Análisis de código y construcción

Software Developers playbook

Transformación desarrollo software - #ONE

Control de versiones

Versión	Fecha de Creación	Responsable	Descripción
1.0	{17-3-2023}	GSD (Global Software Development)	Primera versión oficial de los playbooks

Índice

1. Introducción **4**
1.1. Acerca de este playbook **4**
1.2. Principios básicos **5**
1.3. Niveles de exigencia **5**
2. Prácticas **6**
2.1. Construir y validar de forma automática el software en un entorno local **7**
2.1.1 Construir y validar de forma automática el software en un entorno local 7
2.2. Integrar el código a un ritmo regular, lo más frecuentemente posible, idealmente varias veces al día **9**
2.2.1 Integrar el código a un ritmo regular, lo más frecuentemente posible, idealmente varias veces al día 9
2.3. Integrar el código y validarlo de forma automática **11**
2.3.1 Entorno efímero 11
2.3.2 Pipelines 11
2.3.3 Pull Requests 12
2.4. Arreglar inmediatamente las construcciones fallidas **14**
2.4.1 Arreglar inmediatamente las construcciones fallidas 14
2.5. Mantener los procesos automáticos de construcción rápidos **16**
2.5.1 Mantener los procesos automáticos de construcción rápidos 16
2.6. Asumir el control de nuestras dependencias y artefactos **18**
2.6.1 Asumir el control de nuestras dependencias y artefactos 18
3. Proceso **20**

1. Introducción

1.1. Acerca de este playbook

Este playbook es un componente fundamental para la práctica de desarrollo de software en el Banco. Define los principios, las prácticas, las herramientas y los indicadores que todos los equipos de desarrollo de software del Banco deben adoptar en los procesos de análisis y construcción de código. Abarca las tareas y los procesos automatizados de análisis y construcción de una aplicación de software, incluida la compilación del código fuente, la ejecución de pruebas, la generación de documentación y el empaquetado del software para su distribución.

El playbook debe ser actualizado regularmente basándose en las prácticas y el panorama tecnológico del Banco. El equipo responsable del documento será el responsable de actualizarlo.

