Chapitre 1-3
Les outils techniques

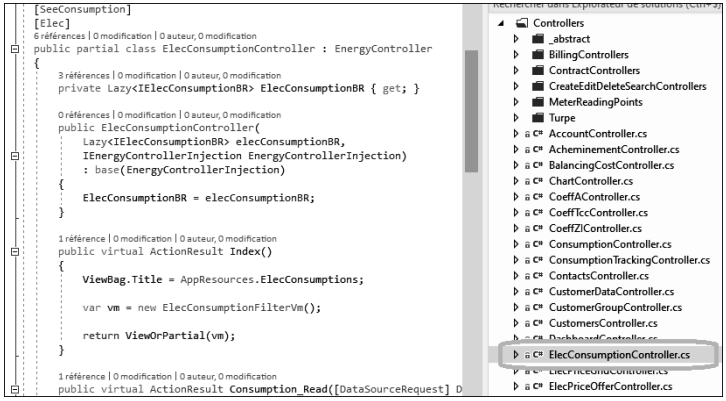
1. Code versioning

En développement informatique, le code d'un projet conséquent se retrouve éclaté en milliers de fichiers. Parmi cette quantité de fichiers, il est tout à fait possible que plusieurs développeurs soient amenés à travailler sur un même fichier à un moment donné.

Chaque développeur tape son code en local sur sa machine ; quand il a fini, il enregistre son code sur un tronc commun situé sur un serveur distant.

Or, si deux développeurs ont travaillé sur un même code d'un même fichier, au moment d'enregistrer sur le tronc commun, il risque d'y avoir des conflits. Quel code faut-il alors conserver ? Il n'y a pas de système de verrouillage sur les fichiers, de sorte que quand j'utilise le fichier, les autres ne peuvent le modifier ! Un bon logiciel de gestion de code versioning sait parfaitement traiter ce genre de conflit. C'est la base pour travailler en équipe sans perdre de temps à attendre que les fichiers soient disponibles. La référence dans le domaine est de loin l'application **GitHub**. Ce type d'application permet d'effectuer toutes les actions possibles dans le quotidien du développeur : récupérer tout le code du projet à partir du tronc commun, afin de le copier en local.

C'est ce qu'on appelle la création de l'espace de travail du développeur où il pourra ensuite ajouter, supprimer ou modifier les fichiers du projet. Une fois son travail terminé, il pourra expédier le résultat sur le tronc commun. Cette action nécessite quelques vérifications avant de valider les modifications. Ce type d'application permet aussi d'avoir toute sorte d'information sur les fichiers : par exemple, qui a saisi tel code, quand, pourquoi, qui a supprimé, qui a modifié, qui a fait le café... non, oublions ce dernier, il ne faut pas exagérer !



Vous avez ci-dessus une photo de mon écran, je suis en train de taper du code sur le fichier ElecConsumptionController.cs.

Un projet peut contenir des milliers de fichiers de ce type et l'ensemble des lignes de code du projet peut largement dépasser le million. Dans le jargon informatique, le tronc commun s'appelle la **branche Main** ou branche principale, ou encore dépôt distant ! Cette branche contient le code du projet dans sa totalité à un instant t et dans un état pur, c'est-à-dire sans aberration de code. Le code de la branche Main respecte toutes les règles de codage définies par l'équipe ; et bien sûr, ce code doit absolument être compilable. La compilation consiste à traduire un code comme celui ci-dessus (lisible par un humain) en un autre code qui, cette fois-ci, ne contiendra que des 0 et des 1 ! C'est le fameux code binaire, seul code compréhensible par les microprocesseurs des ordinateurs. On ne passe pas tout de suite en code binaire quand on développe sous des langages comme C# ou Java, mais nous n'allons pas entrer dans les détails. **Un développeur qui fusionne dans la branche Main un code non compilable commet un sacrilège !**

Avant de vouloir fusionner son code sur la branche Main, le développeur doit tout d'abord le tester en local, puis vérifier que son code compile et que les tests unitaires sont tous validés. Notons une étape très importante : avant de démarrer un nouveau code, il doit absolument se synchroniser avec la branche Main pour récupérer les dernières modifications validées sur cette branche par ses collègues. Ainsi, il pourra traiter plus facilement d'éventuels conflits avant de faire valider son propre code. Finalement, il doit s'assurer que l'intégration continue n'est pas bloquée ! On en reparlera, mais sachez qu'on ne doit sous aucun prétexte forcer un code sur la branche Main si l'intégration continue est au rouge, c'est-à-dire si un problème a été détecté. Le but de l'intégration continue est de vérifier que les nouveaux codes intégrés dans la branche Main ne cassent pas l'harmonie qui règne sur la branche. Autrement dit, il s'agit de s'assurer que le nouveau code introduit ne va pas provoquer des erreurs en tout genre.

