ShopNow

Building a better tomorrow for buyers and vendors (especially vendors)

Part 3

- Quelle exigence permet de neutraliser la menace STRIDE identifiée ?

Spoofing	Implémenter le MFA pour les admins et utiliser des attributs de cookies sécurisés (HttpOnly, Secure).
Tampering	Utiliser des requêtes préparées contre l'injection SQL et la signature HMAC pour les requêtes sensibles.
Repudiation	Activer des logs horodatés, signés et immuables pour tracer chaque action utilisateur.
Information	Forcer le TLS 1.3 en transit et le chiffrement AES-256 au repos pour les données clients.
Denial of Service	Appliquer du Rate Limiting (100~req/min/IP) et déployer un WAF.
Elevation of Privilege	Appliquer un RBAC strict, la séparation des rôles et l'utilisation de comptes DB minimaux.

- L’exigence est-elle mesurable, testable, vérifiable ?

. Mesurabilité (Indicateurs précis) Une exigence est mesurable lorsqu'elle utilise des seuils ou des configurations spécifiques.

Exemple du Rate Limiting : L'exigence de "100 req/min/IP" est directement mesurable en comptant les requêtes sur une période donnée.

Exemple du chiffrement : L'utilisation de l'algorithme "AES-256" ou du protocole "TLS 1.3" constitue une mesure technique exacte et non ambiguë.

Testabilité (Validation technique) Chaque exigence est associée à un critère d'acceptation qui permet de confirmer sa mise en œuvre par un test.

Test de sécurité (Penetration Testing) : L'exigence sur les requêtes préparées est testable en tentant une injection SQL (SQLi) : si l'input n'est pas concaténé, le test réussit.

Test de configuration : Pour le stockage des tokens, on peut tester si le flag HttpOnly est présent ; si le JavaScript ne peut pas lire le cookie, l'exigence est validée.

Vérifiabilité (Audit et Preuve) L'exigence est vérifiable si l'on peut prouver son état à tout moment, notamment via des logs ou des revues de code.

Preuve par les logs : L'exigence sur les logs immuables et horodatés permet de vérifier a posteriori chaque action sans doute possible (anti-répudiation).
Vérification de l'accès : Le respect du RBAC est vérifiable en contrôlant que les comptes avec des privilèges minimaux ne peuvent pas exécuter de fonctions "superuser".

- L’exigence est-elle proportionnée à l’impact métier

Protection des données critiques (RGPD et Réputation)

- Impact métier : La fuite des données clients (PII) a un impact Très fort (sanctions RGPD, perte de réputation).

- Proportionnalité : L'exigence de chiffrement AES-256 et l'utilisation de TLS 1.3 sont des mesures fortes mais nécessaires pour garantir la confidentialité de ces données.

Sécurisation des revenus (Finances)

- Impact métier : La fraude sur les paiements et la modification des commandes ont un impact Très fort (perte financière directe).

- Proportionnalité : L'exigence de signer les requêtes sensibles (HMAC) et d'utiliser des requêtes préparées est proportionnée car elle empêche la manipulation frauduleuse des montants et des transactions.

Continuité de service (Disponibilité)

- Impact métier : Les attaques de bots subies par le passé saturent l'API, provoquant une perte de revenus.

- Proportionnalité : Le Rate Limiting (100~req/min/IP) et le WAF sont des mesures adaptées pour maintenir la boutique en ligne sans bloquer les clients légitimes.

Contrôle des accès privilégiés (Contrôle total)

- Impact métier : La compromission d'un compte administrateur a un impact Très fort car elle permet un contrôle total et des actions destructrices.

- Proportionnalité : L'exigence de MFA obligatoire et de RBAC strict (séparation des rôles) est indispensable pour protéger les fonctions les plus sensibles du système.

- L’exigence introduit-elle un coût ou une complexité excessive ?

Utilisation de standards existants

La plupart des exigences reposent sur des bonnes pratiques de développement qui ne coûtent pas plus cher à implémenter si elles sont intégrées dès la conception (Security by Design):

- Requêtes préparées : L'utilisation de requêtes paramétrées pour éviter les injections SQL est une fonction native des frameworks modernes (Node.js/PostgreSQL) et ne rajoute aucune complexité au code.

