
Sistemas de control de versiones.

Entornos de desarrollo

ÍNDICE DE CONTENIDOS

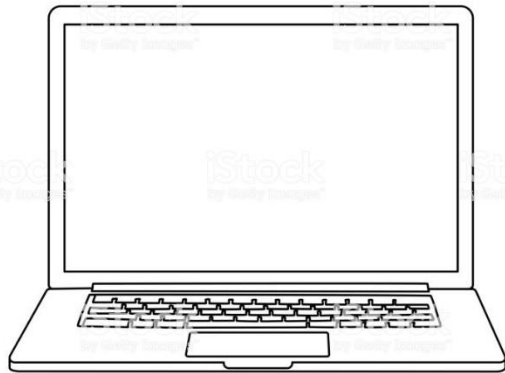
- Introducción a los sistemas de control de versiones.
- Tipos de sistemas de control de versiones.
- Git y GitHub.
- El gestor de versiones Git
- Principales comandos de Git.

¿Qué es el de control de versiones?

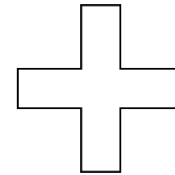
- Los sistemas de control de versiones, son un conjunto de herramientas que permiten registrar los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo.
- **Tienen que proporcionar:**
 - ✓ Mecanismos para realizar cambios sobre los documentos almacenados.
 - ✓ Permiten recuperar versiones anteriores.
 - ✓ Mantener un historial de los cambios realizados en el proyecto.

Control de versiones: versión local

- Los archivos de versiones diferentes son tratados como el mismo, es decir, estamos constantemente generando nuevas versiones.
- Tenemos una sola copia del sistema, en nuestro ordenador, por lo que el programador no podía permitirse eliminar o confundir versiones.



Conjunto de archivos de la versión actual, originalmente.

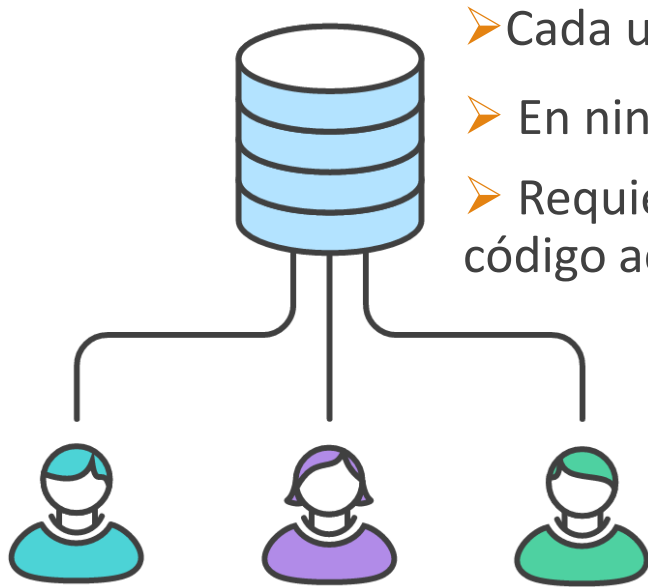


Conjunto de archivos locales

Control de versiones: Tipos

➤ Los sistemas de control de versiones, se agrupan en dos grandes grupos:

1) **Centralizados:** Los sistemas centralizados almacenan todo nuestro código fuente y sus versiones en un único directorio de un ordenador, servidor. Los desarrolladores trabajan con copias locales sobre del código original. Tras realizar sus cambios en local, el sistema de control de versiones guardará los cambios realizados como una nueva versión, nuevo original.



➤ Cada usuario/desarrollador posee una copia local del repositorio.

➤ En ningún momento disponen de la versión actual.

➤ Requiere un esfuerzo continuo por parte del equipo de trabajo para mantener el código actualizado, sin **conflictos**.

