

Architectures Logicielles Modernes



Séance 3 — Le monolithe modulaire : des frontières entre les métiers, vérifiées par un test

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et Techniques, Université Cheikh Anta Diop de Dakar

<https://ebtoure.sn/cours/architectures-logicielles-modernes/chapitres/03-le-monolithe-modulaire.html>

Version du 11 octobre 2026



Version en ligne

Architectures Logicielles — Séance 3

Le monolithe modulaire : des frontières entre les métiers, vérifiées par un test

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et Techniques, Université Cheikh Anta Diop de Dakar

Objectifs de la séance

Contenu

L'hexagone du Lab 2 protège le métier de la technique — mais tous les métiers de SénResto vivent dans le *même* hexagone. Cette séance trace les frontières **entre les métiers** : les **bounded contexts**, matérialisés en **modules** dont **Spring Modulith** garde les frontières par un simple test, et qui collaborent par **appels d'API publique** ou par **événements internes**.

- Découper un domaine en **bounded contexts** — et savoir où passe le trait.
- Structurer un **module** : API publique au packaging racine, implémentation cachée dans `interne/`.
- Faire vérifier les frontières **par le build** : une violation = un test rouge.
- Choisir entre **appel direct** et **événement** — et publier vos premiers événements métier.

La frontière manquante

Un hexagone, cinq métiers

Le constat

Ouvrez **domaine/** : commandes, restaurants, livreurs, notifications y cohabitent librement — demain, le catalogue pourra fouiller les données du paiement sans que rien ne l'en empêche. L'hexagone protège le métier *de la technique* — pas les métiers *les uns des autres*.

- À 5 classes métier, ce n'est pas grave. À 50, c'est le God controller qui renaît — à l'échelle du domaine entier.
- La frontière qui manque n'est pas technique : elle est **métier**. Où finit « commander » et où commence « payer » ?

Le consensus industriel

« *Monolith first* » (Fowler) : on structure d'abord le monolithe en **modules à frontières explicites** — réversible, testable, et meilleur brouillon d'un futur découpage en services... si un jour il se justifie.

Bounded contexts

Le bounded context

Définition (DDD, version allégée)

Un **bounded context** est une zone du domaine à l'intérieur de laquelle les mots ont *un* sens précis et le modèle est cohérent. La frontière passe là où **le vocabulaire change de sens** — pas là où la technique le suggère.

Le test linguistique, sur SénResto

Que veut dire « commande » ? Pour **Commandes** : un panier, un client, un statut. Pour **Livraison** : une course, une adresse, un ETA. Pour **Paiement** : un montant à encaisser. Même mot, trois modèles — trois contextes.

À retenir

On ne découpe pas un domaine avec un couteau technique mais avec une oreille : **écouter où le langage change de sens.**

Les cinq contextes de SénResto

Module	Sa responsabilité	Son langage
catalogue	les restaurants et leurs plats	plat, prix, spécialité
commandes	le cycle de vie d'une commande	statut, quantité, montant
livraison	assigner et suivre les livreurs	course, position, ETA
paiement	encaisser (mobile money simulé)	transaction, encaissement
notifications	prévenir les clients	SMS, destinataire

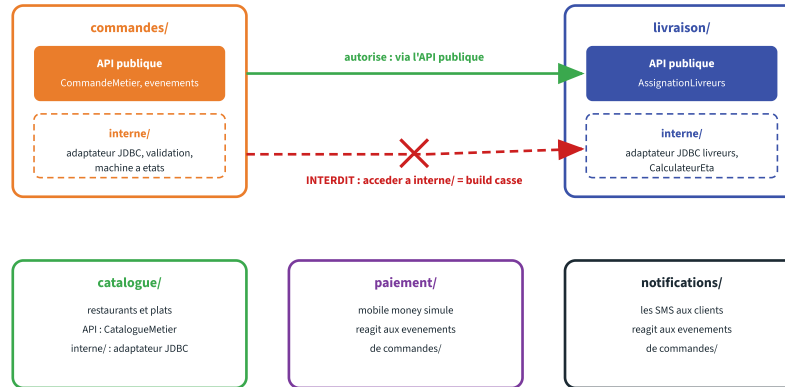
Ce n'est pas un hasard

Cinq contextes, cinq membres par groupe : au Lab 3, **chacun devient propriétaire d'un module** — de son API publique, de ses internes, de ses frontières.

