Master 2 Développement Logiciel

Rédigé avec LaTeX
Version du 12 novembre 2015

Documentation de BandStorm

Intégration, Vérification, Validation, Qualification

Université Toulouse III – Paul Sabatier

— Julian Bironneau
— Antoine de Roquemaurel
— Steve Magras
— Dylan Roletto
— Zaccaria Zyat

12 novembre 2015

Table des matières

1 Le projet **3**

1.1 Le projet Bandstorm 3

1.2 Les informations utiles 3

1.3 Fonctionnalités 4

2 Le *workflow* de développement **5**

2.1 Tests et couverture 6

2.2 Intégration continue 6

2.3 Déploiement continu 6

2.4 Analyse statique de code 7

3 Agilité **8**

3.1 Sprints 8

3.2 Planning Poker et vélocité 8

3.3 Objectifs des sprints 9

3.4 Définition de «fini» 9

3.5 Releases 10

4 Architecture **11**

1 Le projet

1.1 Le projet Bandstorm

Bandstorm est un projet qui a pour but de réaliser un réseau social pour des amateurs de musique. Celui-ci a pour objectif de mettre en relation des utilisateurs par le biais de communautés. Ces communautés sont constituées en fonction de préférences musicales.

1.2 Les informations utiles

1.2.1 Github

Lien du dépôt https://github.com/BandStormTeam/BandStormProject

Liens vers les profils

Julian Bironneau https://github.com/johnSilver7

Antoine de Roquemaurel https://github.com/aroquemaurel

Steve Magras https://github.com/smagras

Dylan Roletto https://github.com/vlaks

Zaccaria Zyat https://github.com/SirAbel

1.2.2 Intégration et déploiement continu

Travis-CI https://travis-ci.org/BandStormTeam/BandStormProject

Heroku https://m2dl-bandstorm.herokuapp.com/

> Il est possible de se connecter avec l'utilisateur «merry» et le mot de passe «password».

1.2.3 Qualité du code

Coveralls https://coveralls.io/github/BandStormTeam/BandStormProject

SonarQube http://sonar.joohoo.fr/dashboard/index/BandStormProject

Attention, le serveur sonar n'est pas disponible 24/24. Il est accessible entre 10h et 23h.

1.3 Fonctionnalités

Notre application possède de multiples fonctionnalités qui vont être détaillées ci-dessous. Chaque utilisateur de BandStorm possède un compte personnel sécurisé par un système de login.

— Les utilisateurs ont la possibilité de suivre d'autres utilisateurs.
— Les utilisateurs peuvent créer, rejoindre et visualiser un groupe.
— Les utilisateurs peuvent créer et visualiser un évènement.
— Les utilisateurs ont la possibilité de rechercher des entités via une barre de recherche (utilisateurs, groupes, évênements).
— Les utilisateurs peuvent poster des statuts.
— Les utilisateurs peuvent visualiser les statuts de leurs abonnements.
— Les utilisateurs ont la possibilité de "lighter" (aimer) un statut.
— Les utilisateurs ont la possibilité de changer leurs informations personnelles.

2 Le *workflow* de développement

Afin de faciliter le travail collaboratif et l'intégration de nos développements respectifs, nous avons utilisé Git et Github. Via un *workflow* en branches par fonctionnalités, nous avons pu développer celles-ci en parallèles

Lorsqu'un développeur choisi de répondre à une *issue*, il va créer une nouvelle branche, nommée suivant la convention suivante `sprintN/noIssue-recapitulatifIssue`.

Lorsque le développeur estime avoir terminé sa fonctionnalité, il ouvre une *Pull Request*, celle-ci nous permet d'effectuer de la revue de code. La fonctionnalité ne sera fusionnée avec la branche de niveau inférieur que lorsque l'équipe estime avoir terminé l'*issue*.

