Cours - Projet ECOM

1. Principe du 2ème Audit
2. Divers Types de Tests
3. Rappels sur les Workflows Git
4. Revues de Code
5. Audits de Projet

Principe du 2ème Audit

Principe du 2ème Audit

Cet audit aura lieu le 5 novembre au matin.
Déroulement :

1. Un groupe entre et tire au sort le nom d'un autre groupe.
2. Le projet du 2ème groupe est audité par le premier.
3. Les remarques seront consignées.
4. Le groupe en question devra dans la journée, répondre, justifier ou rectifier suite à chaque remarque.

Votre note sera basée sur l'évaluation de votre projet par un autre groupe, mais sur la justesse de votre évaluation. Les évaluations de projets se feront à voix haute, en direct avec un enseignant. **Les remarques seront reportées sur le GitLab du projet évalué**.

Focus sur les ressources dans la forge (GitLab) et sur comment est fait le projet (code, documentation, tests, outils utilisés, etc). Le fonctionnel est hors-périmètre : pas d'évaluation des plate-formes déployées (production, etc).

Divers Types de Tests

Principe de Base

1. **Pré-conditions** : on garantit un état particulier de notre système.
2. On exécute une séquence d'actions impliquant notre système.
3. **Post-conditions** : on vérifie l'état attendu de notre système.

Plusieurs types de tests (automatisables) :

- **Unitaires** : très proches du code. Faciles à écrire.
- **Intégration** : plusieurs parties du système sont testées ensemble.
- **Tests de bout en bout** : tests qui impliquent des scénarios joués avec l'ensemble du système. Cela peut aller jusqu'à des conditions réalistes, proches de la production.
- **Performances / charge** : vérifier le comportement du système face à des charges diverses. Utilisés pour vérifier le bon dimmensionnement de l'architecture, les zones de tension, et voir le comportement du système en cas de forte charge. A réaliser sur un environnement similaire à la production. Généralement réalisés dans les dernières étapes.
- **Tests de sécurité** : vérifier des éléments liés à la sécurisation (ports exposés, protocoles utilisés, etc). Souvent, les dossiers d'architecture et de conception font l'objet d'un audit / vérification de sécurité.

Tests Unitaires

Ces tests servent à vérifier la logique de votre code.
C'est le premier niveau de validation d'un projet. Ils sont rapides et peuvent être lancés sur la machine du développeur.

Différents *frameworks* :

- Karma, Jasmine, Chai... pour JS.
- JUnit, TestNG pour Java

```
@Test
public void testWriteString() {

    try {
        String msg = "Test\n2";
        File tmpFile = File.createTempFile();
        Utils.writeString( msg, tmpFile );

        String readMsg = ...
        Assert.assertEquals( msg, readMsg );

    } finally {
        Assert.assertTrue( tmpFile.delete()),
    }
}
```

Règles JUnit

JUnit propose la notion de règle.
Ce sont des attributs de classe annotés, qui fournissent des facilités et allègent la quantité de code à écrire.

```
@Rule
public TemporaryFolder folder = new TemporaryFolder();

@Test
public void testWriteString() throws Exception {

    String msg = "Test\n2";
    File tmpFile = folder.newFile();
    Utils.writeString( msg, tmpFile );

    String readMsg = ...
    Assert.assertEquals( msg, readMsg );
    // Le fichier temporaire sera supprimé automatiquement par la règle
    // (pas besoin de try / catch ici)
}
```

Il existe plusieurs types de règles JUnit.
En général, essayer de voir s'il en existe une selon vos besoins.

Bouchonnage

Bouchonnner certaines classes permet d'écrire des tests unitaires sur des classes qui traînent de nombreuses dépendances ou relations.

En JS, cela peut être solutionné en passant des fonctions en tant que paramètres. En Java, cela peut être résolu par les frameworks d'injection de dépendances (DI) comme dans Spring, ou bien en utilisant des librairies comme Mockito, EasyMock et JMockit.

Exemple avec Mockito :

```
// Définir les bouchons
ServletRequest req = Mockito.mock( ServletRequest.class );
ServletResponse resp = Mockito.mock( ServletResponse.class );
FilterChain chain = Mockito.mock( FilterChain.class );

// Exécuter notre action avec les bouchons en paramètres
this.filter.doFilter( req, resp, chain );

// Vérifier les actions effectuées sur nos bouchons
Mockito.verifyZeroInteractions( req );
Mockito.verifyZeroInteractions( resp );
Mockito.verify( chain, Mockito.only()).doFilter( req, resp );
```

Tests Unitaires en Javascript

Contrairement à Java, le monde Javascript combine plusieurs solutions : une pour lancer les tests, d'autres pour l'écriture des assertions et structurer les tests.

```
// Déclarer le test (Jasmine)
// Mettre un message significatif pour décrire l'objectif du test
it('should filter and be case insensitive', function() {
  var array = [{ instance: { path: '/vm' }}, { instance: { path: '/vm/server' }}];

  // Exemple d'assertions (chai)
  expect(filter(array, 'vm')).to.have.length(2);
  expect(filter(array, 'VM')).to.have.length(2);
  expect(filter(array, 'server')).to.have.length(1);
  expect(filter(array, 'SERVer')).to.have.length(1);
  expect(filter(array, 'invalid')).to.have.length(0);
});
```

Quelles remarques peut-on faire sur cet exemple ?
Le test pourrait être plus complet : au-delà de la taille du résultat, on devrait tester le résultat exact de la commande.

Dépendances VS. Dépendances de Tests

Certaines librairies sont indispensables au fonctionnement du projet.
D'autres ne sont nécessaires que pour les tests. Elles sont donc déclarées différemment dans les outils.

