejas (o incluso más)
- Usando un terminal:
  - Cada uno por separado:
    - Cambiar el entorno de trabajo a nuestra rama
    - Hacer nuevos cambios sobre nuestro fichero, sin commitearlos ni subirlos al repositorio remoto
    - Intentar cambiar el entorno de trabajo a la rama /master
      - ¿Qué ocurre?
    - Guardar los cambios temporalmente en el pila sin necesidad de commitearlos
      - $ git stash
    - Cambiar el entorno de trabajo a /master
    - Listar los cambios guardados en la pila
      - $ git stash list
      - ¿Qué ocurre?
    - Cambiar el entorno de trabajo a nuestra rama
    - Listar los cambios guardados en la pila
    - Restaurar los cambios guardados en la pila
      - $ git stash pop
    - Hacer más cambios, guardarlos en la pila, listarlos y luego borrarlos
      - $ git stash drop

# Material extra

- Aprende Git en 15 minutos (interactivo):
  - https://try.github.io/levels/1/challenges/1
- Curso completo (interactivo):
  - http://gitreal.codeschool.com/levels/1

# ¡La he liado parda!

- https://vimeo.com/82408340

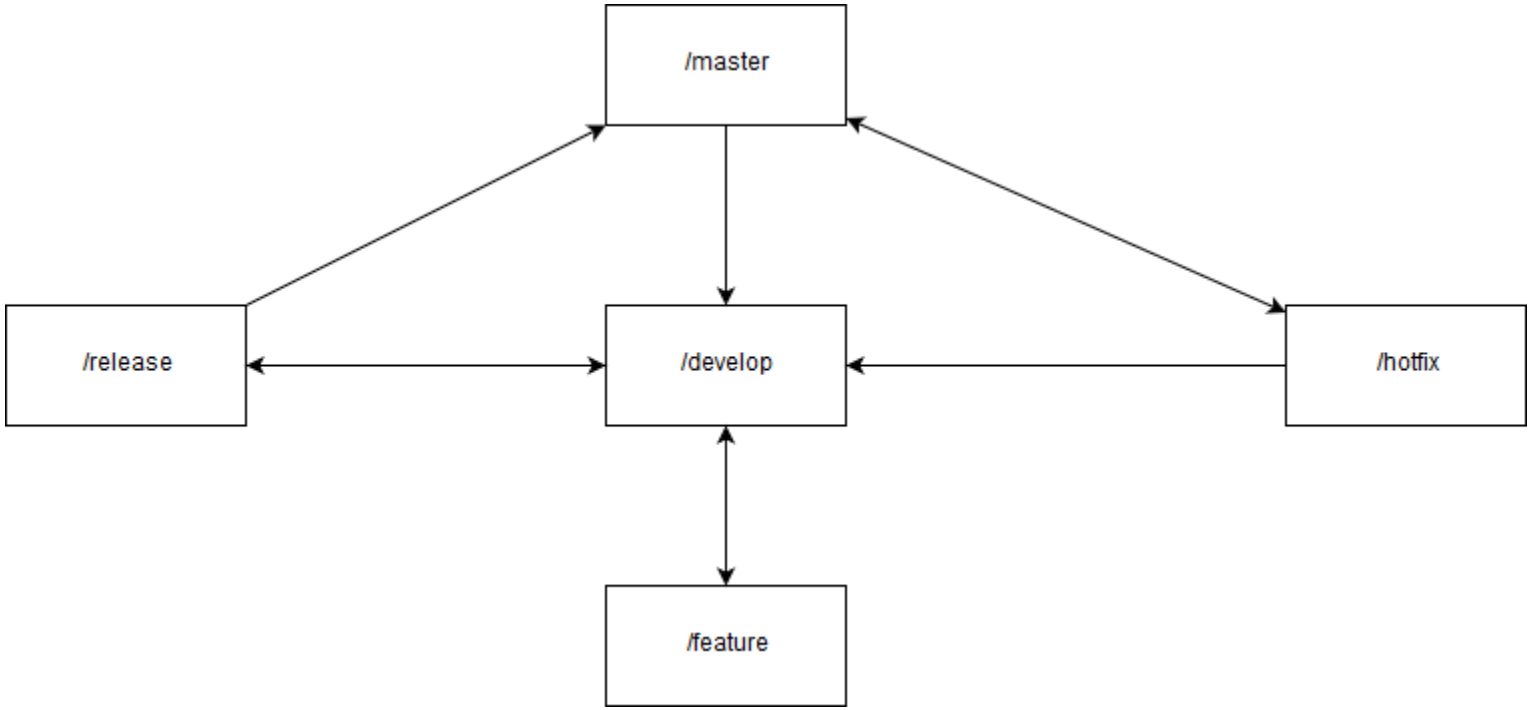
# Git-Flow

- Es un flujo de trabajo para equipos y proyectos en los que se use Git como herramienta de control de versiones
- Define un modelo de creación de ramas estricto (pero no cerrado) diseñado alrededor de la generación de versiones del proyecto
- Ventajas:
  - Desarrollo en paralelo
    - Nueva rama independiente por cada feature
  - Colaboración
    - Varios desarrolladores puedes colaborar en el desarrollo de una feature en una rama aislada
  - Área de preparación de releases
    - La rama de desarrollo es una rama independiente en donde se van introduciendo nuevas features que aún no han sido lanzadas
  - Soporte para correcciones críticas
    - Da soporte a los temidos hotfixes, creando ramas directamente desde tags de master y aplicando cambios atómicos sólo sobre lo que queremos, de manera que no incorporemos accidentalmente nuevas features a master

# Git-Flow

- Flujo:
  - Se crea una rama /develop desde /master
  - Se crea una rama /release desde /develop
  - Cada rama /feature se crea desde /develop
  - Cuando una /feature se completa, se lleva a /develop
  - Cuando una /release se completa, se lleva a /develop y /master
  - Si se detecta un issue crítico en producción se crea una rama /hotfix desde /master
  - Una vez que el /hotfix está listo, se lleva a /develop y a /master

# Git-Flow



# Git-Flow

- Veamos unos ejemplos:
  - https://datasift.github.io/gitflow/IntroducingGitFlow.html