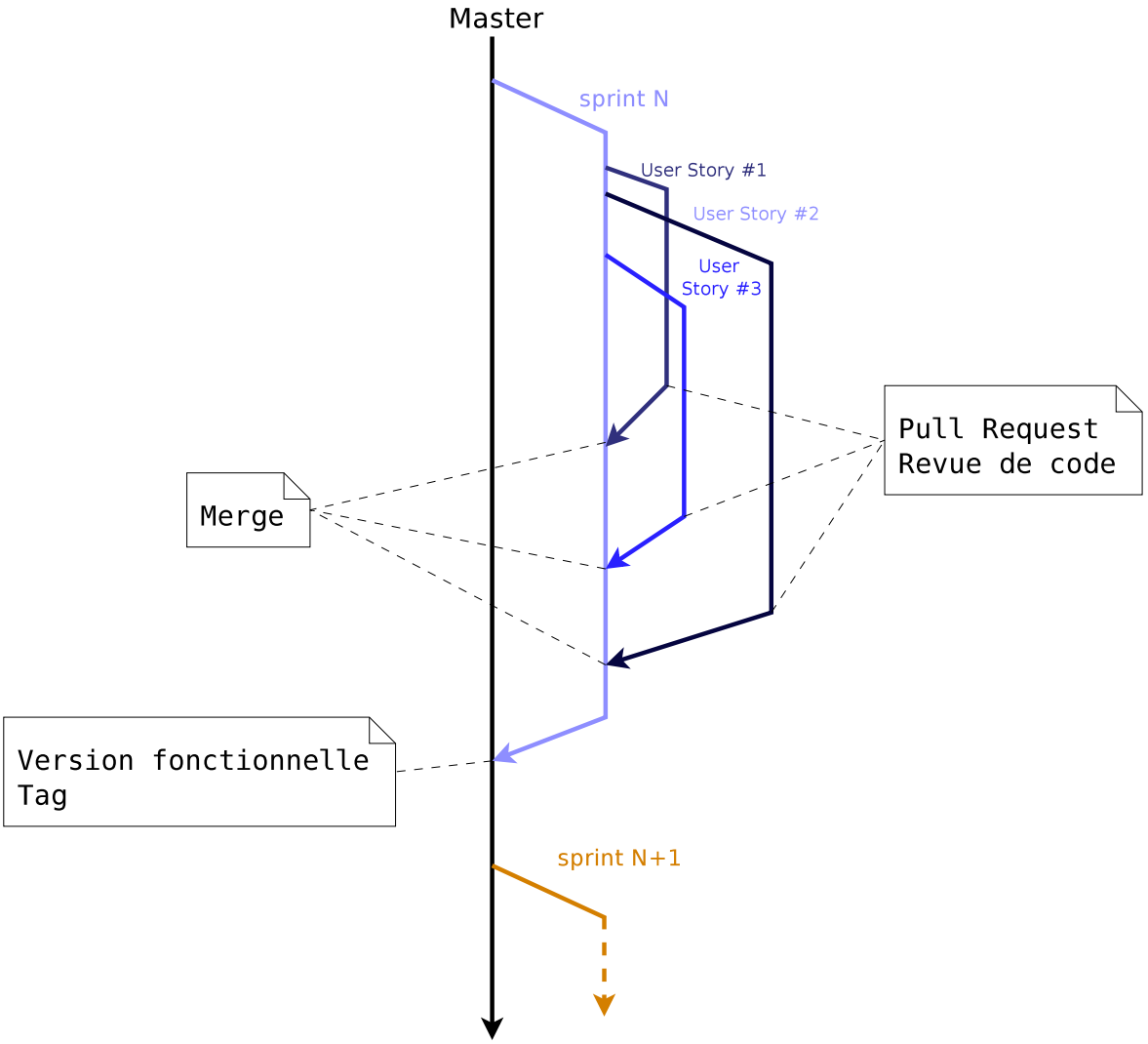

FIGURE 2.1 – *Workflow* en *feature* branches

Pour plus de détails, notre *workflow* est disponible ici :
https://github.com/BandStormTeam/BandStormProject/blob/master/workflow-feature.md

2.1 Tests et couverture

Lors du développement d'un projet complexe, il est indispensable de tester correctement le projet afin de pouvoir le vérifier et le valider. Ainsi, dans le cadre de BandStorm nous avons définis des pourcentages de couverture de tests à respecter impérativement ; Le bon respect de ces chiffres contribue à effectuer un projet de qualité.