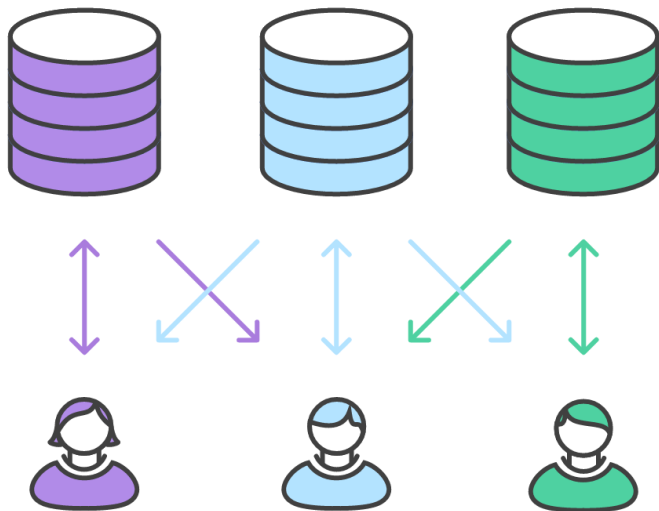
➤ Algunos de estos sistemas son:



Control de versiones: Tipos

➤ Los sistemas de control de versiones, se agrupan en dos grandes grupos:

2) **Distribuidos:** Estos sistemas no tienen un repositorio central. Cada miembro del equipo de trabajo tiene su propia copia del repositorio, con todas las versiones y toda la historia. A medida que avanza la fase de desarrollo, los cambios se van haciendo relevantes tanto en el código fuente como en las versiones. Para alinear los trabajos del equipo, estos sistemas permiten **sincronizar**(esto será posible siempre que no existan conflictos entre las versiones del códigos, las cuales tendremos que solucionar) los repositorios locales.



- Cada usuario/desarrollador posee su propio repositorio.
- Disponen de su versión actual.
- Hay una persona encargada de gestionar los cambios.

➤ Algunos de estos sistemas son:



git



mercurial

Control de versiones: Comparativa

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

Git y GitHub

Git es un software de *control de versiones distribuido* y gratuito diseñado por **Linus Torvals** (2005). El objetivo era mejorar la forma de gestionar el desarrollo de las distintas versiones de kernels de Linux.

Se basa en que cada persona tenga su propio **repositorio local** y se suban los cambios a un **repositorio remoto** de forma que todo el trabajo sea colaborativo y *no sea necesario obligar a trabajar directamente sobre el repositorio remoto*. El repositorio local recibe el nombre de Git, mientras que el repositorio remoto se llama GitHub.

GitHub permite almacenar nuestros proyectos **de forma pública**, aunque utilizando una **cuenta de pago**, también permite hospedar **repositorios privados**.

Además de GitHub, existen más repositorios remotos como: BitBucket, GitLab...

