Capítulo 1-3
Las herramientas técnicas

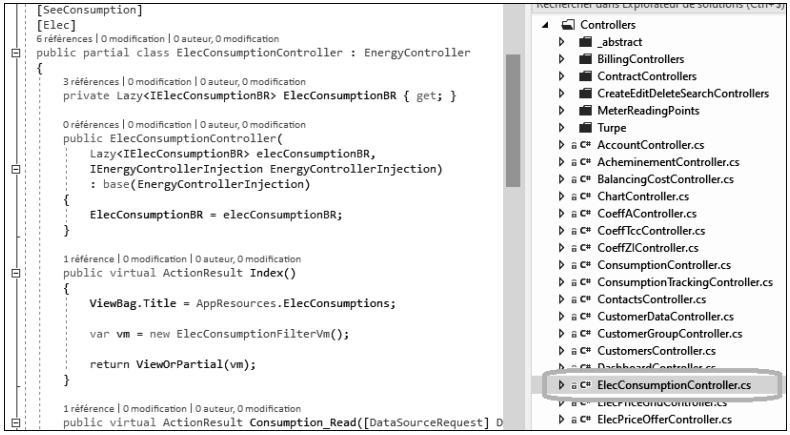
1. Code versioning (Control de versiones de código)

En el desarrollo informático, el código de un gran proyecto se divide en miles de archivos. Es muy probable que varios desarrolladores trabajen en el mismo archivo a la vez.

Cada desarrollador escribe su código localmente en su máquina y cuando termina, lo registra en un tronco común ubicado en un servidor remoto.

Sin embargo, si dos desarrolladores han trabajado con el mismo código en el mismo archivo, existe el riesgo de que surjan conflictos a la hora de guardarlo en el tronco común. ¿Qué código debe conservarse en ese caso? No existe un sistema de bloqueo sobre los archivos, que haga que cuando yo lo uso, los demás no puedan modificarlo. Un buen software de gestión de versiones de código es perfectamente capaz de hacer frente a este tipo de conflictos. Es la base para trabajar en equipo sin perder tiempo esperando a que los archivos estén disponibles. La referencia en este ámbito es, con diferencia, la aplicación **GitHub**. Este tipo de aplicación permite realizar todas las acciones que un desarrollador puede llevar a cabo en su día a día: recuperar todo el código del proyecto del tronco común y copiarlo localmente.

Esto se denomina crear el espacio de trabajo del desarrollador, donde podrá añadir, eliminar o modificar los archivos del proyecto. Una vez finalizado su trabajo, puede cargar el resultado en el tronco común. Esta acción requiere algunas comprobaciones antes de validar los cambios. Este tipo de aplicación también permite disponer de todo tipo de información sobre los ficheros: por ejemplo, quién introdujo qué código, cuándo, por qué, quién borró, quién modificó, quién hizo el café... bueno eso no, olvidémoslo, ¡tampoco nos pasemos!



Esto es una imagen de mi pantalla, donde estoy escribiendo código en el archivo ElecConsumptionController.cs.

Un proyecto puede contener miles de archivos de este tipo, con lo que el número total de líneas de código del proyecto puede superar con creces el millón. En la jerga informática, el tronco común se denomina **branch Main (por rama principal)** o repositorio remoto. Esta rama contiene todo el código del proyecto en un momento dado y en estado puro, es decir, sin aberraciones de código. El código de la rama principal cumple todas las reglas de codificación definidas por el equipo y, por supuesto, debe ser absolutamente compilable. Compilar quiere decir traducir un código como el anterior (legible por un humano) en otro código que, esta vez, solo contendrá 0 y 1. Es el conocido código binario, el único tipo de código que pueden entender los microprocesadores de los ordenadores. No pasamos inmediatamente al código binario cuando desarrollamos en lenguajes como C# o Java, pero no entraremos aquí en los detalles. **Un desarrollador que fusiona código no compilable en la rama principal está cometiendo un sacrilegio.**