- Attributs de cookies : Ajouter les flags HttpOnly ou Secure est une simple ligne de configuration qui ne coûte rien en temps de développement.

Automatisation et outils intégrés Le coût est limité par l'utilisation d'outils déjà présents dans l'architecture:

- TLS 1.3 : L'activation du protocole se fait au niveau du WAF ou du Reverse Proxy (C8), ce qui centralise la configuration sans toucher au code des applications.

- Rate Limiting : Cette fonction est souvent native dans les solutions de WAF ou de passerelles d'API, évitant de développer une solution complexe en interne.

Justification par le ROI (Retour sur Investissement) Le coût de mise en œuvre est largement inférieur au coût potentiel d'un incident:

- MFA pour les admins : Bien que cela ajoute une étape de connexion, le coût d'une licence ou de l'intégration est minime par rapport au risque de compromission totale du système par un compte admin volé.

- Chiffrement AES-256 : L'impact sur les performances est aujourd'hui négligeable grâce aux processeurs modernes, tandis que les amendes RGPD en cas de fuite de données (D1) représenteraient un coût financier majeur.

Gestion de la complexité via le Backlog Pour éviter de surcharger les équipes, la complexité est gérée par la priorisation:

- Seules les exigences critiques (Priorité Haute) sont traitées immédiatement.

- Les mesures plus complexes (comme les logs immuables ou le Captcha adaptatif) sont planifiées pour plus tard afin de ne pas ralentir les livraisons produits.

- L’exigence respecte-t-elle les principes Zero Trust ?

Vérification explicite et continue

Le modèle Zero Trust exige que chaque accès soit authentifié et autorisé, peu importe d'où il vient.

- MFA et Authentification : L'exigence de MFA obligatoire pour les administrateurs et la détection d'anomalies sur l'API Auth garantissent que l'identité est vérifiée de manière forte avant tout accès.

- Validation des Tokens : L'exigence de vérifier la signature des JWT (RS256/HS256) à chaque requête assure que l'identité du porteur est valide en permanence.

Utilisation du moindre privilège Le système limite l'accès aux ressources au strict nécessaire pour accomplir une tâche.

- RBAC et Séparation des rôles : L'exigence de RBAC strict (rôles administrateur vs utilisateur) empêche une personne d'avoir plus de droits que nécessaire.

- Comptes DB minimaux : L'interdiction d'utiliser des comptes "superuser" pour le backend limite les mouvements latéraux en cas de compromission d'un service.

Segmentation et micro-segmentation On ne considère plus le réseau interne comme sûr. Les actifs sont isolés dans des zones de confiance.

- Isolation des zones : Les exigences renforcent l'isolation entre la Zone 2 (DMZ), la Zone 3 (Backend) et la Zone 4 (Données).

- Filtrage granulaire : L'utilisation du WAF et du Rate Limiting au niveau des points d'entrée (C8) permet de filtrer le trafic avant qu'il n'atteigne les ressources sensibles.

Assumer la compromission (Assume Breach) On conçoit le système comme si un attaquant était déjà présent sur le réseau.

- Chiffrement partout : Les exigences de chiffrement AES-256 (au repos) et TLS 1.3 (en transit) garantissent que même si un attaquant intercepte des flux ou accède au stockage, les données restent illisibles.

- Immuabilité des logs : L'exigence de logs signés et immuables permet de détecter et de retracer les actions d'un attaquant après une intrusion.

- Comment équilibrer sécurité et performance lorsque certaines exigences.

Optimisation des protocoles et du chiffrement

- Utilisation de TLS 1.3 : Ce protocole réduit la latence lors de la phase de "handshake" par rapport aux versions précédentes, tout en offrant un chiffrement plus robuste.

- Accélération matérielle : L'utilisation de processeurs modernes supportant des instructions spécifiques (comme AES-NI) permet d'effectuer le chiffrement AES-256 avec un impact négligeable sur le CPU.

