

Empezando con el desarrollo

Una vez tenemos definido lo que tenemos que hacer, es el turno del desarrollador de software. Este va a hacer uso de una serie de herramientas para hacer su trabajo.

En un mundo cada vez más interconectado, los profesionales requieren autonomía, por lo que tendrán estas herramientas instaladas en su ordenador (probablemente un portátil con grandes capacidades), configuradas para poder trabajar con los proyectos. Al conjunto de estas herramientas se las conoce como “entorno local de un desarrollador”. Vamos a ver algunas de las más comunes.

El uso del IDE

Un IDE (Integrated Development Environment) es una interfaz de desarrollo, una herramienta que nos permite realizar desde las tareas más básicas, como codificar, hasta otras más avanzadas o accesorias, como hacer cambios en múltiples ficheros al mismo tiempo.

A menudo se pueden extender instalando plugins, que son extensiones que han desarrollado terceros y que añaden funcionalidad extra o utilidades que van a simplificar nuestras tareas diarias. Estos plugins nos permiten, por ejemplo, ver el historial de cambios que ha tenido un fichero y quienes lo han modificado. Para ello harán uso de un sistema distribuido de control de versiones como Git (que veremos más adelante). Hay muchos IDEs distintos que ofrecen características y opciones diferentes y pueden estar centrados en distintos lenguajes de programación. Entre los más populares, encontramos Eclipse, IntelliJ IDEA o Visual Studio Code.

Git

Git es, como dice su propia página web, un sistema de control de versiones distribuido, de código abierto y gratuito, es decir, donde vamos a guardar el código fuente de nuestras aplicaciones y todos los cambios que se hagan sobre el mismo. La palabra clave es “distribuido”. En el caso de Subversion, CVS o similares hay un repositorio central con el cual se sincroniza todo el mundo. Este repositorio central está situado en una máquina concreta y es el repositorio que contiene el histórico, etiquetas, ramas, etc. En los sistemas de control de versiones distribuidos, como es el caso de Git, esta idea de repositorio central no existe, está distribuido entre los participantes; cada participante tiene en su local el histórico, etiquetas y ramas. La gran ventaja de esto es que no necesitas estar conectado a la red para hacer cualquier operación contra el repositorio, por lo que el trabajo es mucho más rápido y tiene menos dependencias. Ante esta idea hay algunas preguntas comunes que suelen asaltar nuestra cabeza:

- ¿Si tenemos todo el repositorio en local, no ocupará mucho espacio?

Lo normal es que no, porque al ser un repositorio distribuido, solamente tendremos las partes que nos interesan. Habitualmente, si tenemos un repositorio central, tendremos muchos proyectos, ramas, etiquetas, etc. Al tratarse de un repositorio distribuido, solo tendremos en local la parte con la que estamos trabajando (la rama de la que hemos partido).

