**Document présentation**

|  |  |
| --- | --- |
| **Description** | Cahier des tests ProxyStat-Gestion |

**Document certification**

|  |  |  |  |
| --- | --- | --- | --- |
|  | *Name* | *Fonction* | *Date livraison* |
| *Author* | RAKOTOBE Eric | Responsable qualité |  |
| *Decidor* | CRASKE Antoine | Chef de projet |  |

**Document Version history**

|  |  |  |  |
| --- | --- | --- | --- |
| *Version* | *Date* | *Change Summary* | *Change Author* |
| 1.0 | 19 février 2013 | Initialisation du document | RAKOTOBE Eric |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Objectif du document

# Table des matières

[Objectif du document](#h.f55k05uc5fys)

[Table des matières](#h.z3592ryvhrwo)

[Méthodologie](#h.b5kn9bt2v2f)

[Tests unitaires](#h.yppjxa380x7j)

[Tests d’intégration](#h.b4gvgvc83owu)

[Tests de performances](#h.wvspxeay4els)

[Périmètre couvert](#h.el5prynba71q)

[Revues de code](#h.mh81sk4efyv)

[Améliorations](#h.t2ecaj2tjlr5)

[Configuration](#h.7g6kundn90cc)

[Tests de performances](#h.xzh74u63ym1r)

[Scénario page d’accueil](#h.b5kmjod4i8br)

[Scénario authentification valide](#h.so212i748elp)

[Scénario authentification invalide](#h.xvc5vewq5er5)

[Enregistrement des résultats](#h.nnhabvsuzzzl)

[V1 : 22 février 2013](#h.6das7k1zcsap)

[Tests unitaires](#h.vhas2ty3gxhs)

[Tests d’ingration](#h.dmiqapf4n44o)

[Tests de performances](#h.awi11ta7ex92)

# 

# 

# Méthodologie

Il n’est pas possible de tout tester dans un projet. Les paragraphes suivants présentent les choix que nous avons choisi d’effectuer afin d’obtenir le meilleur rendement en terme de qualité logicielle.

Nous utilisons l’outil Sonar afin de suivre les différents indicateurs mentionnés. Sonar est installé sur le serveur d’intégration continue mis à disposition ce qui dispense les développeurs de lancer l’analyse en local.

## Tests unitaires

Les tests unitaires nous permettent de valider le fonctionnement d’un composant en isolation. Nous utiliserons des mécanismes de mocking afin de simuler l’environnement autour d’un composant afin d’assurer le testing en isolation.

Nous avons fait le choix d’avoir une classe gérant l’accès aux données de manière générique, cela nous permettant d’avoir toutes les opérations standards (CRUD, count) en étendant cette classe dans n’importe quel DAO. Nous avons donc testé entièrement ces méthodes car étant utilisées partout nous devons en assurer la robustesse.

Au niveau de la base de données nous utilisons un script qui rafraichit les données à chaque passage des tests via maven. Cela nous permet d’assurer une qualité de jeux de données fiable et une reproductivité des tests. Nous n’utilisons pas de base de données en mémoire pour tester sur la même base que celle de production Postgresql et éviter toute surprise.

Le nommage des tests unitaires respectent les standards. Toute méthode de test est préfixée par ‘test” et doit décrire en anglais l’action à réaliser et dans quelle condition. Par exemple “testBookingServiceThrowsExceptionWhenGuestsAreLessThanOne()”. Cela nous évite d’avoir à gérer des commentaires expliquant le but du test, nous avons une documentation vivante des fonctionnalités en regardant tous les noms des tests exécutés. Ceci est bien sur valable à condition de conserver un taux de couverture suffisant.

Les tests unitaires sont rédigées en trois blocs de code : given, when, then. Le premier décrivant le contexte du test avec les varibles et initialisation du contexte, le when qui effectue l’action proprement dite par exemple appeler un service et le then qui vérifie l’état attendu à la fin du test, c’est donc l’étape de vérification.

Le taux de couverture défini est de 100% sur la couche DAO et de 80% sur la couche de service. Nous ne pouvons en effet pas tout tester - les 20% non testés sur la couche service sont les services les moins utilisés et les moins critiques pour la société ProxyStation qui seront les méthodes appelant uniquement les DAOs sans aucun traitement particulier. Nous effectuons également une revue des tests lors des spécifications pour se focaliser sur les cas de tests et pas uniquement la couverture du code. On peut par exemple tester quatre fois le même code mais avec des paramètres différents pour vérifier le comportement.

Les tests unitaires sont effectués avec le framework standard Java JUnit en version 4. L’environnement est mocké avec le framework Mockito. Ces deux outils sont des standards du marché qui ont fait leur preuve. Ils sont automatisés avec maven.

Les métriques suivies sont le taux de couverture de code par branche de code car le taux de couverture par ligne n’est pas fiable car il prend en compte le nom des classes. Evidemment nous suivons le taux de réussite des tests qui doit être de 100%. Aucun changement dans le gestionnaire de versions n’est poussé par un développeur si le périmètre complet de tests n’est pas validé sur la machine. Ceci pour éviter les bugs en cascade et qu’un développeur ne bloque ou ralentisse pas le développement des autres membres de l’équipe.

## Tests d’intégration

Les tests d’intégrations nous permettent de valider le fonctionnement des composants entre eux. Nous avons fait le choix d’utiliser un outil permettant de simuler des actions sur un naviguateur web afin d’être au plus près de la vision utilisateur et assurer l’intégration de bout en bout. Les tests d’intégrations sont tous automatisés et doivent être validés pour packager l’application et la déployer sur les serveurs d’applications. Nous avons mis cette contrainte dans l’outil maven utilisé pour gérer le cycle de vie de l’application.

