**The Agent**

**Documentation technique.**



Table des matières

[I. Intégration continue 3](#_Toc426970953)

[1. Définition 3](#_Toc426970954)

[2. Pratiques de l'intégration continue 3](#_Toc426970955)

[ Maintenir un gestionnaire de sources unique 3](#_Toc426970956)

[ Automatiser la construction (Build) 3](#_Toc426970957)

[ Rendre le Build « auto-testant » pour capturer les bugs 3](#_Toc426970958)

[ Tout le monde soumet son travail (commit) au même endroit. 4](#_Toc426970959)

[ Chaque commit doit déclencher un build sur la machine d’intégration 4](#_Toc426970960)

[ Garder le Build rapide afin d’avoir un feedback le plutôt possible 4](#_Toc426970961)

[ Tester dans un environnement qui une copie de celui de production 4](#_Toc426970962)

[ Rendre facilement disponibles les exécutables les plus récents 4](#_Toc426970963)

[ Tout le monde peut voir ce qu'il se passe 5](#_Toc426970964)

[ Automatiser le déploiement 5](#_Toc426970965)

[3. Avantages de l’intégration continue 6](#_Toc426970966)

[ Réduire les risques 6](#_Toc426970967)

[ Réduire les tâches répétitives 6](#_Toc426970968)

[ Générer un logiciel déployable. 6](#_Toc426970969)

[4. Mise en œuvre 6](#_Toc426970970)

[5. Outils facilitant la mise en œuvre 6](#_Toc426970971)

[II. Stratégie de branches 7](#_Toc426970972)

[III. Jenkins 8](#_Toc426970973)

[1. Le Build 8](#_Toc426970974)

[2. La Release 8](#_Toc426970975)

[3. Le Déploiement 8](#_Toc426970976)

[4. Enlever le tag 9](#_Toc426970977)

[IV. Rabit MQ 9](#_Toc426970978)

[1. Installation 9](#_Toc426970979)

[2. Créer une queue 9](#_Toc426970980)

[3. Publier des messages 10](#_Toc426970981)

[4. Recevoir les messages 11](#_Toc426970982)

# Intégration continue

## Définition

L'intégration continue est une pratique de développement logiciel où les membres d'une équipe intègrent leur travail fréquemment.

En général, chacun intègre au moins quotidiennement - conduisant à de multiples intégrations par jour. Chaque intégration est vérifiée par un système automatisé de construction (y compris les tests) pour détecter les erreurs d'intégration le plus rapidement possible.

Beaucoup d'équipes trouvent que cette approche conduit à une réduction considérable des problèmes d'intégration et permet à une équipe de développer un logiciel de qualité plus rapidement. « Martin Fowler »

## Pratiques de l'intégration continue

### Maintenir un gestionnaire de sources unique

Cela permet au code source de ne pas être éparpillé. Il permet également de garder un audit complet sur l’ensemble des changements qui ont eu lieu au fils des années.

Il existe plusieurs gestionnaires de source ; Git, SVN, CVS …

Le logiciel doit être construit à partir du gestionnaire de sources. Celui-ci doit être comme le lieu unique qui contient le code source de référence.

### Automatiser la construction (Build)

Le build d’un logiciel peut être un processus long et fastidieux. Son automatisation permet d’éviter les erreurs humaines (oublie de certains fichiers ou certaines étapes …) et de gagner un temps considérable que peut être alloué à des tâches à plus forte valeur ajoutée.

### Rendre le Build « auto-testant » pour capturer les bugs

Traditionnellement, un build signifie compiler, lier et toutes les autres tâches requises pour obtenir un programme exécutable.

Or un programme peut compiler mais cela ne signifie en aucun cas qu’il marche et qu’il est exempt de tout bug.

C’est pour cette raison qu’il faut associer à la phase de build une batterie de tests automatisés.

Ces tests sont notamment ce qu’on appelle les tests unitaires, des framworks tel junit et xunit aident à leur réalisation.

### Tout le monde soumet son travail (commit) au même endroit.

Le fait de faire des commits au même endroit améliore la communication. Ça permet et chaque développeur de savoir ce qu’a fait son voisin.

Avant de faire un commit le développeur doit récupérer le contenue du SCM, faire un build et exécuter les tests afin de s’assurer que son travail n’introduit pas de régression.

Il est aussi souhaitable que les développeurs fassent des commits tous les jours de manière à faciliter l’intégration de leur code. Ça permet également d’avoir des feedbacks réguliers.

