



Séance 4 — L'extraction : un module quitte le monolithe et devient un microservice

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/04-l-extraction.html>

Version du 11 octobre 2026



Version en ligne

Architectures Logicielles — Séance 4

L'extraction : un module quitte le monolithe et devient un microservice

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

Le module **livraison** quitte le monolithe : il devient **livraison-service**, un microservice **FastAPI** en Python, extrait selon le pattern **Strangler Fig**. L'appel qui traversait la JVM traverse désormais **le réseau** — avec ses promesses (déploiement indépendant, écosystème Python) et ses dangers, que nous éprouverons en **provoquant une panne**. Au passage : **docker compose** orchestre nos quatre conteneurs, et une **carte Leaflet de Dakar** suit le livreur en temps réel.

- Savoir répondre à la vraie question : *quand* un microservice mérite-t-il son coût ?
- Extraire sans interrompre : le pattern **Strangler Fig**, rendu possible par le modolith.
- Mesurer ce que change un **appel réseau** — latence, pannes, et la troisième issue.
- Orchestrer avec **docker compose** ; visualiser avec **Leaflet** — zéro clé API.

Pourquoi extraire ?

La demande de la direction

Le scénario

La direction veut faire évoluer le calcul d'itinéraires : optimisation, trafic, géospatial — le terrain de **l'écosystème Python** (geopandas, OR-Tools). Et déployer ces évolutions **chaque semaine, sans redéployer** SénResto.

- Deux besoins qu'un monolithe, même modulaire, ne peut pas offrir : **une autre technologie** pour un module, et **un déploiement indépendant** — les deux bonnes raisons classiques, avec une troisième : le **dimensionnement** indépendant.

Les mauvaises raisons existent aussi

« C'est moderne », « Netflix le fait », « le CV »... Un microservice injustifié, c'est tout le coût du réseau sans aucun bénéfice. La charge de la preuve incombe *toujours* à l'extraction.

Pourquoi livraison — et pas un autre ?

Les critères d'un bon candidat à l'extraction, appliqués à nos modules :

Critère	livraison	contre-exemple
Frontière déjà nette	oui (Lab 3)	—
API étroite	3 opérations	<code>commandes</code> : au cœur de tout
Données peu partagées	les livreurs	<code>catalogue</code> : lu par tous
Raison technologique	Python géo	<code>notifications</code> : aucune
Raison de déploiement	évolutions fréquentes	<code>partage</code> : jamais

À retenir

On n'extraît pas « les modules » : on extrait **un** module, celui dont l'extraction a une raison d'être. Les autres restent — et c'est très bien ainsi. Votre ADR n°4 devra défendre ce choix *et* les non-choix.

Le Strangler Fig

L'idée simple : déménager sans fermer

La boutique du coin

Un boutiquier veut plus grand. Il ne ferme pas pour tout casser : il **construit la nouvelle boutique juste à côté**, puis déplace **un rayon à la fois**. Les clients achètent **tous les jours, sans rien remarquer**. Un rayon pose problème ? Il le **ramène**. L'ancienne est vide ? Il la ferme.

La boutique, mot pour mot, dans notre lab

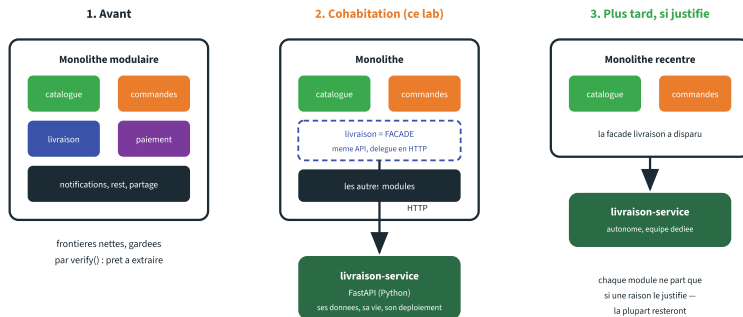
- la nouvelle boutique = `livraison-service` (Python), construit **à côté** du monolithe qui continue de tourner ;
- déplacer le rayon des boissons = basculer **un seul module** (livraison) vers le nouveau service ;
- les clients ne remarquent rien = l'application **fonctionne à chaque instant** ;
- ramener le rayon si problème = le **retour arrière**, gratuit (un profil Spring) ;
- fermer l'ancienne boutique vide = supprimer le code que **plus personne n'appelle**.

