Análisis y Diseño de Sistemas 2

Integración continua

Material de apoyo del curso Análisis y Diseño de Sistemas 2 de la USAC

Ing. Ricardo Morales

Primer semestre 2016

Análisis y Diseño de Sistemas 2

Integración continua

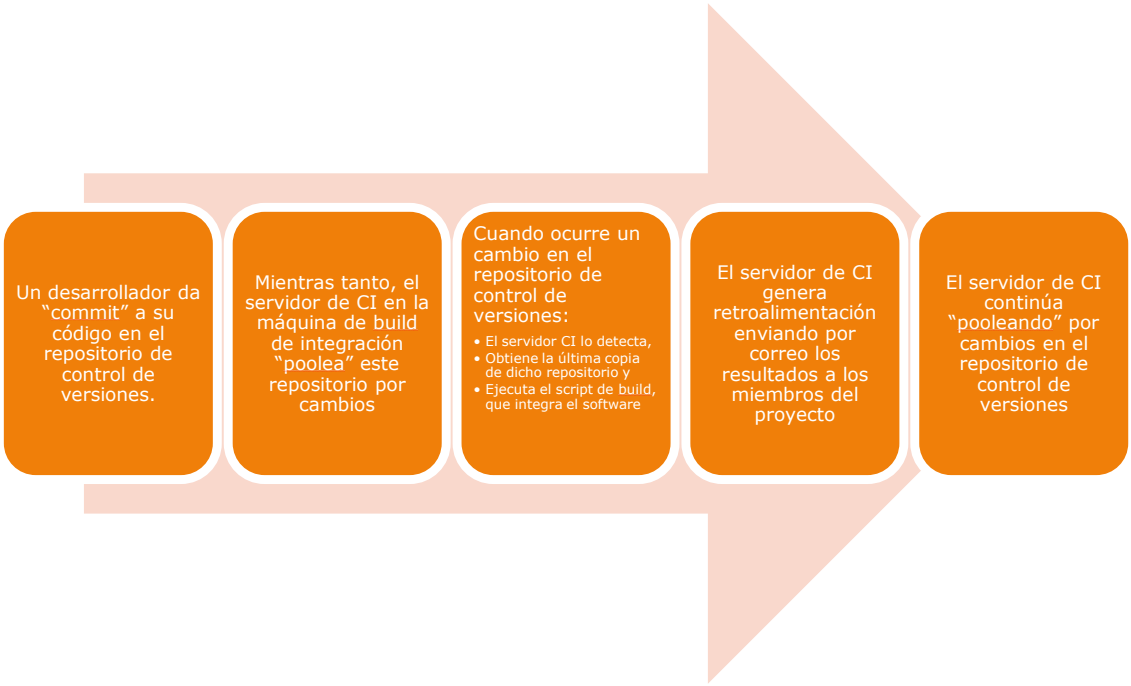
Tabla de contenido

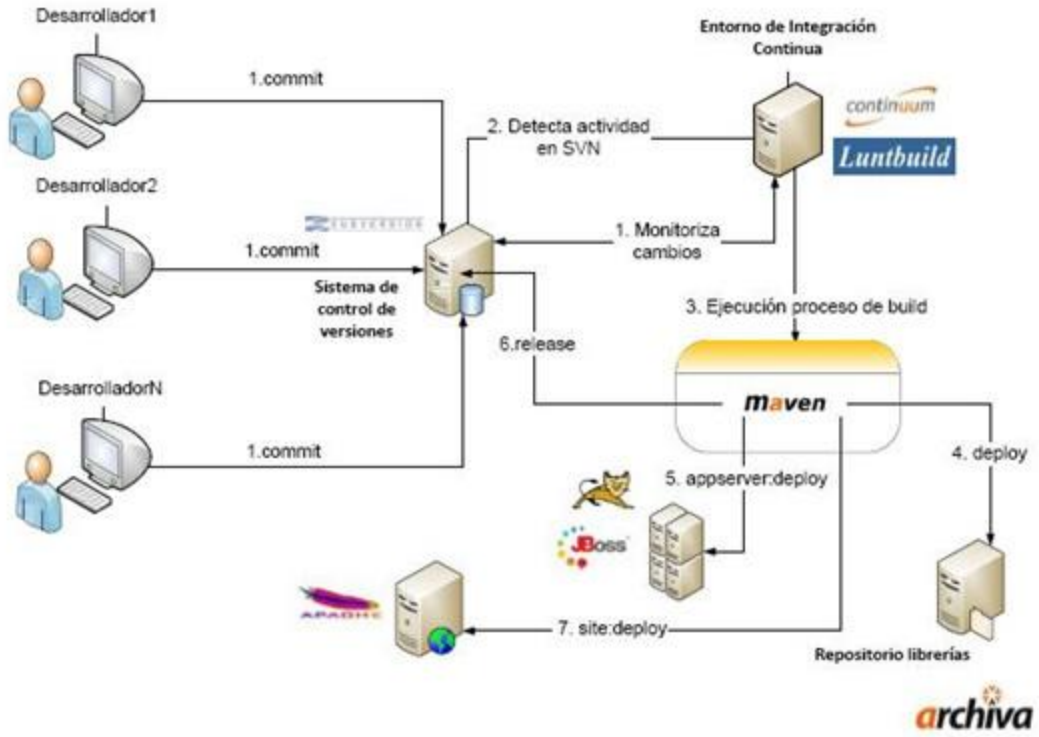
INTEGRACIÓN CONTINUA

Objetivos

- Describir el concepto de integración continua
- Conocer y entender los elementos que integran un sistema de integración continua y su relación con un sistema de control de versiones
- Identificar el valor de las prácticas recomendadas para implementar la integración continua

Pasos





https://jbravomontero.wordpress.com/tag/integracion-continua/

Herramientas y actores

Desarrollador

Luego que el desarrollador realiza modificaciones en el código, corre un build privado (que integra cambios del resto del equipo) y luego "da commit" en el repositorio de control de versiones

Repositorio de control de versiones

El propósito del repositorio es administrar cambios al código fuente y otros activos de software usando un acceso controlado.

Se corre CI contra la línea principal del repositorio (head/trunk) o contra las ramas que se defina.

Servidor CI

Corre un build de integración siempre que a un cambio se le "da commit" en el repositorio de control de versiones. Obtiene los archivos fuente y corre el script(s) de build

Script de build

Es un solo script o un conjunto de ellos, que se usan para compilar, probar, inspeccionar y deployar software.