Nous utilisons l’outil Selenium (dernière version avec WebDriver) avec le driver Firefox. Les tests de compatibilité multi-naviguateurs sont eux effectués avant chaque livraison. Nous avons fait ce choix pour garder un temps d’exécution de ces tests dans une limite de 5 minutes maximum en parallélilsant. Nous ne pouvons pas donc pas exécuter les tests sur tous les navigateurs à chaque changement de code pour des raisons de délais de livraisons et que les développeurs garde un feedback rapide sur leurs modifications.

Nous mesurons les tests d’ingrations par leurs taux de succès qui doit être à 100%. Nous validons également que les tests implémentés correspondent bien à ceux présents dans le document de spécifications détaillées et que nous passons bien par tous les écrans.

## Tests de performances

ProxyStation n’a pas été en mesure de nous fournir le traffic prévisionnel sur leur site internet. Nous nous sommes donc basé sur le nombre actuels de clients en appliquant le taux de transformation moyen des sites web de 2%.

ProxyStation gère environ 1500 réservations par an ce qui nous donne un traffic prévisionnel annuel de 75 000 visites soit 205 visiteurs uniques par jour. Détails du calculs ci-dessous :

1500 réservations = 2% du traffic

75 000 visiteurs uniques annuels = 100% du traffic

Afin de garantir une solution robuste et évolutive, Dream IT a décidé d’assurer des temps de réponse stables inférieurs à 1 seconde pour un traffic annuel double de 150 000 visites.

Les tests de performances sont réalisés de façon incrémentale en enregistrant les résultats pour être capable de détecter un seuil de non-performance. Ils sont également effectué pendant plus de 30 minutes pour assurer la fiabilité de la plateforme dans le temps.

Nous mesurons donc les métriques suivantes :

1. Nombre d’échantillons par minute
2. Durée du test
3. Temps de réponse en ms

Nous utilisons l’outil JMeter qui effectue des appels HTTP directement sur les serveurs, nous ne dépendont de cette façon pas des contraintes des postes clients mais testont bien la plateforme applicative.

Les tests de performances sont effectués manuellement à chaque livraison. Il n’y a pas de valeur ajoutée à les automatiser car ils sont longs pour être effectué à chaque modification du code. Les résultats détaillées sont archivés par la société Dream IT et les rapports agrégés sont disponibles dans ce document.

Nous avons choisi de tester les scénarios qui seront les plus sollicités dans un cadre d’utilisation réelle : affichage de la page d’accueil, connexion avec une connexion réussie et une autre incorrecte.

Les tests ont étés effectués sur un serveur virtualisé ayant la configuration suivante :

* 10Go de disque
* 1Go de RAM
* 1 VCPU
* OS debian squeeze 64 bits

De plus, cette VM est également utilisée comme serveur d’intégration continue effectuant des builds réguliers et hébergant le serveur web, le serveur applicatif et la base de données. Cela impacte donc également les résultats des tests de performances.

Cette configuration ne reflète pas de la configuration cible pour un environnement de production. La société ProxyStation ne mets pas à disposition de machine de production pour les tests de performances.

Dream IT conseille vivement un environnement de production redondé actif/passif ou actif/actif avec 4 serveurs physiques. 2 de bases de données et 2 applicatifs avec le serveur web et le serveur d’application disponibles.

# Périmètre couvert

Nous couvrons toutes les couches logicielles de tests en assurant les tests unitaires en isolation et les tests d’intégration. Les tests unitaires ne sont pas effectués sur la couche controlleur de la partie MVC car nous jugeons que les tests d’intégration suffisent et que la simulation d’un environnement web (requêtes etc) est trop lourde à maintenir.

# Revues de code

Les revues de code nous servent à partager la connaissance au sein de l’équipe et également de réduire la dette technique sur le projet.

Chaque membre de l’équipe présente chaque semaine le code qu’il a développé de façon globale en parcourant le code afin de donner les clefs des fonctionnalités aux autres membres. Cela permet de diffuser la connaissance et de faciliter l’évolution d’une fonctionnalité par différents développeurs.

Toutes les semaines le responsable qualité édite une liste des correctifs à corriger dans les 2 jours qui suivent. Le but de cette réunion est d’éviter d’accumuler trop de dette technique qui coûte cher sur le long terme. En effet, l’accumulation de petits défauts conduits sur le long-terme à des problèmes de maintenance dus à de mauvais choix. Les correctifs peuvent concerner du nommage de méthode, de variable, des problèmes de conception. L’outil sonar nous aide à détecter les erreurs en cas de code non conforme aux standards du marché.

# Améliorations

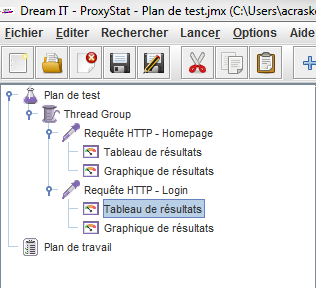
En 1 mois nous n’avons pas eu le temps d’effectuer tous nos objectifs mais avons les plans d’actions adéquats. Nous allons viser le taux de couverture de tests unitaires pour la V2.

