unir
LA UNIVERSIDAD DE INTERNET

FORMACIÓN PROFESIONAL

UNIDAD FORMATIVA 5

Herramientas de gestión de ciclo de vida

Ciclo de vida

Índice

Ciclo de vida **2**

Objetivos **2**

Definición de ciclo de vida del software **2**

Qué hace falta para empezar **4**

Sistemas de control de versiones **5**

Mejores prácticas en procesos ágiles **12**

Tipos de sistemas de control de versiones **14**

Introducción a Git **15**

Feature branching **19**

Semantic versioning **23**

Ciclo de vida

En esta lección se definirá un concepto de suma importancia en DevOps: el ciclo de vida del desarrollo de software (*software development life cycle* o SDLC).

Según la Wikipedia, el ciclo de vida está compuesto por una serie de fases, claramente definidas y diferenciadas, que son utilizados por ingenieros y desarrolladores de sistemas para planificar, diseñar, construir, probar y entregar sistemas de información. El objetivo no es otro que producir sistemas de alta calidad que cumplan o excedan las expectativas del cliente y siempre dentro de las restricciones de tiempo y coste.

Si bien existen muchas y muy diferentes maneras de hacerlo, aquellas en las que nos centraremos como DevOps son las que nos permitan hacerlo de manera ágil (por Agile*),* como se ha visto al principio del temario, y veremos un poco por encima el por qué de esto.

A continuación, se introducirá el concepto fundamental de los sistemas de control de versiones, sin los que el ciclo de vida en DevOps no tendría sentido alguno, y se introducirá el que sin duda es el más importante de todos los sistemas de control de versiones a día de hoy: Git.

Objetivos

1. Conocer qué es un ciclo de vida en desarrollo software y por qué el modelo Agile nos interesa desde un punto de vista DevOps.

2. Aprender qué son los sistemas de control de versiones y cuál es el papel fundamental que juegan en SDLC.

3. Introducir la herramienta Git y el concepto de los flujos de trabajo en equipo.

Definición de ciclo de vida del software

Como ya hemos introducido, el concepto de ciclo de vida del desarrollo software (SDLC) está asociado a una serie de **fases,** estructuradas y definidas, siguiendo un proceso **bien definido** según el modelo elegido.

Gestión de ciclo de vida de las aplicaciones (ALM)

La gestión del ciclo de vida de las aplicaciones (o *application lifecycle management*, ALM) es un proceso de desarrollo integral que, en los últimos años, ha crecido en complejidad debido a la naturaleza robusta y rápida de las aplicaciones de escritorio y móviles actuales. Abarca de extremo a extremo, desde la concepción de una aplicación hasta el desmantelamiento, sin importar si la organización trabaja con métodos de gestión de proyectos *Agile*, en cascada o híbridos.

Es común equiparar ALM con el ciclo de vida de desarrollo de software. Sin embargo, este enfoque tan simple es limitado: ALM es **mucho más** que solo SDLC. De hecho, el ciclo de vida de una aplicación incluye todo el tiempo durante el que una organización está gastando e invirtiendo dinero en ese activo, desde la idea inicial hasta el final de la vida de la aplicación.

El software de ALM ayuda a las organizaciones a través del proceso de gestión de aplicaciones. Algunas suites de software de ALM se despliegan localmente y otras se ofrecen como servicio en la nube; algunas funcionan mejor para metodologías ágiles, otras para proyectos en cascada, etc. Por tanto, elegir una herramienta de gestión del ciclo de vida de la aplicación es una decisión compleja, pero sea cual sea la decisión, hay muchos beneficios al usar software ALM, por ejemplo:

- Puede ayudar a reducir el tiempo de la salida del producto al mercado (*time to market*).
- Mejora la colaboración y la comunicación entre los miembros del equipo.
- Ayuda a las organizaciones a cumplir con las normas y estándares del gobierno o de la industria.
- Aumenta la trazabilidad, visibilidad y estabilidad de la gestión de proyectos.

Modelos

Existen una serie de modelos definidos, de los cuales el ejemplo más conocido es el modelo en cascada o *waterfall*, llamado así por su enfoque lineal y estructurado, desde la captura de requisitos hasta el despliegue de la aplicación:

- **Análisis de requisitos** - Implica contacto con clientes, conocer el mercado y hacer un análisis de viabilidad del producto antes de empezar a volcar recursos en él.
- **Planificación** - Se crean objetivos concretos y se ponen en forma de requisitos que el producto final deberá cumplir, identificar *gaps* y potenciales problemas y crear propuestas.
- **Diseño** - Se formalizan los requisitos en un diseño concreto de arquitectura, UI, reglas… En este punto se debe contar con el acuerdo de todas las partes para saber si tiene sentido el producto o hay que volver a reconsiderar el todo o la partes antes de pasar a la siguiente fase.
- **Desarrollo de software** - La parte que quizá nos resulta más conocida: usando como entrada los requisitos que hay que cumplir, se pasa a crear el producto que los cumpla.
- **Testing** - Todas las baterías de pruebas que nos garantizarán que el producto cumple con las expectativas.
- **Despliegue** - La puesta en marcha y mantenimiento de la aplicación.

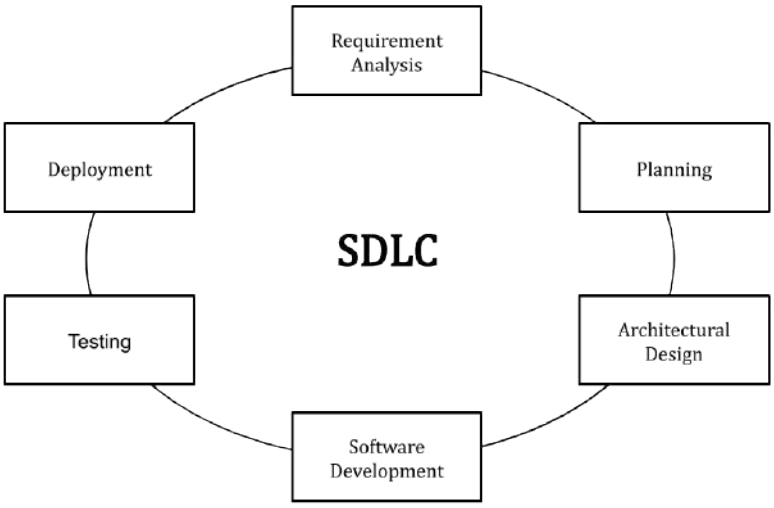


Imagen: fases del SDLC clásico (modelo en cascada). Fuente: Stackify.com.

Como se puede ver, este modelo clásico no se corresponde con todo lo que entendemos por un modo de trabajo ágil: de hecho, vamos a recordar los principios del manifiesto por el desarrollo ágil de software:

- **Individuos e interacciones** sobre procesos y herramientas.
- **Software funcionando** sobre documentación extensiva.
- **Colaboración con el cliente** sobre negociación contractual.
- **Respuesta ante el cambio** sobre seguir un plan.

Con esto queda claro que estos dos modelos **no son compatibles** y, desde un punto de vista DevOps, nos interesamos más por el modelo Agile, pues es el que nos permite aplicar las técnicas y disciplinas que estamos repasando, una y otra vez:

- Ciclos rápidos de pruebas y desarrollo.
- Integración continua.
    - Si las condiciones se dan, incluso **despliegue** continuo.
- Adaptación a requisitos cambiantes mediante metodologías como Scrum o XP.