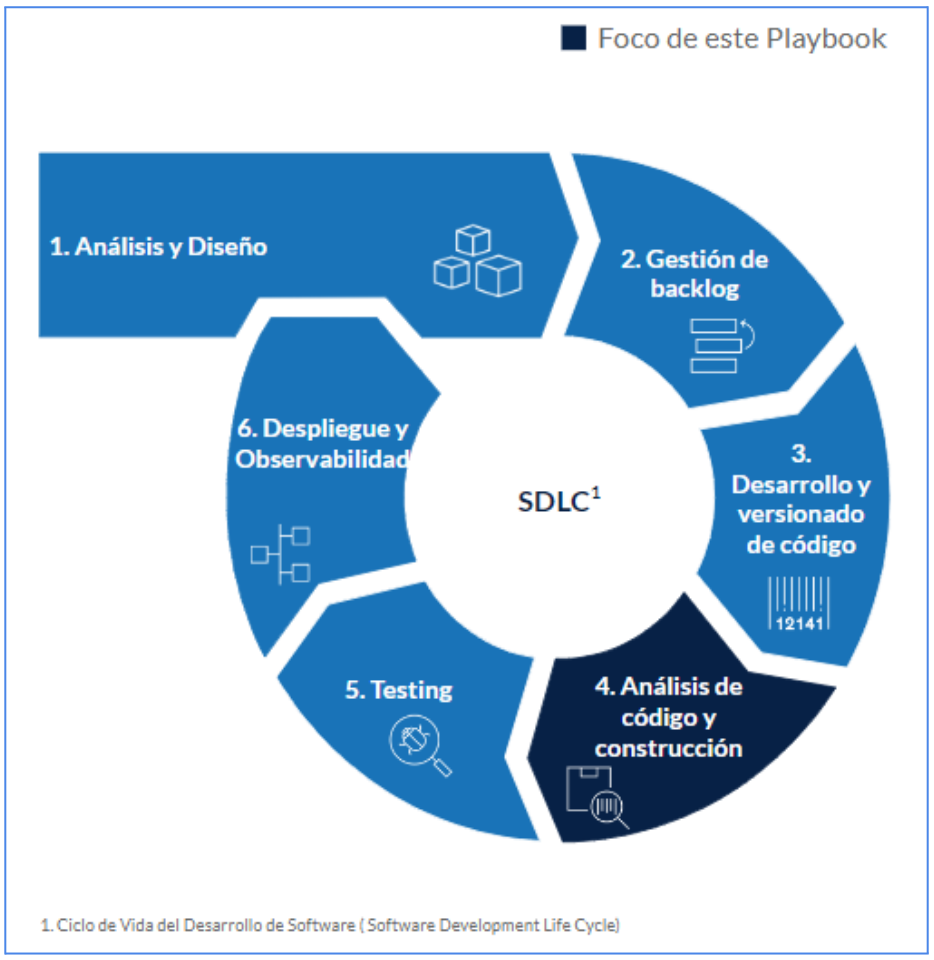

Figura 1 - Ciclo de vida del desarrollo de software

1.2. Principios básicos

Este playbook se guía por un conjunto de principios que todos los desarrolladores debemos seguir:

- **Estandarización:** Usar una taxonomía global y formas de trabajo alineadas entre los equipos y geografías permitirá a los desarrolladores colaborar mejor, compartir mejor el conocimiento y rotar entre los equipos.
- **Calidad sobre cantidad:** Escribir código de calidad debe ser nuestra prioridad ya que la mayor parte de nuestro tiempo lo empleamos manteniendo software heredado y abordando deuda técnica (Maxim and Pressman), cumplir con esto mejorará la velocidad de desarrollo de los equipos a medio y largo plazo.
- **Transparencia y trazabilidad:** El total control y visibilidad de los sistemas y el código del Banco mejora la experiencia de los desarrolladores para navegar en los distintos entornos, responder a incidentes y garantizar el cumplimiento de los requisitos regulatorios.
- **Propiedad:** La alta demanda de calidad requerida en los servicios críticos del banco implica un control detallado de responsabilidad y propiedad de los activos de código. Debemos ser propietarios del código que desarrollamos y sentirnos orgullosos de ello.
- **Pequeños incrementos frecuentes frente a grandes cambios:** Grandes cantidades de código pueden poner en riesgo la estabilidad de nuestros sistemas ya que son más difíciles de probar y deshacer. Además, la mayoría de los sistemas de software evolucionan en función de los comentarios de los usuarios finales y los requisitos son difíciles de anticipar de antemano. Por tanto, es importante desarrollar rápidamente pequeños incrementos para aprender rápidamente y rectificar.
- **Simplicidad:** Los sistemas de software rara vez se escriben una vez y no se modifican. Más bien, muchas personas trabajan con el mismo sistema a lo largo del tiempo. Por eso es importante buscar las soluciones más sencillas que puedan comprenderse fácilmente y evolucionar en el futuro.
- **Automatización:** Los procesos manuales que son repetitivos deben automatizarse para mejorar nuestra productividad y enfocarnos en el desarrollo.

1.3. Niveles de exigencia

Este playbook utiliza intencionadamente las siguientes tres palabras para indicar los niveles de exigencia según la norma **RFC2119**:

- **Debe** - Significa que la práctica es un requisito absoluto.
- **Debería** - Significa que pueden existir razones válidas en circunstancias particulares para ignorar una práctica, pero deben comprenderse todas las implicaciones y sopesar cuidadosamente antes de elegir un camino diferente.
- **Podría** - Significa que una práctica es realmente opcional.