(Le nom officiel, *Strangler Fig*, vient d'un arbre. Retenez la boutique.)

Le figuier étrangleur

Définition (Martin Fowler, 2004)

Plutôt que réécrire d'un bloc (le « big bang », qui échoue presque toujours), on fait **pousser le nouveau autour de l'ancien**, on migre progressivement, et l'ancien meurt quand plus rien ne l'appelle. À chaque instant, **l'application fonctionne**.



Le figuier étrangleur : le nouveau pousse autour de l'ancien, qui continue de fonctionner à chaque instant.

Pourquoi pas le big bang ?

L'autre méthode — « on ferme, on reconstruit tout, on rouvre dans six mois » — échoue presque toujours. Trois raisons simples :

- **Six mois sans rien vendre** : pendant la réécriture, rien n'est livré — et les besoins, eux, continuent de changer.
- **Tout change le même jour** : si ça casse à l'ouverture, impossible de revenir en arrière.
- **On apprend trop tard** : les vrais problèmes se découvrent *après* avoir tout construit.

Les trois règles du figuier

1. Le nouveau se construit *à côté*, jamais *à la place*. **2.** Chaque bascule peut être *annulée* — chez nous : un profil Spring. **3.** L'ancien code n'est supprimé que quand *plus personne ne l'appelle*.

Le monolith paie ses dividendes

- Extraire **livraison** du God controller du Lab 1 aurait exigé des semaines d'archéologie. Aujourd'hui, la frontière est **déjà tracée**, l'API **déjà étroite** (**assigner**, **libérer**, **tous**), les données **déjà isolées**.
- La clé du lab : **AssignmentLivres** reste en place, **même API pour commandes** — mais devient une **façade** qui délègue en HTTP. **commandes** ne saura jamais que livraison a déménagé.

À retenir — la phrase du semestre

« *Monolith first, microservices when justified* » n'était pas de la prudence : c'était une **stratégie**. Le monolithe modulaire est le meilleur brouillon d'un système distribué — chaque frontière interne est une extraction en attente, qu'on ne paiera que si elle se justifie.

Traverser le réseau

Ce que change une flèche qui sort du processus

Hier : dans la meme JVM



un appel de methode :
~100 nanosecondes, ne rate jamais,
reussit ou leve une exception claire

Aujourd'hui : a travers le reseau



un appel HTTP : ~des millisecondes (10 000x plus),
et TROIS issues possibles :

1. reponse OK
2. reponse d'erreur (409, 500,...)
3. **PAS de reponse du tout (timeout, panne)**

— et dans le cas 3, l'assignation a-t-elle eu lieu ?
Impossible de le savoir. Bienvenue dans le distribue.

Le reseau n'est ni fiable, ni instantane, ni gratuit : chaque fleche rouge est une decision d'architecture.

Le lexique du distribué

Quatre mots que ce lab installe pour de bon :

Latence	le temps aller-retour d'un appel ; 100 ns en JVM, des millisecondes en HTTP — un facteur 10,000
Timeout	la durée maximale qu'on <i>accepte</i> d'attendre avant de déclarer l'échec ; toujours explicite , jamais infini
503	<i>Service Unavailable</i> : « je vais bien, mais quelqu'un dont je dépends ne répond pas — réessayez »
Couplage temporel	deux services qui doivent être vivants <i>au même instant</i> pour coopérer — la vraie dette du synchrone

À retenir

Un timeout n'est pas un aveu d'échec : c'est une **décision d'architecte** — combien de temps la confirmation d'une commande a-t-elle le droit de retenir un client ?

