Log me tender

Speaker : Olivier Croisier (Freelance)
Format : Conférence
Date : 6 avril 2017
Slides : LMT-DevoxxFR-2017.pdf

Le sujet des logs n'est pas à la mode, mais touche tous les développeurs.

De System.out à nos jours

A l'origine, System.out.println("Hello World")
On peut réassigner System.out vers un fichier.

En 2001, Ceki Gulcu a développé une API (LOG4J). Influence très forte dans l'univers Java
API : Factory Statique : LogManager.getLogger(id) + des méthodes log permettant de logger messages et exceptions.
Cette API a persisté jusqu'à aujourd'hui.

En 2002, l'API Logging est apparue dans le JDK. Pas utilisée.

En 2012, projet Log4J2.

Plein de frameworks utilisant chacun son système de log.
En 2002, création d'une façade Commons-Logging.

En 2015, Geki Gulcu invite SLF4J qui est utilisé partout en 2017. API toujours similaire à SLF4J.

Des problèmes persistants

Probléme 1 : concaténation dynamique
LOG.debug(« User » + id)
Résolu avec une String template avec des placeholder {}. Plus de concaténation en avance.

Problème 2 : évaluation immédiate des paramètres
Exemple : DB.dump()
Solution 1 : if (LOG.isDebugEnabled())
Inconvénients : couplage technique avec la configuration du logger. Prend de la place (3 lignes). API non-DRY et possibilité d'erreur : isDebugEnabled et LOG.info.
Solution 2 : évaluation lazy grâce au supplier de Java 8. Plus besoin de test « externe » :
LOG.debug(() -> "DB :" + DB.dump()) ;
Implémenté dans JUM, Log4J2 mais pas dans SLF4J. SLF4J devient le facteur limitant.
Solution 3 : « Façader » SLF4J à travers d'un LogService.

Problème 3 : Factory statique
Depuis Spring et l'IoC, plus d'appel de méthodes statiques, mis à part les loggers.

Référence statique : non configurable, non testable, non mockable
Solution : référence non-statique. Notion de « service de log »

- Injectable par IOC
- Testable (test du service de log)
- Mockable (test des services utilisateurs)
- AOP-able (performances, coupe-circuit) : mieux vaut ne pas logger que de ralentir toute la chaine si problème dans logger

Problème n°4 : frameworks trop bas niveau

Non-Dry : harmonisation externe nécessaire

- Niveau de log : code de log éparpillé dans toutes les classes. Le format des logs et niveaux de logs ne seront pas identiques d'une classe à l'autre
- Format de messages : un log n'a d'utilité que s'il est exploitable

Couplage fort :

- Producteur / Consommateur : en 2017, on réfléchit en termes de trace. Mais on pourrait raisonner en termes d'événement (ex : tel service est appelé). Cet événement peut donner lieu à 0 ou N logs (audit, log de performance …)
- Framework de log

Solution : un service de log de haut niveau

- Façade du framework technique
- Expose une API technique et métier ad hoc : plus limité au niveau info, debug, error. On peut ajouter des méthodes. Exemple : beginProcess.
- Centralise la gestion des messages (format, niveau)
- Possibilité de raisonner en termes d'évènements
- Un événement => 1..N logs

Proposition LogService

Ce n'est pas un framework, mais une simple réflexion.
Les méthodes dépendent du métier, du framework de log …
Design goals :

- Une API restreinte et moderne : ne prend plus que des supplier Java 8
- Rétro-compatibilité et extensibilité : transition en douceur avec les méthodes debug, info …
- Façade pour SLF4J : standard
- Composant IOC : service Spring
- Factory toujours nécessaire pour le cas d'utilisation « 1 logger par classe ». Le LogService n'est alors pas accéder directement mais on passe par un LogServiceFactory

Conclusion

Modernes, Splunk et ELK permettent d'exploiter des logs.
La façon dont les logs sont produits aujourd'hui est encore archaïque.

Voir aussi ExceptionContext : trying to improve business exceptions http://bit.ly/2nQQqB5
Les exceptions métiers peuvent venir qu'à des endroits bien ciblées : nul besoin de la stacktrace. Or, la stacktrace coute très cher à générer.