Stratégies de mise en cache et Edge Computing

- Utilisation du Cache Redis (C4) : Placer les données de session et les paniers en mémoire permet de compenser le temps de calcul nécessaire aux vérifications de sécurité du backend.

- Déchargement (Offloading) sur le WAF/CDN : En effectuant la terminaison TLS et le filtrage des attaques au niveau du périmètre (Zone 2 - DMZ), on libère les ressources du backend pour la logique métier.

Architecture asynchrone et intelligente

- Signatures et vérifications : Pour les actions non critiques, les signatures peuvent être vérifiées de manière asynchrone ou via des "circuit breakers" pour éviter de bloquer l'utilisateur en cas de ralentissement.

- Filtrage adaptatif : Utiliser des solutions de sécurité (comme le Captcha adaptatif) qui ne s'activent qu'en cas de détection d'anomalies, évitant ainsi de ralentir les utilisateurs légitimes dans un flux normal.

Monitoring et ajustement continu

- Mesure de la latence : Il est impératif de mesurer la latence induite par chaque couche de sécurité (WAF, authentification) pour ajuster les seuils de Rate Limiting si nécessaire.

- Backlog Agile : Intégrer des tests de performance dans la "Definition of Done" pour s'assurer que l'ajout d'une exigence de sécurité ne dégrade pas les temps de réponse au-delà des limites acceptables

- Comment intégrer ces exigences dans un backlog Agile sans ralentir la livraison produit ?

Transformer les exigences en "User Stories" de sécurité

Ne traitez pas la sécurité comme un bloc à part. Intégrez-la directement dans les fonctionnalités sous forme de critères d'acceptation ou de stories techniques.

- Exemple : "En tant qu'administrateur, je veux me connecter avec un MFA pour protéger l'accès aux données clients."

Utiliser la "Definition of Done" (DoD) Incluez des standards de sécurité de base dans la DoD de l'équipe pour qu'ils soient appliqués à chaque ticket, sans avoir besoin de créer des tâches spécifiques.

- Exemples : "Toute nouvelle API doit avoir un Rate Limiting configuré" ou "Toute nouvelle requête DB doit être préparée".

Priorisation par le risque (Risk-Based Backlog) Utilisez l'analyse STRIDE pour prioriser les tâches.

- Traitement immédiat : Les exigences "Haute priorité" qui corrigent des menaces critiques (ex: vol de tokens, injection SQL).

- Traitement différé : Les exigences "Moyenne" ou "Basse" sont planifiées dans les futurs Sprints selon la roadmap produit.

Automatisation des tests (DevSecOps) Pour ne pas ralentir la livraison, automatisez la vérification des exigences dans la pipeline CI/CD.

- Vérification automatique : Scanner les dépendances, tester la configuration TLS 1.3, ou vérifier la présence des flags HttpOnly sur les cookies.

Équilibrer par la vélocité Plutôt que de tout livrer d'un coup, utilisez une approche itérative :

- Sprint N : Sécurisation des accès critiques (MFA Admin, HTTPS).

- Sprint N+1 : Renforcement de la traçabilité (Logs immuables).

- Continu : Rotation des secrets et revues RBAC.