Qué hace falta para empezar

Adaptar una organización para modificar los procesos y migrar a un modelo Agile no es inmediato y tiene una serie de requisitos fundamentales que se deben cumplir:

- **Control de versiones**. Todo debe estar versionado en un repositorio: código, pruebas, scripts de base de datos, scripts de construcción y despliegue, etc. Da igual el tamaño del proyecto y el número de individuos involucrados: cualquier proyecto de software debe usar un sistema de control de versiones.
- **Una construcción automática.** Aunque los IDEs facilitan la construcción de un proyecto sencillo, la situación se complica con aplicaciones profesionales que se ofrecen como servicio. En cualquier caso, la construcción del proyecto, que puede implicar compilación, empaquetado de recursos y almacenamiento en un repositorio, debe estar automatizada de principio a fin. Si el lenguaje de programación o el sistema de despliegue ofrecen utilidades de compilación, empaquetado, etc., hay que aprovecharlas. Además, todos los scripts de esta tarea deben estar también versionados.
- **Batería de pruebas completa**. Si la batería de pruebas no es lo suficientemente completa, una ejecución satisfactoria realmente no dice nada sobre la fiabilidad de los cambios. Las pruebas deben cubrir suficientes casos como para ser capaces de detectar alteraciones en el comportamiento del código y en las funcionalidades de la aplicación. Al igual que la fase de construcción, las pruebas deben ejecutarse de manera automática y deben estar versionadas junto al código.

- **Aceptación del equipo**. Los conceptos de integración, entrega y despliegue continuos son una metodología, no una herramienta. Requiere un cierto nivel de implicación y disciplina por parte del equipo. Si uno de los desarrolladores, acostumbrado a trabajar en largas iteraciones y a reportar sus tareas cada mes, no se adapta a integrar sus cambios habitualmente, no recibirá valor alguno del sistema de CICD, por muy avanzada que sea la implementación del *pipeline*. El objetivo final no es integrar rápido, sino aumentar la calidad del producto final. Los equipos deben empezar por asimilar este objetivo, hacer suyos los procesos y la calidad aumentará por sí sola. La aceptación del equipo es una transformación cultural que, si bien puede partir de la iniciativa de unos pocos miembros de los equipos DevOps, debe contar con el apoyo de la organización para su cumplimiento.

En este tema, nos vamos a centrar en los dos primeros puntos.

Sistemas de control de versiones

A medida que los equipos diseñan, desarrollan y despliegan software es común que se desplieguen varias versiones del mismo software en diferentes sitios y que los desarrolladores trabajen simultáneamente en las actualizaciones. Por lo tanto, los fallos a corregir y las características nuevas (*bugs* y *features*) solo están presentes en ciertas versiones.

Con el propósito de localizar y corregir errores, resulta de vital importancia poder recuperar y ejecutar diferentes versiones del software para determinar en qué versión o versiones se produce el problema. También puede ser necesario desarrollar dos o más versiones del software **al mismo tiempo**; por ejemplo, cuando una versión tiene errores corregidos, pero no hay nuevas características, mientras que en otra se trabajan las nuevas características.

Para trabajar en equipo es necesaria alguna forma de coordinar el código fuente, pruebas y otros elementos importantes del proyecto. Un sistema de control de versiones proporciona un **repositorio central** que ayuda a coordinar los cambios en los archivos y también proporciona un historial de cambios.

- Un proyecto **sin** control de versiones puede tener fragmentos de código dispersos entre ordenadores de los desarrolladores, unidades en red e incluso medios extraíbles. El proceso de construcción puede involucrar a una o más personas que buscan encontrar las últimas versiones de varios archivos, tratando de ponerlos en los lugares correctos. Esto imposibilita automatizar los despliegues y aumenta el riesgo de estos.

- Un proyecto **con** control de versiones centraliza este proceso de administración de cambios. Es un proceso ordenado en el que los desarrolladores obtienen el código más reciente del servidor, realizan su trabajo, ejecutan todas las pruebas para confirmar que su código funciona y, a continuación, suben sus cambios al repositorio mediante uno o varios *commits*. Este proceso puede ocurrir múltiples veces al día. Si en cada subida de cambios se construyen los paquetes y se ejecutan las pruebas se puede decir que es un proceso de integración continua.

Vocabulario común para el control de versiones

Para entender lo que implica trabajar con control de versiones es necesario utilizar un vocabulario común:

- **Repositorio**

Es el almacenamiento maestro de todos los archivos y el historial de cambios de estos. Se almacena en el servidor de control de versiones. Cada proyecto autónomo debe tener su propio repositorio, aunque un único proyecto puede estar repartido en más de un repositorio.

- **Sandbox**

También se conoce como copia de trabajo. Contiene una copia de todos los archivos del repositorio de un punto en particular. Cada desarrollador mantiene su propia copia de trabajo a partir del contenido del repositorio.

- **Check-out**

Es el proceso de inicializar una copia de trabajo a partir de un punto concreto de un repositorio. En algunos sistemas de control de versiones este proceso se define con el término *update and lock* o actualizar y bloquear.

- **Update**

Actualización de la sandbox para obtener los últimos cambios desde el repositorio. También se puede actualizar a un punto en particular en el pasado.

- **Lock**

Un bloqueo hace posible que nadie pueda editar un archivo si el desarrollador lo ha bloqueado.

- **Check-in o *commit***

Registro de los cambios efectuados en la copia de trabajo. Es el proceso fundamental para guardar los cambios en el repositorio. Los cambios de la copia de trabajo son efímeros desde el punto de vista del repositorio; es decir, aunque los archivos estén guardados en el disco duro, los cambios entre el último *commit* y el estado actual no están registrados en el repositorio y, por tanto, no existen en el histórico del control de versiones a menos que se registren en un *commit*.

- **Revert**

Destruye la sandbox para descartar los cambios y volver al punto de la última actualización. Esto es útil cuando el código de la copia de trabajo actual se ha vuelto inestable y no es posible hacerlo funcionar de nuevo. A veces, revertir es más rápido que depurar, especialmente si hay *commits* recientes.

- **Tip o head**

La cabecera del repositorio contiene los cambios más recientes que se han registrado. Al actualizar la copia de trabajo, los archivos quedan en el estado de la cabecera. Si el sistema de control de versiones soporta ramas, cada rama tiene su propia cabecera, por lo que la copia de trabajo se puede actualizar a la cabecera de cada una de las ramas.

- **Tag o label**

Una etiqueta marca un *commit* determinado en el historial del repositorio, lo que permite acceder fácilmente a él de nuevo. Pueden usarse para indicar una versión liberada, o *release*, o un punto de pase a producción. Aunque es posible borrarlas, no deberían ser modificadas.

- **Rollback**

El proceso de deshacer un *commit* para que los cambios que ha introducido desaparezcan de la cabecera del repositorio. El mecanismo para hacerlo varía dependiendo del sistema de control de versiones: en unos casos se genera un segundo *commit* B que anula los cambios del anterior *commit* A, por lo que los archivos vuelven al estado anterior: al *commit* A; en otros casos se puede eliminar el *commit* A completamente.

- **Rama**

Las ramas permiten dividir el repositorio en distintos historiales alternativos. Las ramas suelen partir de un *commit* común, pero a partir de ese momento los cambios registrados en cada rama pueden divergir tanto como sea necesario. Es habitual, por ejemplo, separar en ramas el trabajo en diferentes características nuevas, en arreglos o en entornos de trabajo. Las ramas apuntan a un *commit*, al igual que las etiquetas, pero se actualizan con cada nuevo *commit* (las etiquetas no se modifican una vez asignadas a un *commit*).