JS (package.json) => propriété "devDependencies"

```
"dependencies": {
    ...
},
"devDependencies": {
    "glob": "^7.1.0"
}
```

Java (pom.xml) => attribut **scope**

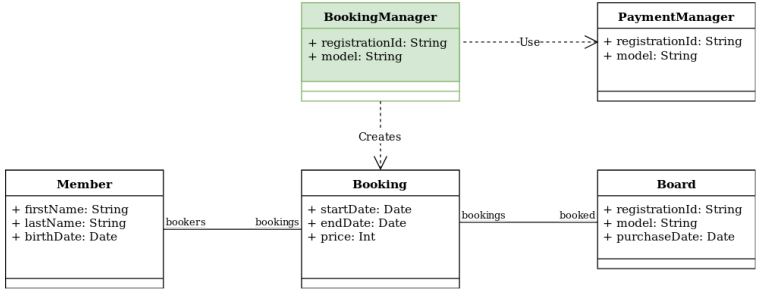
```
<dependency>
    <groupId>org.apache</groupId>
    <artifactId>apache-commons</artifactId>
    <scope>test</scope>    <!-- compile par défaut -->
</dependency>
```

Cela évite permet de faire le tri entre les dépendances.
Distinction indispensable lors de l'empaquetage.

Tests d'Intégration

Un test unitaire ne devrait évaluer qu'un seul élément.
Autrement dit, les tests unitaires évaluent les éléments indépendamment les uns des autres. Exemples : une classe Java, un composant Angular, etc.

Sur le diagramme ci-dessous, un test unitaire de BookingManager nécessiterait de bouchonner PaymentManager, et idéalement, Booking.



Utiliser les véritables composants au lieu de bouchons relève du test d'intégration. Dans les faits, on distingue rarement tests d'intégration et tests unitaires : les frameworks supportent l'un et l'autre.

Tests d'Intégration

Les tests d'intégration peuvent aussi impliquer des briques logicielles supplémentaires, ce que ne fait pas un test unitaire. Exemple ici en Java, où l'on lance une vraie base de données depuis un test.

```
// Règle pour lancer une BD PostgreSQL éphémère
@Rule
public SingleInstancePostgresRule pg = EmbeddedPostgresRules.singleInstance();

@Before
public void before() throws Exception {

    // Récupération de la source de données
    this.ds = this.pg.getEmbeddedPostgres().getPostgresDatabase();

    // Création des tables (utilitaire maison)
    TestUtilities.initializeDatabase( this.ds );
}
```

Autres cas possibles :

- Test d'API REST sur un vrai serveur web : Jetty, GrizzlyServer...
- Test avec des conteneurs Docker avec TestContainers

Faisable pour tout langage, même si un peu plus compliqué en Javascript.

Tests de Charge - 1/3

Deux outils de référence :

- JMeter
- Gatling

Gatling est plus récent.
Pour rappel, ces tests aident à vérifier le dimmesionnement de son infrastructure et de son déploiement. Cela permet aussi de vérifier le bon fonctionnement du projet sous de fortes charges.

Exemple de scénario Gatling :

```
class PostEndingSessions extends Simulation {

  // Definition d'un agent HTTP
  val httpProtocol = http
    .baseUrl("http://localhost:9090/soda-stats-ws")
    .doNotTrackHeader("1")
    .acceptLanguageHeader("fr-FR,en;q=0.5")
    .acceptEncodingHeader("gzip, deflate")
    .userAgentHeader("Mozilla/5.0 (Windows NT 5.1; rv:31.0) Gecko/20100101 Firefox/31.0")
```

Tests de Charge - 2/3

```
// Définition d'un scénario simple
val scnStop = scenario("Post / end")
  .exec(
    http("End session")
    .post("/stop")
    .body(RawFileBody("request.stop.json")).asJson
    .check(
      status.is(204)
    )
  )
  .pause(1)
```

Ici, on envoie une requête statique sur un *end-point* REST, et on s'assure que le code de retour du traitement est 204.

Il est possible de variabiliser un scénario.
Voire d'en définir plusieurs, qui peuvent tourner simultanément.

Tests de Charge - 3/3

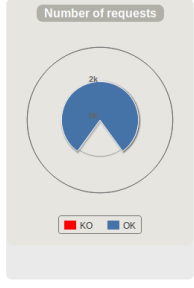
```
// Conditions de l'exécution.
setUp(
  scnStop.inject(
    nothingFor(4 seconds),
    atOnceUsers(10),
    rampUsers(10) during (5 seconds),
    constantUsersPerSec(10) during (15 seconds),
    constantUsersPerSec(10) during (15 seconds) randomized,
    rampUsersPerSec(10) to 20 during (1 minute),
    nothingFor(2 seconds),
    rampUsersPerSec(10) to 20 during (1 minute) randomized
  )
  .protocols(httpProtocol)
)
.assertions(
  global.responseTime.max.lt(200),
  global.successfulRequests.percent.is(100)
)
```

Exécutable automatiquement, y compris dans Maven / Gradle.
Pertinence ? Génération de rapport HTML.

Il faut parfois des machines dédiées pour exécuter ces tests (saturation des ressources).