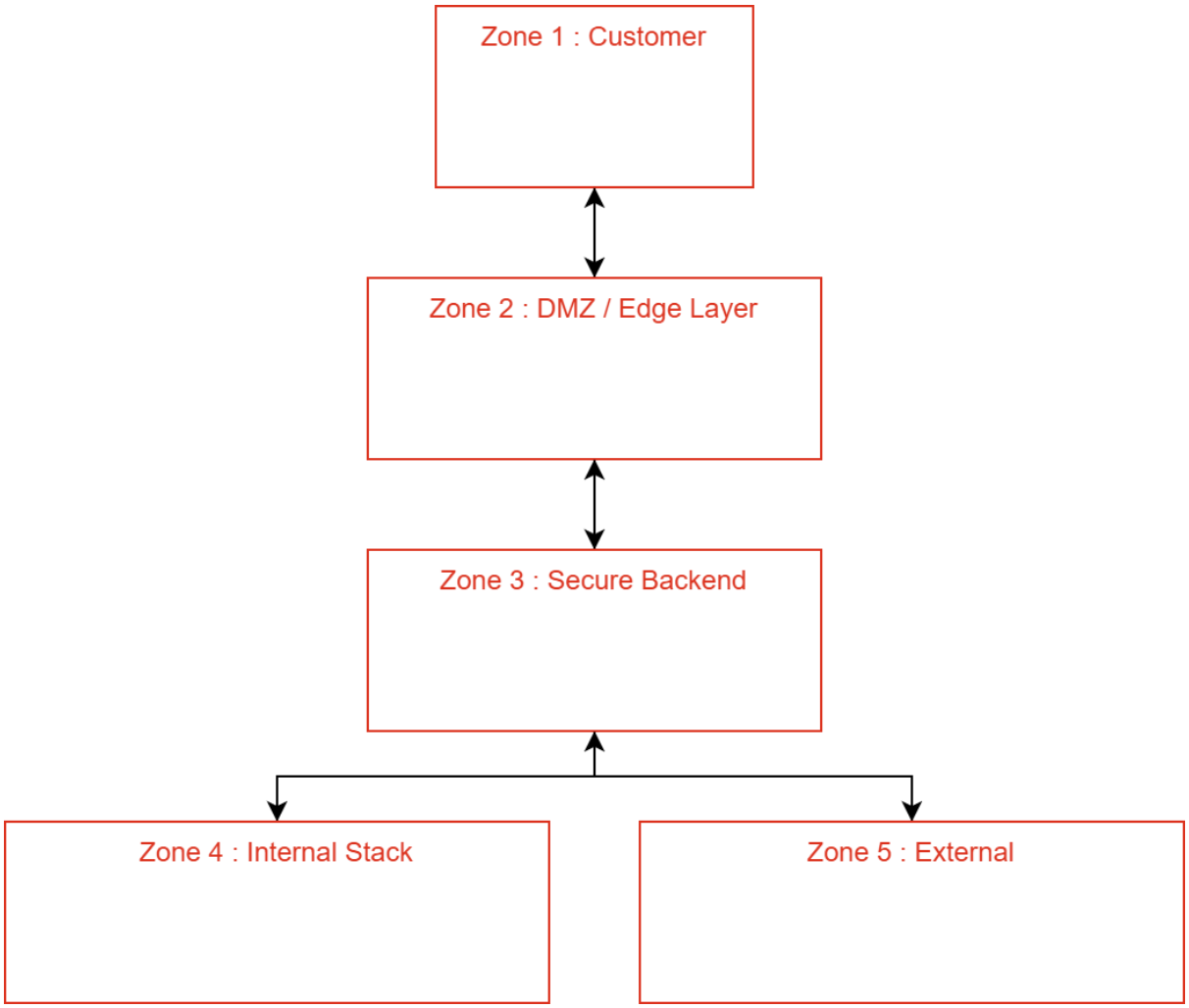
Part 4

On the exigency and feasibility of a thorough Zero-Trust enforcement company-wide.

One should always get a reminder of advanced security measures on a stack-level to avoid any major human-based failure. A leaked password ? Some backdoor ? Worry not, my brave sir, for thy counselor shall help you understand the principle of resource rights… and wrongs.