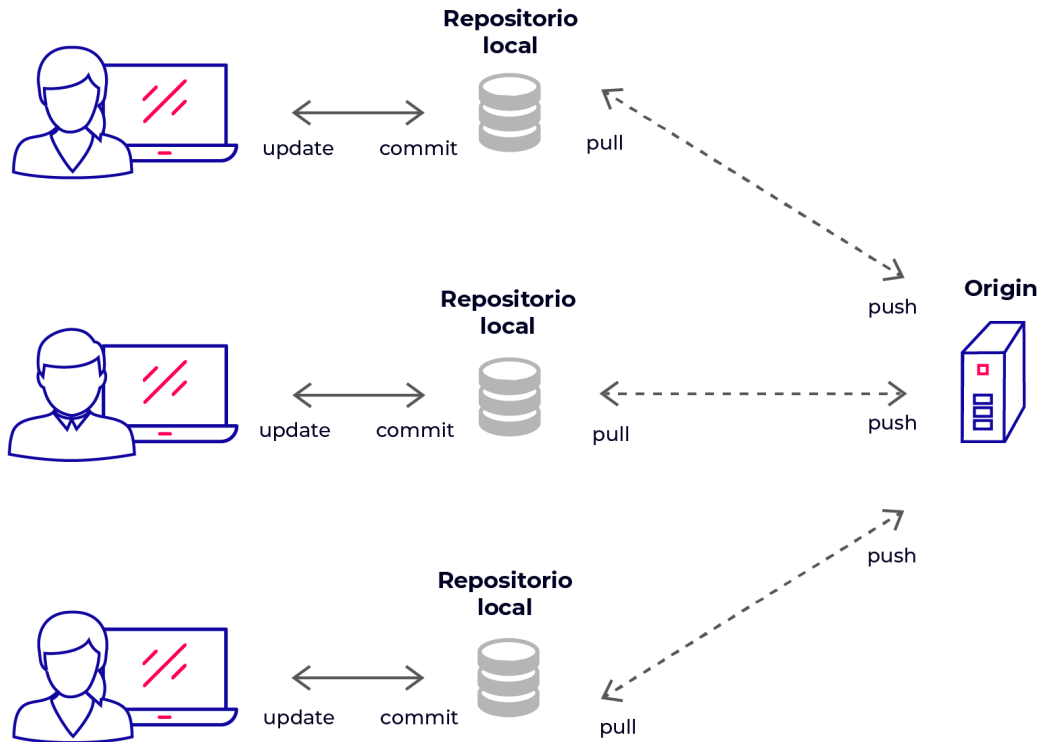
- ¿Si todo el mundo trabaja en su local, no puede resultar en que distintas ramas diverjan?

Efectivamente, esto puede ocurrir y es natural. En un desarrollo tipo open-source no hay mucho problema, es lo habitual y constantemente están saliendo nuevos “forks” (desarrollos paralelos y distintos del original) de los productos. En un desarrollo “interno” tampoco hay problema si ocurre esto, porque luego vamos a acabar haciendo “merge” (fusionando los cambios que tenemos en local con los que hay en el repositorio, para que tanto la versión local como la del repositorio sean iguales). Al final, todo depende de nuestro proceso de desarrollo, no tanto de la herramienta que usemos; es normal que los desarrolladores abran nuevas ramas constantemente para desarrollar partes del sistema y que luego hagan merge para unir estos cambios sobre una rama principal.

Git Flow

Git Flow surge de un post de Vincent Driessen en 2010 donde explica un modelo de desarrollo basado en las ramas de git utilizando una serie de convenciones para el nombrado de las ramas y definiendo un uso específico de las mismas.

Aunque el enfoque de Git es descentralizado, se necesita un repositorio central que contenga los cambios hechos por los componentes del equipo. A este repositorio se le llama “origin”. Cada desarrollador hace “pull” (traerse los cambios del origen a tu carpeta local) y “push” (subir los cambios de la carpeta local al repositorio origen) al origen. Además de las relaciones centralizadas, cada desarrollador puede hacer pull de cambios de otros compañeros para formar subequipos. Esto es útil cuando trabajas con dos o más desarrolladores en una gran característica nueva, antes de llevar el trabajo en progreso al “origin” prematuramente y causar que haya un desarrollo incompleto.



Ramas principales

El repositorio central tiene dos ramas principales, paralelas, que nunca se borran y sobre las que todo el sistema de ramas de soporte se va a apoyar:

- **Master:** rama que contiene el código listo para subir a producción o hacer una nueva versión; este código será el del commit al que apunte HEAD (el último commit de la rama).
- **Develop:** rama que siempre refleja el estado del desarrollo con los últimos cambios listos para ser entregados en la próxima versión; este código será el del commit al que apunte HEAD (el último commit de la rama). Esta rama puede tener otros nombres como, por ejemplo, rama de integración.