Exemple de Rapport d'exécution de Gatling

> Global Information



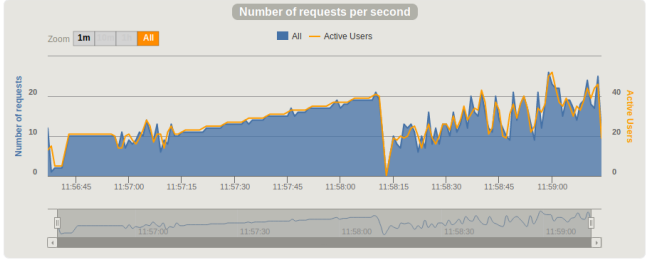
ASSERTIONS

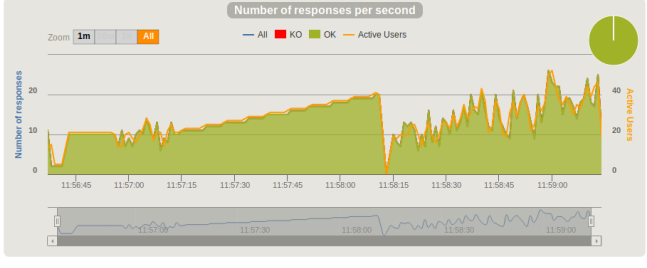
Assertion	Status
Global: max of response time is less than 200.0	OK
Global: percentage of successful events is 100.0	OK

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	2114	2114	0	0%	13.38	2	6	7	10	14	58	6	4
Start session	2114	2114	0	0%	13.38	2	6	7	10	14	58	6	4

Exemple de Rapport d'exécution de Gatling





Tests de Bout en Bout

Aussi appelés « *tests end-to-end* » ou « e2e ».
Ils consistent à jouer des scénarios qui impliquent tous les composants logiciels d'un projet, dans des conditions aussi proches que possibles de la production.

Contrairement aux tests de charge, où l'on se concentre sur la tenue du système, les tests de bout en bout se focalisent sur le fonctionnel. On parle de tests de validation, en conditions réelles.

Pour les tests automatisés, en open source, on voit surtout Selenium sur le marché. Pour JS, Cypress est la solution la plus à jour (avec Protractor).

Autrement, on voit encore très souvent des personnes jouer des procédures de tests à la main, notamment pour les interfaces graphiques. D'où l'importance d'**avoir des plans de tests qui documentent ces procédures**.

Tests de Sécurité

Ces tests sont difficiles.
On parle plutôt de sécurisation et de détection des failles. Les tests en eux-mêmes sont plus souvent le fait d'audits, par des spécialistes.

Pour un projet, il faut plutôt miser sur :

1. Définir une politique de sécurité (gestion des mots de passe, rôles, qui contrôle quoi et comment). C'est de l'organisation plus que du code. Bien poser les choses par écrit et que chacun en soit informé.
2. Mettre en place des vérifications automatisées au niveau du code (des outils comme Sonar ou SpotBugs peuvent repérer certaines failles).
3. Segmenter les zones réseaux dans l'architecture (zone données, zone applicative, DMZ). Limiter les ports et les moyens d'accès.
4. Sécuriser les flux dans l'architecture (crypter ce qu'il faut, **là où il faut** - ex : HTTPS). Attention, crypter a un coût. Gare aux performances.
5. Mettre en place des vérifications automatisées sur les binaires.
   Exemple : Clair Scanner pour Docker, Harbor (toujours pour Docker), OWASP's Depdency Check, Nexus, etc.

Industrialisation des Développements

1. Jouer des tests a un coût.
2. Ecrire des tests automatisés implique de les maintenir.
3. Et ne pas avoir de tests automatisés implique de tester à la main.
4. **Ceux qui ne testent pas échouent souvent. Et très vite.**

Il faut donc choisir ses combats et placer le curseur au bon endroit, en fonction du projet.

Tout cela doit s'inscrire dans une démarche de long terme :

- Documenter ce que l'on veut avoir.
- Mettre en place les procédures, outils et moyens pour atteindre l'objectif…
- … et maintenir l'ensemble dans le temps, voire augmenter les ambitions.

Le responsable des tests, ou de la sécurité, ou de la qualité, n'est pas celui qui fait tout, mais qui édicte ce qui doit être fait et comment y arriver.

Les tests limitent les risques de régression et **assurent une continuité au projet**, même en cas de changement d'équipe ultérieur. Ils doivent être **consignés dans un document**, et/ou **automatisés**, afin de les dérouler de **manière continue** et ainsi de **gagner du temps**.

JHipster

A partir d'un modèle, JHipster génère non seulement du code applicatif, mais aussi des tests automatisés, tant pour le code Java que pour Javascript. Il existe aussi des générateurs, comme pour Gatling.

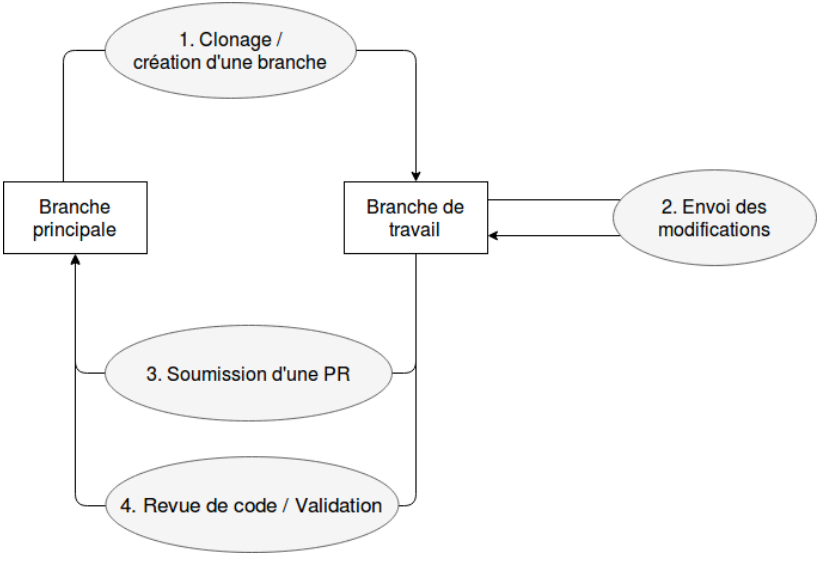
Il est attendu de vous que vous complétiez ces tests !
Vous allez rajouter de la logique (des algorithmes) dans votre code. Il faut les tester. A minima, il est attendu que vous démontriez votre capacité à en écrire, en particulier si la logique fonctionnelle est complexe.