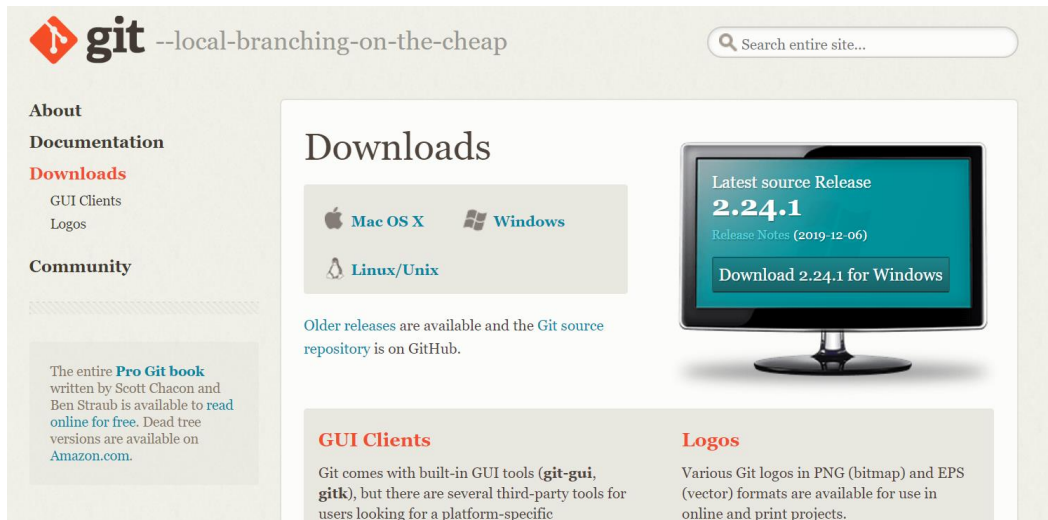


Git: Características

- ✓ **Desarrollo no lineal**, es decir, que un proyecto pueda tener varias “versiones” de si mismo en desarrollo de forma simultanea. De esta forma, las posibles ramas pueden ser fusionadas al ser revisadas por varios programadores.
- ✓ **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.
- ✓ Los almacenes de información pueden publicarse por ***diversos protocolos de comunicación*** (*http, ftp, ssh*, protocolo nativo Git, etc).
- ✓ Los repositorios de otros gestores de control de versiones tales como Subversion y SVK se pueden usar directamente con git-svn, es decir, Git permite la convergencia entre diferentes gestores de control de versión.
- ✓ Gestión eficiente de proyectos grandes, dada la rapidez de gestión de diferencias entre archivos, entre otras mejoras de optimización de velocidad de ejecución.
- ✓ Una de las desventajas de Git, es su curva de aprendizaje, nuestro equipo de trabajo necesitará una introducción al flujo de trabajo de esta; para evitar así posibles errores en la realización de cambios.

Git: Instalación

En primer lugar, nos dirigiremos al sitio WEB de la aplicación (<https://www.git-scm.com/>) para la descarga del software. Hay versiones disponibles para Microsoft Windows, GNU Linux, y Mac en sus versiones de 32 y 64 bits.



De esta forma se descargará el gestor de comandos de consola de Git. Este gestor nos permite instalar una GUI.

El primer comando que introduciremos en la GUI, será: `git --version` para ver la versión que tenemos actualmente instalada en el sistema.

```
$ git --version
git version 2.24.1.windows.2
```

Git: Configuración

Antes de empezar a trabajar con Git, lo primero que tenemos que hacer, es configurar la **identificación de usuario**. Para ello, se usa el comando ***“git config”***

Ya que Git está orientado a un entorno de trabajo multiusuario, lo ideal, es configurar como mínimo nuestro nombre y el correo electrónico para poder ser identificado en el proyecto. Para ello usaremos, respectivamente:

- **git config --global user.name**
- **git config --global user.email**

```
$ git config --global user.name "Francisco Javier Cano"
$ git config --global user.email "fjcanofp@gmail.com"
```

Git: Configuración

Para verificar que hemos realizado la configuración con éxito, usaremos la opción: ***git config list***

```
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
credential.helper=manager
user.name=Francisco Javier Cano
user.email=fjcanofp@gmail.com
```


Git: Crear un repositorio

Nos situaremos en el directorio (*path*) donde queramos crear nuestro repositorio y usaremos el comando: ***git init nombre_repositorio***

```
$ git init ej_repositorio  
Initialized empty Git repository in D:/IES Clara del Rey 19-20/ENTORNOS DE DESAR
```

Tras la creación del directorio, podemos visualizar que se ha creado un directorio .git. Este contiene una base de datos con el registro de cambios realizados en el proyecto.

☐ Nombre

 .git

Git: áreas de trabajo

Los ficheros de un proyecto pasan por tres áreas diferentes:

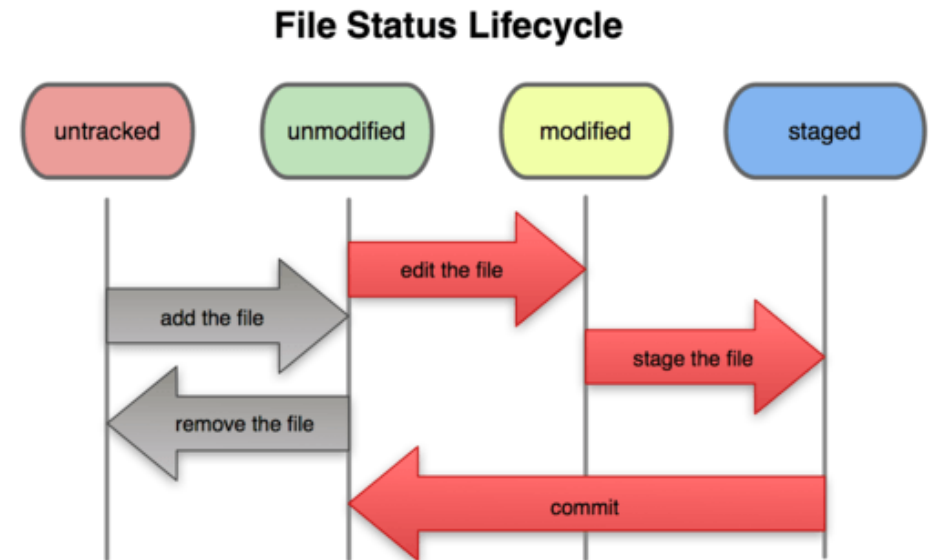
- ✓ **working directory:** Son ficheros que el usuario está trabajando en ellos y aun no hay indicado que son susceptibles de añadir al repositorio.
- ✓ **staging area:** Los ficheros han sido marcados para ser incluidos en el proyecto y actualizados en el sistema de control de versiones de Git.
- ✓ **commit area:** Los ficheros han incluidos en el proyecto para que otros usuarios puedan tener acceso a ellos.



Git: Estado de los ficheros

Cada archivo de tu directorio de trabajo puede estar en uno de estos dos estados:

- **Tracked** (bajo seguimiento): son aquellos que existían en la última instantánea; pueden estar sin modificaciones, modificados, o preparados
- **Untracked** (sin seguimiento): son todos los demás, cualquier archivo de tu directorio que no estuviese en tu última instantánea ni está en tu área de preparación



Git: Estados de los ficheros

Para comprobar el estado de los ficheros en nuestro proyecto, usaremos el comando **git status**:

```
ESARROLLO/MIModelo1920/4_Optimización y Documentación/GitRepositorio/ej_Reposi
rio (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    f1.txt
    f2.txt

nothing added to commit but untracked files present (use "git add" to track)
```


Git: Estados de los ficheros

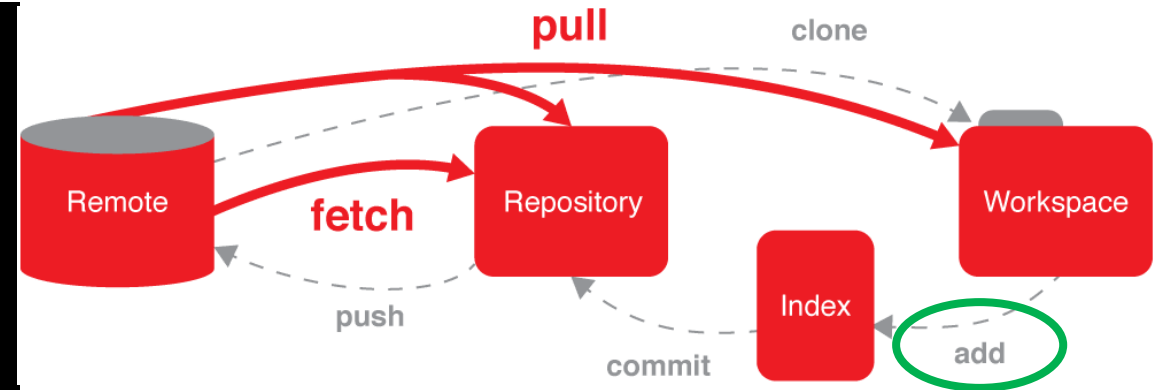
Para cambiar el estado de los ficheros como *tracked* (staging area), usaremos el comando **git add nombre_fichero**

```
$ git add f1.txt f2.txt

$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   f1.txt
    new file:   f2.txt
```