Spring Modulith

Un module : une API publique, des internes cachés

Un module = une API publique (racine) + des internes caches



Cinq contextes, cinq modules, cinq membres de groupe. Un paquetage racine par module = son API publique ;
tout ce qui est dans interne/ est invisible des autres modules — et Spring Modulith le verifie par un test.

La convention Spring Modulith

La règle tient en deux lignes — c'est sa force :

```
sn/ucad/senresto/  
|-- commandes/          <- paquetage direct = MODULE "commandes"  
|  |-- CommandeMetier.java  <- racine = API PUBLIQUE du module  
|  |-- CommandeConfirmer.java <- les evenements aussi  
|  \-- interne/           <- INVISIBLE des autres modules  
|      \-- AdaptateurCommandesJdbc.java  
\-- livraison/           <- module "livraison", meme structure
```

- Chaque paquetage directement sous l'application est un **module** ; seuls les types de son paquetage *racine* sont accessibles aux autres.
- Pas de nouveau framework à apprendre : c'est **une convention de paquetages** — et un gardien qui la fait respecter.

Le gardien : la frontière vérifiée par un test

```
class ModulithArchitectureTest {  
  
    @Test  
    void lesFrontieresDesModulesSontRespectees() {  
        ApplicationModules.of(SenRestoApplication.class).verify();  
    }  
}
```

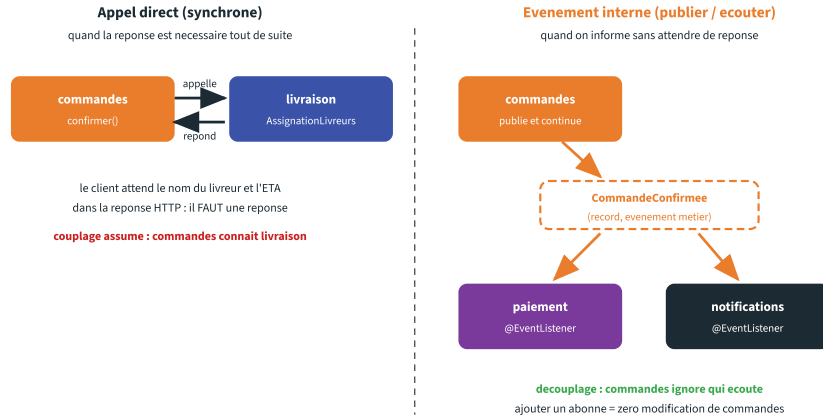
- Qu'un développeur importe l'**interne/** d'un autre module, et ce test devient **rouge** — le build casse, la violation est nommée, fichier et ligne à l'appui.
- L'architecture cesse d'être un dessin sur un slide : elle devient une **propriété vérifiée en continu**, au même titre qu'un comportement.

À retenir

La différence entre une convention et une architecture, c'est un **test qui échoue**. Vous en ferez l'expérience au lab : violation volontaire, build cassé, message lu, violation retirée.

Les événements internes

Deux façons de collaborer entre modules



Deux outils, deux usages — et au Lab 5, ces memes evenements traverseront le reseau via RabbitMQ.

Publier un événement métier

Un événement est un simple *record* — un fait passé, nommé au participe : *la commande a été confirmée*. Il vit dans l'API publique du module qui le publie :

```
// commandes/CommandeConfirmer.java (racine du module = public)
public record CommandeConfirmer(long commandeId, String numero,
    String clientTelephone, int montantTotal, String livreurNom,
    int etaMinutes) {}
```

```
// Dans CommandeMetier.confirmer(), a la place de l'appel aux notifications :
evenements publier(new CommandeConfirmer(id, numero, telephone,
    montant, livreurNom, etaMinutes));
```

- `commandes` annonce un **fait** et continue sa route : il ignore qui écoute, et n'attend rien de personne.