Cela fait partie des éléments qui seront contrôlés lors du deuxième audit.

Rappels sur les Workflows Git

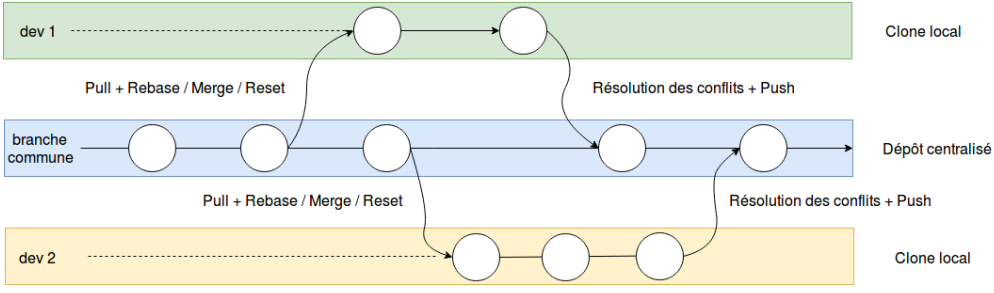
Pull Requests (ou Merge Requests)

Mécanisme introduit par GitHub et repris par GitLab.
Cela facilite les contributions externes et encourage les revues de code.



Utilisation "centralisée"

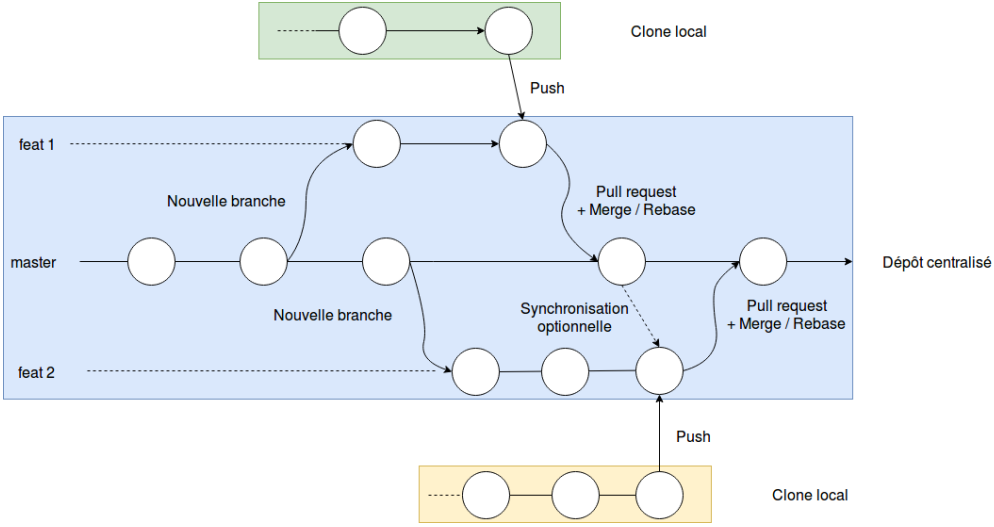
Un seul dépôt, souvent une seule branche.
Ce que les gens faisaient avec CVS et SVN.



On évite !
Admissible quand on est seul et avec des tâches « courtes ».

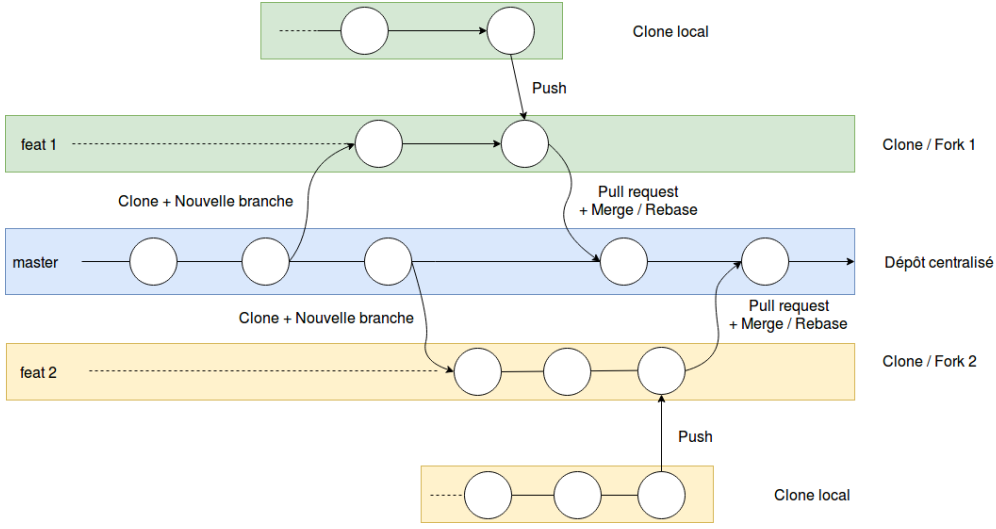
Feature-Branching

On créé une branche par fonctionnalité.
Réintégration dans la branche principale via une *pull request*.



Forks

Un *fork* est une copie distante d'un dépôt Git.
Intérêt : gestion des droits, bifurcation dans la feuille de route, etc.