Como vemos en el mensaje, para quitar un fichero de la zona de preparación, usamos el comando **git rm --cached nombre_fichero**

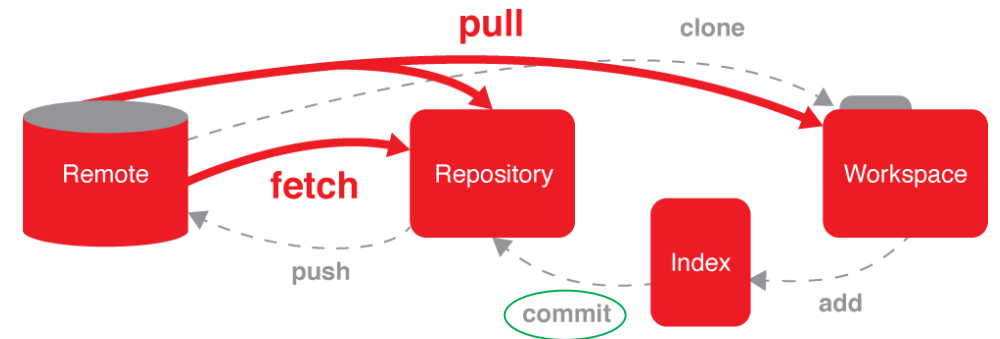
Git: commit

Por último, registraremos el conjunto de cambios realizados en nuestro “repositorio” local mediante el commando: **git commit -m “mensaje”**.

Si te has equivocado en el mensaje del commit, puedes modificarlo añadiendo la opción **-amend**.

La opción **-m** nos permite añadir un mensaje, el cual tiene que permitir “comprender” a tus compañeros de trabajo el motivo de este.

```
$ git commit -m "mi primer commit"
[master (root-commit) d4069bc] mi primer commit
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 f1.txt
create mode 100644 f2.txt
```



Nota: Commit identifica los cambios realizados en tu entorno de trabajo. Este es a nivel de tu repositorio local.

Git: log

Para comprobar las acciones que se han realizado en nuestro proyecto, el comando git log.

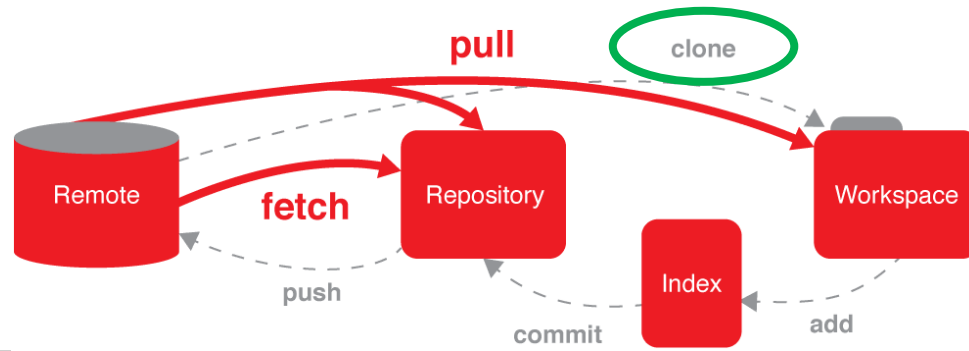
```
$ git log
commit d4069bc53be4220f88e62de9d1fc58aa4f111c21 (HEAD -> master)
Author: Francisco Javier Cano <fjcanofp@gmail.com>
Date:

    mi primer commit
```

Si deseamos visualizar la información más resumida, usaremos la opción `–oneline` en el comando anterior

Nota: Cuando dudes sobre las opciones disponibles en cada comando, o simplemente refrescar la información sobre este, utiliza el comando **git help comando_aVisualizar**

Git: clone




Utilizamos **clone** cuando queremos **obtener una copia de un repositorio Git existente**, por ejemplo, un proyecto en el que te gustaría contribuir, el proyecto que tenías en casa pero estás en otro ordenador...