- **Fusión o *merge***

Es el proceso de combinar cambios de dos ramas. Si dos programadores hacen un cambio a uno o varios archivos, cada uno en una rama por separado, y ambos hacen un check-in de los cambios, el segundo programador tendrá que fusionar los cambios de la primera persona. Las herramientas más modernas ayudan en este proceso e incluso lo hacen automáticamente si los cambios no afectan a las mismas líneas de código.

- **Resolución de conflictos**

Una fusión, o *merge*, se puede llevar a cabo automáticamente si los cambios de dos *commits* afectan a archivos diferentes o a partes diferentes de un archivo. Sin embargo, si los cambios afectan a la misma sección de un archivo, las herramientas no son capaces de resolverlo y el desarrollador debe encargarse de identificar el estado original del fichero, los cambios introducidos por el otro desarrollador y sus propios cambios, y decidir cómo resolverlo. Los sistemas de control de versiones suelen resaltar los conflictos y muestran, en un mismo editor, los cambios de ambos *commits* con el objetivo de facilitar la vida al desarrollador.

Ventajas del control de versiones

Las siguientes son algunas de las ventajas y facilidades que aporta un sistema de control de versiones.

- **Edición simultánea**

Si varios desarrolladores modifican el mismo archivo sin utilizar el control de versiones (por ejemplo, si los archivos se almacenan en una unidad compartida), es probable que uno sobrescriba accidentalmente los cambios de otro. Para evitar este inconveniente, algunos sistemas de control de versiones recurren a un modelo de bloqueo de versiones: cuando un desarrollador trabaja en un archivo, lo bloquea para evitar que alguien más realice cambios. Los archivos, por tanto, son de solo lectura mientras están bloqueados.

Si bien este enfoque resuelve el problema de sobrescribir accidentalmente los cambios, puede causar otros problemas. **Un modelo de bloqueo hace que sea difícil hacer cambios**. Los miembros del equipo tienen que coordinar cuidadosamente quién está trabajando en qué archivo y esto limita su capacidad de trabajar en nuevas funcionalidades que impliquen cambios en múltiples archivos. Para evitar esto, los equipos a menudo recurren a la propiedad de código fuerte, o *strong code ownership*, el peor de los modelos de propiedad de código: solo una persona tiene la autoridad para modificar un archivo en particular. El enfoque opuesto, propiedad de código colectivo, o *collective code ownership*, facilita el reparto de tareas en un equipo, pero es difícil de llevar a cabo si se utiliza el bloqueo de archivos.

En lugar del bloqueo, es mejor utilizar un modelo concurrente de control de versiones. Este modelo permite a dos personas editar el mismo archivo simultáneamente. El sistema de control de versiones fusiona automáticamente los cambios: nada se sobrescribe accidentalmente. Si dos desarrolladores modifican exactamente las mismas líneas de código, el sistema de control de versiones detecta un conflicto y ofrece la posibilidad a los autores de combinar sus cambios manualmente.

Las fusiones automáticas pueden parecer arriesgadas, pero en la práctica funcionan bien en entornos de integración continua con construcción automatizada. La integración continua reduce el alcance de las fusiones a un nivel manejable (hacer múltiples cambios de código pequeños en lugar de uno solo grande) y la construcción, con su batería de pruebas completas, confirma que la integración de código funciona correctamente.

- **Posibilidad de viajar en el tiempo**

Uno de los usos más potentes de un sistema de control de versiones es la capacidad de retroceder en el tiempo. Un desarrollador puede actualizar su copia de trabajo con todos los archivos a un punto en particular en el pasado. Por ejemplo, si se ha detectado un error en una versión ya publicada de la aplicación, pero no se encuentra la causa, el desarrollador puede aplicar diffdebugging:

- Actualiza la copia de trabajo a un punto en el que el error no existía.
- A continuación, actualiza la copia de trabajo con los siguientes cambios por orden histórico hasta que logra aislar el *commit* exacto que introdujo el error.
- En ese punto, el desarrollador solo tiene que revisar los cambios de ese *commit* para localizar el error. Si se trabaja con integración continua y el número de cambios es pequeño, el error será fácil de corregir.

El viaje a través del tiempo también es útil para reproducir errores. Si un usuario reporta un error y el equipo de soporte no puede reproducirlo en la última versión, siempre puede intentar usar la misma versión del código que está usando quien lo reporta. Este método es fácil de llevar a cabo para versiones públicas de aplicaciones discretas, ya que probablemente el equipo de soporte tendrá acceso a un almacén con los binarios de todas las versiones públicas, pero si el error ocurre en una aplicación ofrecida como servicio con múltiples despliegues al día, es probable que solo se pueda llegar a la versión exacta que estaba usando el cliente seleccionando un *commit* concreto en el árbol de cambios.

- **Almacenamiento del proyecto completo en el repositorio**

En todo lo explicado hasta el momento se ha dado por supuesto que el control de versiones está destinado a almacenar el código fuente. No obstante, cualquier otro fichero de texto es susceptible de ser versionado: ficheros de configuración, scripts de base de datos, documentación, etc. La funcionalidad de retroceder en el tiempo no brinda ningún beneficio a menos que sea posible construir el entorno exacto de la aplicación en un momento dado. Almacenar todo el proyecto en el control de versiones, incluido el sistema de construcción, compilación, pruebas y despliegue, ofrece la posibilidad de volver a crear versiones antiguas del proyecto.

Siempre que sea posible, es recomendable mantener en el control de versiones todas las herramientas, librerías, documentación y cualquier elemento relacionado con el proyecto. Las herramientas y librerías son particularmente importantes ya que, si se dejan fuera, en algún momento se actualizará una de ellas y ya no se podrá volver a un punto anterior a la actualización, salvo manualmente (con la posibilidad de fallo humano que esto implica).

Por razones similares, puede ser recomendable almacenar todo el proyecto en un único repositorio. A pesar de que pueda parecer natural dividir el proyecto en múltiples repositorios, quizás uno para cada módulo, o uno para el código fuente y otro para la documentación, este enfoque aumenta las oportunidades para que aparezcan desfases entre repositorios. Sin embargo, decidir el límite del repositorio tiene unas fronteras menos definidas y hay situaciones en las que tiene sentido repartir el código en múltiples repositorios. Una regla útil es identificar los componentes que se van a desplegar juntos.

Si en cada versión se van a publicar siempre una serie de piezas del código, esas piezas deben ir en el mismo repositorio (en este contexto, pieza puede significar un módulo, un paquete, un servicio, etc., en función del tipo de aplicación). Si el sistema está compuesto por servicios independientes con flujos de vida y fechas de despliegue diferentes, es posible que las tareas de integración y entrega continua sean más sencillas si cada servicio se almacena en un repositorio diferente.

Lo único relacionado con el proyecto que **nunca debería guardarse** en el control de versiones es el código generado y las librerías de terceros:

- En el caso de lenguajes compilados, la compilación debería volver a crear el mismo binario automáticamente. Esto se puede garantizar si, además del código, se versionan las herramientas de compilación, es decir, el fichero Makefile con los parámetros de 'gcc' o el fichero 'package.json' con las dependencias y comandos de construcción de JavaScript.

- Los ficheros autogenerados pueden ser, por ejemplo, documentación en formato HTML, con tablas de contenidos e índices, construida a partir de ficheros 'markdown', o clientes de API generados a partir de una definición de OpenAPI.