2. Prácticas

Todos los equipos de software deben seguir las **seis prácticas** siguientes relativas al análisis y la construcción de código, independientemente de la tecnología y el lenguaje de programación utilizados:

1. Construir y probar de forma automática el software en un entorno local
2. Integrar el código a un ritmo regular, lo más frecuentemente posible, idealmente varias veces al día.
3. Integrar el código y validarlo de forma automática
4. Arreglar inmediatamente las construcciones rotas
5. Mantener los procesos automáticos de construcción rápidos
6. Asumir el control de nuestras dependencias y artefactos

A continuación se definen cada una de las prácticas siguiendo la siguiente **estructura**:

- Sus beneficios (que conseguimos).
- Precondiciones (condiciones para poder aplicar la práctica).
- Adopción (cómo implementarla).
- Herramientas necesarias.
- Indicadores para los desarrolladores (como la medimos).
- Enlaces de interés.

2.1. Construir y validar de forma automática el software en un entorno local

Beneficios

Tener disponible un entorno local para probar nuestro código de forma automática y eficiente tiene grandes beneficios:

- Mejora la productividad y motivación de todos los desarrolladores.
- Reduce el número de errores en el código.
- Elimina la dependencia con otros servicios o entornos remotos.
- Aumenta la colaboración, al poder integrarnos más rápidamente con otros compañeros del equipo.
- Hace los ciclos de desarrollo más cortos al poder probar los cambios lo antes posible.

Precondiciones

Para poder aplicar la práctica se deben dar las siguientes **precondiciones**:

- La tecnología debe poder **ejecutarse en el entorno local**. Los frameworks y arquitecturas deben garantizar esta capacidad.

Adopción

2.1.1 Construir y validar de forma automática el software en un entorno local

Todos los ingenieros de software **deben** poder garantizar que los desarrollos cumplen los requisitos para los que fue creado. Para poder cumplir con este cometido es esencial tener un entorno local de desarrollo dónde poder validar nuestra funcionalidad de la forma más automatizada posible.

De forma agnóstica a la plataforma, tecnología y lenguaje de programación los equipos de desarrollo:

- **Deben** disponer de un **entorno local,** donde puedan validar que el software que desarrollan cumple con los requisitos establecidos. **Dicho entorno** debe ser similar al de producción en cuanto a funcionalidad ofrecida, permitiendo ejecutar pruebas automáticas y manuales sobre el software bajo las siguientes premisas:
  - **Debe** contener versiones de software alineadas con las de producción y las arquitecturas con las que se trabaja.
  - **Debe** ser un entorno autocontenido y aislado (que permita al desarrollador trabajar sin bloquear a otros y sin tener dependencias externas).
  - **Debe** contener datos o sistemas de datos que permitan validar la funcionalidad desarrollada.
  - **Debería** ser un entorno contenerizado o en su defecto virtualizado. Habrá soluciones que no lo requieran (e.g., librerías).
- **Deben** tener scripts de construcción, validación y despliegue del código automáticos alineados con los existentes para su arquitectura / plataforma que permiten:
  - La construcción del artefacto que están desarrollando.
  - La creación de los entornos locales mencionados anteriormente.
  - El despliegue del software en dicho entorno local.