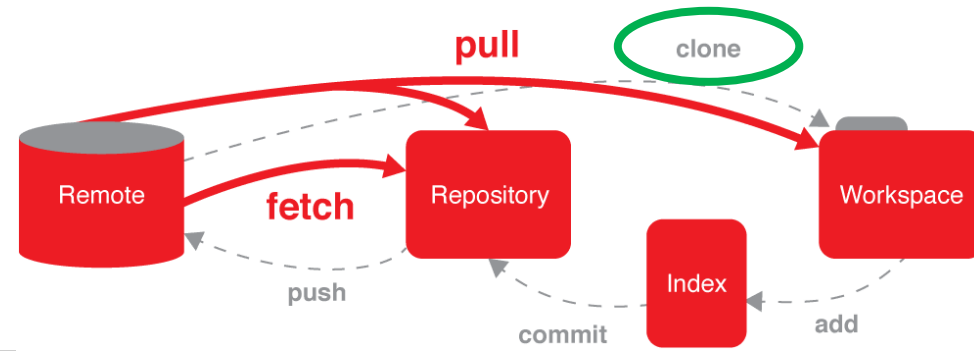
Puedes clonar un repositorio con **git clone url**. Por ejemplo, si quieres clonar el repositorio coding-cheat-sheets, harías :

```
$ git clone https://github.com/aspittel/coding-cheat-sheets.git
Cloning into 'coding-cheat-sheets'...
remote: Enumerating objects: 351, done.
remote: Total 351 (delta 0), reused 0 (delta 0), pack-reused 351
Receiving objects: 100% (351/351), 378.64 KiB | 1.66 MiB/s, done.
Resolving deltas: 100% (181/181), done.
```

Esto crea un directorio llamado "coding-cheat-sheets", inicializa un directorio .git en su interior, descarga toda la información de ese repositorio, y saca una copia de trabajo de la última versión.

 coding-cheat-sheets

Git: clone



Si quieres clonar el repositorio a un directorio con otro nombre que no sea su nombre sin la extensión .git, puedes especificarlo con la siguiente opción de línea de comandos:

```
$ git clone https://github.com/aspittel/coding-cheat-sheets.git codingcheat2
Cloning into 'codingcheat2'...
remote: Enumerating objects: 351, done.
remote: Total 351 (delta 0), reused 0 (delta 0), pack-reused 351
Receiving objects: 100% (351/351), 378.64 KiB | 1.47 MiB/s, done.
Resolving deltas: 100% (181/181), done.
```

Git: .gitignore

El fichero **.gitignore** se coloca en la raíz del proyecto en el se indica que fichero no serán subidos al repositorio. Esto es interesante porque siempre hay ficheros que no deben ser subidos al repositorio, por ejemplo:

.jar , .war : Es el resultado de compilar un proyecto java normal o web.

.class: Son el resultado de compilación de una clase en java.

/target/: directorio sobre el que se descargan la librerías en Maven.

Los motivos por los que no se suben estos ficheros son:

- Son **específicos** de cada equipo ordenador.
- **Ralentizan** las subidas y bajadas de cambios.
- **Ocupan** el repositorio de tamaño inservible. Tamaño medio de ficheros **.war** 100 MB.
- Los fichero **compilables** son distintos para cada máquina.

Git: .gitignore

Ejemplo:

comentario

*.[ao] ignora los ficheros con a o

#ignora todos los ficheros cc

bin/

#ignora el contenido del dire

```
# Compiled class file
*.class

# Log file
*.log

# BlueJ files
*.ctxt

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.nar
*.ear
*.zip
*.tar.gz
*.rar

# virtual machine crash logs, see http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
```

Git: .gitignore

Los ficheros que no hay que subir de un proyecto Java:

```
# Compiled class file
*.class

# Log file
*.log

# BlueJ files
*.ctxt

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.nar
*.ear
*.zip
*.tar.gz
*.rar

# virtual machine crash logs, see http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
```