Mecanismo de retroalimentación

Uno de los propósitos clave de CI es producir retroalimentación del build de integración, porque se necesita saber lo antes posible si ocurrió algún problema con

el último build. Al recibir rápidamente esta información, se puede corregir el problema

Máquina de build de integración

Es una máquina separada cuya única responsabilidad es integrar software. En esta máquina funciona el servidor CI

Características de integración continua

Los siguientes son elementos que pueden agregarse como parte del script de build, para obtener retroalimentación:

- Compilación de código fuente

  Es una de las características básicas y comunes de un sistema CI. Implica crear código ejecutable a partir del código fuente.

- Integración de base de datos

  Algunas personas consideran la integración de código fuente y la de base de datos, procesos separados. Esto es infortunado porque la base de datos es una parte integral de la aplicación de software.

  Se debe tratar el código fuente de la base de dato – scripts DDL, scripts DML, procedimientos almacenados, etc. – de la misma manera que cualquier otro código fuente en el sistema.

- Pruebas

  Muchos consideran que si no existen pruebas automatizadas, no puede ser CI.

  Sin pruebas automatizadas, es difícil para los desarrolladores u otros participantes del proyecto tener confianza en los cambios al software.

  Además de las pruebas unitarias, es posible correr diferentes categorías de pruebas desde el sistema CI.

