Les Applications Réactives : un nouveau paradigme pour relever les défis de l'économie numérique !

Speakers : **Fabrice Croiseaux**, **Antoine Detante** – InTech (ESN au Luxembourg)
Devoxx France 2014 – Jeudi 17/04/2014 – 13h25 - 14h15

Animent le Podcast : nipdev.com
Autre titre : comment expliquer à son patron qu'il serait utile de développer en utilisant les patterns réactifs.
Evocation du livre blanc Reactive Manifesto

Définition maison d'un système réactif : système qui répond à des variations importantes de paramètres prédéfinis sans remettre en cause sa structure et sans perturber son fonctionnement nominal => typiquement le nombre de threads

Etre réactif au niveau business : modèle de développement lean startup : on part d'une idée, on livre et on regarde comment faire évoluer l'idée d'origine / business modèle. Capacité à pivoter.
Groupon, Paypal, Instagram ont pivoté.

Un point majeur du Lean startup : mesure systématique de tout ce qui se passe sur l'application. Pas fait : à cause des perfs, du temps de dév …
Les mesures doivent être asynchrones, non bloquantes, systématiques (au fil de l'eau, cas nominaux et cas d'erreur).

Réagir de façon non bloquante : place au code

Exemple d'un GET /user/1234 avec recherche d'utilisateurs puis des transactions. Enchainement synchrone avec blocage des threads le temps de récupérer les données.
Or, les tâches sont //. Une partie du temps côté serveur est perdu à attendre les données.
Modèle non-bloquant : requête en // + fonction de callback
L'objet getUser renvoie une instance de Future. La Future va être complétée par les 2 appels exécutés en //. Avantages : plus de threads bloqués. Le serveur réagit aux évènements.

Au niveau API, réseau et OS, il faut un mécanisme permettant d'effectuer des appels synchrones.

Exemple en Scala : type Future[T]. Une liste de Future peut être exécutée en parallèle. Possibilité de mettre des dépendances. Le Future a 2 états : complétée et non complétée. Des fonctions de callback peuvent être ajoutés sur l'évènement onSuccess.
On peut pas écrire dans une Future (lecture seule). Nécessité d'utiliser le type Promise pour compléter une Future. Nécessaire pour créer soit même des traitements asynchrones.

Autre exemple en JavaScript et jQuery : $.when().then() + promise()

Une Future complétée peut avoir 2 états : Success ou Failure. Gestion des erreurs plus difficile qu'en synchrone. Failure lorsqu'une exception a été levée lors d'un des appels.

En JS comme en Scala, l'erreur est traitée comme un évènement non exceptionnel.
En Scala, map et flatMap n'exécute que les cas en succès

Pourquoi être réactif à l'ère de l'économie digitale ?
4 piliers : Social, Mobile, Cloud et Big Data
Une application réactive est une application qui attend un évènement.
Sur les applis mobiles, il faut penser **Event**.
Le Social permet de créer des communautés auprès de ses services. Nécessite de proposer des **API**.
Le Big Data est la capacité à traiter d'énormes volumes de données. Peu d'applis produisent de la Big Data. Par contre, des applis peuvent utiliser les données produites par les autres. Savoir consommer des Data sous forme de **Stream**. Comparaison avec les vidéos (téléchargement vs streaming).
L'application doit pouvoir commencer à traiter les données au fur et à mesure qu'elles arrivent. La méthode getBody() de l'API Servlet renvoie tout le body.
Sur le Cloud on utilise du parallélisme pour profiter des capacités du serveur.
Les 4 piliers en temps réel.

Comment on peut monter en charge ?
1. En parallélisant avec des Future et des Promises
2. Résilience : gérer les erreurs comme les cas nominaux
3. Limiter et ne pas bloquer les IO

Exemple de Code Story 2013 : calcul d'un planning de voyage optimal de 50000 trajets. Il fallait commencer par trier les vols.
Approche basique : on attend la fin de la réponse http qui arrive dans une liste puis on utilise un SortedSet.
Approche réactive : création d'un Stream (Iteratee en Play Framework), les données sont ajoutées directement dans un SortedSet.

Réagir aux évènements

Développement d'applis avec un système producteurs / consommateurs :

- Faible couplage
- Meilleure utilisation de ressources

Travailler avec des flux d'évènements, potentiellement infinis.

Cas d'utilisation : appli qui logge toutes les actions effectuées par les utilisateurs.
Exemple en JS à l'aide de **bacon.js** :

- Pattern de type « Observable »
- Permet de produire et travailler avec des flux d'évènements en JS
- Apporte des patterns de programmations fonctionnelles

```
var pageStream = $('.links a').asEventStream('click')
```

3 évènements click, keyup, unload, mappé, mergé et onValue posté sur le serveur.

Bacon permet d'envoyer des évènements en mode « push ».

Conclusion

Beaucoup de nouveaux frameworks permettant de faciliter l'approche Reactive.
Le regain d'intérêt de la programmation fonctionnelle est en partie liée à la possibilité de faire du Reactive.
Librairie Rx : Reactive Extension existe en .NET, Java et Scala.
Pas besoin d'apprendre Scala ou Clojure pour faire du Reactive.

Questions / Réponses

1. Lorsque les drivers JDBC n'offre pas de mécanisme asynchrone, c'est difficile.
MongoDB et Redis offrent des drivers asynchrones.
Plugin asynchrone pour MySQL.
En attente d'Oracle ?

2. Risque de contention du nombre de threads ? Aucun thread n'est pas en attente. Pas plus de multiplications de threads.