# Configuration

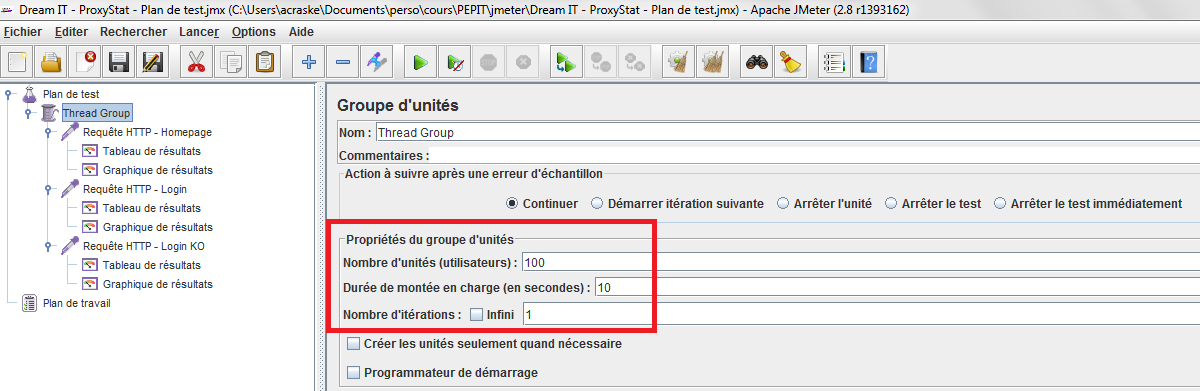
## Tests de performances

L’outil JMeter a été choisi car éprouvé dans le cadre de tests de performances. C’est un logiciel open-source client lourd Java. La version utilisée est la 2.8, dernière version stable à cette date.

Les 3 cas de tests ont étés configurés en respectant le standard de configuration JMeter : 1 thread group puis une déclinaison des cas de tests.



La configuration des paramètres de chargement de la plateforme se configurent au niveau du Thread group. Ces derniers sont changés à chaque nouvelle campagne de test en fonction de la charge cible.

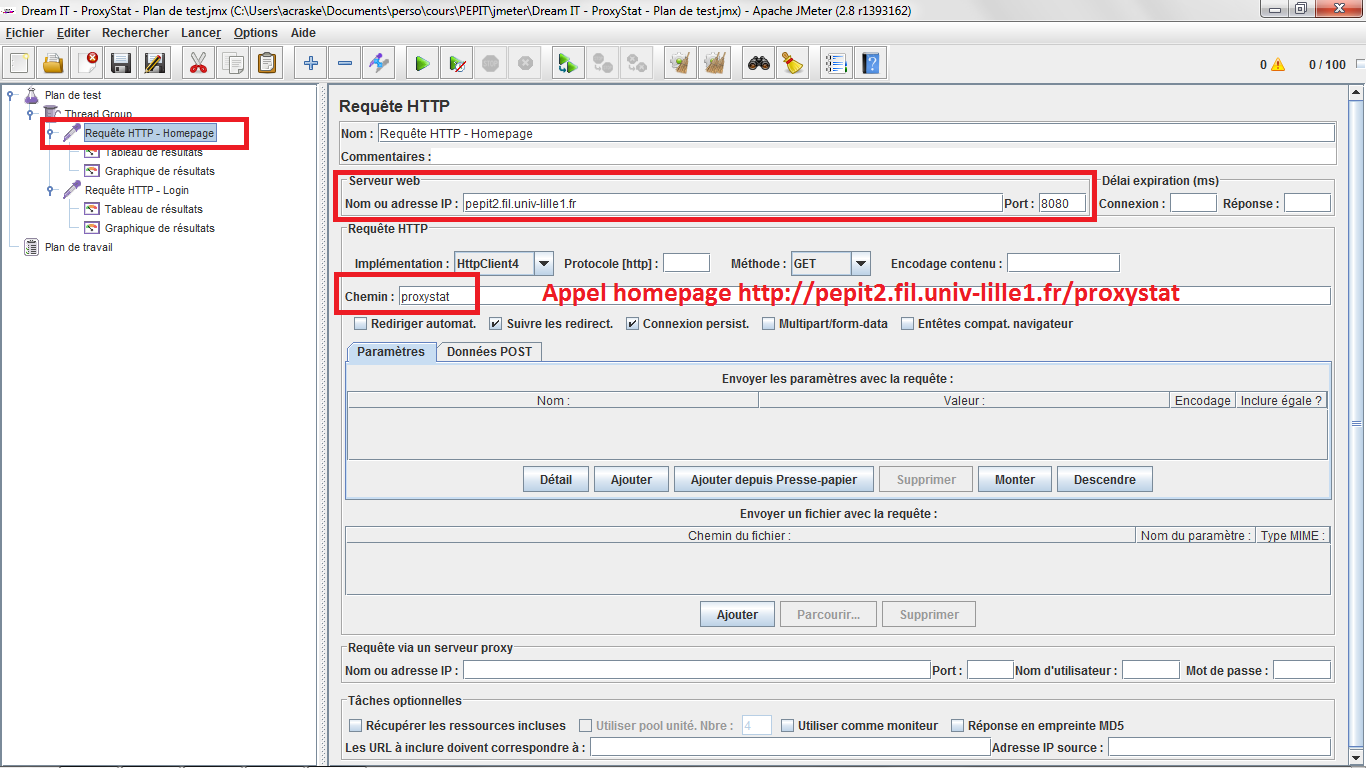


Chacun des scénarios de tests est un appel HTTP. On utilise un tableau stockant tous les résultats qui seront archivés, un tableau agrégant les résultats qui sont présents dans ce document et le graphique qui permet de détecter rapidement les anomalies.

Chacun des scénarios est décrit dans les paragraphes suivants.