Resources isolation

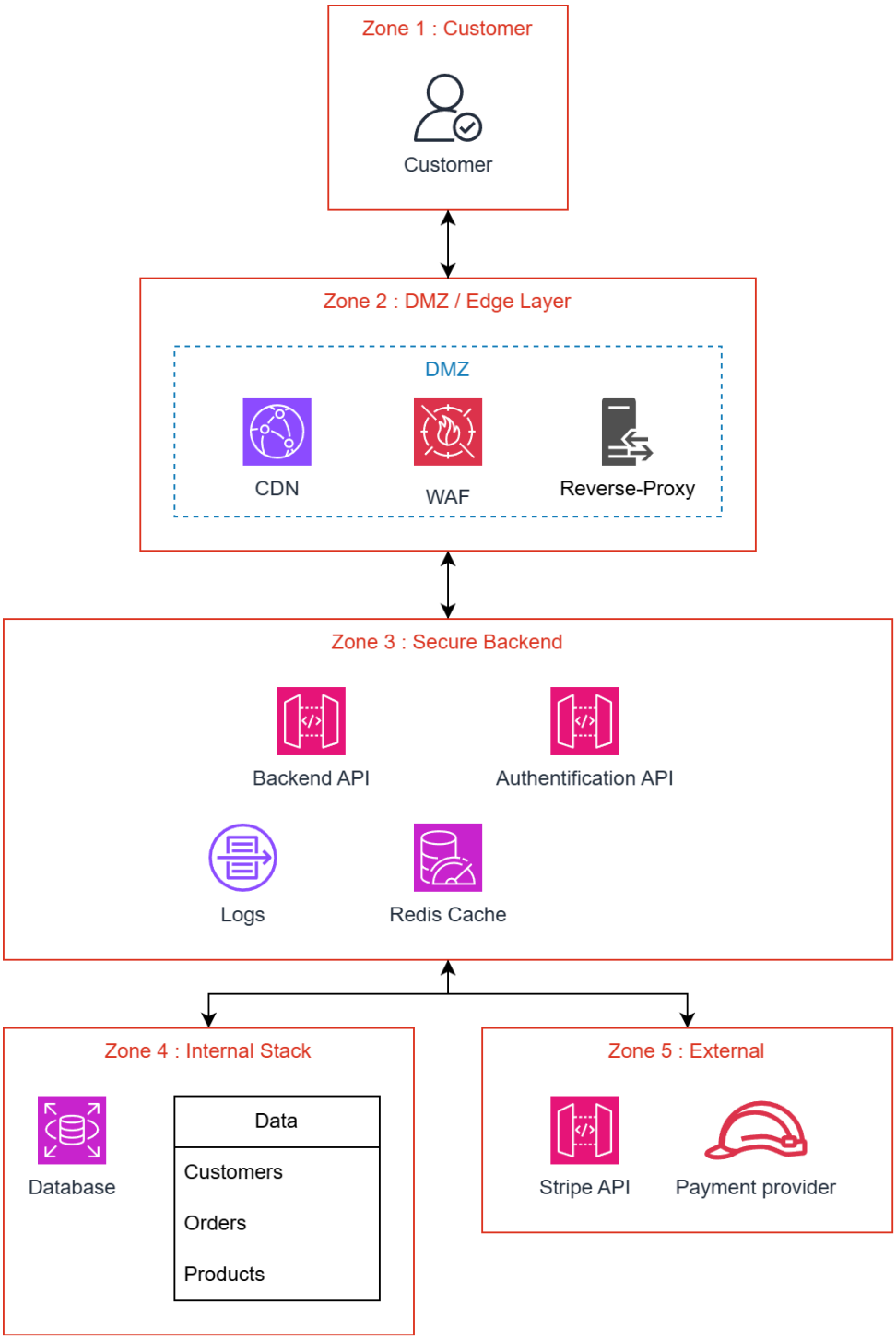
Every resource yields powerful capacities for the stack it's attached to.
Currently, the stack has no real permissions or policies enforcing security for resources. Considering the actual stack :



1. The resources are not clearly defined
2. Thus, the interactions between each resources inside zones are not defined

Then, a complete topology of the stack is necessary to better understand each one role and flow to better enforce policies.

There goes the updated stack flow as it is to this day :



- The flows between zones is to be DESTROYED to avoid zone-wide control of resources.
    - The only clear “global” flows existing is from the DMZ, led by the WAF handling protocols following and connectionst.
    - NO arrow should start from a zone and goes to another.
- Permissions have to be clearly defined and associated from a resource to another, each interaction having it’s own policy defined.

Less-privilege policy, stack-wide

Just as mentioned before, no resource should have more permissions than required. The REDIS cache is only useful as to tamper with the requests made for the backend database, and thus should only get a Read & Write access to the PostgreSQL instance.