Antes de proceder a fusionar su código en la rama Main, los desarrolladores deben probarlo primero localmente y, a continuación, comprobar que compila y que supera todas las pruebas unitarias. Hay un paso muy importante: antes de empezar cualquier código nuevo, el desarrollador debe sincronizarse con la rama Main para recuperar las últimas modificaciones validadas por sus colegas en esta rama. Esto les facilitará la resolución de cualquier conflicto antes de validar su propio código después. Por último, debe asegurarse de que la integración continua no se bloquea. Hablaremos más sobre esto más adelante, pero debe saber que en ningún caso debe forzar el código en la rama Main si la integración continua está en rojo, es decir, si se ha detectado un problema. El objetivo de la integración continua es comprobar que el nuevo código integrado en la rama Main no rompe la armonía que reina en ella. En otras palabras, se trata de garantizar que el nuevo código introducido no provoque cualquier tipo de errores.

En la jerga, no hablamos de fusionar, sino de que vamos a hacer un **merge** de nuestro código. No tiene nada de esotérico: *to merge* significa fusionar. Cada desarrollador escribe su código por su cuenta y, una vez terminado, lo fusiona con la rama principal. También se podría decir que integra su código en la rama principal. La rama principal es, por tanto, una rama de integración para todo el código producido; en otras palabras, nuestra posible aplicación en un momento dado. La acción de hacer un merge en la rama Main nunca es directa, hay que tomar precauciones porque, como acabamos de decir, violar la integridad de la rama Main puede ser fatal. En la medida de lo posible, se debe evitar que cualquier nuevo código cause errores en esta rama. Cada desarrollador que quiera fusionar su código en la rama Main debe hacer una **Pull Request (PR)**, también conocida como **Merge Request (MR)**. Se trata de una solicitud para fusionar su código local en la rama Main. Sin entrar en demasiados detalles, se trata de un humano que puede o no validar el código. En los equipos ágiles, una persona de control de calidad (Quality of Service, QA) puede asumir este papel, pero también puede ser un arquitecto o incluso uno o varios desarrolladores dedicados; o incluso, por qué no, ¡el propio equipo de desarrollo! En este último caso, el código desarrollado por fulano será comprobado por otra persona para asegurarse de que quie codificó no valida su propia PR. Si se rechaza la Pull Request, el desarrollador tendrá que hacer los ajustes necesarios y rehacer la PR. Una vez aceptada la PR, el código se integra en la rama Main y la integración continua se activa. Veremos más sobre esto dentro de poco.

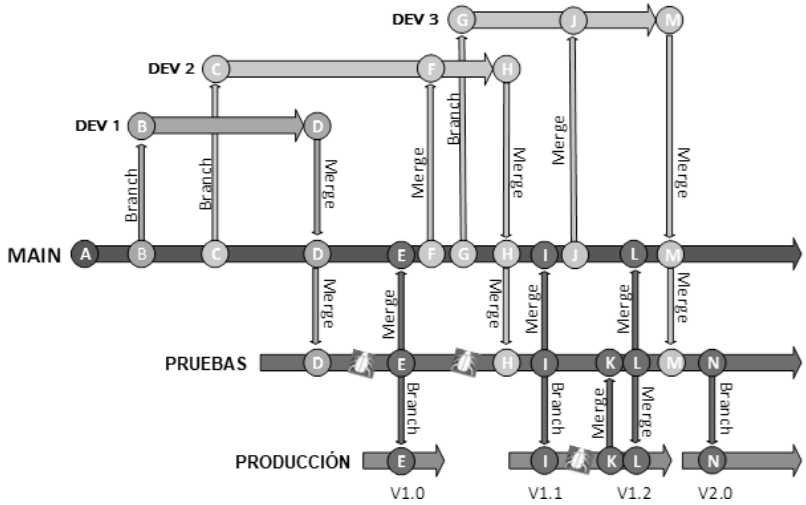
Cuando un nuevo desarrollador se une a un equipo, antes de que pueda codificar una sola línea, tiene que conectarse al servidor que contiene la rama Main para extraer todo el código del proyecto en su ordenador. Para ello, crea en este último una nueva rama, basada en la rama Main. Esta nueva rama, creada localmente, es una copia de la rama Main. Nuestro desarrollador codifica entonces su parte del código localmente solo en su rama, y luego hace su PR una vez que su código ha sido finalizado.