- Un fichero OpenAPI define los servicios ofrecidos por una API rest y los métodos y esquemas permitidos en cada servicio. Los ficheros OpenAPI se suelen definir en JSON o YAML, por lo que son candidatos ideales para ser versionados. Herramientas con Swagger Codegen permiten crear librerías en múltiples lenguajes a partir de una definición de OpenAPI. Incluso aunque la librería se genere en un lenguaje no compilado y, por tanto, sea posible almacenar el código fuente en el control de versiones, es más útil almacenar solo el fichero OpenAPI junto a los comandos para generar el cliente (por ejemplo, en un script de bash) para permitir que en cada construcción del cliente siempre se use la versión del fichero OpenAPI de la copia de trabajo.

- Paquetes de instalación o distribución, como instaladores de Windows, paquetes 'deb' o 'rpm' o archivos 'tgz' o 'jar'.

- Librerías como paquetes de Python o NPM, o las dependencias de un proyecto Maven.

La información de usuario también debe ir al repositorio. Esto incluye la documentación, notas sobre requisitos, escritura técnica (como manuales y guías) y pruebas de usuario.

Hay una excepción adicional en cuanto a lo que no debe almacenar en control de versiones: código que se va a desechar. Estos son los experimentos, test, proyectos de investigación que permanecen sin integrarse, etc. Algunos desarrolladores mantienen repositorios propios para almacenar este tipo de información.

Usualmente, los programadores se preocupan de que el sistema de control de versiones no sea demasiado complejo para que los usuarios lo utilicen, ya que hay una curva de aprendizaje importante para quien no lo haya usado. Si bien es cierto que algunos sistemas de control de versiones son complejos, la mayoría tienen interfaces fáciles de usar. Por ejemplo, el cliente de TortoiseSvn Windows para el sistema de control de versiones de código abierto Subversion es sencillo de usar y hay múltiples interfaces gráficas para Git, como Sourcetree o SmartGit.

- **Ficheros binarios**

En el apartado anterior, se mencionaba que cualquier fichero de texto es susceptible de ser versionado. Los sistemas de control de versiones suelen permitir el almacenado de ficheros binarios, pero en ese caso ofrecen una funcionalidad más limitada a la hora de detectar los cambios. Mientras que en un fichero de texto es posible identificar qué líneas se han modificado en cada *commit* (y quién y cuándo las modificó), en el caso de los archivos binarios solo se almacena la información de cambios a nivel de archivo, es decir, si el archivo ha sido modificado o no, sin especificar qué bytes o secciones del fichero.

Sin embargo, sistemas como Git permiten usar herramientas externas para extraer contenido de texto de un fichero binario y guardar los cambios sobre este texto. El control de versiones almacenará los cambios sobre el fichero binario, pero las herramientas de comparación de cambios podrán trabajar con ficheros de texto. Por ejemplo, se puede versionar un fichero PDF y usar la herramienta 'pdfinfo' para comparar detalles del contenido entre dos versiones.

- **Código limpio y ordenado**

Uno de los enunciados más importantes en XP (*eXtreme Programming*, una célebre práctica Agile) es el de mantener el código limpio y listo para ser empaquetado. Esto comienza por la copia de trabajo de cada desarrollador. Aunque es necesario romper la construcción, es decir, trabajar con código que no es compilable durante un intervalo de tiempo para seguir avanzando en una funcionalidad, esta situación solo debe quedarse a nivel de dicha copia de trabajo.

Un desarrollador no debería hacer un *commit* de código que no sea compilable o no complete las pruebas satisfactoriamente. Esto permite que cualquier persona pueda hacer actualizaciones en cualquier momento, sin preocuparse por romper el código y que, a su vez, todo el equipo trabaje sin problemas y comparta los cambios fácilmente. A grandes rasgos, el código atraviesa **cuatro niveles de completitud**:

- **Roto**. El código no compila o no pasa las pruebas. Esto solo debería ocurrir en una sandbox.
- **Se construye y pasa todas las pruebas**. Todas las versiones del repositorio deben estar, al menos, en este nivel.
- **Listo para demo**. Cualquier versión identificada con etiquetas por el desarrollador o enlazada con sistemas de gestión del ciclo de vida de la aplicación, para que las partes interesadas lo prueben.
- **Listo para ser entregado a los usuarios y clientes reales**. Esta versión está lista para pasar a producción y debe haber pasado no solo las pruebas unitarias sino pruebas de integración, validación de usuario, etc.

- **Código base único**

Uno de los errores más devastadores que puede cometer un equipo es duplicar su código base. Lamentablemente, es fácil llegar a esta situación. Por ejemplo, un cliente solicita inocentemente una versión personalizada del software. Para entregar esta versión rápidamente, lo más simple parece ser duplicar el código base, realizar los cambios necesarios, construir el paquete y enviarlo. Sin embargo, copiar y pegar esa personalización dobla el número de líneas de código a las que hay que hacer mantenimiento. Es casi imposible recombinar una base de código duplicada de manera inmediata. Este error contribuye, además, a aumentar la deuda técnica considerablemente (la sección 'A fondo' incluye un recurso con más detalles sobre el concepto de la deuda técnica).

Desafortunadamente, los sistemas de control de versiones realmente hacen que este error sea aún más fácil de cometer. La mayoría de estos sistemas ofrecen la opción de crear ramas, es decir, dividir el repositorio en dos líneas separadas de desarrollo. Esto es esencialmente lo mismo que duplicar el código, incluso aunque a nivel de almacenamiento solo se almacenen los cambios.

Es beneficioso utilizar las ramas, pero usarlas para proporcionar múltiples versiones customizadas de software es arriesgado. Aunque los sistemas de control de versiones proporcionan mecanismos para mantener sincronizadas varias ramas, hacerlo es un trabajo tedioso que se hace cada vez más difícil con el tiempo. En su lugar, es recomendable diseñar código que admita varias configuraciones. Por ejemplo, con una arquitectura de plugins, con un archivo de configuración editable por cada usuario o con un sistema de perfiles en el que cada usuario tenga acceso a unas funcionalidades u otras en función de la configuración de su perfil. El sistema de construcción puede, además, generar múltiples versiones a partir de múltiples ficheros de configuración.

Mejores prácticas en procesos ágiles

El control de versiones no es un proceso exclusivo de las metodologías ágiles. No obstante, merece la pena resaltar lo siguiente:

- Aunque es raro, todavía hay equipos que utilizan herramientas o prácticas de control de versiones desactualizadas, e incluso que no han adoptado ninguna herramienta.
- El control de versiones no es meramente una buena práctica sino un habilitador de varias prácticas ágiles, tales como la integración continua.
- La comunidad ágil se inclina hacia herramientas y prácticas que facilitan el trabajo concurrente y favorecen modelos distribuidos en lugar de los centralizados.
- Es, por tanto, beneficioso para un equipo ágil reflexionar explícitamente sobre su infraestructura y políticas de control de versiones.

Estas son algunas de las **mejores prácticas** a tener en cuenta cuando se trabaja con control de versiones:

- Si no está en control de versiones, no existe.
    - El código que reside en la máquina de un desarrollador no le vale a nadie más y se puede perder.
    - Se recomienda habitualmente subir el código al control de versiones al final del día (siempre que no rompa la construcción, o en cualquier caso si se trata de ramas privadas).
- Conviene hacer *commit* con frecuencia.
    - Los *commits* deben ser una agrupación lógica de cambios normalmente pequeña.
    - Los *commits* frecuentes y pequeños hacen que los procesos de fusión y de retroceso sean más simples y manejables.
- Hay que verificar los cambios antes de hacer *commit*.
    - Es útil usar una herramienta diferente para inspeccionar los cambios. Así se evita incluir cambios temporales (como comentarios o sentencias de log).
    - Esto es especialmente necesario tras solucionar conflictos. Un conflicto mal solucionado puede introducir cambios inesperados en el código, incluso aunque este compile correctamente.
- Cada *commit* debe ser probado.
    - Debe ser razonablemente seguro retroceder a cualquier *commit* y así asegurar que la aplicación se construye y las pruebas finalizan correctamente.