Dans le jargon, on ne parle pas de fusion mais on dit qu'on va **merger** son code ! Il n'y a rien d'ésotérique ici, *to merge* en anglais signifie fusionner ! Chaque développeur tape donc son code de son côté, puis une fois fini, il le merge sur la branche principale. On peut aussi dire qu'il intègre son code dans la branche principale. Cette dernière est par conséquent une branche d'intégration de tout le code produit ; autrement dit, notre potentielle application à un instant t. L'action de merger sur la branche Main n'est jamais directe, il faut prendre des précautions car, comme nous venons de le dire, la violation de l'intégrité de la branche Main peut être fatale. On évite au maximum que tout nouveau code vienne provoquer des erreurs sur cette branche. Chaque développeur qui veut fusionner son code dans la branche Main doit faire une **Pull Request (PR)**, également appelée **Merge Request (MR)**. Il s'agit d'une demande de fusion de son code local vers la branche Main. Sans entrer dans les détails, c'est un humain qui fera ou pas la validation du code. Dans les équipes agiles, une personne de la QA (Qualité de service) peut tenir ce rôle mais cela peut aussi être un architecte ou même un ou plusieurs développeurs dédiés ; ou encore, pourquoi pas, l'équipe Dev elle-même ! Ainsi, dans ce dernier cas, un code développé par untel sera vérifié par tel autre collègue pour éviter bien sûr que celui qui a codé ne valide sa propre PR ! Si on refuse la Pull Request, le développeur devra faire les correctifs et refaire une PR. Une fois celle-ci acceptée, le code est intégré dans la branche Main et l'intégration continue se met alors en marche. On en reparlera sous peu.

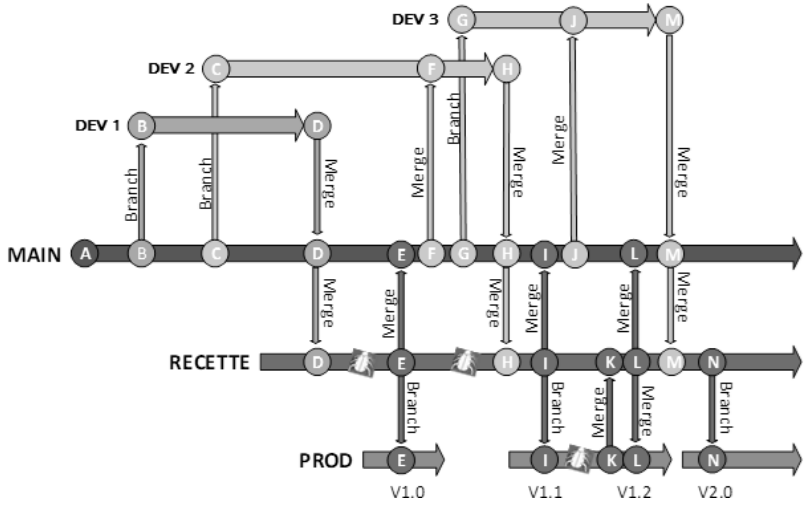
Quand un nouveau développeur intègre une équipe, avant de pouvoir coder la moindre ligne, il doit se connecter sur le serveur contenant la branche Main pour extraire tout le code du projet sur son ordinateur. Pour cela, il crée une nouvelle branche sur ce dernier, à partir de la branche Main. Cette nouvelle branche créée en local est donc une copie de la branche Main. Notre développeur code ensuite uniquement en local sur sa branche sa partie de code, puis fait sa PR une fois son code finalisé.