En este ámbito, la regla de oro es no codificar líneas y líneas antes de hacer una Pull Request. Se codifica una tarea y se hace su PR; se avanza tarea a tarea. Esto facilita la gestión de conflictos si varios desarrolladores están trabajando en los mismos archivos, así como facilita el procesamiento de las Pull Requests. Como hemos mencionado antes, otro buen reflejo es recuperar siempre la última versión de la rama Main antes de comenzar un nuevo código, ya que otros desarrolladores seguramente habrán realizado fusiones mientras tanto; este reflejo le evitará encontrarse con demasiada diferencia entre su versión local y la versión de la rama Main. Lo ideal es recuperar el código de la rama Main tan pronto como esta haya sido actualizada en una PR validada.

Por último, el merge (la fusión entre el trabajo local del desarrollador en local y el código completo central) puede realizarse en ambas direcciones: se puede hacer un merge de nuestro trabajo local hacia la rama Main, o un merge de la rama Main hacia nuestra rama local, lo que equivale, en este último caso, a recuperar la última versión de la rama Main. Por lo tanto, el merge no es una copia que sobrescribe lo existente, sino una copia que fusiona el contenido de dos ramas.

Nos detendremos aquí en lo que respecta a las ramas, pero ha comprendido la esencia del trabajo de un desarrollador: crear ramas en las que codificar y hacer PR para validar su trabajo; así es como, a medida que aumentan las fusiones, el tamaño del código del proyecto aumenta en la rama Main.

A continuación se muestra un ejemplo de estrategia de sucursales adoptada por un equipo:



El equipo ha creado su rama Main en A. Un desarrollador crea una rama local en B para recuperar el código de la rama Main y copiarlo en su nueva rama local; se dice que ha tirado una rama. Luego puede iniciar su código localmente. Cuando ha terminado su trabajo, realiza un merge en D; su código local ahora está fusionado en la rama Main. También realiza un merge en la rama Pruebas. Es en esta rama Pruebas donde se corregirán los errores. Está prohibido desarrollar directamente en la rama Main y mucho menos en la rama Producción (o Prod). Mientras nuestro desarrollador trabajaba en su código, otro desarrollador a su vez tiró una rama local en la etapa C.

Cuando se detecta un error en la rama Pruebas, lo corregimos desarrollando directamente en esta rama. Una vez corregido el fallo y validada la prueba, se dibuja una rama Prod y se fusiona con la rama Main para transferir las correcciones, porque si el fallo existe en Pruebas, también existe en Main. En nuestro ejemplo, el primer error detectado después del merge en D se corrige en E y se reportan las correcciones en la rama Main; y como la primera puesta en producción era esperada, se crea una nueva rama Prod. Cuando la rama Main se actualiza, siempre es bueno para un desarrollador recuperar rápidamente las modificaciones haciendo un merge. Esto es lo que hacen, por ejemplo, el segundo y el tercer desarrollador respectivamente en F y J.

Pero nuestro tercer desarrollador no recuperó la corrección de errores de Prod realizada en L. No importa, cuando fusione su código en la rama Main (en M), verá si tiene conflictos con las correcciones hechas previamente en la rama Main. Cualquier corrección hecha en la rama Pruebas debe reflejarse en la rama Main, y tarde o temprano en la rama Prod. Todo nuevo código en la rama Main debe reflejarse en la rama Pruebas para ser probado antes de llevar el código a la rama Prod. Finalmente, a partir de cualquier rama no local, somos capaces, en un instante t, de crear una versión del proyecto siempre que, por supuesto, las ramas Main y Pruebas estén perfectamente sincronizadas.

Hoy en día, en los equipos ágiles, es al final del sprint cuando el código se transfiere a Pruebas para la Sprint Review; sin embargo, después de n sprints, las cosas se complican: no solo hay que continuar con las Sprint Reviews, sino que, además, durante el sprint, también hay que corregir los posibles errores detectados en Pruebas. Si se ha hecho una entrega en producción desde entonces, también se pueden encontrar errores que deberán corregirse rápidamente. En nuestro ejemplo de rama, para corregir el error encontrado en Prod, como no se puede desarrollar directamente en esta rama, necesitamos recuperar el código en Prod para fusionarlo en Pruebas, donde corregiremos el error en L. Una vez corregido y validado, se refleja el arreglo en Prod y luego en Main.