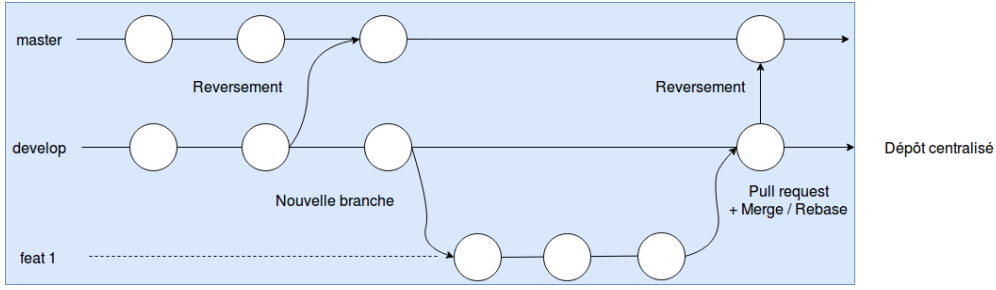


Git Flow - 1/3

Du *feature branching* avec des conventions de nommage sur les branches.

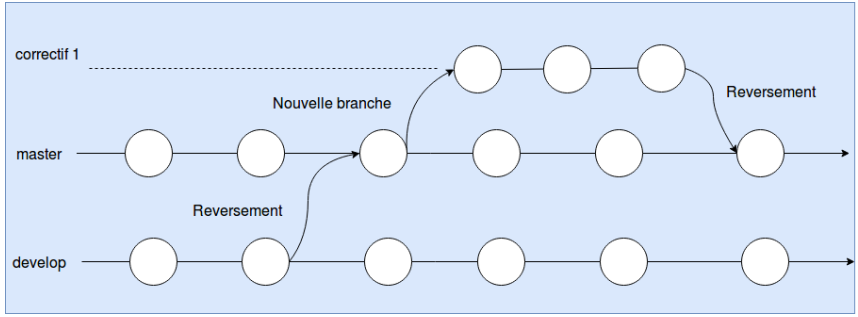
Le **master** correspond à la production.
develop correspond à la dernière version de développement.

Les évolutions se font sur d'autres branches.
feat- désigne l'ajout d'une nouvelle fonctionnalité.



Git Flow - 2/3

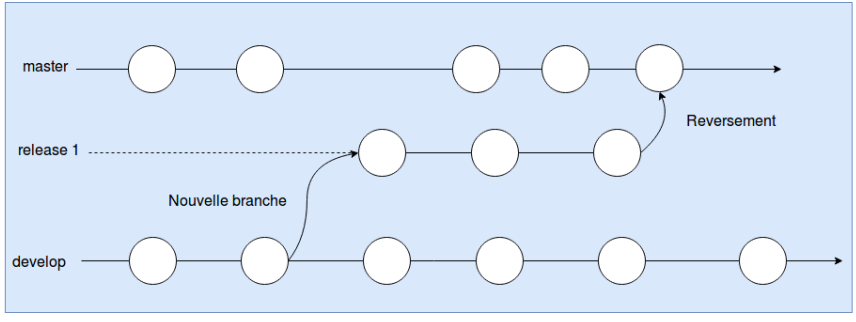
Les correctifs de bugs sont gérés dans des branches préfixées par *fix-*.
Ils sont rapatriés soit dans la branche **develop**, soit dans le **master**.



Le passage d'une branche à l'autre se fait via *pull request*.

Git Flow - 3/3

Une mise en production correspond à rapatrier la branche **develop** dans **master**. Par définition, **master** décrit l'état du code en production.



Le passage d'une branche à l'autre se fait via *pull request*.

Relation avec CI / CD

Intégration continue : exécution des tests.
Livraison continue : intégration continue et mise à disposition des livrables. Exemple : envoi d'une image Docker sur un registre Docker, envoi sur dépôt Maven / NPM *staging*…

Déploiement continu : livraison continue et déploiement des éléments.
Attention ! Tout le monde ne fait pas du déploiement continu ! Et parfois, on s'arrête sur de la pré-production. Le passage en production est validé ou lancé manuellement (**impact possible sur la vraie vie**).

Il faut choisir son combat.

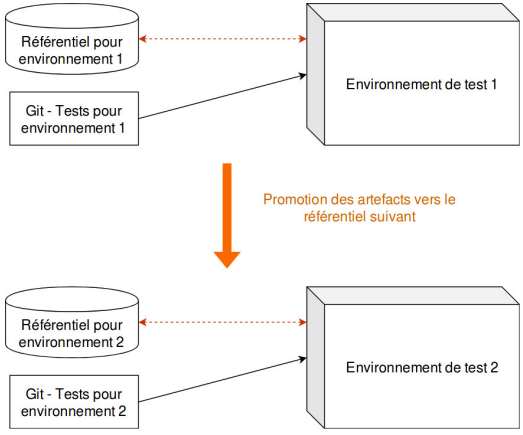
- Intégration continue sur toutes les branches et toutes les PR.
- Livraison continue si commit direct sur des branches précises (ex : **master**).
- Déploiement continu si branche précise (ex : master). Et si souhaité.

Intégrer dans les pipelines la gestion des environnements / secrets.
Impératif pour le déploiement continu.

Pipelines et Promotions

Approche classique, automatisable entièrement ou partiellement.

1. On teste le code et produit les livrables **une fois** (ex : image Docker).
2. On les publie ensuite dans un dépôt (ex : registre Docker).
3. On les déploie et utilise dans un environnement de test (*staging*).
4. Ensuite, on peut les **promouvoir** dans un nouvel environnement.



Les livrables sont envoyés d'un dépôt N vers un dépôt N+1. C'est le principe de la promotion.

L'environnement peut ensuite avoir ses spécificités (performances, sécurité, etc).