Les huit illusions du réseau

Peter Deutsch (Sun, 1994) a listé les huit croyances fausses que tout débutant du distribué paie un jour :

- | | |
|----------------------------------|-------------------------------|
| 1. Le réseau est fiable | 1. La topologie ne change pas |
| 2. La latence est nulle | 2. Il y a un administrateur |
| 3. La bande passante est infinie | 3. Transporter ne coûte rien |
| 4. Le réseau est sûr | 4. Le réseau est homogène |

Celles qui nous mordent aujourd'hui

La n°1 justifie nos timeouts et le 503 ; la n°2, notre réflexion avant chaque extraction ; la n°5, l'usage des *noms* de services dans compose plutôt que des adresses. Les cinq autres vous attendent en entreprise.

La troisième issue

Le vrai piège du distribué

Un appel de méthode a deux issues : il réussit, ou il lève une exception. Un appel réseau en a **trois** : réponse OK, réponse d'erreur... ou **pas de réponse du tout**. Et dans ce troisième cas, impossible de savoir si l'opération a eu lieu côté serveur — le livreur a-t-il été réservé ? Vous ne le savez pas.

- D'où les outils du distribué que ce cours introduit : **timeouts** (ne pas attendre éternellement), codes d'erreur traduits (503 quand le service est injoignable), et — au Lab 5 — l'asynchrone et l'**idempotence**.
- Règle d'hygiène : chaque appel réseau sortant a un **timeout explicite** et un **comportement d'échec décidé**. Jamais d'appel « optimiste ».

À retenir

La latence est **10,000 fois** supérieure et l'échec devient un cas *normal*. Un microservice, c'est un module qui a accepté ce contrat — c'est pour cela qu'on n'extraît pas sans raison.

Le contrat HTTP du nouveau service

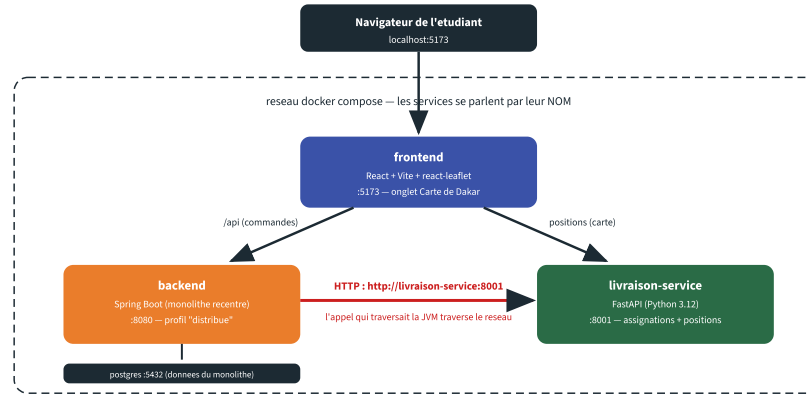
L'API du module devient une API HTTP — traduction presque mot à mot :

```
POST    /assignments    {"quartier": "Medina", "restaurant_quartier": "Ouakam"}
      200 {"livreur_id": 1, "livreur_nom": "Ibrahima Ndiaye", "eta_minutes": 43}
      409 {"erreur": "Aucun livreur disponible pour le moment, ..."}
DELETE  /assignments/1    # liberer le livreur 1
GET      /livreurs        # la liste (pour l'onglet Livreurs)
GET      /livreurs/1/position # {"latitude": ..., "longitude": ...} (carte)
```

- Même vocabulaire, mêmes règles, mêmes messages d'erreur : le contrat métier **survit au changement de langage**. Seule la syntaxe d'appel change.
- Le trajet est réel : livreur → restaurant → client. Le service possède **ses** données (les livreurs) : c'est le principe *database per service* — plus de table **livreur** consultée par le monolithe.