Cette couverture doit être de 80% en ligne de code,
- 100% pour les classes du domaine.
- 80% pour les contrôleurs.
- 80% pour les services.

Les classes du domaine, services et contrôleurs devront être couverts par des tests unitaires. Les services de type DAO devront être couverts par des tests d'intégrations.

Afin d'avoir des tests les plus exhaustifs possibles, nous avons également mis en place des tests fonctionnels en utilisant Geb. Ces tests nous permettent de vérifier le bon fonctionnement global de notre application.

2.2 Intégration continue

Nous avons utilisé `Travis-CI` afin d'intégrer et de vérifier cette intégration de manière continue. Notre serveur d'intégration effectue plusieurs actions :
- Compilation du projet
- Lancement des tests unitaires
- Lancement des tests d'intégration
- Calcul de la couverture de code
- Éventuel déploiement sur heroku (uniquement master)

Le *build* ne peut être correct que si l'ensemble de ces actions est bien effectué.

> À noter que Travis CI ne lance pas les tests fonctionnels. En effet, les tests fonctionnels nécessitent un serveur graphique ainsi que le démarrage d'un navigateur. Travis-CI propose une aleternative pour simuler un environnement graphique, mais celui-ci ne semble pas fonctionner avec Geb. Il pourrait être intéressant d'installer un serveur d'intégration continue sur une machine personnelle tel que Jenkins ou Gitlab-CI, nous n'aurions pas cette limite de serveur X.

Les tests fonctionnels étant les tests les plus proches d'un utilisateur permettraient ainsi de pouvoir effectuer de la non-régression de façon systématique et automatique.

2.3 Déploiement continu

La vérification de la bonne intégration de notre application ne suffit pas à avoir un processus de mise en production fiable. Il est indispensable de vérifier le bon déploiement de l'application sur

un serveur autre que celui de production. C'est dans ce but que nous utilisons Heroku. La branche *master* de l'application est déployée de façon continue et automatique.

2.4 Analyse statique de code

Pour avoir un code de qualité, il est faut également prendre en compte la maintenance que celui-ci va coûter. Cette maintenance peut être réduite en développant avec des conventions claires, précises et mesurables. Ainsi, nous avons utilisé Codenarc qui permet de vérifier les règles de nos conventions d'écritures.

SonarQube quant à lui permet d'une part de centraliser toutes les informations nécessaires, tout en conservant un historique de celles-ci, mais il nous permet également d'avoir de nouvelles métriques tel que le nombre de ligne de code, le pourcentage de commentaires ou la complexité cyclomatique de notre projet.