### Chaque commit doit déclencher un build sur la machine d’intégration

Cela permet de s’assurer que le code est toujours dans état stable. Le but est de corriger les erreurs le plutôt possible. Ça évite leur accumulation.

Le fait de corriger tout de suite, évite au développeur de se replonger dans code ancien et de switch de contexte.

### Garder le Build rapide afin d’avoir un feedback le plutôt possible

L’intérêt de l’intégration continue est d’avoir un feedback rapide. La méthode XP (eXtreme Programming) recommande de ne pas dépasser dix minutes.

L’intérêt c’est aussi d’inclure le moins de commits possible à chaque build pour faciliter la correction en cas de bug.

### Tester dans un environnement qui une copie de celui de production

L’intérêt des tests et de trouver des bugs dans un environnement maitrisé.

Si vous testez dans un environnement différent, chaque divergence entraine un risque que ce qui arrive en cours de test n'arrivera pas en production.

### Rendre facilement disponibles les exécutables les plus récents

Toute personne participant à un projet de logiciel devrait être en mesure d'obtenir les derniers exécutables et être capable de les exécuter : pour des démonstrations, des essais exploratoires, ou juste pour voir ce qui a changé cette semaine.

### Tout le monde peut voir ce qu'il se passe

L’idée c’est de rendre visible les résultats de builds via une interface graphique out tout autre système.

Certaines équipes utilisent un système de lampes qui passent de vert en cas de succès à rouge en cas d’échec.

### Automatiser le déploiement

Pour éviter des erreurs humaines mais aussi gagner du temps. Notamment lorsqu’il s’agit de déployer les scripts de base de données ou toute autre ressource sensible.

|  |
| --- |
| Intégration Continue |

## Avantages de l’intégration continue

### Réduire les risques

Elle permet notamment d’éviter les cas où le logiciel marche sur le PC du développeur mais pas sur la machine de déploiement.

### Réduire les tâches répétitives

Toutes les tâches répétitives comme la compilation le build les tests et l’inspection du code sont exécutées de manière automatique.

### Générer un logiciel déployable.

Grâce à l’automatisation de l’ensemble des tâches, l’intégration continue permet de générer un logiciel que l’on peut déployer à tout moment de manière rapide et efficace.

## Mise en œuvre

L’ensemble des pratiques peuvent être mise en œuvre de manière progressive.

On peut commencer par mettre en œuvre un gestionnaire de source et un Build automatisé et ajouter ensuite les autres briques au fur et à mesure.

## Outils facilitant la mise en œuvre

Il existe toute une panoplie d’outils facilitant la mise en œuvre d’un environnement d’intégration continue, en voici un échantillon parmi les outils les plus communément utilisés :

* **Gestionnaire de** **source** : SVN, Git …
* **Build et Tests unitaires** : Maven et Junit ou TestNG.
* **Inspection de la qualité du code** : Sonar.
* **Déploiement des changements de base de** **données** : Liquibase.
* **Serveur d’intégration continue** : Jenkins, hudson, TeamCity, Bamboo.

# Stratégie de branches

|  |
| --- |
| Stratégie de branches |

1. Les développements seront commités sur la branche **master**.
2. Lors de la release un **tag** est créé.
3. Si un bug se présente, on tire une **branche** depuis le **tag,** et on **release** à partir de cette branche et finalement on **merge** dans **master**.
4. Dans certains cas on peut créer une branche dans le cadre d’un POC (Proof Of Concept) par exemple, et puis faire un **merge** sur **master**.

# Jenkins

Jenkins joue le rôle de chef d’orchestre dans l’intégration continue.

Il nous permet des réaliser les taches suivantes :

## Le Build

Le job **theagent\_build** récupèrele code source depuis github, examine la qualité du code et effectue les tests unitaires.

Les rapports produits sont dans le répertoire **build**.

## La Release

Le job **theagent\_release** permet de tagger le code qui est sur la branche master afin d’être déployé sur la préprode et la prod.

## Le Déploiement

Lorsque le tag créer, on va utiliser le job **theagent\_deploy\_preprod** pour déployer et réaliser les tests fonctionnels (le job theagent\_functional\_tests est exécuté automatiquement par theagent\_deploy\_preprod) sur la machine de préprod.

En cas d’échec, on utilise theagent\_rollback\_preprod, pour revenir à la version précédente.

Les jobs **theagent\_deploy\_pprod** et **theagent\_rollback\_prod** permettent respectivement de déployer et de faire un rollback sur la production.

## Enlever le tag

Quand un tag est créé suite à une release qui est rollbackée, on supprime le tag avec le job theagent\_remove\_tag.