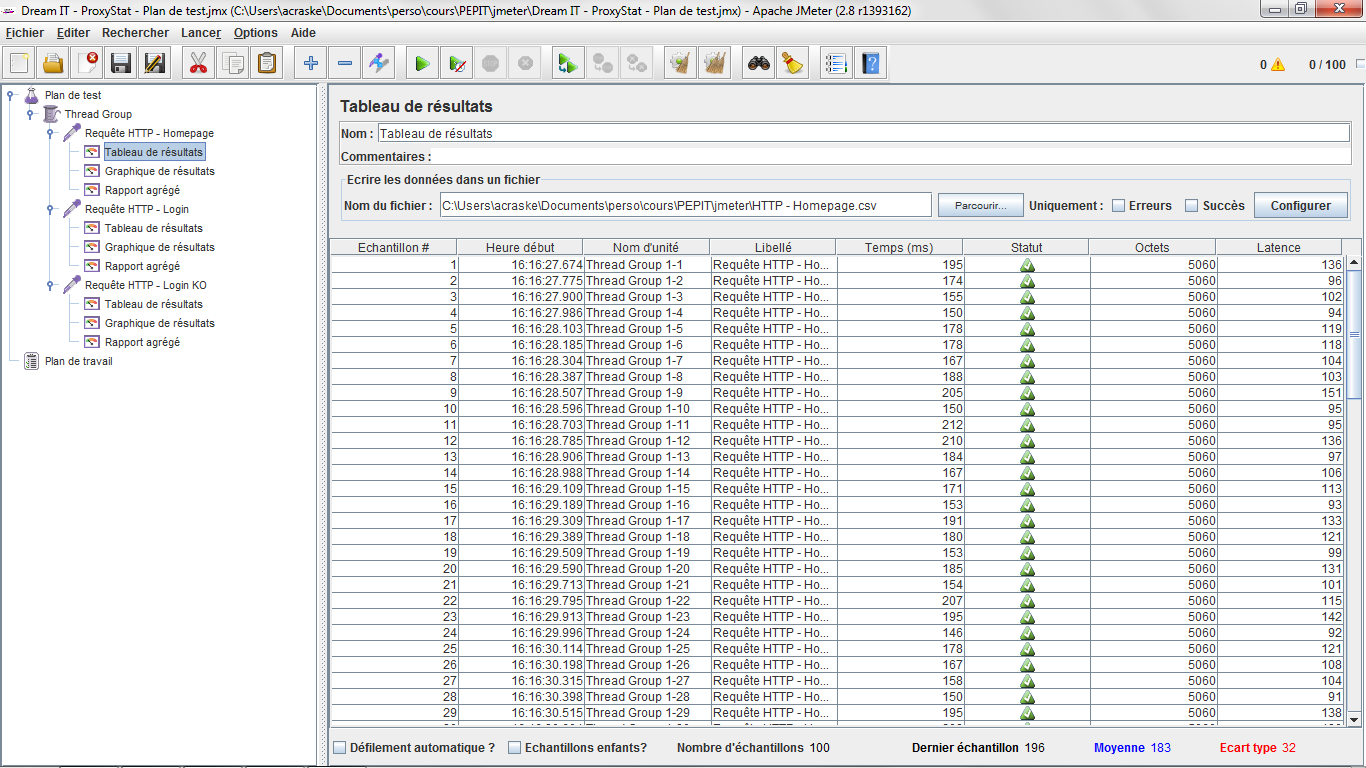
## Scénario page d’accueil

Ce scénario fait un appel HTTP simple sur l’URL de la page d’accueil. Le but est d’assurer un affichage de la page d’accueil dans des temps de réponses acceptables pour l’utilisateur afin qu’il reste sur le site La configuration est présentée ci-dessous.

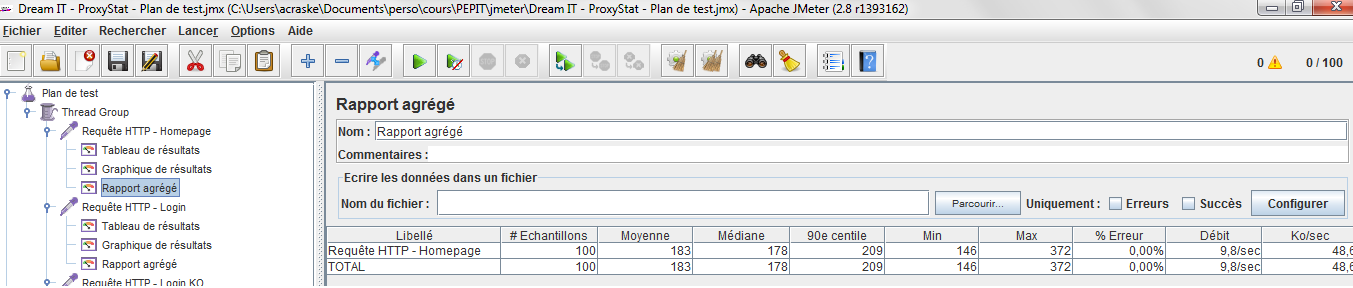


Pour rendre les choses plus concrète les différents types de résultats sont présentés ci-après uniquement pour ce scénario.