Cada error reportado se convierte, en el sprint, en una US técnica a tener en cuenta. Los errores aumentan, por tanto, la velocidad del equipo. Un buen KPI sería conocer la proporción de errores en una velocidad.

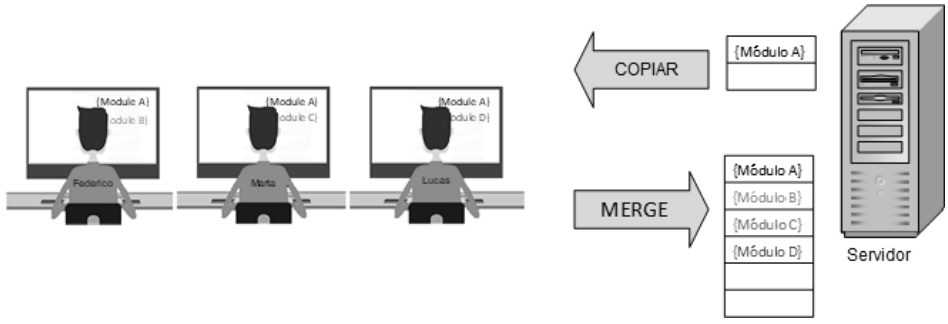
Por supuesto, existen casos en los que un error detectado en producción es tan grave que debe corregirse absolutamente en cuestión de minutos. Por tanto, no se espera a que termine el sprint; se debe lanzar una operación Commando para corregir rápidamente el fallo en Pruebas y validar para un merge en Prod. Cuidado, sería una imprudencia corregir el error directamente en la rama Prod. Recuerdo, por ejemplo, el caso de un cambio urgente que había que realizar en un parámetro de un archivo web.config de un sitio de comercio electrónico. A primera vista, no tenía nada de complicado, pero la modificación directa en producción hizo caer el sitio web. De repente, ¡diez millones de clientes no pudieron acceder! Afortunadamente, el error se corrigió en menos de cinco minutos. El responsable se había equivocado de archivo, pensando que estaba trabajando en el archivo en Prod, cuando en realidad era el archivo web.config de la rama Pruebas que había subido a producción.

Esta experiencia fue una gran lección de gestión ágil: ningún miembro del equipo perdió la calma, ni un solo directivo increpó al culpable; todos estaban concentrados en resolver el problema. En pocos minutos, todo el equipo había restablecido la situación y se felicitaba por su perfecta colaboración.

Como decíamos, cuando un desarrollador ha terminado su código, lo valida. Este código se comprueba y, si todo está bien, se fusiona en la rama Main, luego en Pruebas, y finalmente en Prod. La integración continua permite automatizar todas las etapas que siguen a la validación de una Pull Request. Todavía hay muchas cosas que comprobar antes de instalar la aplicación para los usuarios.