- **Deben** disponer de un equipo local adecuado y adaptado a las necesidades para soportar el entorno de desarrollo local.
- **Deberían** (siempre que la tecnología y el lenguaje de programación lo permita) hacer uso de herramientas locales para análisis estático de código y de seguridad.
- **Deberían** hacer uso de fakes (i.e., réplica consumible de un servicio no productivo actualizada a la última versión) y el servicio **debe** proveerlos, siendo accesibles desde el propio entorno local mencionado.
- **Deberían** hacer uso de mocks en el entorno local (más detalle en el Playbook de testing).
- **Deberían** contar con un conjunto coherente de datos que le permitan probar su funcionalidad.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Bitbucket:** El código de los desarrolladores debe estar alojado en bitbucket para poder copiarlo al entorno local y construirlo.
- **Gestor de artefactos (Artifactory):** Los equipos deben almacenar sus artefactos e imágenes en un servicio único de gestión de artefactos (Artifactory).
- **IDE:** El entorno de desarrollo integrado de los desarrolladores debe permitir el desarrollo de código e integrarse con el entorno local de validación.
- **Equipo local**: El equipo local debe tener la capacidad suficiente para desplegar el entorno local contenerizado o virtualizado descrito en la práctica.
- **Análisis de código estático:** Lint, PMD, checkstyle, SonarQube y herramientas similares deberían ser adoptadas para el análisis estático de código de forma local.
- **Chimera:** Se debería aplicar (cuando lo permita la tecnología y el lenguaje) para llevar a cabo análisis de seguridad y detección de vulnerabilidades en el código.
- **Samuel**: Repositorio unificado de enforcement automático, en relación a la práctica debería dar soporte a los pre-hooks locales.
- **Docker**: Los equipos deberían hacer uso de docker para generar contenedores del entorno local mencionado que les permita construir y ejecutar su software. Los equipos deberían disponer de licencias de la herramienta si lo necesitan.
- **Catálogo utilidades (Herramienta pendiente de definición):** Debería existir un catálogo de utilidades para que los desarrolladores puedan agilizar su trabajo, dicho catálogo contendrá utilidades para trabajar en el entorno local (e.g., pre-hooks, scripts ansible, scripts ssh).

Enlaces

- https://martinfowler.com/articles/developer-effectiveness.html
- https://www.martinfowler.com/articles/mocksArentStubs.html
- https://es.wikipedia.org/wiki/Objeto_simulado

2.2. Integrar el código a un ritmo regular, lo más frecuentemente posible, idealmente varias veces al día

Beneficios

Los de esta práctica son los siguientes:

- Mejorar la calidad del software al identificar errores lo antes posible.
- Recibir feedback sobre nuestro código antes.
- Ayudar a dividir e integrar el código en pequeños incrementos.
- Incrementar la velocidad de entrega.
- Fomentar el trabajo en equipo y la colaboración.

Adopción

2.2.1. Integrar el código a un ritmo regular, lo más frecuentemente posible, idealmente varias veces al día

Los ingenieros de software **deberíamos** hacer commits idealmente de forma diaria e integrarnos lo antes posible con el código de nuestros compañeros evitando grandes conflictos que retrasan la puesta en producción de nuestro software.

El proceso de integración debe ejecutar las pruebas que hayamos ejecutado en nuestro entorno local y las propias pruebas del entorno integrado (definidas en el playbook de testing).

La adopción de esta práctica por parte de los equipos requiere aplicar las siguientes directrices:

- Los incrementos de software **deben** ser pequeños y las funcionalidades que sean distintas **deben** estar desacopladas.
- Los equipos **deben** conocer las prácticas de integración y entrega continua alineadas con los principios de este playbook.
- Los equipos **pueden** aplicar mecanismos de programación por parejas para fomentar la integración regular del código aplicando pequeños incrementos en el software (para más detalle sobre técnicas de revisión de código ir al playbook de desarrollo y versionado de código).

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Bitbucket:** El código de los desarrolladores debe estar alojado en bitbucket para poder disponer de él en local y construirlo.
- Samuel: Repositorio unificado de enforcement automático, en relación a la práctica debería dar soporte a los pre-hooks locales.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Frecuencia de commits	Número medio de commits en ramas de nueva funcionalidad (feature branches). Para equipos haciendo uso de dichas ramas
Número de integraciones al día	Número medio de integraciones en la rama principal al día

Enlaces

- https://martinfowler.com/books/duvall.html
- https://martinfowler.com/articles/continuousIntegration.html#EveryoneCommitsToTheMainlineEveryDay
- https://martinfowler.com/bliki/PullRequest.html
- https://martinfowler.com/articles/on-pair-programming.html

2.3. Integrar el código y validarlo de forma automática

Beneficios

Los beneficios de esta práctica son los siguientes:

- Mejorar la colaboración y el feedback.
- Incrementar la velocidad de desarrollo de todo el equipo de desarrollo.
- Validar el código integrado rápidamente, sin bloqueos y con claridad.
- Aumentar la calidad del código.
- Evitar promocionar código a entornos finales que sólo funciona en el entorno local.