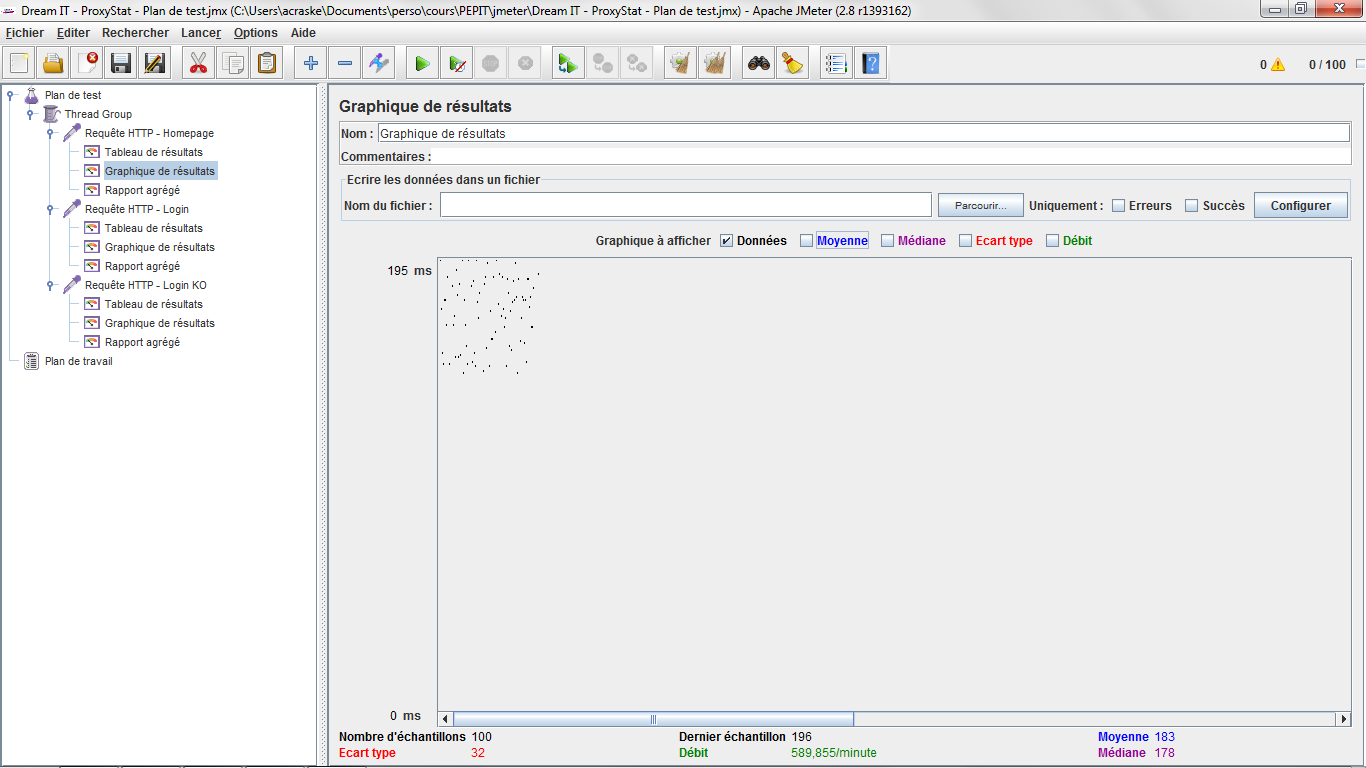
Le tableau complet de résultat :



Les résultats aggrégés :

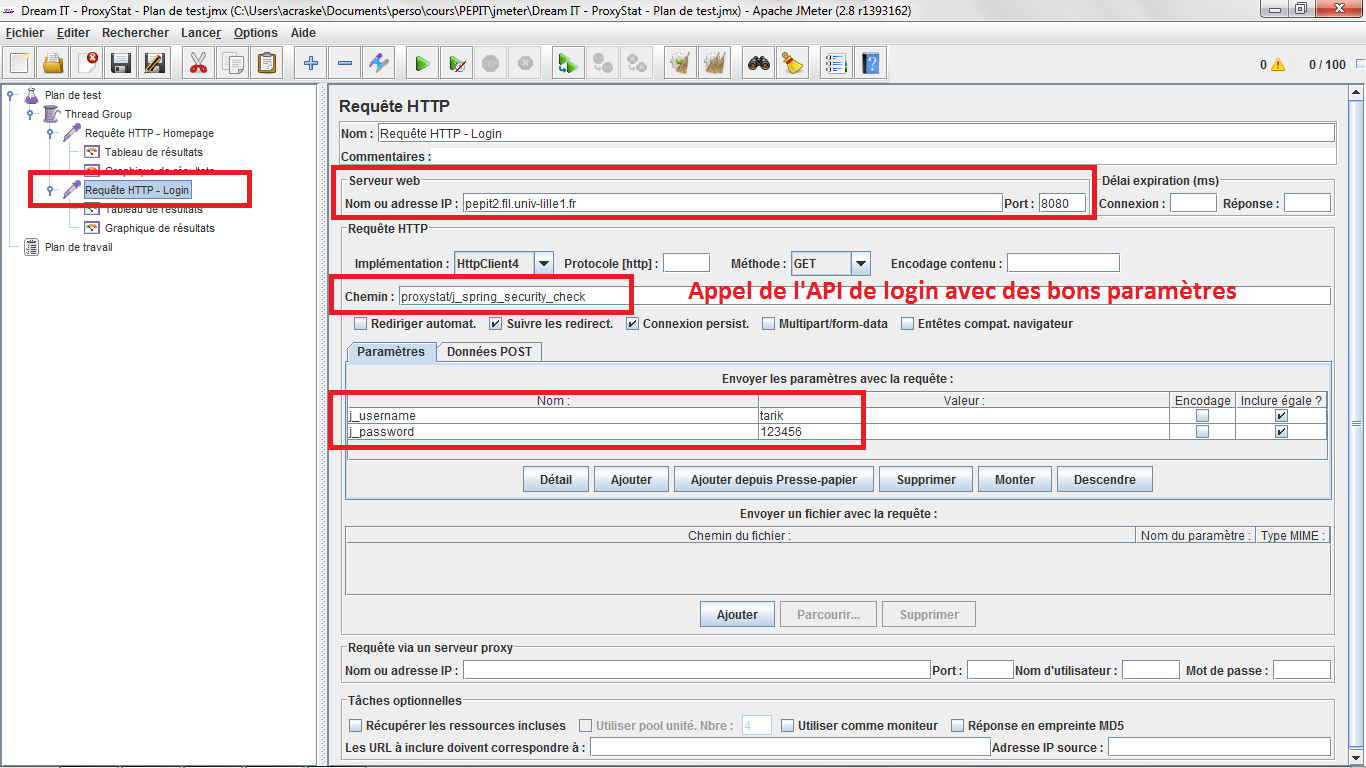


Le graphique de résultat :