# Rabit MQ

## Installation

|  |
| --- |
| echo "deb http://www.rabbitmq.com/debian/ testing main" >> /etc/apt/sources.list  apt-get update  wget https://www.rabbitmq.com/rabbitmq-signing-key-public.asc  apt-key add rabbitmq-signing-key-public.asc  apt-get install rabbitmq-server  rabbitmq-plugins enable rabbitmq\_management  echo "[{rabbit, [{loopback\_users, []}]}]." >>/etc/rabbitmq/rabbitmq.config  service rabbitmq-server restart |

Pour configurer le firewall, utiliser : vi /etc/init.d/firewall

et ajouter

|  |
| --- |
| iptables -t filter -A INPUT -p tcp --dport 15672 -j ACCEPT  iptables -t filter -A INPUT -p tcp --dport 4369 -j ACCEPT  iptables -t filter -A INPUT -p tcp --dport 25672 -j ACCEPT  iptables -t filter -A INPUT -p tcp --dport 5672 -j ACCEPT  iptables -t filter -A INPUT -p tcp --dport 5671 -j ACCEPT  iptables -t filter -A INPUT -p tcp --dport 61613 -j ACCEPT  iptables -t filter -A INPUT -p tcp --dport 61614 -j ACCEPT  iptables -t filter -A INPUT -p tcp --dport 1883 -j ACCEPT  iptables -t filter -A INPUT -p tcp --dport 8883 -j ACCEPT |

L’interface de gestion est sur : <http://212.129.16.12:15672/> (guest/guest)

## Créer une queue

* Sous l’onglet Queues, cliquer sur « Add new queue »
* Choisir le nom (« Name »)
* En sélectionnant **Durable** dans « Durability », les messages sont persistés (Pas de perte en cas d’arrêt du serveur).

|  |
| --- |
|  |

## Publier des messages

Le code est disponible entièrement sur : <https://github.com/omebarki/rabitmq_poc>

|  |
| --- |
| **package** com.theagent.rabitmq.pub;  **import** com.rabbitmq.client.Channel; **import** com.rabbitmq.client.Connection; **import** com.rabbitmq.client.ConnectionFactory; **import** com.theagent.rabitmq.Constants;  */\*\*  \* Created by Omar MEBARKI on 07/08/2015.  \*/* **public class** RabitPublisher {  **public static void** main(String[] args) **throws** Exception {  ConnectionFactory factory = **new** ConnectionFactory();  factory.setHost(Constants.QUEUE\_HOST);  Connection connection = **null**;  Channel channel = **null**;  **try** {  connection = factory.newConnection();  channel = connection.createChannel();   String message = **"Hello World!"**;  channel.basicPublish(**""**, Constants.***QUEUE\_NAME***, **null**, message.getBytes());  System.***out***.printf(**"message sent:"** + message);   } **finally** {  **try** {  channel.close();  } **catch** (Exception e) {  }  **try** {  connection.close();  } **catch** (Exception e) {  }  }   } } |

## Recevoir les messages

Dans cet exemple, l’acquittement est manuel (autoAck = **false**).

Une fois affiché le message est acquitté  (channel.basicAck(delivery.getEnvelope().getDeliveryTag(), **false**);)

|  |
| --- |
| **package** com.theagent.rabitmq.sub;  **import** com.rabbitmq.client.Channel; **import** com.rabbitmq.client.Connection; **import** com.rabbitmq.client.ConnectionFactory; **import** com.rabbitmq.client.QueueingConsumer; **import** com.theagent.rabitmq.Constants;  */\*\*  \* Created by Omar MEBARKI on 07/08/2015.  \*/* **public class** RabitConsumer {     **public static void** main(String[] args) **throws** Exception {  ConnectionFactory factory = **new** ConnectionFactory();  factory.setHost(Constants.***QUEUE\_HOST***);  Connection connection = **null**;  Channel channel = **null**;  **try** {  connection = factory.newConnection();  channel = connection.createChannel();   QueueingConsumer consumer = **new** QueueingConsumer(channel);  **final boolean** autoAck = **false**;  channel.basicConsume(Constants.***QUEUE\_NAME***, autoAck, consumer);    **while** (**true**) {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = **new** String(delivery.getBody());  System.***out***.println(**" [x] Received '"** + message + **"'"**);  channel.basicAck(delivery.getEnvelope().getDeliveryTag(), **false**);  }   } **finally** {  **try** {  channel.close();  } **catch** (Exception e) {  }  **try** {  connection.close();  } **catch** (Exception e) {  }   }   } } |