Dans ce domaine, la règle d'or est de ne pas coder des lignes et des lignes avant de faire une Pull Request. On code une tâche et on fait sa PR ; on avance tâche par tâche. Cela facilite la gestion des conflits au cas où plusieurs développeurs auraient travaillé sur les mêmes fichiers, et cette rigueur facilite aussi le traitement des Pull Requests. Comme nous l'avons rapidement évoqué, un autre bon réflexe est de toujours récupérer la dernière version de la branche Main avant de démarrer un nouveau code, car d'autres développeurs auront certainement fait des merges entre-temps ; ce réflexe vous évitera donc de vous retrouver avec trop d'écart entre votre version en local et la version de la branche Main. L'idéal est de récupérer le code de la branche Main dès que cette dernière a été mise à jour lors d'une PR validée.

Pour finir, le merge (la fusion entre le travail du développeur en local et le code complet central) peut se faire dans les deux sens : soit on merge notre travail local vers la branche Main, soit on merge de la branche Main vers notre branche locale, ce qui revient dans ce dernier cas à récupérer la dernière version de la branche Main. Le merge n'est donc pas une copie qui écrase l'existant mais une copie qui fusionne le contenu de deux branches.

Nous allons nous arrêter là en ce qui concerne les branches, mais vous avez compris l'essentiel du travail d'un développeur : créer des branches pour coder et faire des PR pour valider son travail ; c'est ainsi qu'au gré des merges la taille du code du projet augmente dans la branche Main.

Voici ci-dessous l'exemple d'une stratégie de branche adoptée par une équipe :



L'équipe a créé sa branche Main en A. Un développeur crée une branche locale en B afin de récupérer le code de la branche Main pour le copier dans sa nouvelle branche locale ; on dit qu'il a tiré une branche. Il peut alors commencer son code en local. Quand il a fini son travail, il fait un merge en D ; son code local est désormais fusionné sur la branche Main. Il fait aussi un merge sur la branche Recette. C'est sur cette branche Recette qu'on corrigera les bugs. Il est interdit de développer directement sur la branche Main ! Encore moins sur la branche Prod. Pendant que notre développeur travaillait sur son code, un autre développeur a tiré à son tour une branche locale à l'étape C.

Quand un bug est détecté sur la branche Recette, on le corrige en développant directement sur cette branche. Une fois le bug corrigé et la recette validée, on tire une branche Prod puis on merge sur la branche Main pour y reporter les corrections, car si le bug existe en Recette, c'est qu'il existe aussi en Main ! Sur notre exemple, le premier bug détecté après le merge en D est corrigé en E et on reporte les corrections sur la branche Main ; et comme la première mise en production était attendue, on crée une nouvelle branche Prod. Quand la branche Main est mise à jour, il est toujours bien pour un développeur de récupérer rapidement en local les modifications en faisant un merge. C'est par exemple ce que font le deuxième et le troisième développeur respectivement en F et J.

Mais notre troisième développeur n'a pas récupéré le correctif du bug Prod fait en L. Ce n'est pas grave, quand il fusionnera son code sur la branche Main (en M), il verra s'il est en conflit avec les correctifs précédemment apportés sur la branche Main. Toute correction faite sur la branche Recette doit être répercutée sur la branche Main, et tôt ou tard sur la branche Prod. Tout nouveau code sur la branche Main doit être répercuté sur la branche Recette pour être testé avant de répercuter le code sur la branche Prod. Finalement, à partir de n'importe quelle branche non locale, on est capable, à un instant t, de créer une version du projet sous condition bien sûr que les branches Main et Recette soient parfaitement synchronisées.

À présent, dans les équipes agiles, c'est en fin de sprint qu'on bascule le code en Recette pour la Sprint Review ; or, au bout de n sprints, les choses se compliquent : non seulement il faut continuer les Sprint Reviews, mais entretemps il faut aussi, durant le sprint, corriger les éventuels bugs détectés en Recette. Si une livraison en production a été faite depuis, on peut aussi y trouver des bugs qu'il faudra rapidement corriger. Dans notre exemple de branche, pour corriger le bug trouvé en Prod, comme on ne peut développer directement sur cette branche, il faut récupérer en K le code en Prod pour le fusionner en Recette où on corrigera l'erreur en L. Une fois corrigé et validé, on répercute le correctif en Prod puis en Main.