Precondiciones

- El código debe estar versionado en la **herramienta de control de versiones**.
- Se debe disponer de un **sistema de automatización de pipelines** de construcción y validación con acceso a los repositorios de código.
- Las tecnologías deben permitir la ejecución de **entornos efímeros**.

Adopción

Desarrollar software es un trabajo en equipo. Por tanto, **debemos** **integrar nuestro código con el resto de compañeros y validar que funciona correctamente** de forma automática lo antes posible.

El objetivo de integrar nuestra funcionalidad es ejecutar las mismas validaciones que hicimos en el entorno local (e.g., unit tests, pruebas manuales, análisis estático de código) en un entorno integrado, junto con los últimos cambios en el código.

2.3.1. Entorno efímero

Los equipos **deben** tener un **entorno efímero** donde poder integrar su código, dicho entorno debe ser similar al entorno local referido en la práctica 2.1 y los desarrolladores deben poder validar su software de forma independiente (sin bloquear a otros).

- Las validaciones realizadas en este entorno **deben** hacerse sobre versiones oficiales de los servicios de terceros (a.k.a. Fakes). El desarrollador que valida su funcionalidad **debe** garantizar que las versiones de los servicios de terceros son correctas.

2.3.2. Pipelines

El entorno efímero **debe** ser orquestado por **pipelines** alojados en un servidor que los automatice (servidor automatización) con las siguientes características:

- **Debe** construir el software de forma automática y en un entorno similar al que tendrá la versión final en producción.
- **Debe** validar el código, ejecutando:
  - Los equipos que lo requieran, pueden añadir un paso de validación de políticas de desarrollo (e.g., validar que las dependencias sean cerradas).
  - Análisis de código estático.

        - Baterías de tests y pruebas necesarias (descritas en el playbook de testing).
    - Los pipelines **deben** ser rápidos y no bloquear el desarrollo (menos de 1 hora).
    - Los logs de cada ejecución de los pipelines **deben** ser legibles y accesibles.
    - **Debería** integrarse el nuevo código con la rama principal antes de ser construido.
    - Se **debería** notificar al equipo cuando haya nuevas construcciones.

2.3.3. Pull Requests

Los equipos **deberían** aplicar mecanismos de **Pull Requests** para asegurar la integración y validación del código de forma automática.