- La división en ramas facilita el trabajo.
    - La mayoría de las funcionalidades o los arreglos de errores importantes deben tener una rama propia.
    - No es recomendable trabajar en la rama por defecto o principal (antes denominada máster). Ramificar aísla el trabajo en secciones limpias y manejables y mantiene la rama máster lo más estable posible.
- Los mensajes de los *commits* son esenciales en el trabajo colaborativo.
    - El mensaje incluido en un *commit* debe describir en detalle qué cambios incluye. Los mensajes del estilo 'arreglos' o 'añadida recomendación de mi compañero' aportan poco valor a quien repase el histórico de cambios de un fichero. Sin embargo, comentarios con referencias a la funcionalidad o al error, como 'soporte de SAML en el inicio de sesión' o 'corrección de la vulnerabilidad *heartbleed*' aportan contexto a los desarrolladores que necesiten trabajar en esas secciones del código más adelante.
    - Se pueden incluir referencias a tareas, incidentes o proyectos de otras herramientas. Algunos sistemas son capaces de detectar estas incidencias y enlazar automáticamente los *commits* con dichos sistemas.
- No hay que versionar los artefactos generados.
    - Los artefactos tales como clases compiladas, documentos autogenerados, logs, etc., no deben incluirse en el control de versiones. En su lugar, hay que incluir las herramientas con las que se han compilado o generado.
    - Muchos sistemas de control de versiones soportan el uso de ficheros de configuración para ignorarlos automáticamente y asegurar que esos ficheros no se versionan.
- Las dependencias deben estar versionadas siempre que sea posible.
    - Otros desarrolladores deben ser capaces de instalar dependencias automáticamente para poder reproducir el estado de la aplicación.
    - El servidor que se encarga de la construcción también debe tener acceso a las dependencias.
- Los *commits* deben reflejar los cambios en la estructura de la base de datos.
    - Cuando un desarrollador actualiza su copia de trabajo con un *commit*, su entorno de desarrollo debe ser capaz de ejecutar ese código. Si el código de la aplicación asume que hay una columna nueva en una tabla concreta de la base de datos, el *commit* debe incluir los cambios necesarios en el esquema de la tabla. De lo contrario, el desarrollador debe investigar el fallo y añadir la columna manualmente.
    - Las bases de datos sin esquema también pueden aprovechar el control de versiones para definir índices o migraciones a nivel de documento.

- El control de versiones permite experimentar y optimizar el código.
    - Siempre hay un sitio al que retroceder, así que la copia de trabajo permite experimentar con una idea sin desestabilizar el código de producción.
    - La decisión de borrar un módulo o una función que dejan de ser necesarios es más sencilla, ya que se pueden recuperar del control de versiones si hacen falta. Muchas bases de código mantienen código muerto por si acaso, que no hace más que dificultar el mantenimiento de la aplicación.

Tipos de sistemas de control de versiones

Hasta ahora se ha hablado de los sistemas de control de versiones en general. Los conceptos de *commit* o rama no son específicos de un producto ni de un tipo de sistema. Para poder tratar temas más avanzados como los flujos de trabajo es necesario explicar la diferencia entre sistemas de control de versiones locales, centralizados y distribuidos.

Sistema de control de versiones locales

Los primeros sistemas de control de versiones fueron **locales**. Los cambios sobre los ficheros se registraban en una base de datos única que se almacenaba localmente. Esto implicaba dos desventajas principales: si se perdía el disco duro de ese equipo, el trabajo se echaba a perder y el trabajo en equipo no era una opción.

Dos de estos sistemas son Source Code Control System, o SCCS, y Revision Control System, o RCS. Aparte de las limitaciones ya mencionadas, estos sistemas solo podían registrar los cambios en un único fichero, no en un proyecto.

Sistema de control de versiones centralizados

Estos sistemas almacenan el histórico de cambios en un servidor al que los desarrolladores se conectan. Solo hay una base de datos, pero los ficheros pueden existir en múltiples equipos. Permiten el trabajo en equipo y siguen un modelo tan sencillo que sigue actualmente en uso. Un error en el servidor puede implicar la pérdida del proyecto, por lo que es fundamental protegerlo y mantener copias de seguridad del mismo.

El principal problema para el trabajo colaborativo es la necesidad de bloquear archivos en uso (*lock)*. Aunque esta situación es mejor que la que ofrece un sistema local, dista mucho de una situación ideal en la que cualquier desarrollador puede editar cualquier fichero. El bloqueo de ficheros es una fuente de frustración para los desarrolladores, sobre todo, cuando existen grandes ficheros con detalles comunes a todo el proyecto.

Concurrent Version System, o **CVS**, fue uno de los primeros sistemas centralizados. Apache Subversion, o **SVN**, apareció más tarde y mejoró muchas funcionalidades de CVS, como la capacidad de versionar proyectos enteros, no solo grupos de archivos.

Sistemas de control de versiones distribuidos

En un sistema distribuido, o descentralizado, no hay un único servidor que almacene la base de datos con el histórico de cambios. Cada cliente, es decir, el equipo de cada desarrollador, almacena una copia del repositorio junto al histórico de cambios. Cada repositorio puede actualizarse con los cambios de cualquier otro repositorio (del mismo proyecto, claro): por ejemplo, un desarrollador que esté trabajando en una nueva característica puede recuperar un arreglo que haya hecho un compañero suyo e incorporarlo a su trabajo.

El concepto de servidor central sigue teniendo sentido: la situación habitual es que exista un repositorio principal en un servidor y cada cliente cree un *fork*. Un *fork* no es más que un clon del repositorio original. Cuando un desarrollador quiere compartir unos cambios que tenga en su clon local, puede decidir subirlos al repositorio original o a algún otro *fork*. En un equipo de desarrolladores trabajando en el mismo proyecto, el repositorio original suele ser el principal y todos los desarrolladores envían sus cambios contra él. Esta facilidad de colaboración es la principal ventaja de los sistemas distribuidos.

Un sistema distribuido suele ser más rápido que los centralizados, ya que no necesita acceso continuo al servidor. Además, los cambios se registran como parches: al no haber una base de datos central que mantenga la historia de cambios, cada parche es autocontenido y se puede intercambiar entre repositorios.

Introducción a Git

El eje principal en la gestión de software es su almacenamiento versionado. Y, en esto, la herramienta por excelencia es Git, con el permiso de otras herramientas que ni siquiera vale la pena mencionar. La trascendencia de esta herramienta es tal, que el concepto de open source como lo conocemos no tendría sentido sin ella.

Git es un sistema de control de versiones **distribuido**, en el que el código fuente se almacena en un servidor central organizado en repositorios, pero cada usuario es libre de almacenar una copia local y subirla a donde desee.

Este servidor central almacena una o más *revisiones* del código, que popularmente conocemos como *branches* o ramas. Los usuarios se 'traen' (*pull)* los cambios de esas ramas a sus estaciones de trabajo y hacen lo que tengan que hacer.

Cuando se tienen cambios que deben llegar a los demás usuarios, los añadimos (*commit)* a esa rama y, después, los 'empujará' (*push)* al servidor central. Si todo va bien y nadie más ha modificado esa rama, el código estará disponible para que cualquiera pueda volver a hacer *pull* y tener esa aportación disponible.