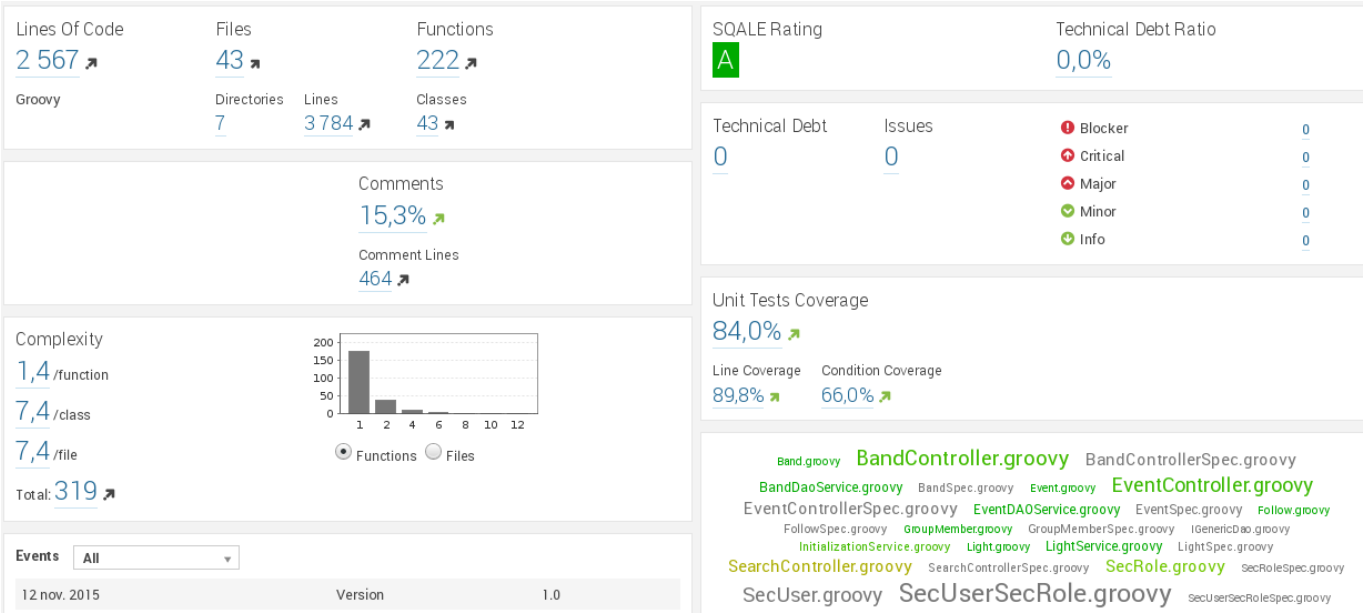

FIGURE 2.2 – Tableau de bord SonarQube

3 Agilité

3.1 Sprints

Pendant toute la durée du projet, l'équipe a suivi les principes du processus agile « Scrum ». La durée du projet a été découpée en trois sprints d'une durée de deux semaines chacun. Un sprint 0 a été réalisé afin de mettre en place l'architecture, le processus de développement et la première version du Backlog.

Il a été décidé que la répartition des rôles de *Product owner* et de *Scrum master* changent à chaque Sprint afin que tout le monde puisse essayer au moins un des rôles.

Une revue et une rétrospective ont été effectuées à la fin de chaque Sprint.
Le tableau ci-dessous recense les différents sprints associés aux rôles et aux dates de revues et rétrospectives.

	Sprint 1	**Sprint 2**	**Sprint 3**
Dates	30/09/2015 → 13/10/2015	14/10/2015 → 27/10/2015	28/10/2015 → 11/11/2015
Product Owner	Julian Bironneau	Dylan Roletto	Zaccaria Zyat
Scrum *master*	Steve Magras	Antoine de Roquemaurel	Julian Bironneau
Dates revues	14/10/2015	28/10/2015	12/11/2015
Dates rétrospectives	14/10/2015	28/10/2015	12/11/2015

3.2 Planning Poker et vélocité

Pour attribuer un nombre de points aux *issues*, l'équipe a effectué des *planning poker*. Ceci au travers d'un outil disponible gratuitement sur internet, *Pointing Poker* (https://www.pointingpoker.com/).

Après avoir terminé un sprint, il nous était donc possible de calculer la vélocité de celui-ci. Le plugin ZenHub nous a permis de représenter facilement et de manière graphique la courbe *burn-down* de notre vélocité.

Ci-dessous, le *burn-down* représentant le premier sprint :

