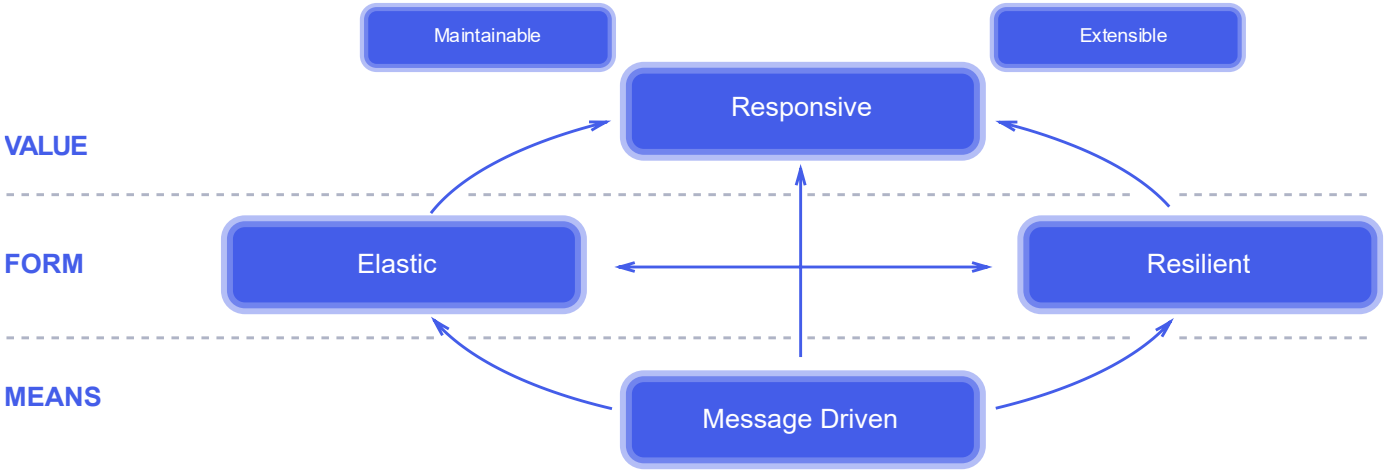
Reactive programming avec Spring Webflux

Pourquoi la programmation réactive ?

Nous avons fait face à une montée en charge brutale, avec des contraintes temps réel sur des millions d’appareils connectés.

La programmation réactive permet de traiter des flux asynchrones sans bloquer, tout en maîtrisant la charge.

Le Manifest Reactif



Réactif/Responsive

- Répond rapidement et de manière cohérente
- Temps de réponse prévisibles (bornes fiables)
- Détection rapide des problèmes
- Qualité de service constante
- Renforce la confiance utilisateur

Résilient/Resilient

- Reste fonctionnel malgré les pannes
- Contient et isole les défaillances (bulkhead)
- Récupération automatique (retry, circuit breaker)
- Pas de propagation en cascade
- Expérience utilisateur préservée

Élastique/Elastic

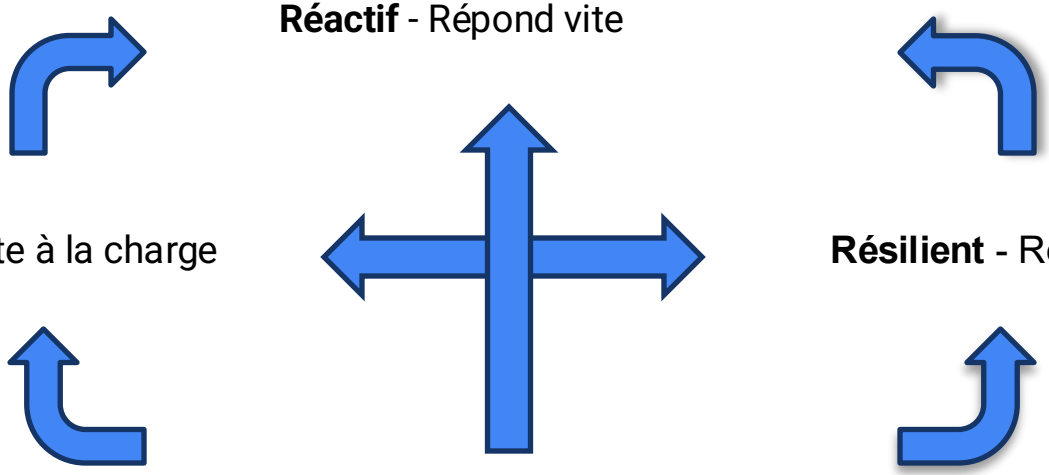
- S’adapte dynamiquement à la charge
- Scale up / scale down selon la demande
- Utilisation efficace des ressources
- Maintient les performances sous forte charge
- Coût optimisé

Orienté message/Message Driven

- Communication asynchrone via messages
- Couplage faible entre composants
- Isolation naturelle des services
- Supporte la distribution et la scalabilité
- Permet le backpressure

Le Manifeste Réactif

Réactif - Répond vite



Elastic - S'adapte à la charge

Résilient - Résiste aux pannes

Orienté message

Xavier Bouclet

- ~19 d’expérience en IT
- CTO / Technical Leader
- Co-organisateur de la conférence /dev/mtl
- Blogger https://xavierbouclet.com, Youtube @XavierBouclet
- Formateur Kotlin, Spring Boot