# Git-Flow

- Material extra
  - http://nvie.com/posts/a-successful-git-branching-model/
  - https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
  - http://aprendegit.com/que-es-git-flow/
  - https://datasift.github.io/gitflow/IntroducingGitFlow.html

# Pull request

- Es un mecanismo que permite a los desarrolladores notificar al resto del equipo que una /feature ha sido terminada y está pendiente de revisión, aprobación y mezclado en /develop si procede.
  - Se aplica también a otros casos, pero este es el caso principal.
- No forma parte de Git ni de Git-Flow, pero es una práctica adoptada por la práctica totalidad de proyectos serios y soportada por los principales servicios de repositorios.
- Es una manera muy eficaz de hacer pair-programming.

# Pull request

- El desarrollador será el encargado de:
  - Crear la pull request, indicando las ramas origen y destino.
  - Haber creado los tests suficientes y necesarios para poder probar el código.
  - Si hay un ticket asociado, indicar en el mismo cómo probar el código.
- El revisor será el encargado de:
  - Revisar que la calidad del código es correcta.
  - Revisar que existen tests para probar el código.
  - Bajarse la rama y probar la funcionalidad.
  - Si hay un ticket asociado, revisar que se indica cómo probar el código y seguir las indicaciones.
  - En caso de rechazarse la pull request, indicar por qué se rechaza, siempre de manera clara, objetiva y constructiva.
  - En caso de aprobarse, hacer el merge.
- Más info: https://www.atlassian.com/git/tutorials/making-a-pull-request

# Git merge vs rebase

- Merge da una mayor trazabilidad del histórico, aunque puede llegar a ser imposible de seguir
- Rebase deja un histórico más limpio, pero es menos “seguro”: es más propenso a errores, sobre todo con gente menos experimentada
- Rebase Golden Rule: nunca hagas rebase sobre una rama compartida
- Pull Request Rule: si usas pull request, no uses rebase
- Más info:
  - https://www.atlassian.com/git/articles/git-team-workflows-merge-or-rebase
  - https://www.atlassian.com/git/tutorials/merging-vs-rebasing
  - https://stackoverflow.com/questions/30400222/using-git-flow-how-to-work-rebase-into-the-mix
  - https://blog.sourcetreeapp.com/2012/08/21/merge-or-rebase/

# Herramientas

- SourceTree (Windows/MacOs)
  - https://www.sourcetreeapp.com
- SmartGit (multiplataforma):
  - https://www.syntevo.com/smartgit
- GitHub:
  - https://github.com/
- GitLab:
  - https://gitlab.com
- BitBucket:
  - https://bitbucket.org