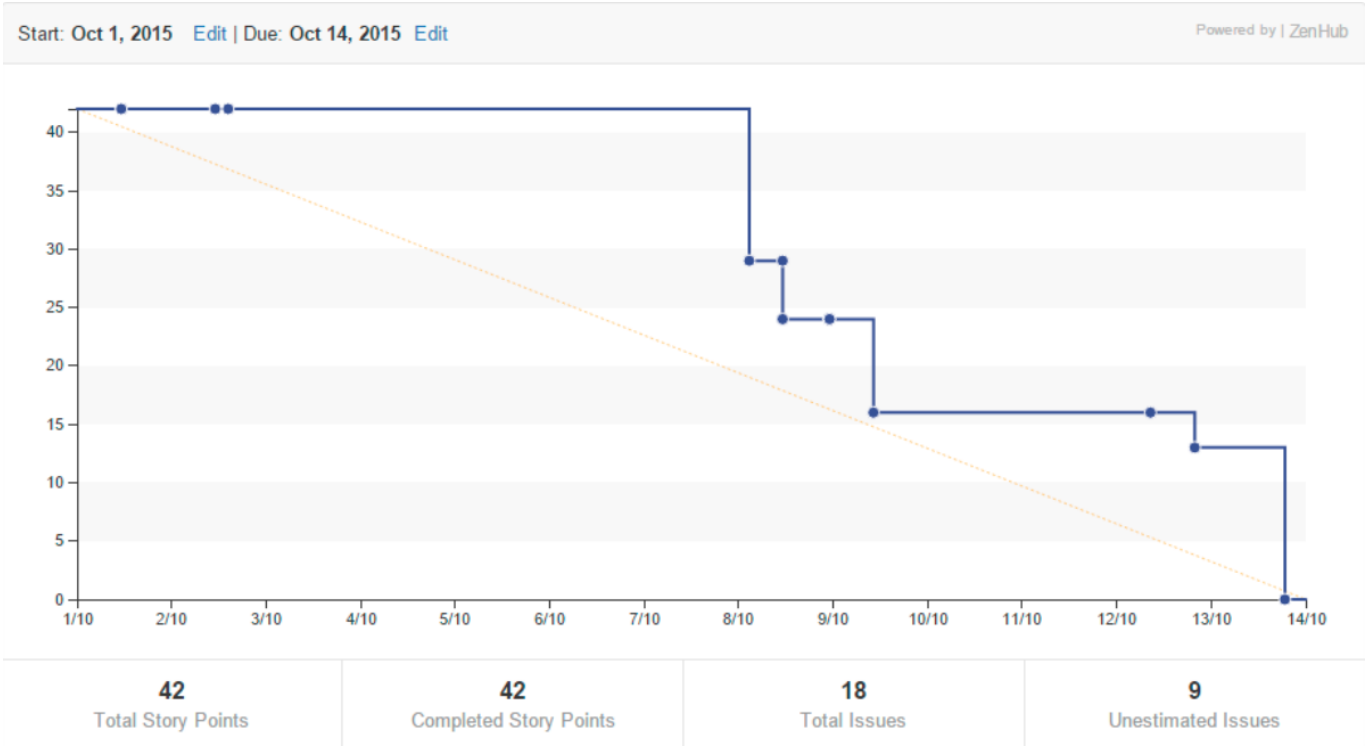

FIGURE 3.1 – Burndown chart du premier Sprint

Les résultats obtenus pour la vélocité de ce projet sont indiqués dans le tableau ci-dessous :

Sprint	Points prévus	Points effectués	Nombre d'issues
Sprint 1, v0.1	42	42	18
Sprint 2, v0.2	44	44	13
Sprint 3, v1.0	23	20	15

3.3 Objectifs des sprints

Les sprints ont été guidés par les objectifs de sprints ci-dessous :

Sprint 1 Avoir les fonctionnalités basiques d'identification d'un réseau social.

Sprint 2 Permettre aux membres de BandStorm de pouvoir intéragir sur le site.

Sprint 3 Corriger les bogues et de rajouter des fonctionnalités d'interactions entre les utilisateurs.

3.4 Définition de «fini»

On détermine qu'une *user story* est terminée lorsque :

— Les tests unitaires et d'intégrations sont passés
— 100% de couverture de code pour les classes du modèle
— 80% de couverture de code pour les classes contrôleurs
— 80% de couverture de code pour les classes services et DAO
— Les tests d'acceptation sont passés
— La *pull request* est revue par le *product owner*
— Le *build* est passé sur Travis CI
— Les critères Codenarc sont respectés
— Correction des *criticals* et *majors* sur SonarQube
— La javadoc est rédigée

3.5 Releases

Après avoir effectué les trois sprints, une release a été plannifée. Cette release est le résultat de tous les travaux effectués au cours du projet. Le projet doit être livré à Franck Silvestre le 12/11/2015 et être déployé sur le serveur Heroku.

4 Architecture

BandStorm s'appuyant sur le framework grails, une grande partie de l'architecture et de l'organisation est structurée par celui-ci. Ainsi, nous utilisons le modèle MVC, c'est-à-dire la séparation des contrôleurs vis-à-vis des vues HTML et des modèles.