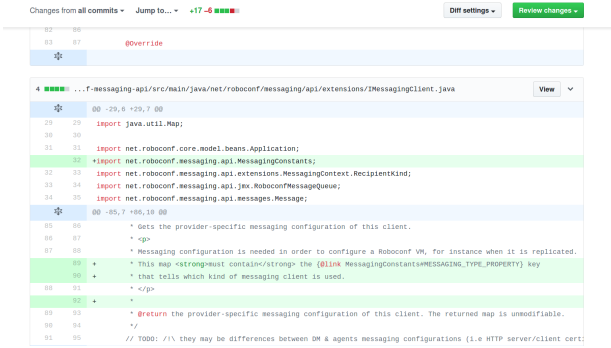
Tout au bout, il y a l'environnement de production (« la vraie vie »).

Revues de Code

Qu'est-ce qu'une revue de code ?

C'est le processus de relecture et de validation du code.
Peut être appliqué aléatoirement. Mais généralement, on préfère le faire sur chaque évolution du code. C'est plus simple.

Des plate-formes comme GitHub (et ensuite GitLab) ont révolutionné et démocratisé cette pratique, grâce à des portails web riches.



Finalité : **faire ce que l'on a à faire, avec le minimum de code, tout en le gardant compréhensible et maintenable**.

Objectif 1 : efficacité du code

```
// En pseudo-langage C
for( int i=0; i<myCollection.size(); i++ ) {
    sb.append( myCollection.get( i ));
    if( i != myCollection.size() - 1 )
        sb.append( ", " )
}
```

```
// Meilleure approche : les itérateurs
Iterator<String> it = myCollection.iterator();
while( it.hasNext()) {
    sb.append( it.next());
    if( it.hasNext())
        sb.append( ", " )
}
```

```
// Pourquoi un while ? Utiliser un for.
for( Iterator<String> it = myCollection.iterator(); it.hasNext(); ) {
    sb.append( it.next());
    if( it.hasNext())
        sb.append( ", " )
}
```

```
// Et en Java 8+
String.join( ", ", myCollection );
```

Objectif 2 : lisibilité du code

```
// Opérateur ternaire à la chaîne
return t % 2 == 0 ? 2 : t %3 ? 3 : T % 5 == 0 ? 5 : 1;
```

```
// Opter pour un switch ou des if
if( t % 2 == 0 )
    return 2;
else if( t % 3 == 0 )
    return 3
else if( t % 5 == 0 )
    return 5;

return 0;
```

```
// Un seul return, plus facile pour debogage
int res = 0;
if( t % 2 == 0 )
    res = 2;
else if( t % 3 == 0 )
    res = 3
else if( t % 5 == 0 )
    res = 5;

return res;
```

Objectif 3 : documentation du code

Commentaire mal agencé, qui n'a aucun intérêt.

```
// Because this is not
    // possible
```

Commenter son code est important.
Mais ne documenter que ce qui mérite de l'être, et le faire avec soin alors.

```
// It is not possible to rely on the distributed map.
// To keep on working in case of network failures, we use a local lock.
```

Certaines personnes ou certains projets ont adopté une approche rigoriste sur les commentaires, à savoir aucun commentaire. Ils exigent en compensation des noms signficatifs pour les méthodes, variables, etc. **Il reste conseillé de commenter les éléments importants du code** (explications, aération).

Penser à la Javadoc et JSDoc !

Ne pas oublier qu'avec Javascript, il y a des mécaniques de minification et d'enlaidissement (*uglify*) du code. Les commentaires sont retirés lors de ces phases, mais restent bien sûr accessibles dans la forge.

Faire preuve de Savoir-Vivre

Faire une revue de code revient à juger le travail des autres. Il faut donc…

1. Evaluer le bien-fondé de ses remarques. **Réfléchir avant d'écrire**.
2. Faire preuve de tact, de politesse et de respect dans ses échanges.
3. Savoir accepter que l'on n'a pas toujours raison. Cela doit reste une discussion, et ne pas devenir une mise en accusation.
4. Ne pas hésiter à définir des règles pour un projet (documentation).
5. Et les faire respecter à l'aide d'outils. Il suffit alors d'indiquer que les règles sont dans telle configuration d'outil.

Exemples d'outils :

- Checkstyle (indentation, entêtes de fichiers, bonnes pratiques…).
- PMD / SpotBugs / Sonar : vulnérabilités liées au code (fuite mémoire possible, risque de NPE, etc)
- Linters (pour le monde Javascript).

A intégrer dans la chaîne d'assemblage (*build*) standard. Vérifiable en intégration continue mais **aussi sur les machines des développeurs**. Et à appliquer à tout le monde (vérification systématique). C'est toujours ça qu'il n'y aura pas à commenter lors des revues de code.

Audits de Projet

Qu'est-ce qu'un audit ?

L'audit (…) est un ensemble « d'opérations d'évaluations, d'investigations, de vérifications ou de contrôles, regroupées sous le terme d'audit en raison d'exigences réglementaires ou normatives.

(…)

L'audit est perçu comme un outil d'amélioration continue, car il permet de faire le point sur l'existant afin d'en dégager les points faibles ou non conformes (suivant les référentiels d'audit). Ce constat, nécessairement formalisé sous forme de rapport écrit, permet de mener les actions nécessaires pour corriger les écarts et dysfonctionnements relevés. »

Source : wikipedia

Sources d'Informations

Sur quoi se base-ton pour un audit ?
Plusieurs sources en général :

- Entretiens avec des personnes impliquées dans le projet. Définir quelles personnes, leurs rôles, la pertinence de les rencontrer dans le cadre de l'audit.
- Documents : spécifications, documents d'architecture, de conception, d'exploitation, documentation pour utilisateurs, etc. Tout ce qui est susceptible d'aider à comprendre le projet et son contexte. **A condition qu'il y ait de la documentation**.
- Forges : inclut le code, la gestion des tickets, les forums, salons de discussion, etc. Ce n'est pas le plus facile pour commencer. Tout dépend de la demande formulée. Parfois, un audit doit se limiter au code. Mais même quand c'est la demande, le code en lui-même est rarement suffisant.