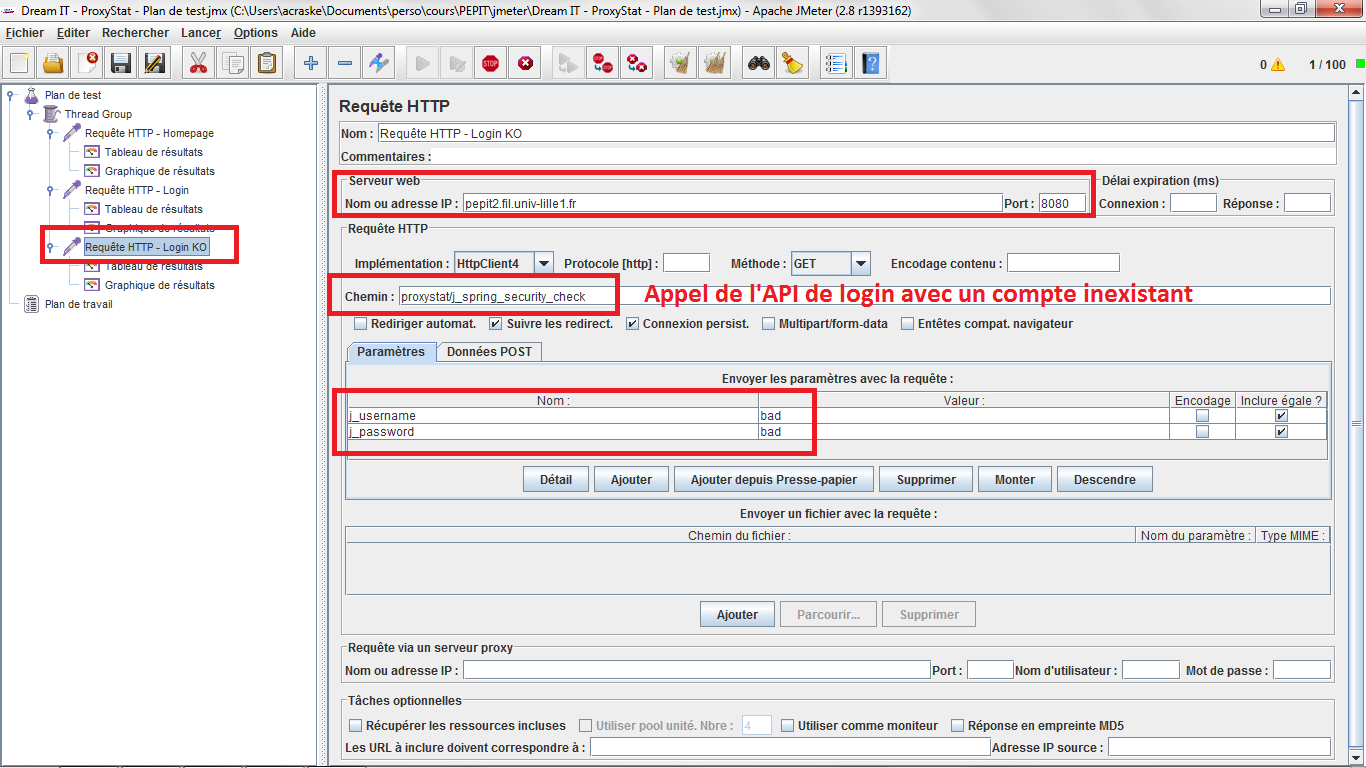
## Scénario authentification valide

Ce scénario fait un appel HTTP GET sur l’URL d’authentification avec des identifiants existants en base de données. Le but est de valider qu’on pourra authentifier les clients dans des temps de réponses acceptables pour qu’ils restent sur le site et achètent potentiellement.



## Scénario authentification invalide

Ce scénario fait un appel HTTP GET sur l’URL d’authentification avec des identifiants inexistants en base de données. Le but est de valider que la plateforme peut résister à attaques visant à surcharger la plateforme et/ou à supporter un grand nombre d’utilisateurs en simultanés se trompant de mot de passe.



# Enregistrement des résultats

## V1 : 22 février 2013

### Tests unitaires

### Tests d’ingration

### 

### 

### Tests de performances

Durée : 30 minutes

Nombre d’utilisateurs : 1000

Résultats :

|  |  |  |  |
| --- | --- | --- | --- |
| **Scénario** | **Temps réponse moyen (< 1s ?)** | **% Erreur** | **Résultat validé / invalidé** |
| Homepage | 779 ms | 0,30% | Validé |
| Login OK | 488 ms | 0,10% | Validé |
| Login KO | 351 ms | 0% | Validé |

Homepage



Login OK



Login KO



Durée : 30 minutes

Nombre d’utilisateurs : 5000

Résultats :

|  |  |  |  |
| --- | --- | --- | --- |
| **Scénario** | **Temps réponse moyen (< 1s ?)** | **% Erreur** | **Résultat validé / invalidé** |
| Homepage | 362 | 0 % | Validé |
| Login OK | 176 | 0 % | Validé |
| Login KO | 154 | 0 % | Validé |