- Las Pull Request **deben** usarse exclusivamente para los siguientes fines:
    - La revisión del código (por parte de otro compañero de equipo) manual.
    - La validación de políticas de un catálogo global (e.g., validar semántica de ramas) adoptadas por el equipo y totalmente automáticas.
    - La validación del código (mediante pruebas y tests) automático.
    - El análisis de código (mediante herramientas de análisis estático de código) automatizado.
- Las Pull Requests **deben** orquestar los pipelines automáticos de integración y validación definidos en la sub-práctica anterior (pipelines).
- **Debe** existir una última construcción correcta del pipeline en la Pull Request para que pueda ser aceptada y por tanto mergeada.
- Para poder aceptar una Pull Request los equipos **deberían** integrar la rama del nuevo código con la rama destino y si hubiera cambios nuevos volver a pasar el proceso de integración.
- Los equipos **pueden** definir templates de Pull Request como mecanismo para especificar qué información mínima es relevante incluir en ellas.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Jenkins:** Servidor de automatización único dónde alojar los pipelines de construcción e integración.
- **Bitbucket:** El código de los desarrolladores debe estar alojado en bitbucket para poder copiarlo en local y construirlo.
- **Gestor de artefactos (Artifactory):** Los equipos deben almacenar sus artefactos e imágenes en un servicio único de gestión de artefactos (Artifactory).
- **Análisis de código estático:** Lint, PMD, checkstyle y herramientas similares deben ser adoptadas para el análisis estático de código de forma local.
- **Chimera:** Se debería aplicar (cuando lo permita la tecnología y el lenguaje) para llevar a cabo análisis de seguridad y detección de vulnerabilidades en el código.
- **Samuel**: Repositorio unificado de enforcement automático, en relación a la práctica debería dar soporte a los pre-hooks locales.
- **Docker**: Los equipos deberían hacer uso de docker para generar contenedores del entorno local mencionado.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Tiempo medio Pull Requests	Tiempo medio en días que las Pull Requests permanecen abiertas (en caso de aplicar Pull Requests).
Porcentaje Pull Requests con construcciones automáticas	Porcentaje de Pull Requests que orquestan pipelines de construcción y validación automáticos
Rechazo Pull Requests	Ratio de Pull Requests rechazadas y abiertas en los últimos 3 meses
Tiempo medio interacción Pull Requests	Tiempo medio en horas entre interacciones en una Pull Requests, entendiendo interacción eventos como comentarios, cambios en el código o aceptación de la misma
Construcciones fallidas	Ratio entre construcciones fallidas y exitosas en un periodo ventana de 2 semanas

Enlaces

- https://martinfowler.com/articles/continuousIntegration.html#EveryCommitShouldBuildTheMainlineOnAnIntegrationMachine
- http://www.jamesshore.com/v2/blog/2005/continuous-integration-is-an-attitude

2.4. Arreglar inmediatamente las construcciones fallidas

Beneficios

Los beneficios de esta práctica son los siguientes:

- Incrementar la calidad del código al resolver problemas antes de integrar código.
- Mejorar la productividad al no tener que cambiar de tarea por encontrarnos con flujos "rotos".
- Mejorar la colaboración entre los ingenieros al reducir los cuellos de botella.
- Fomentar el sentimiento de responsabilidad y la cultura del equipo.
- Facilitar la identificación de errores lo antes posible.

Precondiciones

- Debemos disponer de un **sistema de automatización de pipelines** de construcción y validación.

Adopción

2.4.1. Arreglar inmediatamente las construcciones fallidas

El proceso automático de construcción e integración de software puede fallar por distintos motivos (e.g., problemas de entorno, tests con errores, herramientas fuera de servicio) y es responsabilidad de los desarrolladores que están integrando el nuevo código arreglar o escalar el problema lo antes posible:

- Los desarrolladores **deben** vigilar los pipelines de construcciones orquestados por sus commits, evitando el concepto de "commit and run".
- Los desarrolladores **deben** identificar el origen de las construcciones fallidas lo antes posible y arreglarlas.
- **Debe** existir un sistema de alarmado visible para todo el equipo.
- Los desarrolladores **deben** estar preparados para revertir el código a una versión anterior en caso que no sea posible arreglar fácilmente el fallo en la construcción.
- **Deben** existir unas políticas de actuación clara en caso de pipelines de construcción no satisfactorias.
    - Los equipos **pueden** definir un tiempo máximo en que una construcción se encuentra fallando para tomar las actuaciones pertinentes (e.g. revertir código, alertar sistemas de terceros fallidos).
- Los desarrolladores **no deberían** añadir código mientras exista una construcción fallida para facilitar a las personas que están reparando la construcción su tarea.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Jenkins:** Servidor de automatización único dónde alojar los pipelines de construcción e integración.
- **Chimera:** Se debería aplicar (cuando lo permita la tecnología y el lenguaje) para llevar a cabo análisis de seguridad y detección de vulnerabilidades en el código.