- Inspección

  Las inspecciones automatizadas de código pueden usarse para mejorar la calidad del software a través de forzar reglas. Por ejemplo, que ninguna clase tenga mas de 300 líneas de código no documentado.

- Deployment

  El deployment continuo permite producir software que funciona en cualquier momento.

  Esto significa que el propósito clave de un sistema CI es generar los artefactos de software en un entregable con los últimos cambios al código y hacerlo disponible para el ambiente de pruebas.

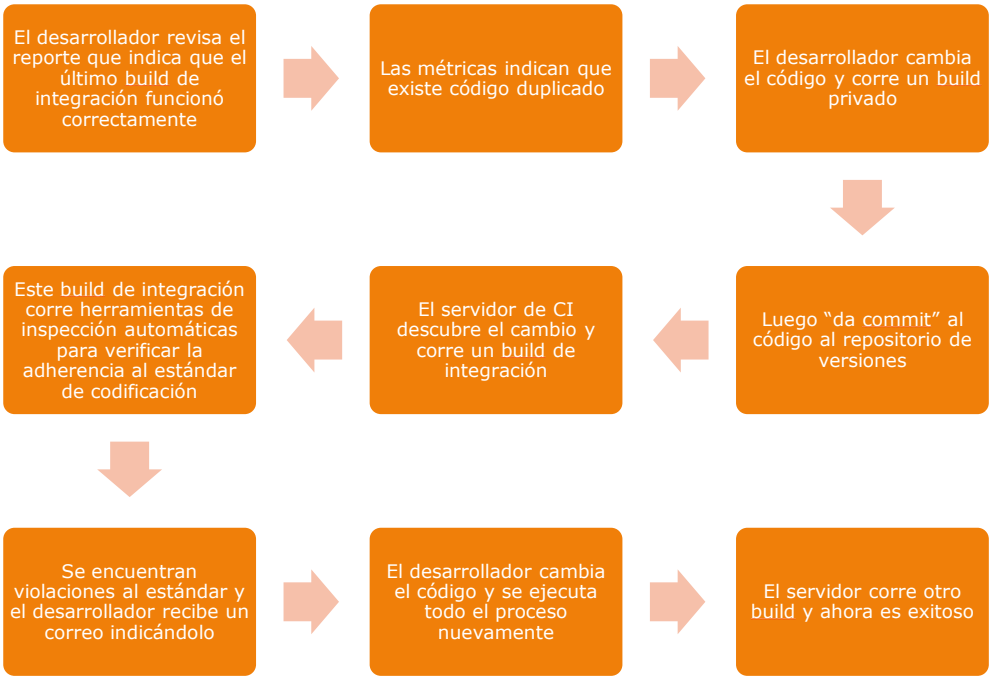
  Entre otras cosas, se deben obtener los archivos fuente del repositorio de control de versiones, ejecutar el build, ejecutar exitosamente las pruebas e

inspecciones, etiquetar el release, generar los archivos de deploy y deployar o instalar los archivos en el ambiente apropiado.

- Documentación y retroalimentación

  Un sistema CI también puede generar documentación actualizada durante el proceso.

Un día en integración continua



Términos

Integración

El acto de combinar artefactos separados de código fuente para determinar cómo funcionan como un todo

Integración continua

Práctica de desarrollo de software donde los miembros de un equipo integran su trabajo frecuentemente, usualmente cada persona integra como mínimo de forma diaria, generando múltiples integraciones por día

Cada integración es verificada por un build automatizado que detecta errores de integración tan pronto como es posible

Muchos equipos encuentran que este enfoque lleva a una reducción significativa de problemas de integración y le permite al equipo desarrollar software cohesivo mas rápido

Build de integración

Es el acto de combinar componentes de software (programas y archivos) en un sistema de software. El build incluye múltiples componentes o solo archivos fuente compilados

Build privado

Consiste en correr un build localmente antes de dar commit en el repositorio de control de versiones

Build de release

Obtiene software listo para liberar a los usuarios

¿Qué se necesita para empezar?