Fonctionnel

Pour comprendre un projet, il est nécessaire de savoir à quoi il sert.

- Quelle est sa finalité ?
- A qui s'adresse-t-il ?
- Est-ce un projet utilisé uniquement en interne ?
- Ou bien distribué chez des clients ?
- Est-ce un logiciel commercial ou avec des facettes commcerciales ?
- Si oui, quels sont les marchés visés (national, continental, international) ?
- Quel modèle économique ? Licence, support…
- Disponible en plusieurs langues ?

Tout ce qui peut vous aider à comprendre le projet sans avoir à rentrer sur les détails techniques (ou plus exactement, sur la manière dont il est fait).

Aspects Organisationnels

Identifier les rôles et les personnes.

- Chef de projet, architectes, experts, administrateurs…
- Qui sont les leaders, les personnes avec un impact fort ?
- Qui produit quoi ?
- En termes de code, y a-t-il des contributeurs plus essentiels ?

Quels processus et outils autour du projet ?

- Gestion des sources : Git, SVN, CVS…
- Gestion des tickets ?
- Intégration continue.
- Processus de validation, plans de tests…
- Démarche qualité.

Historique

Comprendre l'historique du projet.

- Année de création ?
- Événement marquants de la vie du projet (ex : changement d'équipe) ?
- Nombre de livraisons et mises en production, version actuelle ?
- Code : nombre de commits et de contributeurs.
- Fréquence des livraisons / mises en production ?
- Y a-t-il eu beaucoup de *turn-over* dans l'équipe ?

Le projet utilise-t-il une gestion sémantique des versions ?
Y a-t-il eu des changements majeurs de technologies et/ou d'architecture ?

Parfois, il arrive que réécrire un projet coûte moins cher que de le faire évoluer. Un projet souvent remanié peut avoir accumulé une énorme dette technique.

Architecture

Y a-t-il un document d'architecture ?
Exemple : DAT (Dossier d'Architecture Technique)

Un document d'architecture ne se résume pas à un dessin.
Il inclut certes des diagrammes, mais ils sont accompagnés d'explications.

Idéalement, il retrace le cheminement qui a aboutit au résultat actuel.
Exemples : volumétrie cible (tant d'utilisateurs à la seconde), tolérance aux pannes (oui, mais quelles pannes ?).

Il est utile de regarder également l'existant d'un document d'exploitation. Quelles sont les procédures et informations mises à disposition (et enrichies par) les équipes en charge de la production (« la vraie vie ») ?

Est-ce que la conception et l'exploitation coïncident ?
Tout écart devrait inquiéter. Attention aussi aux évolutions liées à la charge (**une architecture qui évolue alors qu'elle s'éloigne des conditions initialement prévues**).

Livraisons

La question est de savoir ce qu'il faut pour déployer le projet.

- Quels sont les prérequis ?
- Qu'est-ce qui est livré par les équipes projets ? Images Docker, RPM, JAR…
- Où sont stockés ces livrables ?

Il faut aussi se demander comment ils sont produits.

- Automatiquement, dans un pipeline ? Y a-t-il des procédures manuelles ?
- Quelle est la fréquence de livraison ?
- Livraison globale ou bien différenciée selon les briques ?

Démarche Qualité

Y a-t-il une démarche qualité ?
Il y en a une si 2 conditions sont remplies :

1. Les éléments de la démarche sont documentés quelque part.
2. Des procédures ont été définies pour vérifier leur mise en œuvre.

On préférerait que ces procédures soient outillées, et si possible, automatisées. Voir le sujet des revues de code, et des outils comme Sonar et Checkstyle.

Les *pull requests*, avec revue de code, ou la programmation par paire, sont des éléments qui contribuent à la démarche qualité, tout comme l'automatisation des tests et la rédaction de plans de tests pour ce qui ne peut pas être automatisé.

Tests

Il s'agît de déterminer comment le projet est testé.
Pour toute mise en production (« la vraie vie »), on doit tester l'application. Une erreur peut avoir des répercussions organisationnelles, financières, voire même mortelles.

=> Penser à l'aéronautique par exemple.

Ces tests sont-ils automatisés ? Documentés ?
De quels types sont-ils ? Unitaires, intégration, e2e ? Ce qui n'est pas automatisé doit être documenté.

Combien de temps faut-il pour valider une livraison logicielle et la mettre en production ? L'automatisation permet de gagner du temps.

Y a-t-il des indicateurs liés aux tests ?
Exemples : taux de couverture de code, taux de couverture de branches.

Juridique

Attention aux aspects juridiques. Cela porte sur :

- Le respect de la propriété intellectuelle (code copié ?).
- Le respect des licences (dépendances incluses). Licences contagieuses ?
- Le respect des lois là où la solution sera exploitée.

Tout cela tient aussi au contexte, et notamment de savoir si le projet est uniquement utilisé en interne, ou bien s'il est distribué à des clients.

Distinguer mention de propriété intellectuelle…

```
/*
 * Copyright (c) 2019 - Université Grenoble Alpes
```

… et mention de licence.

```
 * Licensed under the terms of the MIT Licence.
 */
```

Cela devrait toujours figurer dans vos sources. Certains projets n'ont pas de licence. Quand ils en ont une, le texte intégral de la licence devrait apparaître dans le dépôt de sources. Vérification automatique avec Checkstyle.