Cuando el código fuente en la rama de desarrollo alcanza un punto estable y está listo para ser lanzado en forma de nueva versión, todos los cambios deben fusionarse nuevamente en la rama master y luego etiquetarse con un número de versión.

Por lo tanto, cada vez que los cambios se fusionan en la rama master, por definición, se genera una nueva versión de producción. Podríamos compilar y desplegar automáticamente el software en producción cada vez que haya un commit en master.

Además de las ramas master y develop, el modelo de desarrollo utiliza una variedad de ramas de soporte para ayudar al desarrollo paralelo entre los miembros del equipo, facilitar el seguimiento de las características, prepararse para los lanzamientos de producción y ayudar a solucionar rápidamente los problemas de producción en vivo. A diferencia de las ramas principales, estas ramas siempre tienen un tiempo de vida limitado, ya que una vez cumplido su propósito, se eliminan.

Los diferentes tipos de ramas que podemos usar son: feature, release y hotfix.

Cada una de estas ramas tiene un propósito específico y están sujetas a reglas estrictas sobre qué ramas pueden ser su rama de origen y a qué ramas pueden hacer merge.

Estas ramas no son especiales desde una perspectiva técnica, son ramas normales de git y esta clasificación se hace por el tipo de uso que pretendemos darle al código dentro de la rama:

Ramas Feature:

Utilizadas para desarrollar nuevas funcionalidades para las próximas versiones. Al comenzar el desarrollo de una funcionalidad, la versión de destino en la que se incorporará puede ser desconocida en ese momento. La esencia de una rama feature es que existe mientras la funcionalidad esté en desarrollo, pero finalmente se fusionará de nuevo en la rama develop (para agregar definitivamente la nueva funcionalidad a la próxima versión) o se descartará (en caso de que la característica resulte ser un experimento decepcionante) borrando la rama.

Esta rama se va a crear partiendo de la rama develop y, al terminar, se va a fusionar (merge) con la rama develop. La convención de nombres de estas ramas suele ser feature/(nombre de la característica a desarrollar). Si se está usando un sistema para manejar las tareas del desarrollo y estas tareas tienen un número o identificador, el nombre de estas ramas suele ser feature/(identificador)-(nombre de la característica a desarrollar).

Ramas Release:

Apoyan la preparación de una nueva versión de producción. Además, permiten pequeñas correcciones de errores y preparan metadatos para un lanzamiento (número de versión, fechas de compilación, etc.). Al hacer este trabajo en una rama release, la rama de desarrollo puede ser usada para recibir características para la próxima gran versión sin crear problemas al cruzar trabajo entre dos versiones.

El momento clave para crear una nueva rama release desde la rama de desarrollo es cuando la rama desarrollo (casi) refleja el estado deseado de la nueva versión. Antes de crear la rama release desde la de develop, todas las funcionalidades que tienen que ir en la nueva versión tienen que ser primero fusionadas con la rama develop; luego ya se puede crear la rama release. Las ramas feature que contengan funcionalidades no

pertenecientes a la versión que se va a lanzar a la rama release, no deben ser fusionadas con la rama develop.

Es exactamente en la **creación de una rama release cuando se le asigna un número de versión**, no antes (de acuerdo a las reglas de versionado del proyecto). Hasta ese momento, la rama de desarrollo contiene cambios para la "próxima versión", pero no está claro si esa "próxima versión" se convertirá finalmente en la 0.3 o 1.0 hasta que se cree la rama. Aún así, no se va a sacar versión del proyecto hasta que se haga el merge con master y, por lo tanto, los eventuales bugs que pueda tener esta rama estén resueltos.

Esta rama se va a crear partiendo de la rama develop y al terminar se va a fusionar (merge) con las ramas develop y master. La convención de nombre de estas ramas suele ser **release/(número de versión)**.