Homepage



Login OK



Login KO

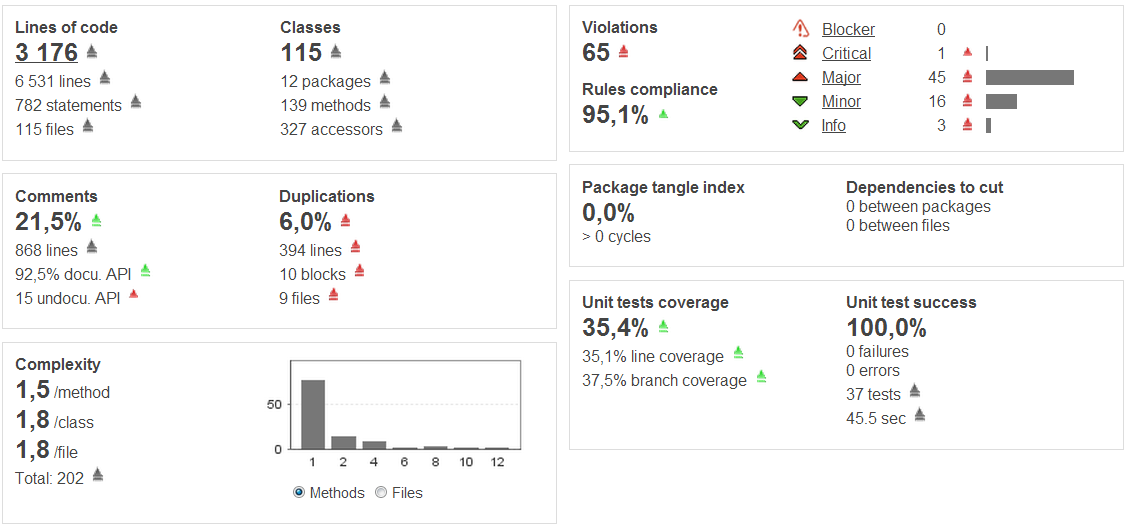


## Mesures avec l’outil Sonar

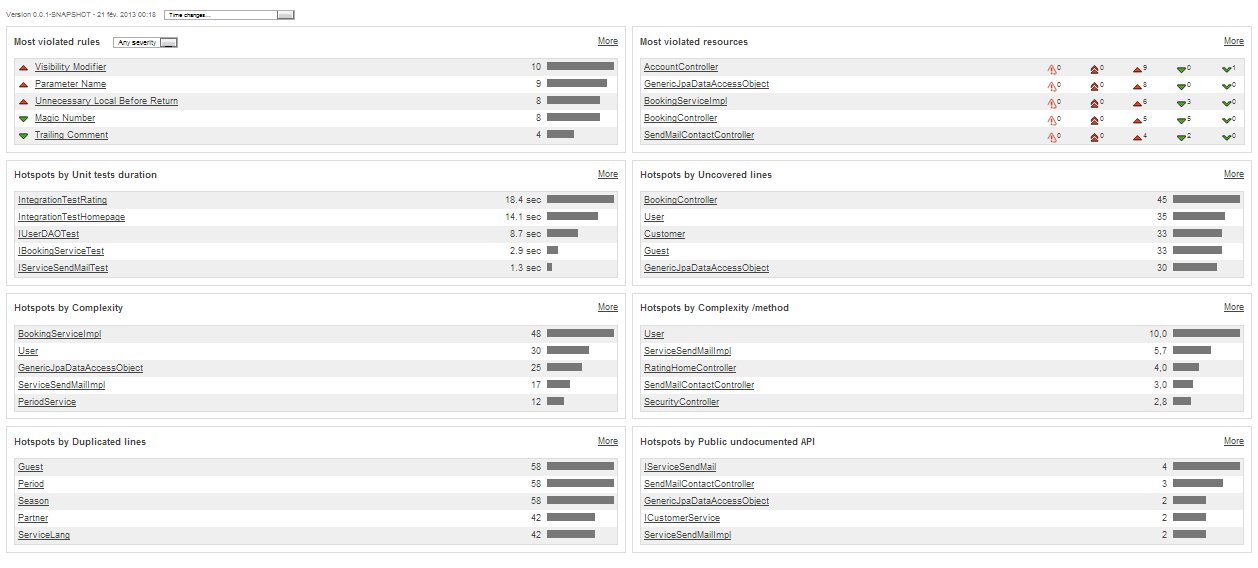
Ci-dessous le tableau de bord de notre projet sous Sonar. Toutes les alertes ne sont pas traités mais nous nous sommes focalisé sur les bloquantes et critiques. Le responsable qualité définira les actions les plus importantes à mener pour les réduire. Nous atteignons bien les 100% de succès des tests unitaires.

Nous avons 4% de fausses-alertes sur la duplication dû à des problèmes d’analyse de l’outil. Preuve qu’il faut toujours investiguer le calcul d’une mesure avant de s’y fier.

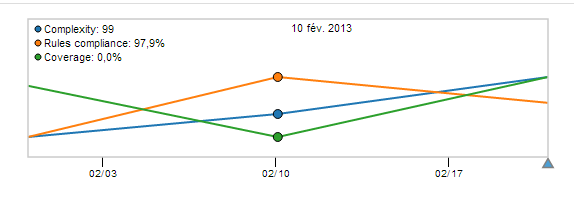
Le bon point est la documentation du code, nous faisons en effet le choix de documenter au maximum dans le code ce qui est utile aux développeurs au lieu de le faire dans un document séparé qui est vite oublié ou perdu.