- **Samuel**: Repositorio unificado de enforcement automático, en relación a la práctica debería notificar de errores en las construcciones.
- **Radiadores de información (herramienta pendiente de definición)**: Los equipos deben disponer de un lugar centralizado para visualizar el estado de sus pipelines de construcción y validación del software.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Tiempo medio construcciones fallidas	Tiempo medio en horas desde que una construcción falla y se arregla (siguiente construcción con éxito)
Ratio construcciones fallidas y correctas	Número de construcciones fallidas entre el número de construcciones correctas en el periodo de una semana
Porcentaje construcciones fallidas notificadas	Porcentaje de construcciones finalizadas con error que han sido notificadas al equipo

Enlaces

- https://learning.oreilly.com/library/view/97-things-every/9780596800611/ch15.html
- https://martinfowler.com/articles/continuousIntegration.html#FixBrokenBuildsImmediately

2.5. Mantener los procesos automáticos de construcción rápidos

Beneficios

Los beneficios son los siguientes:

- Recoger feedback temprano sobre el estado de nuestro código.
- Mejorar el foco en las tareas, al no tener que cambiar de contexto por tiempos de espera prolongados en los procesos de construcción y validación.
- Incrementar la velocidad de entrega de la nueva funcionalidad.
- Mejorar la calidad de los tests y el código.

Precondiciones

- El código debe estar versionado en la **herramienta de control de versiones**.
- Se debe disponer de un **sistema de automatización de pipelines** de construcción y validación con acceso a los repositorios de código.

Adopción

2.5.1. Mantener los procesos automáticos de construcción rápidos

Todo el código que desarrollemos **debe** pasar por un proceso de construcción y validación, por tanto, cuanto más rápido sea este proceso mejor será la experiencia de desarrollo.

Reducir estos tiempos no significa dejar de validar nuestro código o reducir la validación al máximo, el objetivo es buscar un equilibrio entre el tiempo de ejecución y las validaciones necesarias para garantizar que nuestro código está libre de errores y cumple con los requerimientos funcionales y las buenas prácticas establecidas. En relación al o anterior:

- Se **deben** establecer timeouts en los pipelines de construcción.
- Los equipos **deben** disponer de un sistema multitenant con recursos necesarios para lanzar los pipelines de construcción y validación (incluyendo tests y análisis de código).
- Los equipos **deben** disponer de tiempo para optimizar el proceso de construcción y validación, además de los pipelines.
- **Debe** existir un radiador de la información para el equipo que les permita recibir feedback de sus procesos de construcción y validación.
- Los equipos **pueden** saltarse algunas de las fases del pipeline de forma manual y para situaciones que lo requieran, pero el software **no debe** pasar a la siguiente etapa de puesta en producción sin tener una última ejecución completa y satisfactoria.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Jenkins:** Servidor de automatización multitenant que debe tener capacidad para soportar y optimizar la construcción y validación del software.
- **Equipo local:** El entorno local debe ser capaz de ejecutar los procesos de construcción y validación del código en cuestión de minutos. Es responsabilidad del desarrollador optimizar el proceso, pero se le debe proveer el hardware adecuado para no exceder el tiempo dedicado a este proceso.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Tiempo medio construcciones	Tiempo medio en minutos empleado en los pipelines automáticos de construcción y validación
Número medio de pipelines finalizados con timeout	Número medio de pipelines finalizados con timeout durante el periodo de una semana

Enlaces

- https://martinfowler.com/articles/continuousIntegration.html#KeepTheBuildFast

2.6. Asumir el control de nuestras dependencias y artefactos

Beneficios

Los beneficios de tener artefactos publicados inmutables son los siguientes:

- Incrementar la confianza en los artefactos publicados.
- Mejorar la trazabilidad de los componentes desplegados.
- Aumentar la reproducibilidad, permitiendo ejecutar validaciones o replicar errores en producción con mayores garantías.
- Cumplir con requisitos regulatorios bancarios.

Precondiciones

- Debemos tener implantado un sistema centralizado para la gestión de artefactos.