Ramas Hotfix:

Muy **parecidas a las ramas release**, ya que también **están destinadas a prepararse para un nuevo despliegue en producción**. Surgen **de la necesidad de actuar inmediatamente sobre un estado no deseado (un bug o error)** de una **versión que se está usando en producción**. **Cuando se descubre un error crítico** en una versión de producción **debe resolverse de inmediato**, una rama hotfix **se puede crear partiendo de la rama master que esté marcada (con un tag) con el número** de la versión desplegada en producción.

La esencia de esta rama es que el trabajo de los miembros del equipo (en la rama de desarrollo) puede continuar, mientras que otra persona está preparando una solución rápida para producción.

Esta rama se va a crear partiendo de la rama master y al terminar se va a fusionar (merge) con las ramas develop y master. La convención de nombre de estas ramas suele ser **hotfix/(número de versión, incrementado sobre la versión que hay en producción)**.

Lo importante en la convención de nombres de las ramas auxiliares es indicar el tipo de rama, poner la barra como separador y darle un nombre claro para que cualquiera que vea la rama sepa el trabajo que se está realizando en ella.

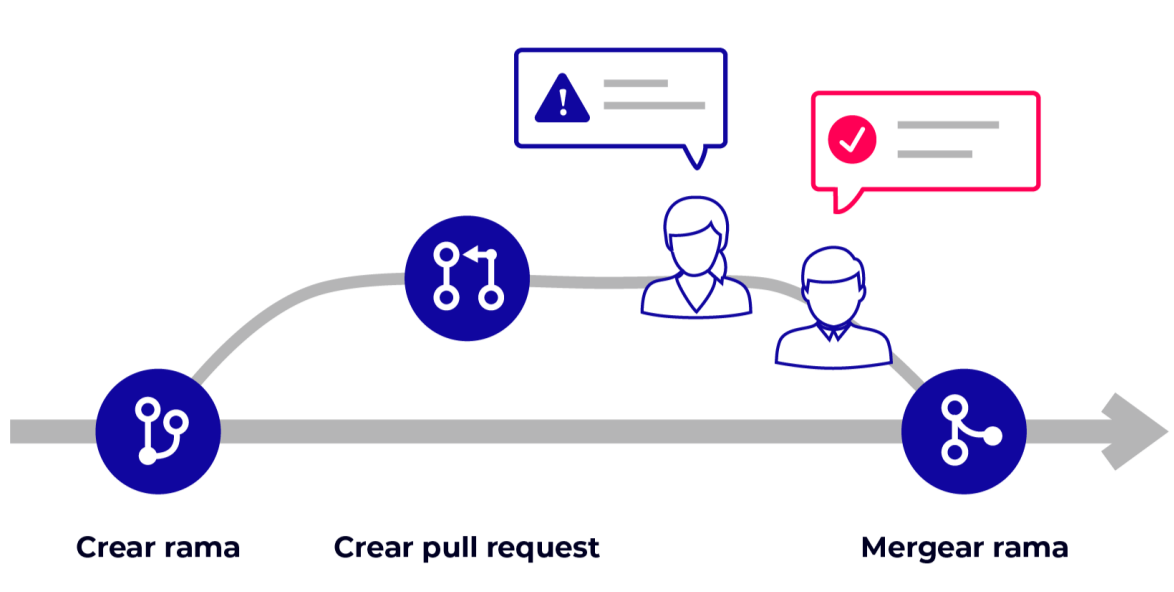
GitHub Flow

Es un **flujo definido por GitHub para estandarizar la forma en la que todas las contribuciones en los proyectos comunitarios de GitHub tienen que hacerse**. Está pensado para equipos y proyectos en los que se realizan **despliegues regularmente**. Este flujo también puede **aplicarse fuera de GitHub como sucede con GitFlow, ya que sigue el mismo patrón de centralizado y distribuido**.