Control de versiones

Todo en el proyecto debe estar en un repositorio de control de versiones: código, pruebas, scripts de base de datos, scripts de build y deployment, y todo lo necesario para crear, instalar, correr y probar la aplicación

Build automatizado

Se debe poder empezar el build desde la línea de comandos. Aunque los IDEs se pueden integrar con una herramienta de integración continua, es mejor tener scripts que se corran desde la línea de comandos porque:

- Se debe poder correr el proceso de build de una forma automatizada desde el ambiente de integración continua para que pueda ser auditado cuando las cosas salen mal
- Los scripts de build deben ser tratados como el código base. Deben ser probados y mejorados constantemente
- Hace que entender, mantener y debugear el build sea mas fácil y permite una mejor colaboración con operaciones

Acuerdo del equipo

La integración continua es una práctica, no una herramienta. Requiere el grado de compromiso y disciplina del equipo de desarrollo

Si la gente no adopta la disciplina necesaria para trabajar, los intentos de integración continua no llevarán a una mejora en la calidad que se espera

Prácticas para iniciar

Hacer check in regularmente

La práctica mas importante para que la integración continua funcione correctamente es hacer check ins frecuentes al trunk, al menos un par de veces al día.

Hacer cambios pequeños, elegir tareas pequeñas, escribir pruebas y código fuente; y hacer commit.

Hacer commit luego de cada tarea, cada desarrollador debe hacer commit de su código luego de finalizar cada tarea.

Crear una suite automatizada de pruebas completas

Las 3 clases de pruebas que deberían correr en la integración continua son: pruebas unitarias, pruebas de componentes y pruebas de aceptación.

Las pruebas unitarias son escritas para probar el comportamiento de piezas pequeñas de la aplicación de forma aislada (por ejemplo un método o función la interacción entre un grupo pequeño de ellas).

Las pruebas de componente prueban el comportamiento de varios componentes de la aplicación. Al igual que las pruebas unitarias, no requieren siempre iniciar toda la aplicación, aunque pueden afectar la base de datos o filesystem.

Las pruebas de aceptación verifican que la aplicación alcance los criterios de aceptación definidos por el negocio, incluyendo la funcionalidad provista por las aplicaciones así como características como capacidad, disponibilidad, seguridad, etc.

Mantener proceso de build y pruebas corto

Si hacer el build y correr las pruebas unitarias toma mucho tiempo, se tendrían los siguientes problemas:

- La gente dejará de hacer builds completos y correr pruebas antes de hacer check in
- El proceso de integración continua tomará tanto tiempo que commits múltiples correrán el build otra vez y no se sabrá cual falló
- La gente hará check in con menos frecuencia porque deben esperar mucho para que el software pase el build y las pruebas

Idealmente el proceso de compilar y probar, que se corre antes de hacer check in y en el servidor de integración continua, debe tomar como máximo 10 minutos, con un ideal de 90 segundos.

Administrar el workspace de desarrollo

Es importante que el ambiente de trabajo de los desarrolladores sea bien manejado, por sanidad y productividad, para ello se necesita lo siguiente:

- Alcanzar una administración de la configuración cuidadosa, que incluya código fuente, datos de prueba, scripts de base de datos, build y deployment
- Administración de la configuración de dependencias con terceros, librerías y componentes
- Asegurarse que las pruebas automatizadas puedan correrse en las máquinas de los desarrolladores

Prácticas esenciales de implementación

No hacer commit en build roto

El pecado capital de la integración continua es hacer commit de un build roto.

Un desarrollador no debe hacer commit de código que no compile, antes debe hacer un build privado. Si un build falla, el desarrollador responsable debe identificar la causa lo mas pronto posible y corregirla.

Cuando esta regla no se sigue, toma mas tiempo que el build sea corregido.