Écouter : les abonnés

Chaque module intéressé déclare un écouteur — dans ses murs :

```
// paiement/interne/EcouteurPaieement.java
@Component
class EcouteurPaieement {

    @EventListener
    void quandCommandeConfirnee(CommandeConfirnee evt) {
        System.out.println("[PAIEMENT] Encaissement mobile money simule : "
            + evt.montantTotal() + " FCFA pour " + evt.numero());
    }
}
```

- `paiement` est né **sans modifier une ligne** de `commandes` — et par défaut ces événements sont **synchrones** : le comportement observable ne change pas... pour l'instant.

Le pont vers la séance 5

Retenez ces records : au Lab 5, les *mêmes* événements traverseront le réseau via RabbitMQ, vers un service Python. Vous faites déjà de l'event-driven — dans un seul processus.

Ce que le modolith achète — et ce qu'il coûte

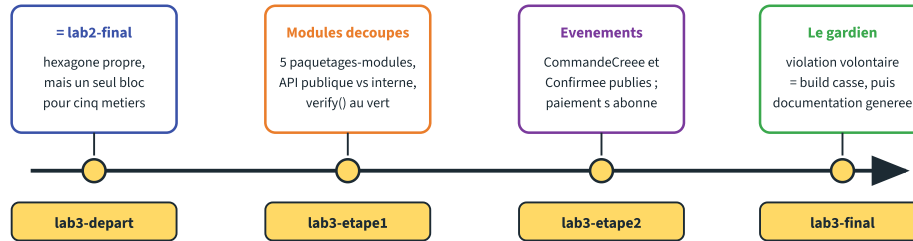
- **Achète** : des frontières métier vérifiées par le build ; un déploiement qui reste *simple* (un seul processus, une seule BDD, pas de réseau interne) ; des modules extractibles plus tard — le Strangler Fig du Lab 4 cueillera un module déjà découpé ; une documentation d'architecture générable depuis le code.
- **Coûte** : la discipline des paquetages ; des API publiques à concevoir avec soin ; la tentation permanente du raccourci (« juste cet import... ») — que le gardien transforme, précisément, en build cassé.

À retenir

Le monolithe modulaire offre **80,% des bénéfices d'organisation des microservices pour 20,% de leur coût opérationnel**. C'est le point d'équilibre moderne — et le meilleur point de départ pour la suite.

Le lab

Le chemin du Lab 3



Bloques ? git checkout du tag suivant : personne ne reste au bord de la route.

Point de départ : votre **lab2-final**. Les 8 tests (intégration + unitaires) restent verts — et un neuvième juge arrive : **verify()**.

Votre mission d'aujourd'hui

Rôle	Propriétaire de module au Lab 3
Le Doyen	<code>commandes</code> — le module qui publie ; merges, tags, ADR
L'Huissier	<code>livraison</code> — l'API <code>AssignmentLivres</code> , appelée en direct
Le Rédacteur	<code>catalogue</code> — et la documentation générée des modules
Le Notaire	<code>notifications</code> — les trois écouteurs, mêmes SMS qu'avant
L'Appariteur	<code>paiement</code> — le module né d'un événement, sans toucher aux autres

Le livrable individuel de la séance

L'Architecte du jour signe l'ADR n°3 : « **Comment tracer les frontières des modules ?** » — une page, deux alternatives sérieuses, deux conséquences négatives assumées.

L'essentiel de la séance

À retenir

- La frontière qui manquait n'est pas technique : elle est **métier** — et se trouve en écoutant où **le vocabulaire change de sens**.
- Un **module** = une API publique au packaging racine, des internes invisibles — et **Spring Modulith** transforme cette convention en **test** : violation = build cassé.
- Deux modes de collaboration : **appel direct** quand il faut une réponse (livraison), **événement** quand on informe (paiement, notifications) — et l'abonné naît sans toucher au publieur.
- Le modulith : 80,% des bénéfices d'organisation des microservices, 20,% de leur coût.

La question qui ouvre la séance 4

La direction veut faire évoluer le calcul d'itinéraires avec l'écosystème Python — et le déployer sans redéployer tout SénResto. Un module peut-il *quitter* le monolithe ? À quel prix ? Réponse au Lab 4 : l'extraction d'un microservice, pattern Strangler Fig.

Ressources de la séance

- **TP** · [Lab 3 — Le monolithe modulaire](#) (488 Ko)