En este modelo solo hay una regla: **todo el código que hay en la rama master tiene que poder desplegarse en cualquier momento.** Por este motivo, siempre que se quiera hacer un cambio, hay que crear una rama a partir de la rama master y darle un nombre descriptivo. En esta nueva rama creada se pueden hacer tantos commits como se quiera y experimentar, sabiendo que la rama no será fusionada (merge) con master hasta que alguien la revise y la apruebe. El mensaje incluido en cada commit tiene que ser descriptivo y detallado para que la persona o personas que revisen los cambios hechos en la rama puedan entender qué se van a encontrar al leer el histórico de commits.

Una vez terminado el trabajo en la rama y pasados todos los tests, se tiene que crear una **pull request** (petición de fusión) de tu rama con la rama master. Una pull request sirve para hacer **"code review"** (revisión de código), que es una técnica de mejora de calidad de código y transmisión efectiva del conocimiento, en el que cada funcionalidad o pieza de código que tenga cierta entidad pasa por un proceso de revisión donde las personas encargadas, que tendrán que ser añadidas para revisar esa pull request, comprueban el código y sugieren mejoras en el mismo a través de comentarios en la pull request. Una vez se resuelvan los comentarios o puntos pendientes y aprueben la pull request, esta se podrá fusionar con master.

Cuando los cambios han sido fusionados, se harán tests de integración para comprobar que funciona correctamente con el código que ya había y se desplegará una nueva versión en producción. Si en algún momento hay algún problema, se puede hacer **"rollback"** (volver a una versión anterior que funciona) usando la versión anterior de la rama master.



Trunk based development

Esta forma de trabajar apuesta **por no usar ramas**, para así evitar los **merges grandes** que pueden surgir de no hacer merge durante días o semanas. Para que esta técnica funcione los **equipos deben ser maduros**, ya que no están haciendo commits en una rama de código que nadie ve, sino en la rama principal que usan todos los desarrolladores. **Romper la build** y subir algo que no pase los tests, o que no compile, **puede parar a decenas de personas**, por ello el equipo debe ser experimentado. Es posible, además, que **se suban cambios a la rama principal que no están acabados**; es código que va a ir a producción y, sin embargo, no debe ejecutarse.

La **ventaja principal de este enfoque es que las integraciones de los cambios en la rama principal aunque se hacen más frecuentemente, éstas son más sencillas de resolver ya que existen menos posibilidad de conflicto al ser más acotadas y por lo tanto se reduce el número de errores que estos conllevan.**

Entorno local

No todos los lenguajes son compilados, **ciertos lenguajes como Java, Kotlin, C y otros, requieren el uso de un compilador.** En el caso de **algunos lenguajes, este viene incluido en el IDE.** Pero normalmente queremos tener el compilador instalado y configurado por fuera del IDE para poder compilar o ejecutar la batería de tests y verificar que los cambios realizados recientemente no hayan introducido nuevos errores o bugs al código ya existente.

Otras herramientas que encontramos **en un entorno local de desarrollo** son aquellas que nos **permiten ejecutar el código en un entorno similar al productivo con la principal finalidad de validar que la aplicación cumple con los requisitos funcionales mínimos y que no tiene fallos antes de publicar dichos cambios en los repositorios y/o promoverlos a otros ambientes.** Entre estas herramientas podemos encontrar **servidores web como Apache o Nginx, servidores de aplicaciones como Apache Tomcat, JBoss Wildfly, Jetty, Liberty,** que incluso pueden ser ejecutados localmente dentro de contenedores Docker, o servidores de desarrollo incluidos en la tecnología que estemos usando como, por ejemplo, Angular CLI. Otras **herramientas que se suelen utilizar para estas pruebas** en el desarrollo mobile son los **emuladores de sistemas operativos móviles, o incluso un móvil conectado al ordenador y** al entorno de desarrollo, para ejecutar y verificar la aplicación en un entorno lo más cercano al real. Hablaremos de la función de estas herramientas más adelante.