Conteneurs et carte

Quatre conteneurs, un fichier



Quatre conteneurs, un fichier docker-compose.yml — et une fleche rouge dont la panne fera tout le Lab 5.

docker compose : l'orchestre local

```
services:
  postgres:          # les donnees du monolithe
    image: postgres:16
  backend:           # Spring Boot, profil "distribue"
    build: ./backend
    environment:
      - SPRING_PROFILES_ACTIVE=distribue
      - LIVRAISON_URL=http://livraison-service:8001
  livraison-service: # FastAPI
    build: ./livraison-service
  frontend:          # React + la carte
    build: ./frontend
```

- Sur le réseau compose, **le nom du service est son adresse** : `http://livraison-service:8001` — pas d'IP, pas de configuration réseau manuelle.
- Un seul geste pour tout l'orchestre : `docker compose up --build`. Et un seul pour la panne de tout à l'heure : `docker compose stop livraison-service...`

La carte : Dakar en temps réel

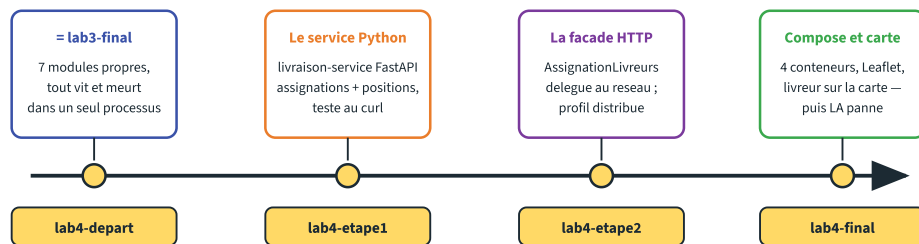
- **Leaflet** + OpenStreetMap : la cartographie libre, **zéro clé API**, zéro compte — [react-leaflet](#) l'intègre à notre frontend en un composant.
- Le nouveau service expose la **position simulée** du livreur (interpolation restaurant → quartier du client au fil de l'ETA) ; le frontend l'interroge toutes les 3 secondes : le marqueur *avance sur la carte de Dakar*.
- Ce n'est pas un gadget : c'est la **démonstration du bénéfice** — cette fonctionnalité vit entièrement dans le nouveau service, développable et déployable sans toucher au monolithe. La promesse de l'extraction, à l'écran.

En pratique au lab

L'onglet « Carte » montre les restaurants (marqueurs fixes) et le livreur de la commande suivie (marqueur mobile). Quartiers, coordonnées : tout vient de [QuartiersDakar](#) — désormais propriété du service Python.

Le lab

Le chemin du Lab 4



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

Point de départ : votre **lab3-final**. Et en clôture, une expérience dont le Lab 5 fera son affaire : **docker compose stop livraison-service**.

Votre mission d'aujourd'hui

Rôle	Pilotage au Lab 4
Le Doyen	merges, tags, ADR — et l'expérience de la panne au vidéoprojecteur
L'Huissier	les profils Spring (monolithe / distribué) et docker compose
Le Rédacteur	l'onglet Carte (react-leaflet, polling des positions)
Le Notaire	la façade HTTP : AdaptateurLivraisonHttp (RestClient, timeouts)
L'Appariteur	le service FastAPI : assignations, positions, données livreurs

Le livrable individuel de la séance

L'Architecte du jour signe l'**ADR n°4** : « **Pourquoi extraire Livraison — et pourquoi pas les autres ?** » — le premier ADR qui doit défendre des *non-décisions*.

L'essentiel de la séance

À retenir

- On extrait un module pour **trois raisons** : technologie, déploiement indépendant, dimensionnement — jamais pour la mode. La charge de la preuve incombe à l'extraction.
- Le **Strangler Fig** : le nouveau pousse autour de l'ancien, l'application fonctionne à chaque instant — et le monolith avait déjà tracé la frontière.
- Un appel réseau a **trois issues**, coûte 10,000 fois plus cher, et exige timeout + comportement d'échec explicites.
- **docker compose** : le nom du service est son adresse ; quatre conteneurs, un fichier, un geste.

La question qui ouvre la séance 5

En fin de lab, nous tuerons [livraison-service](#)... et la confirmation de commande mourra avec lui. Un service peut-il tomber *sans entraîner les autres* ? Réponse au Lab 5 : l'asynchrone — RabbitMQ, et des événements qui attendent patiemment le retour des absents.

Ressources de la séance

- **TP** · [Lab 4 — L'extraction d'un microservice](#) (544 Ko)