Esto es el flujo habitual que, como veremos, admite multitud de matices, ya que primero tendremos que saber qué tipo de problemas vamos a tener que resolver o qué manera de trabajar es la mejor para nuestro equipo.

Interfaces web

Cuando instalamos Git en nuestro sistema operativo, lo que realmente estamos es instalando un cliente que se conectará a uno o más servidores de los que traerá el código fuente.

Estos servidores ofrecen interfaces web muy avanzadas para facilitar el uso de la herramienta, enfocadas siempre a la visualización, compartición y aportación de código fuente a los repositorios.

Sin duda, las más comunes son gitlab, Github y Bitbucket: en todos los casos existen versiones gratuitas y de pago, pensadas para empresas u organizaciones con ánimo de lucro.

Las cuentas gratuitas permiten tener repositorios públicos ilimitados (o un número bastante alto, al menos) y es una práctica usual tener una cuenta en alguno de ellos y subir allí ciertos repositorios personales:

- Programas que hemos hecho y de los que nos apetece fardar un poco, aunque también con la esperanza que alguien encuentre algo que le inspire.
- Katas o similares.
- *Forks* de proyectos open source.
- *Pet projects.*

Recomiendo encarecidamente hacerse una cuenta en alguno de ellos desde ya, aunque creamos que no la vamos a usar.

Flujos de trabajo

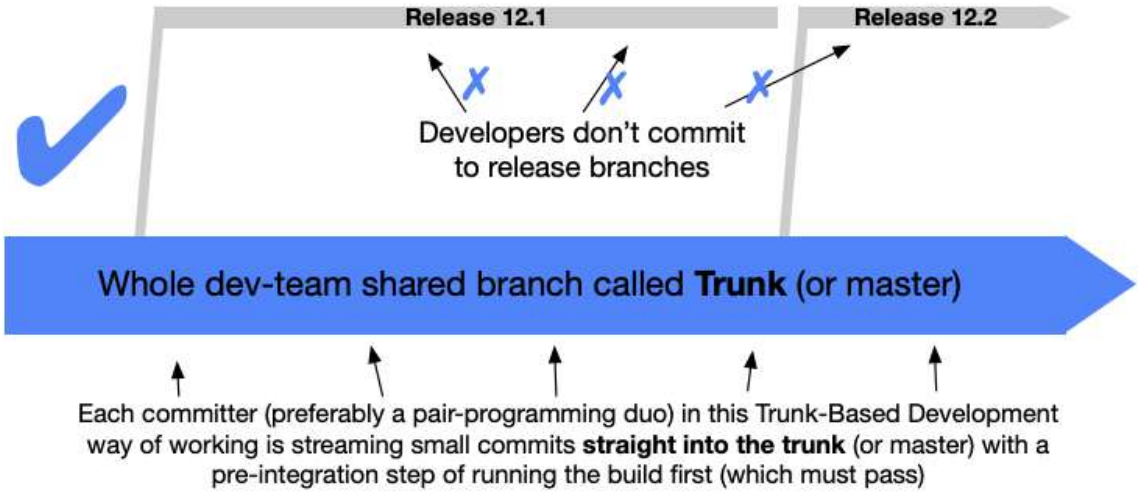
Ahora, ya sabemos donde vamos a guardar nuestro código fuente. Sabemos que podemos tener almacenadas versiones del mismo y estamos en condiciones de entregar una nueva funcionalidad.

Llegados a este punto, y si se ha leído algo de literatura al respecto, es posible que exista un poco de confusión en cómo empezar a trabajar con Git en un equipo. Esto es perfectamente entendible y, desde luego, no existe una respuesta única: depende del tamaño del equipo, de la tipología de los ciclos de *release*, del tiempo que se necesita para pasar test o para desplegar…

En todo proyecto Git hay una rama principal: *main/master* (usaremos el nombre ***main*** a partir de ahora, pues es el nombre oficial en Github desde 2020). Esta rama es en la que se supone que se debe encontrar siempre la fuente de verdad de nuestro código fuente.

Trunk-based development

Este flujo de trabajo toma el nombre de la rama *trunk*, que en SVN representa la rama principal de trabajo. La idea es que todo el equipo trabaja en una única rama de integración, y cuando se decide que tiene todo lo necesario, se genera una *release*, que pasa a ser **intocable** (salvo para *bug fixing*).



Fuente: Trunk Based Development.

Cuando arrancamos un repositorio de cero en nuestra máquina, el flujo puede ser similar a este:

```
# crear un repositorio nuevo en el directorio actual
$ git init

# añadimos un remoto con la direccion del servidor
$ git remote add origin git@github.com:username/reponame.git

# añadir algunos ficheros a la rama actual (master)
$ git add src/ README.md

# hacemos un commit de esos cambios
$ git commit -m "Añadidos ficheros iniciales"

# oops, nos hemos olvidado algo... no pasa nada, aun no hemos subido
$ vim README.md
$ git commit README.md -m "fix: actualizado README"

# Ahora ya tiene buena pinta, lo subimos al servidor
$ git push -u origin master
```

Si lo que queremos es contribuir a ese repositorio, la diferencia en este caso es que nos clonaremos el repositorio remoto y añadiremos los cambios:

```
# clonamos un repo
$ git clone git@github.com:username/reponame.git

# Modificamos ficheros

# Subimos los cambios
$ git add README.md && git commit -m "feature: new README"

# Hacemos push al servidor, ojalá no haya conflictos...
$ git push
CONFLICT (add/add): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.

$  git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
            both added:      README.md
```

Una vez el equipo decide que se han incorporado todas las funcionalidades que haya que despegar, se creará desde la rama principal una nueva rama de integración, usualmente llamada 'release/X.Y' a la que no se pueden hacer cambios (salvo errores graves).

```
# traemos los últimos cambios a master
$ git checkout master
$ git pull

# creamos nueva rama de integración y la subimos
$ git checkout -b release/1.2
$ git push -u origin release/1.2

# La marcamos con una tag (opcional)
$ git tag 1.2.0
$ git push --tags
```

Bug fixing

En el caso de detectar errores graves en la *release*, errores que exigen una acción inmediata, cabría modificar esa *release* para arreglar el error y volver a llevar el código a producción. Esto no es lo más adecuado por varias razones:

- Podría ser que nuestro proceso de despliegue esté preparado para desplegar solo la rama principal.
- Tendríamos cambios en una rama que no están en la rama principal, creando una divergencia que debe ser llevada inmediatamente a la misma.
- Estaríamos modificando una rama que, por definición, debería ser de solo lectura.

En este caso, lo recomendable sería modificar la rama principal y crear una nueva *release*, que podría llevar un sufijo indicando que arregla la anterior (por ejemplo, 'release/1.2.1').

Conclusión

Como vemos, el problema de este método de trabajo es a la hora de modificar ficheros de forma **concurrente** por varios colaboradores: es muy sencillo incurrir en estos errores si se trabaja sobre ficheros muy utilizados, por lo que no parece una buena manera de trabajar si el equipo es de más de unas pocas personas, si se va a modificar el mismo set de ficheros constantemente, o si las nuevas contribuciones van a llevar mucho tiempo de desarrollo.