Java et les technologies réactives

La spécification Reactive stream a été publiée en v1.0.0 en **2015**.
Depuis plusieurs implémentation existe :

- Akka Stream: **2015** (Akka créé en 2009)
- Flow API: **2017**
- Projet Reactor: **2016** (créé en 2011)
- RxJava: **2016** (créé en 2013)
- Mutiny: **2019**

Projet Reactor

Project Reactor est une bibliothèque Java réactive qui permet de manipuler des flux de données asynchrones via deux types principaux :

- **Mono: 0..1**
- **Flux: 0..n**

Spring Webflux

Spring WebFlux est le framework web réactif de Spring, basé sur Project Reactor, permettant de créer des applications non bloquantes et asynchrones.

Spring MVC vs Spring WebFlux

Critères	Spring MVC	Spring Webflux
Modèle	**Thread-per-request** (bloquant)	**Netty: Event-loop** (non-bloquant) + réactif **Tomcat: Servlet Async** (thread switching) + asynchrone
Style de code	Impératif	Réactif (`Mono`/`Flux`) ou coroutines (Kotlin)
Scalabilité	Bonne, mais peut saturer en I/O lente (beaucoup de threads)	Excellente pour **forte concurrence** et **I/O lente**
I/O	I/O bloquante (JDBC, SDKs)	Optimisé pour I/O non-bloquante (WebClient, R2DBC)
Backpressure	Non natif	Oui
Complexité	Faible	Forte

Démonstration

Virtual Threads : game changer ?

Project Loom introduit les virtual threads en Java afin de simplifier la concurrence massive tout en conservant un modèle de programmation impératif.

Un virtual thread Java n'est pas lié à l'OS donc on peut créer plus de virtual thread que de thread. Il est géré par la JVM.

Il est donc possible de gérer plus de charge avec les virtual thread.

Alors... WebFlux a-t-il encore un intérêt ?

Oui, WebFlux a encore un intérêt lorsque le système ne doit pas bloquer pour être capable de gérer beaucoup d'éléments dans un court temps.

En WebFlux c'est pas juste notre code qui est non bloquant c'est toute la stack.

Comment décider entre MVC et WebFlux ?

Nombre d'événements / seconde	Stack
1 000	MVC
1000 -> 10000	MVC + virtual thread
> 10000	MVC + virtual thread ou WebFlux si temps de traitement > 50-100 ms
> 50000	WebFlux

Mais ...

5000 requêtes/seconde
Temps moyen de traitement = **200** ms
=> **1000** requêtes simultanées

MVC → **~1 000** threads nécessaires
WebFlux → **~quelques dizaines** de threads

Alors que ...

20000 requêtes/seconde
Temps moyen de traitement < **10** ms
=> **1000** requêtes simultanées

MVC → **~200** threads nécessaires
WebFlux → **~quelques dizaines** de threads

Règle d'or

Si la charge tient avec ≤ 2–3× le nombre de CPU en threads actifs, la programmation réactive n’est probablement pas nécessaire.

Retour d’expérience / pièges / mythes

- On ne devient pas scalable « par magie » avec WebFlux.
- Le plus gros piège : **blocage involontaire** dans un pipeline réactif.
- Debugguer Reactor est plus complexe (operator names, checkpoints).
- Les libs bloquantes sont encore nombreuses -> difficulté d’intégration.
- Virtual threads améliorent déjà fortement la scalabilité pour peu d’effort.

La programmation réactive n’accélère pas ton code, elle empêche ton système de s’effondrer sous la charge.

Conclusion

Questions ?

Merci

Ressources

- https://dev.to/jottyjohn/spring-mvc-vs-spring-webflux-choosing-the-right-framework-for-your-project-4cd2
- https://www.geeksforgeeks.org/blogs/spring-mvc-vs-spring-web-flux/
- https://www.baeldung.com/spring-mvc-async-vs-webflux
- https://medium.com/@dasbabai2017/spring-mvc-vs-spring-webflux-understanding-the-web-frameworks-in-spring-boot-e0cdb44419e2
- https://www.youtube.com/watch?v=2ficjW95tmo
- https://www.reddit.com/r/java/comments/1aldwzg/is_there_still_a_need_for_webflux/?rdt=64785
- https://stackoverflow.com/questions/46606246/spring-mvc-async-vs-spring-webflux
- https://www.infoq.com/presentations/servlet-reactive-stack/
- https://www.youtube.com/watch?v=-nW36K-3DSU
- https://nurkiewicz.com/2019/02/rxjava-vs-reactor.html
- https://www.javacodegeeks.com/2024/12/reactive-programming-in-java-project-reactor-vs-rxjava.html
- https://julien.ponge.org/posts/publication-performance-and-costs-of-reactive-programming-libraries-in-java/
- https://codingplainenglish.medium.com/spring-webflux-vs-virtual-threads-which-one-wins-in-2026-8bcd74de1ad3
- https://plus8soft.com/blog/virtual-threads-vs-webflux/
- https://medium.com/@reyanshicodes/i-stress-tested-spring-boot-with-1-million-requests-virtual-threads-vs-webflux-3b20bd598724