Chaque bug remonté devient, dans le sprint, une US technique à prendre en compte. Les bugs augmentent donc la vélocité de l'équipe ! Un bon KPI serait de connaître la part des bugs dans une vélocité.

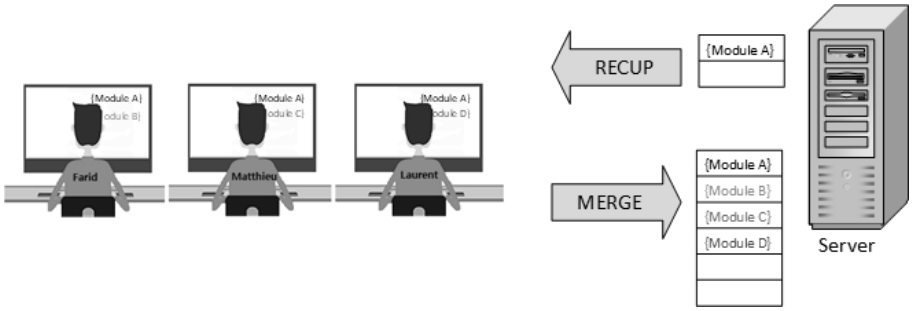
Bien sûr, il existe des cas de figure où un bug détecté en production est d'une telle gravité qu'il faut absolument le corriger dans les minutes qui suivent. On n'attend donc pas la fin du sprint, il faut lancer une opération Commando pour corriger rapidement en Recette et valider pour un merge en Prod ! Attention, il serait imprudent de corriger le bug directement sur la branche Prod. Je me rappelle par exemple la cas d'une modification urgente qu'il fallait effectuer sur un paramètre d'un fichier web.config d'un site e-commerce. Il n'y avait rien de compliqué a priori, pourtant la modification directe en production a fait tomber le site ! D'un coup, dix millions de clients ne pouvaient plus y accéder ! En moins de cinq minutes, l'erreur a heureusement été corrigée. Le fautif s'était trompé de fichier, pensant travailler sur le fichier Prod, alors qu'il s'agissait du fichier web.config de la recette qu'il avait poussé en production !

Cette expérience a été une grande leçon de management agile : aucun membre de l'équipe n'a perdu son sang-froid, aucun dirigeant n'a invectivé le coupable ; tous étaient concentrés sur la résolution du problème. En quelques minutes, l'équipe dans sa globalité a rétabli la situation et s'est félicitée de sa parfaite collaboration.

Comme nous le disions, quand un développeur a fini son code, il le valide. On vérifie ce code et, si tout est OK, le code merge sur la branche Main, puis Recette, puis Prod. L'intégration continue permet d'automatiser toutes les étapes qui vont suivre la validation d'une Pull Request. Il reste encore beaucoup de choses à contrôler avant d'installer l'application pour les utilisateurs.