Adopción

2.6.1. Asumir el control de nuestras dependencias y artefactos

Una vez el software se construye y publica de forma centralizada en el gestor de artefactos esas **versiones no deben sobreescribirse**. Si es necesario generar una nueva versión se procederá a incrementar la versión y guardar un nuevo artefacto en el gestor de artefactos.

Es responsabilidad del ingeniero de software aplicar las siguientes directrices:

- **Debe** garantizar que nuestros artefactos sean inmutables, es decir las releases publicadas no se sobreescriben.
- **Debe** aplicar los principios SOLID relacionados con el empaquetado de software (SRP y DIP).
- **Debe** aplicar los principios de common closure[1].
- **Debe** evitar dependencias abiertas en releases productivas.
- **Debe** disponer de un gestor de artefactos global y único con dominios distintos para versiones snapshot (en desarrollo) y versiones finales (estables).
- **Debería** fijar las dependencias que sean transitivas (i.e. dependencias de terceros que vienen con las propias de nuestro código).

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Jenkins:** Debe proporcionar capacidad de trazabilidad de las ejecuciones y unos logs claros para identificar las dependencias descargadas por nuestro software en el proceso de construcción.
- **Gestor de artefactos (Artifactory):** Debe tener dominios distintos para las releases de desarrollo y producción. También debe ser accesible y navegable por todos los equipos.
- **Samuel**: Debería asegurar que se cumplen las directrices establecidas en la práctica.

[1] Las clases y módulos que cambien por el mismo motivo deberían estar incluidos en el mismo paquete. Por tanto, si algo cambia en un aplicativo debería hacerlo dentro del mismo paquete.

Enlaces

- https://es.wikipedia.org/wiki/SOLID
- https://learning.oreilly.com/library/view/principles-of-package/9781484241196/html/471891_1_En_8_Chapter.xhtml

3. Proceso

Las prácticas revisadas anteriormente se operativizan a través del **proceso de análisis y construcción de código**. Dicho proceso es orquestado mediante pipelines automatizados (tal y como se describe en las prácticas de este playbook) que ejecutan al menos las siguientes fases después de que el ingeniero de software integre el código en una de sus ramas:

1. **Gestión de dependencias:** Esta fase implica gestionar las dependencias del código, incluyendo bibliotecas externas o módulos necesarios para que el código funcione correctamente. Se utilizan herramientas de gestión de dependencias para manejar y resolver estas dependencias (e.g., Gradle, Maven, Setuptools, Npm).

2. **Análisis estático de código:** Se realiza un análisis estático del código para examinarlo sin ejecutarlo. Se verifican estándares de codificación, posibles errores, vulnerabilidades de seguridad y otros problemas de calidad de código. Se utilizan herramientas de análisis estático (e.g., SonarQube, Chimera) para escanear el código y proporcionar retroalimentación sobre posibles problemas.

3. **Construcción**: En esta fase, el código fuente se traduce a código legible por la máquina mediante un compilador (para lenguajes compilados) y/o herramientas de empaquetado de software. El compilador verifica el código en busca de errores de sintaxis (normalmente para lenguajes compilados) y genera archivos binarios o ejecutables.

4. **Validación tests**: Se ejecutan de forma automática los tests asociados al código que validan unidades o componentes individuales (tests unitarios) o la interacción entre diferentes componentes o módulos del código (tests de integración). Para más información sobre los tipos de pruebas ejecutados en esta fase revisar el playbook de testing.

5. **Publicación:** El fichero binario generado se publica en la herramienta de gestión de artefactos para poder ser desplegado en los entornos necesarios.

GLOSARIO

Equipo: Conjunto de personas trabajando sobre un mismo backlog de tareas. Desarrollan nueva funcionalidad en el código.

Servicio: Productos de software (e.g., aplicación web, aplicación móvil, librerías de transacciones).

REFERENCIAS

- Glover, Andrew, et al. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007. Accessed 10 March 2023.
- Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Addison-Wesley Signature Series (Fowler)) 1st Edición de Jez Humble (Author), David Farley (Author)