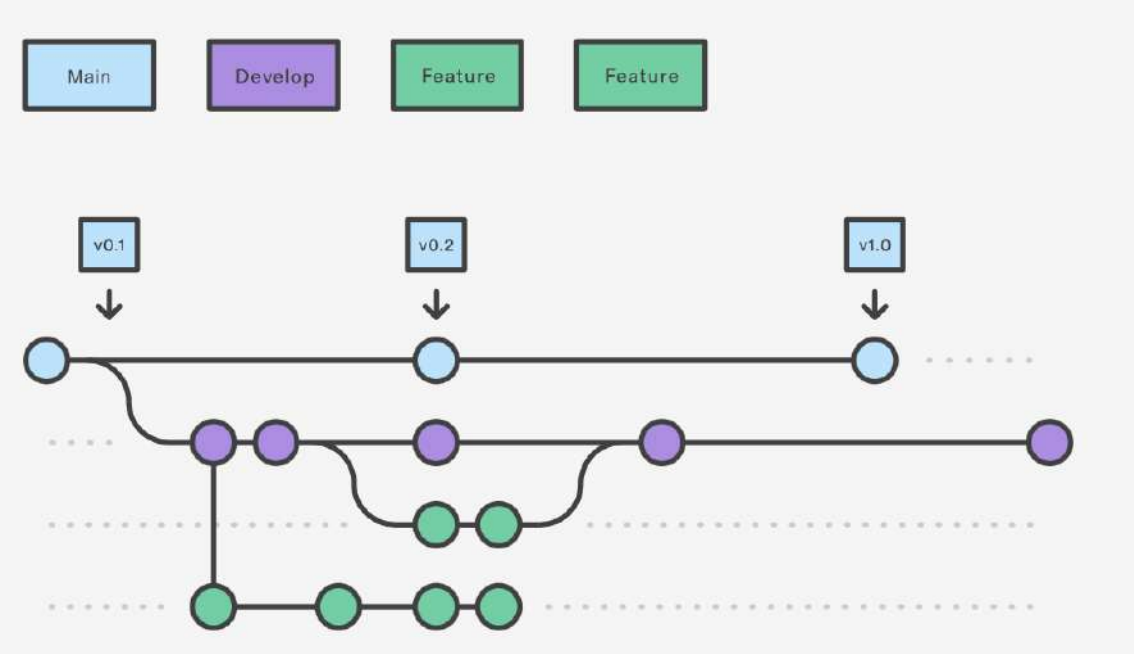
Sin embargo, este modelo **refuerza** buenas prácticas DevOps, ya que lo que precisamente se busca es que los ciclos de cambios sean **lo más cortos posible**, pues eso minimizará los conflictos y aumentará la velocidad. Además, es más fácil asegurar que *master/main* está siempre en estado **desplegable** cuando los cambios son pequeños, siempre y cuando contemos con una base de test automáticos que nos de la confianza suficiente para avanzar.

En general, es un flujo de trabajo recomendado para equipos o proyectos en los que se pueda avanzar por medio de iteraciones rápidas, pues esto permitirá que se puedan hacer despliegues más frecuentemente. Si el volumen de cambios es grande, necesita de muchas pruebas que no sean triviales de hacer o que no estén 100% automatizadas (con esta condición excluimos un porcentaje bastante alto de proyectos), quizás no sea el más apropiado.

En la web de Trunk Based Development se profundiza en este flujo de trabajo y se explora cómo integrarlo en nuestros ciclos de integración continua.

Feature branching

En este modelo de trabajo se busca tratar de aislar un poco más el trabajo de los diversos colaboradores empleando las ramas (*branches*) para cada nueva funcionalidad que se quiera añadir e intercalando una nueva rama de integración sobre la que preparar cada *release* de forma independiente a la rama principal. Esta rama recibe usualmente el nombre de **develop** o **dev**.



Ejemplo de feature branching. Fuente: Atlassian.

Importante

Este modelo es perfectamente válido sin esta rama de desarrollo intermedio. Lo que busca develop es aislar la rama maestra de errores que se producen durante la evolución de una *release*, en la que se integran muchas piezas que no siempre tienen por qué dejar el entorno en un estado desplegable. Por ejemplo, podemos querer integrar una tarea que modifica el esquema de la base de datos, pero sin actualizar la base de datos de producción todavía (proceso crítico).

Al flujo definido anteriormente le añadimos entonces estas dos nuevas ramas, quedando un proceso como el que sigue:

1. Los desarrolladores comienzan una tarea: más adelante se explicará cómo deben ser estas tareas, pero podemos asumir que:
    a. Siempre será algo que aporta valor a la aplicación.
    b. Puede ser desplegado independientemente del trabajo de otros compañeros.
2. Se crea, a partir de develop, una rama de tipo **feature**, con un nombre que incluya el ID del ticket/tarea del sistema gestor de tareas empleado (JIRA, Trac, Bugzilla…) **si existe**, y algunas palabras que resuman la tarea. Ejemplos pueden ser:
    a. feature/JIRA-1234-apply-new-login-page.
    b. feature/ID-abcd-cambio-modelo-datos.

    La razón de este patrón es doble: facilita la integración de control de versiones - gestor de tareas, presente en la gran mayoría de las aplicaciones y facilita la gestión de ramas durante la *release* (ya que se suelen borrar las ramas de tipo feature, ya sea después de integrarlas o después de hacer la *release*).

3. Se trabaja en la nueva rama y cuando creamos que cumplimos todos los criterios de aceptación definidos para la misma (acordados con el equipo o con el dueño del producto) hacemos una solicitud de integración de la misma, proceso usualmente conocido como ***pull request.***
4. Cuando se hayan integrado a develop todas las funcionalidades que el equipo decide que deben formar parte de la siguiente *release*, crearemos una nueva rama 'release/X.Y', esta vez partiendo de develop, que eventualmente será mergeada a máster (y a develop si ha habido cambios).

Pull requests and code reviews

Afortunadamente, no estamos solos en el mundo. No somos, o no deberíamos ser, los únicos responsables del código fuente. Una de las grandes ventajas del control de versiones distribuido es la facilidad que aporta al trabajo en equipo y, concretamente, a la revisión por parte de compañeros del código que aportamos para cumplir una funcionalidad concreta.

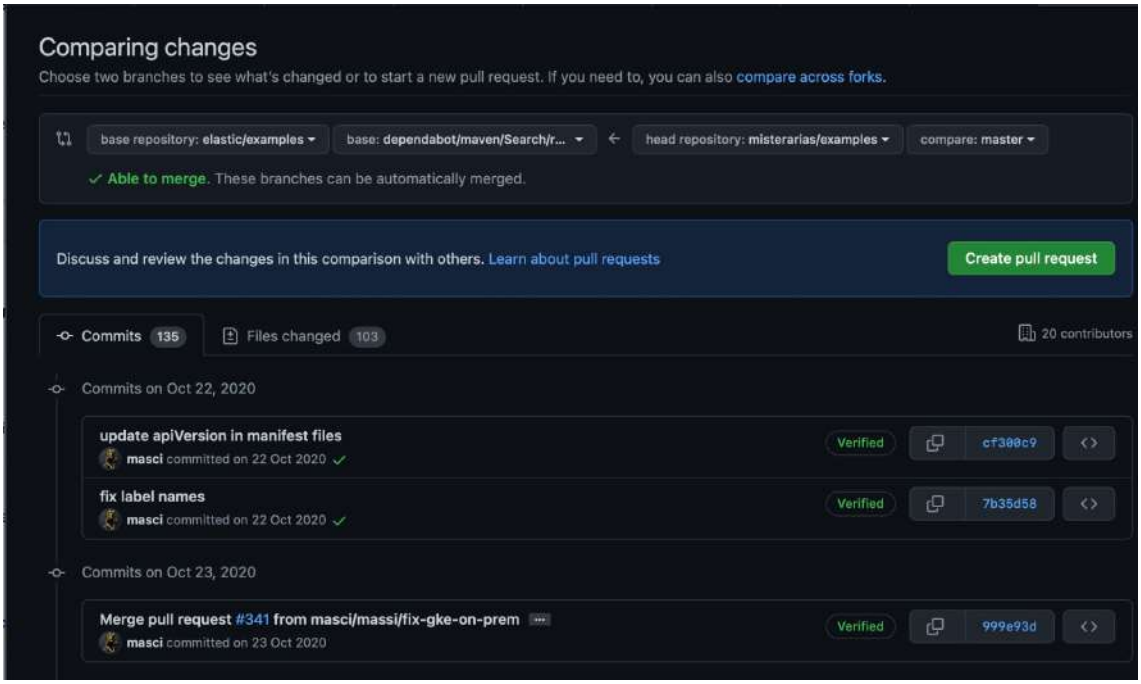
El mecanismo de *pull request* (o *merge request*) es aquel por el que solicitamos aportar código a una rama de integración compartida con el resto de componentes del equipo.