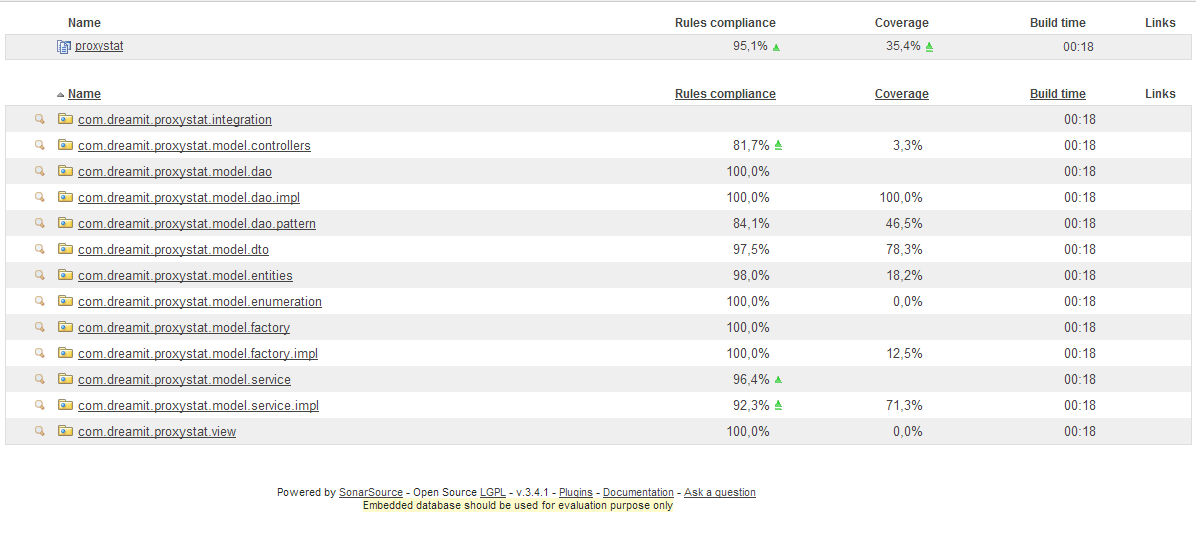
On voit que la majeure partie des erreurs sont dues à des problèmes de visibilité des champs dûs à des contraintes du framework de mapping objet/relationnel.

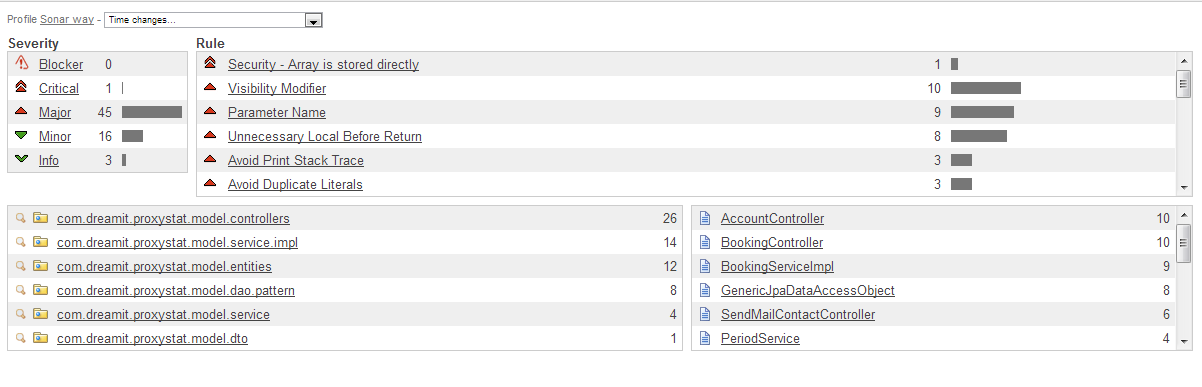


On note bien une diminution du nombre d’erreurs mais une augmentation de la complexité. Nous devons donc continuer notre plan d’action de revue de code et effectuer également une revue d’architecture sur les classes les plus complexes.

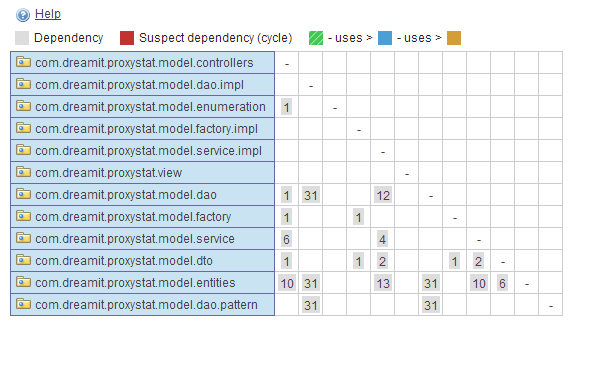


Nous sommes proches de nos objectifs : le 100% de couverture sur la couche DAO est atteint (dao.impl) et 71% sur la couche service par rapport à notre objectif de 80%. Nous montrons notre capacité à suivre les règles établies en début de projet.





Classiquement on retrouve les entités et DAOs avec le plus de dépendances. Les entités sont en effet potentiellement passées via toutes les couches et les DAOs utilisent les entités et sont appelés par les services. Nous n’avons donc pas d’alerte à ce niveau.



Ce nuage de tag nous montre les endroits du code les plus importants en terme d’alertes. Le service de réservation est le plus important fonctionnellement ce qui confirme sa complexité. Nous allons travailler plus particulèrement à l’amélioration de ce service.