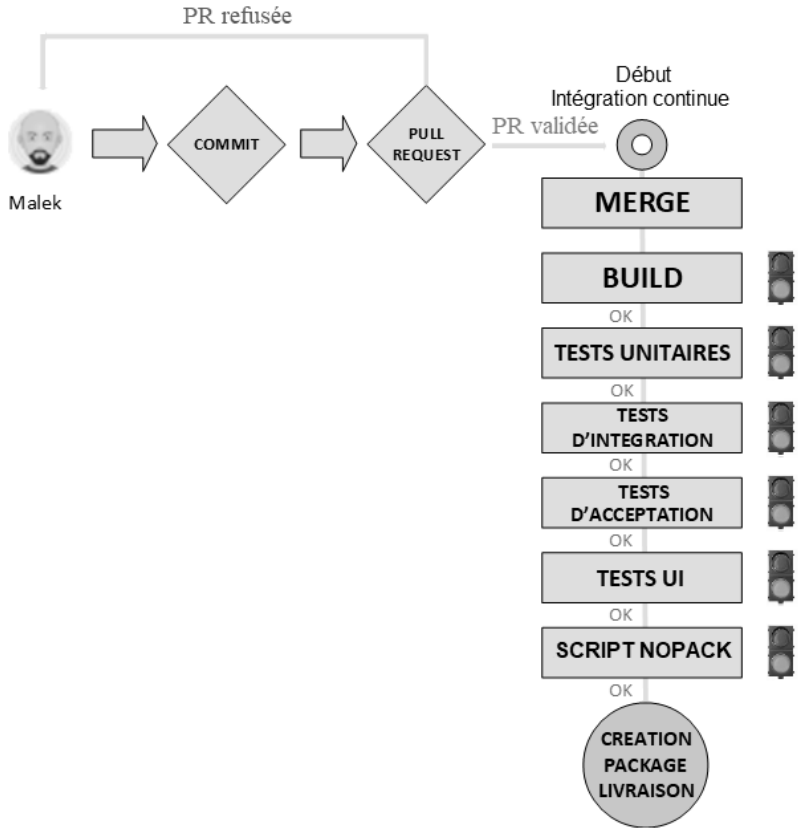
2. Intégration continue

L'intégration continue (CI) a pour but de traiter chaque merge effectué sur la branche Main ; on intègre son code à celui de ses collègues.



Pour rappel, lorsqu'un développeur a fini son code, il effectue une Pull Request pour soumettre son travail. Lors de la validation de cette PR, l'intégration continue démarre. En fait, aujourd'hui, la CI fait bien plus qu'une simple intégration du code (merge) dans la branche Main. Après le merge, on y trouve désormais un ensemble de filtres posés les uns derrière les autres qui vont automatiquement s'exécuter l'un après l'autre dès qu'un code est fusionné dans la branche Main. L'ensemble de ces filtres disposés en file s'appelle un **pipeline** ou **workflow applicatif**.

Ces filtres sont donc des éléments exécutables (une application, un script, un job, un bout de code, un service, un web service, un microservice...) qui vont s'exécuter de manière séquentielle dans l'ordre dans lequel ils se trouvent dans le pipeline.



Le but de la CI est de s'assurer que le merge d'un nouveau code sur la branche Main n'a pas entraîné d'erreur, afin de nous éviter de construire un package de livraison rempli de bugs en tout genre. Aujourd'hui, comme nous l'avons évoqué, la CI est alimentée par d'autres types de contrôle ; par abus de langage, on appelle toute la chaîne de contrôle une intégration continue.

Après le merge, la CI compile (fait un **Build**) le code, puis effectue beaucoup de tests ! La puissance du pipeline, réside dans la possibilité d'ajouter autant de filtres qu'on le souhaite. Dans notre schéma ci-dessus, on a ajouté un script « NoPack » qui pourrait par exemple envoyer un e-mail en cas d'erreur du pipeline. Vous pourriez écrire des scripts de test de performance, par exemple, et les intégrer dans le pipeline.

La moindre erreur dans les filtres arrête immédiatement le pipeline ; il sera donc impossible pour ce dernier de construire le package de livraison. Ce package est l'ensemble des fichiers constituant notre projet. Il suffit de les installer sur un serveur pour que notre projet se transforme en application disponible pour les utilisateurs. Ce package est composé du code binaire de notre projet et de différents fichiers indispensables à la bonne exécution de ce code : fichiers de dépendance, de configuration et de base de données, etc.

Dès qu'un filtre du pipeline détecte une erreur, le pipeline arrête sa course ; à partir de cet instant, chaque membre de l'équipe interrompt sa tâche pour tenter de résoudre le problème. Lors des tests fonctionnels (tests d'acceptation), imaginez que le pipeline détecte des erreurs mais que l'équipe poursuive malgré tout son travail et que chaque membre continue à faire des merges sur une branche Main infectée… Les bugs se cumuleraient et on passerait plus de temps ensuite à les résoudre ! L'intégration continue est un garde-fou, disons même un ensemble de garde-fous. Chaque filtre du pipeline est une tour de contrôle ne laissant passer aucune erreur traitée par le filtre. Prenons par exemple un cas où l'on souhaite tester la couverture de code et interdire que le pipeline ne fabrique le package d'installation dans l'hypothèse où les tests unitaires n'auraient pas été couverts à 80 %. Il faut alors ajouter une application comme **SonarQube** pour configurer un seuil de couverture de code à 80 %. Du coup, si les tests unitaires ont été négligés par l'équipe, avec une telle tour de garde, il n'y a aucune chance que le pipeline ne construise le package de livraison car on n'aura pas atteint nos 80 % de couverture de code !