Muchas aplicaciones utilizan **bases de datos** para el almacenamiento de información o datos del negocio. Para poder desarrollar en un entorno similar al productivo, muchas veces es necesario **utilizar bases de datos de forma local**, ya sean ejecutadas en memoria dentro **de la aplicación cómo podrían ser H2 o SQLite, o instaladas localmente (o bien usando algún mecanismo de virtualización ligero como Docker), como PostgreSQL, MySQL o MariaDB entre otras.** Además, estas bases de datos se suelen utilizar para

tener algunos **datos de prueba**, y probar modificaciones sobre el esquema o los datos **sin impactar al resto de entornos que están siendo usados por muchas personas.**



Base de Datos

autentia


¿Qué es?

Una base de datos es una especie de “almacén” que nos permite guardar grandes cantidades de información para su posterior **recuperación, análisis y/o transmisión**. Podemos encontrar diversos tipos de bases de datos y se clasifican de acuerdo a las necesidades que busquen solucionar en dos grandes bloques.

 TIPOS

Concepto	Definición	Ejemplos
Relacionales	Los datos se relacionan entre ellos a través de identificadores en tablas. El lenguaje predominante es SQL (Structured Query Language). Aportan a los sistemas mayor robustez y ser menos vulnerables ante fallos gracias a su atomicidad, consistencia, aislamiento y durabilidad (ACID) .	MySQL, PostgreSQL, Oracle, SQLite.
No relacionales	No siguen un patrón fijo como estructura de almacenamiento por lo que su uso es muy común cuando no se tiene un esquema exacto de lo que se va a almacenar . Se pueden encontrar bases de datos de gráficos, orientada a documentos, de columnas (Wide column store) o de claves-valor. Algunas desventajas es que no todas contemplan la atomicidad ni la integridad de los datos.	MongoDB, Redis, Elasticsearch, Cassandra, DynamoDB.

No Relacionales	Definición	Ejemplos
Clave-Valor	Cada elemento en la base de datos se almacena como un par (clave-valor) donde la clave sirve como un identificador único. Tanto la clave como valor pueden ser cualquier tipo de primitivo, como objetos compuestos.	DynamoDB, Redis, Riak, Voldemort
Columnas (Wide column stores)	También conocidas como familia de columnas o base de datos columnar. Permiten manejar un gran volumen de datos y mezclan conceptos de las bases de datos relaciones con una base de datos clave-valor. Almacenan tablas de datos como secciones de columnas en lugar de filas de datos y en cada sección se puede encontrar elementos con clave-valor	Cassandra ,HBase, Microsoft Azure Cosmos




Base de Datos

auténtia

¿Qué es?

Una base de datos es una especie de “almacén” que nos permite guardar grandes cantidades de información para su posterior **recuperación, análisis y/o transmisión**. Podemos encontrar diversos tipos de bases de datos y se clasifican de acuerdo a las necesidades que busquen solucionar en dos grandes bloques.


TIPOS

No Relacionales	Definición	Ejemplo
Documentales	Los datos se almacenan y se consultan como documentos tipo JSON. Los documentos pueden contener muchos pares diferentes clave-valor, o incluso documentos anidados. Son más flexibles al cambio ya que si el modelo de datos necesita cambiar, solo se deben actualizar los documentos afectados y no todo el esquema.	MongoDB
De Gráfos	Usan nodos para almacenar entidades de datos y aristas para almacenar las relaciones entre nodos. El valor de estas bases de datos se obtiene de las relaciones entre nodos y no hay un límite para el número de relaciones que un nodo pueda tener. Un borde siempre tiene un nodo de inicio, un nodo final, un tipo y una dirección.	Neo4J, HyperGraphDB

Product

Users

Leads

SQL

Key-Value

Graph DB

Column Family

Document

NO SQL

Adicionalmente, podemos ejecutar en nuestro entorno local herramientas como **Sonar**, para realizar inspecciones preliminares del código y verificar su calidad. Estas verificaciones se pueden lanzar también mediante plugins integrados en el IDE.