Una vez esté hecha esta solicitud, el resto del equipo puede revisar el trabajo, inspeccionar visualmente el código y aportar pistas o consejos que lo puedan hacer mejor. Podrían incluso **rechazar** la solicitud si no se cumplen una serie de requisitos de integración acordados previamente:

- No tener una cobertura mínima de test.
- No seguir los estándares de escritura del código.
- No estar apropiadamente documentado.
- No resolver el problema planteado o no hacerlo en todos los casos…

Este proceso por el cual una solicitud pasa por la revisión de nuestros pares, y tras un periplo llega a formar parte de una rama de integración, se llama ***code review*** y es otra de las razones por las que herramientas web como las mencionadas son tan potentes y tan necesarias: democratizan el código fuente y ayudan a que todo el equipo sienta que ha aportado algo en cada nueva funcionalidad que se entrega.

Los principales servicios de VCS distribuido, como Bitbucket, Gitlab y Github, ofrecen este servicio de forma totalmente gratuita. Cualquiera puede hacer una *pull request* a un repositorio público o privado y comenzar ese proceso de colaboración que permite tanto mejorar el código de las aplicaciones como la manera en que trabajamos y creamos código fuente nosotros mismos.



Ejemplo del previo a la apertura de una *pull request* en Github.

En el caso de que queramos contribuir a un repositorio público, debemos hacernos antes un ***fork*** del mismo, que creará una copia en nuestro espacio personal sobre la que tendremos permisos absolutos. Esto es una forma de controlar quién tiene acceso a modificar el código de un proyecto de forma directa, ya que sin permisos adecuados **no se puede aprobar *pull requests*** ni **hacer *pushes* directos a ramas de integración**.

De este modo, se controla quién puede hacer modificaciones reales al código, algo **muy** importante en repositorios privados o en proyectos open source con muchos usuarios.

Conclusión

Este flujo permite a parte del equipo hacer pruebas aisladas de la nueva release en esa rama 'release/X.Y' mientras el resto continúan trabajando en tareas de la siguiente.

Posiblemente, estas pruebas se realizan en entornos de pruebas específicos y que se parecen más a los productivos que cualquier otro que podamos tener para el desarrollo local, o entornos efímeros basados en máquinas virtuales o contenedores donde se puedan hacer pruebas de carga o integración, que como sabemos son mucho más costosas (en tiempo y recursos) que las pruebas unitarias.

Este modelo de trabajo, cuando incluye la rama develop como acabamos de describir, es también conocido como GitFlow, y se hizo muy famoso hace muchos años por el enfoque estructurado que le da al desarrollo de aplicaciones usando control de versiones. El modelo GitFLow añade además dos ramas de desarrollo nuevas:

1. Las ramas de arreglo de errores '**bugfix/'** que sirven para arreglar los errores detectados en una rama *release* una vez creada desde develop, pero antes de haberlos llevado a máster. Estas ramas siempre parten de una *release* y acaban en ella.

2. Las ramas de arreglo de errores en caliente '**hotfix/'** que corrigen errores detectados solo en la rama máster, usualmente corregido de forma muy urgente pues vienen de una situación no contemplada en ningún tipo de pruebas. Estas ramas siempre parten de máster y van a máster, pero, además, debemos asegurarnos que también se llevan a develop y a cualquier *release* que se encuentre en marcha.

La principal razón por la que parece que este modelo de trabajo ya no tiene tanta aceptación a día de hoy es que toda la gestión de ramas que se hace **parece que** no sigue la filosofía DevOps de avanzar rápido, con ciclos cortos que añadan valor de modo incremental y despliegues constantes a producción.

Sin embargo, muchos somos de la opinión de que la velocidad no solo se debe medir en número de despliegues por día y, además, no todos los proyectos/equipos son iguales, por lo que no se pueden medir de la misma manera.

Es nuestra labor como responsables DevOps la de conocer los diferentes modelos, entender las necesidades del equipo de desarrollo y poder **razonar** qué tipo de modelo es más ventajoso (pudiendo ser, **perfectamente**, un flujo diferente de los mencionados).

Semantic versioning

Un problema recurrente, común a ambos métodos, es el del apropiado nombre de las versiones de las *releases*. Si bien puede parecer sencillo, un enfoque estructurado del problema nos garantizará que no existan dudas por parte de ningún miembro del equipo a la hora de generar versiones, versiones cuyo nombre nos proporcionará pistas sobre lo que representan.

Según la web semver.org, el versionado semántico (SemVer) es: un conjunto simple de reglas y requerimientos que dictan cómo asignar e incrementar los números de la versión. Estas reglas están basadas en prácticas preexistentes de uso generalizado tanto en software de código cerrado como de código abierto, pero no necesariamente limitadas a estas.

A muy grandes rasgos, cada versión de nuestro software estará en el formato 'x.y.z':

- ***x (major)*** - Suele marcar una familia o línea de versiones concreta, su incremento **no garantiza** la retrocompatibilidad con la anterior.
- ***y (minor)*** - Dentro de una línea de versiones, marcan incrementos de funcionalidad que son **retrocompatibles** entre sí.
- ***z (patch)*** - Dentro de una versión concreta, marcan **incrementos sin funcionalidades relevantes**, usualmente correcciones de errores.

Aunque la norma definida por SemVer 2.0 es bastante más compleja, con estos tres axiomas podemos conseguir un sistema de nombrado unívoco y definido, que además ofrece a los posibles consumidores de nuestro software (por ejemplo, otras aplicaciones que nos importen) para saber de qué versión deben depender si quieren tener tal o cual funcionalidad o qué parche deben usar para garantizar que un determinado *bug* ya no se encuentra presente.

Log de cambios

Es habitual acompañar los proyectos con un versionado semántico de un log de cambios o changelog, en el que se detalla qué cambia en cada versión que se despliega; aunque la gestión manual de este fichero es tediosa (y fuente habitual de conflictos de integración de versiones), es un elemento fundamental de cara a posibles consumidores de nuestro software, pues podrán ver qué se ha hecho en cada cambio.

Un ejemplo de formato, sugerido por la conocida web Mantenga un changelog, sería:

```
# Changelog
All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](https://keepachangelog.com/en/1.0.0/),
and this project adheres to [Semantic Versioning](https://semver.org/spec/v2.0.0.html).

## [Unreleased]

## [1.0.0] - 2017-06-20
### Added
- New visual identity by [@tylerfortune8](https://github.com/tylerfortune8).
- Version navigation.
- Links to latest released version in previous versions.
- "Why keep a changelog?" section.
- "Who needs a changelog?" section.
- "How do I make a changelog?" section.
 - "Frequently Asked Questions" section.
```

LA UNIVERSIDAD EN INTERNET | FORMACIÓN PROFESIONAL

PROEDUCA