Siempre correr todos las pruebas de commit localmente antes de hacer commit

Correr las pruebas de commit localmente es un chequeo antes de la acción de commit. Es una forma de asegurar que lo que creemos que funciona realmente lo haga.

Cuando el desarrollador está listo para hacer commit, debe refrescar la copia local del proyecto actualizando del repositorio de versiones e iniciar el build local.

La importancia de correr las pruebas localmente se debe a:

- Otras personas pueden haber realizado un check in antes del último cambio a verificar y la combinación de ambos podría causar una falla de las pruebas
- Una fuente usual de errores e olvidar agregar nuevos artefactos del repositorio

Esperar que pasen las pruebas de commit antes de avanzar

En el punto del check in, los desarrolladores que lo hicieron son responsables de monitorear el progreso del build y la salida final.

Hasta que el check in haya compilado y pasado las pruebas de commit, los desarrolladores no deben iniciar ninguna tarea nueva.

En caso de falla, deben determinar la naturaleza del problema y arreglarlo.

Nunca irse a casa con un build roto

Cuando un build falla, cerca del fin de un día o semana, hay 3 opciones: arreglarlo e irse tarde, revertir los cambios e intentar después o dejar el build roto.

Debido a que se recomiendan commits frecuentes, una estrategia aplicada es hacer commit 1 hora antes de terminar una jornada laboral o al inicio del día.

Siempre estar preparado para revertir a la revisión previa

En la mayoría de los casos, al encontrar una falla, se podrá repararla y publicar nuevamente un release correcto.

En los casos en los que la corrección de un problema lleve mas tiempo, se debe regresar a la última versión conocida que funcionaba bien, eso lo permite el control de versiones.

Establecer tiempo para reparaciones antes de revertir

Establecer una regla en el equipo: cuando el build esté roto luego de un check in, tratar de repararlo por 10 minutos. Si luego de 10 minutos no se ha finalizado la solución revertir a la versión previa.

No comentar pruebas fallidas

Una vez que se ha establecido la regla previa, algunos desarrolladores comentan (inhabilitan) las pruebas que no pasan para que el build funcione.

Esto hará que la confianza sobre el build generado disminuya, por la omisión de pruebas.

Tomar responsabilidad de todas las fallas que resulten de sus cambios

Si alguien hace commit y pasan las pruebas de sus cambio, pero otras ya no pasan, usualmente esto significa que se introducido un error de regresión.

Es la responsabilidad de quien hizo el commit reparar todas las pruebas que no pasaron como resultado del cambio.

Desarrollo dirigido por pruebas (TDD)

Tener un suite de pruebas completo es esencial para la integración continua.

La única forma de tener una buena cobertura de pruebas es a través del desarrollo dirigido por pruebas.

La idea de TDD es que cuando se desarrolla una nueva pieza de código o se repara un error, los desarrolladores creen primero la prueba que es una especificación ejecutable del comportamiento esperado del código a escribir.

Prácticas sugeridas de implementación

Refactoring

Refactoring significa hacer una serie de cambios pequeños e incrementales que mejoran el código, sin cambiar su comportamiento.

Es buscar hacer tunning al código, cada vez que se hace un cambio o mejora a una funcionalidad, como parte del proceso.

Builds fallidos por brechas de arquitectura

En el script de build pueden haber pruebas que verifiquen que se estén siguiendo reglas de arquitectura.

Ejemplo: uso obligatorio de llamadas remotas.

Builds fallidos para pruebas lentas

En el caso que un build tome mas de cierto tiempo, puede considerarse como fallido y buscar las pruebas que tarden demasiado para mejorarlas.

Builds fallidos por warnings y brechas de estilo de codificación

Los warnings del compilador o algunos de ellos podrían llevar a que falle un build, para forzar su corrección.

También se podría verificar que el código cumpla con el estilo o estándar de codificación establecido (Simian, Jdepend, Checkstyle).