Ci-dessous le diagramme de classes des modèles de notre application.

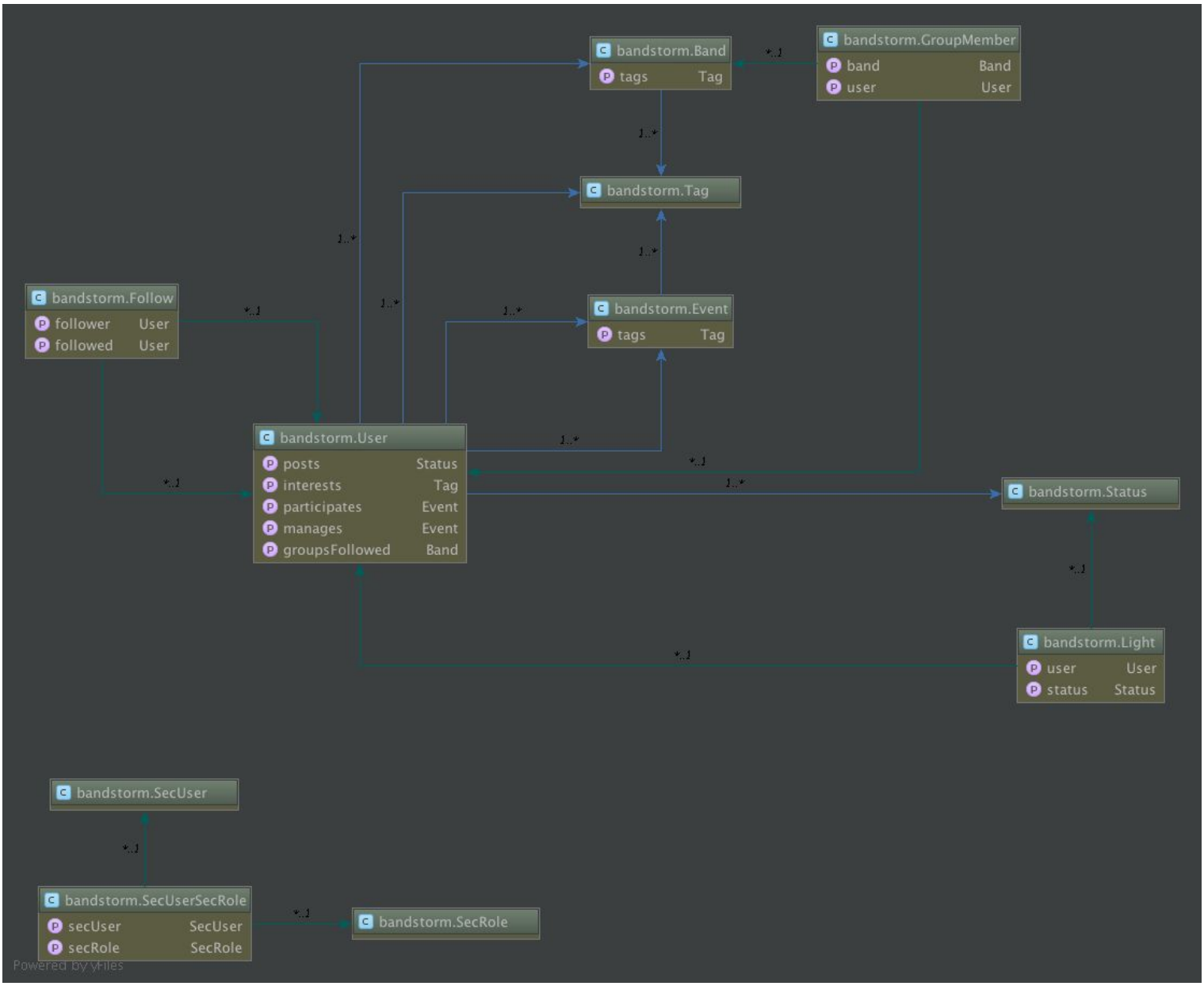

FIGURE 4.1 – Diagramme de classe de BandStorm