Documentation

- Documentation développeurs : quel langage de programmation, comment est organisé le code, quels frameworks sont utilisés, comment compiler le code, quel outil d'assemblage…
- Documentation utilisateurs : certaines applications nécessitent une aide contextualisée, des guides pratiques, des tutoriels. Cela constitue la documentation utilisateur. Potentiellement en plusieurs langues.
- Documentation exploitants : les éléments d'architecture, avec les procédures (installation, mise à jour, sauvegarde des données et de la configuration, restauration, actions d'administration.…). Le tout est destiné à ceux qui vont gérer la production.
- Lisez-moi : pour les forges modernes, un ficher **readme.md** est indispensable. Il est la devanture de votre projet, un point d'entrée vers d'autres ressources (documentation, forum, etc).

Cela s'ajoute aux documents projets, comme documents d'architecture, de conception, etc. Si ce n'est que ceux-là n'ont pas vocation à être partagés avec un grand nombre de personnes.

Un Projet Logiciel, c'est tout ça !

Il ne s'agît pas que de pondre du code.
En tant que futurs ingénieurs, vous devez adopter une démarche scientifique et rigoureuse dans vos projets. Il faut être capable de justifier un choix, de l'argumenter et de le documenter.

Beaucoup de projets rencontrent des difficultés car les éléments de conception et le contexte initial ont été perdus (*turn-over* des équipes, pas de documentation…). Or, la vie d'un projet ne se limite pas à son développement.

Il subsiste néanmoins une part d'artisanat dans notre métier
Ainsi, on peut reconnaître certaines personnes rien qu'en lisant leur code. Mais votre plus-value sur le long-terme sera bien votre capacité à réfléchir et à vous poser les bonnes questions avant de trouver des réponses.

C'est limpide lorsque l'on regarde les TJM pratiqués.
Un développeur n'est pas vendu le même prix qu'un architecte ou un expert technique.

Annexe : Éléments de Conception

Pas de méthodologie universelle.
Mais quelques éléments de méthode pour structurer votre travail.

Principes de Base

- Faible couplage (limiter les adhérences)
- Cohésion forte (qui est responsable de quoi ?)

Patterns et anti-patterns

Il existe une large littérature sur le sujet, dans divers domaines d'application.

Hiérarchiser les besoins

Généralement, on trouve :

1. Le contexte (utilisation, qui, quoi, environnement)
2. Les concepts (qu'est-ce que l'on veut faire ?)
3. La couche logique (on ébauche des solutions - ex : publish / subscribe)
4. La couche physique (comment on met en œuvre la solution - ex : ActiveMQ)

Annexe : Justifier un Choix - 1/4

Exemple de méthodologie.

1. Poser le besoin.
2. Identifier des critères d'évaluation et comment les noter.
3. Définir une pondérations des critères : certains peuvent être plus importants.
4. Identifier les impératifs et contraintes (c'est un critère qui disqualifie toute solution qui ne le remplit pas)
5. Lister les solutions candidates.
6. Evaluer chaque solution par rapport à la grille de critère et de contraintes.

La solution avec la meilleure note est celle retenue.
Si la conclusion n'est pas satisfaisante, c'est soi que les pondérations ne sont pas bonnes, soit qu'ils manquent des critères.

Annexe : Justifier un Choix - 2/4

Exemple basique : choix d'une BD, avec des critères restreints.
Besoin : stocker des données.
Contrainte : la solution doit être open source.
Critères :

Critère	Description	Notation	Pondération
1. Maîtrise dans les équipes	Les équipes sont-elles déjà formées sur la solution ?	0 si non 2 si oui	2
2. Tolérance à une panne de VM	Quelle est la difficulté pour mettre en place une installation résiliente en cas perte d'une VM hébergeant un serveur ?	0 si impossible 1 si possible avec solution annexe 2 si natif	2
3. Impact d'un changement de schéma ?	Quel est l'impact d'un changement de schéma sur le code applicatif ?	0 si compliqué à gérer 1 si simple mise à jour 2 si rien à faire	1

Liste des solutions considérées : PostgreSQL, Oracle, MongoDB

Annexe : Justifier un Choix - 3/4

Remplaçons la grille pour chaque soution.
Oracle ne respecte pas le critère open source. La solution n'est pas évaluée.

Critère \ Solution	**PostgreSQL**	**MongoDB**
1	SQL standard supporté. => 2	Équipe pas formée aujourd'hui => 0
2	Mécanisme de serveurs primaire et secondaires. => 2	Mécanisme de serveurs primaire et secondaires. => 2
3	Code ORM à mettre à jour => 1	L'historique des schémas des collections doit être gérée dans l'application. => 0

En appliquant les pondérations, on obtient les notes suivantes :

- PostgreSQL : 2x2 + 2x2 + 1 = 9
- MongoDB : 0x2 + 2x2 + 0 = 4

PostgreSQL obtient la meilleure note.
C'est la solution retenue.

Annexe : Justifier un Choix - 4/4

Cette méthode est itérative.
Si une nouvelle solution apparaît sur le marché, il suffit de rajouter son évaluation.

De même, si un critère évolue, il suffit de mettre à jour la liste des critères et de revoir l'évaluation des solutions pour ce critère.

Cette procédure devrait être documentée et archivée.
En cas d'évolution du projet, on pourrait aisément rajouter une solution dans l'étude sans avoir à retoucher au reste.

Deuxième audit

Cet audit aura lieu le 5 novembre au matin.
Déroulement :

1. Un groupe entre et tire au sort le nom d'un autre groupe.
2. Le projet du 2ème groupe est audité par le premier.
3. Les remarques seront consignées.
4. Le groupe en question devra dans la journée, répondre, justifier ou rectifier suite à chaque remarque.

Exemples d'éléments à regarder :

- Qualité du lisez-moi, de la documentation développeur, exploitant…
- Organisation de l'équipe
- Qualité du code
- Outils mis en place
- Des tests ont-ils été rajoutés ? Quelle pertinence ?
- Voir plus haut pour d'autres idées…

Des instructions pratiques vous seront données très prochainement.