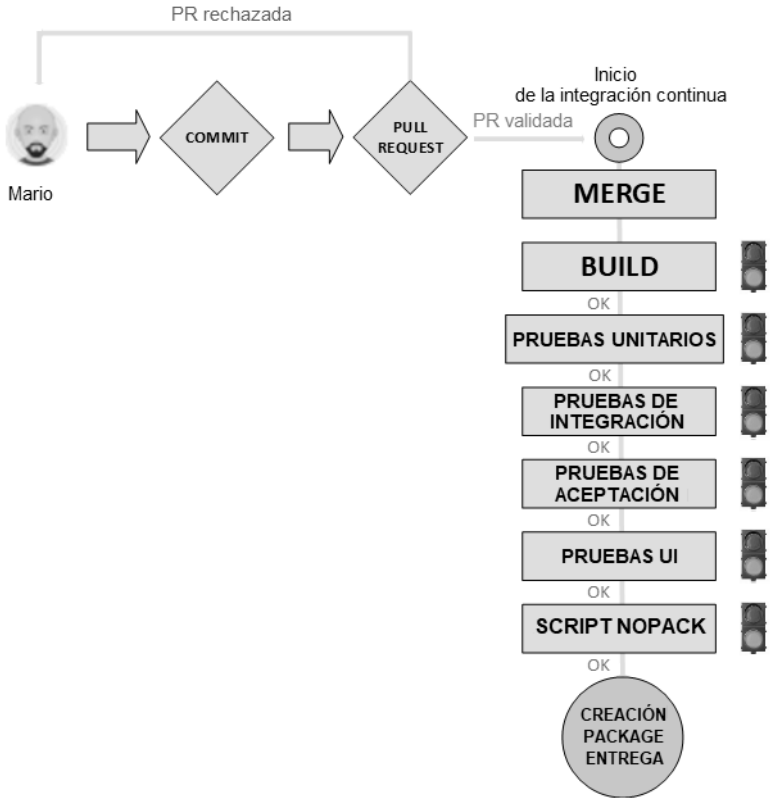
2. Integración continua

El objetivo de la integración continua (CI) es procesar cada merge realizado en la rama Main; se integra su código con el de sus colegas.



Como recordatorio, cuando un desarrollador ha terminado su código, hace una Pull Request para enviar su trabajo. Una vez que esta PR ha sido validada, comienza la integración continua. Hoy en día, la integración continua hace mucho más que una simple integración del código (merge) en la rama Main. Después del merge, se encuentran un conjunto de filtros colocados uno detrás de otro que se ejecutarán automáticamente, uno tras otro, tan pronto como el código se fusione en la rama Main. Todos estos filtros dispuestos en una cola se denominan **pipeline** o **workflow (flujo de trabajo) de la aplicación**.

Estos filtros son, pues, elementos ejecutables (una aplicación, un script, un job, un fragmento de código, un servicio, un servicio web, un microservicio, etc.) que se ejecutarán secuencialmente en el orden en que aparezcan en el pipeline.



El objetivo de la CI es garantizar que el merge del nuevo código en la rama Main no ha provocado ningún error, para que no construyamos un package (paquete) de entrega lleno de bugs (o errores). Hoy en día, como ya hemos mencionado, la CI se alimenta de otros tipos de control; por abuso del lenguaje, llamamos integración continua a toda la cadena de control.

Tras el merge, la CI compila el código (hace un **Build**), y luego realiza un montón de pruebas. El poder del pipeline reside en el hecho de que puede añadir tantos filtros como quiera. En nuestro diagrama anterior, hemos añadido un script « NoPack » que podría, por ejemplo, enviar un correo electrónico en caso de error en el pipeline, o escribir scripts de pruebas de rendimiento e integrarlos en el pipeline.

El más mínimo error en los filtros detiene inmediatamente el pipeline; por tanto, será imposible que el pipeline construya el package de entrega. Este package es el conjunto de archivos que componen nuestro proyecto. Basta con instalarlos en un servidor para que nuestro proyecto se transforme en una aplicación disponible para los usuarios. Este package se compone del código binario de nuestro proyecto y de varios archivos indispensables para que el código funcione correctamente: archivos de dependencias, de configuración, de base de datos, etc.

En cuanto un filtro del pipeline detecta un error, el pipeline deja de funcionar; a partir de ese momento, cada miembro del equipo detiene su tarea para intentar resolver el problema. Durante las pruebas funcionales (pruebas de aceptación), imaginemos que el pipeline detecta errores pero que el equipo sigue trabajando y que cada miembro sigue fusionando en una rama Main infectada... Los errores se acumularían y pasaríamos todavía más tiempo en resolverlos. La integración continua es una salvaguarda, o más bien un conjunto de salvaguardas. Cada filtro del pipeline es una torre de control que no deja pasar ningún error procesado por el filtro. Tomemos, por ejemplo, un caso en el que se desea probar la cobertura del código y evitar que el pipeline produzca el package de instalación en caso de que las pruebas unitarias no hayan sido cubiertas al 80 %. En ese caso, debe añadir una aplicación como **SonarQube** para configurar un umbral de cobertura de código del 80 %. Como resultado, si las pruebas unitarias han sido descuidadas por el equipo, con una torre de vigilancia como esta, no hay posibilidad de que el pipeline construya el package de entrega porque no habremos alcanzado nuestro 80 % de cobertura de código.