An IAM policy would read like this :

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowRDSPostgresIAMAuthConnect",
      "Effect": "Allow",
      "Action": [
        "rds-db:connect"
      ],
      "Resource": [
     "arn::rds-db:::dbuser:Customers/<db-username>"
      ]
    },
    {
      "Sid": "AllowReadOnlyMetadataForTargetInstance",
      "Effect": "Allow",
      "Action": [
        "rds:DescribeDBInstances",
        "rds:DescribeDBClusters"
      ],
      "Resource": [
        "arn::rds:::db:Customers",
        "arn::rds:::cluster:Customers"
      ],
      "Condition": {
        "StringEquals": {
          "rds:DatabaseEngine": "postgres"
        }
      }
    }
  ]
}
```

This way, an IAM role could connect *on behalf* of the REDIS cache and do read & write operations, while only simple Read-only actions are permitted on the PostgreSQL tables.

Also, better security enforcement shall be placed with the concrete separation of Databases instances per scope. So instead of three tables (Customers, Orders, Products) in a signal instance, each table should be migrated to a dedicated (and scaled-down) instance.

Another example of the need of, for instance, the Backend API is too much monolithic and wields too much permissions for too many operations. Breaking down resources to correspond to the need of a single scope is a security isolation process to impose on a long-term vision.

Dynamic controls and monitoring

Dynamic controls and monitoring would be better off with an Infrastructure As Code configuration, with service roles, service users and service accounts dedicated for each major work assigned. **No general/master/omnipotent role, whatever the number needed.**

A finalized IaC work folder should look like this :

```
├── shopnow_infra/
│   ├── dmz/
│   │   ├── _locals.tf
│   │   ├── _providers.tf
│   │   ├── _regions.tf
│   │   ├── _variables.tf
│   │   ├── users.tf
│   │   ├── waf.tf
│   │   ├── cdn.tf
│   │   └── proxy.tf
│   ├── backend/
│   │   ├── _locals.tf
│   │   ├── _providers.tf
│   │   ├── _regions.tf
│   │   ├── _variables.tf
│   │   ├── users.tf
│   │   ├── api_authentication.tf
│   │   ├── api_backend.tf
│   │   └── db_redis.tf
│   ├── internal_stack/
│   │   ├── _locals.tf
│   │   ├── _providers.tf
│   │   ├── _regions.tf
│   │   ├── _variables.tf
│   │   ├── users.tf
│   │   ├── db_customers.tf
│   │   ├── db_orders.tf
│   │   └── db_products.tf
│   └── external_stack/
│       ├── _locals.tf
│       ├── _providers.tf
│       ├── _regions.tf
│       ├── _variables.tf
│       ├── api_stripe.tf
│       └── api_open_banking.tf
```

```
└── shopnow_api/
    └── backend_api/
        ├── nodejs/
        │   └── node-modules
        ├── main.js
        └── etc.js
```

This organization leads to instantly recognizable structure - infrastructure *per se* - and would lift off the burden of permissions applying and tracking.

Again, a dedicated monitoring solution to trace out actions in a centralized cluster for actors AND resources is to be adopted as soon as possible.

Actions traceability

Actions are to be tracked by two monitoring type.

SIEM

Gathering details about connections and actions should be monitored in details and analyse by a human - or to some extent by a trained AI agent - on another layer. Catching security liabilities would require an internal instance directed at it.

It would require the deployment of per-resource agents to leverage logs and metrics before any interpretation-layer control within the solution. Once deployed and configured to be free of impersonation and data poisoning, it should reveal critical to maintain current stack in the better conditions.

Logs monitoring

To leverage optional but useful metrics on the resources usage (connections per minutes, CPU and RAM loading, free storage space, etc) to better scale up or down resources, getting a monitoring solution to keep an eye on the operations would be an addition to implement, not as a security urgency, but as an operative urgency. One can’t pretend to run a smooth operation if one can’t tell if the machines are put to good use, mostly in the case of scaling up operations, as expected for an online shopping platform.

Best choices…

The best combo for optimum monitoring would be the usage of a SIEM and a monitoring solution. On-Premise, on-site controlled combo could be a Wazuh server and a Grafana monitor, in addition to any agent for logs and metrics deployed per-resource.

A unified SaaS solution would be Datadog, leveraging metrics AND logs in a single point, with automatic insights generated and alerts posting in the most common IM solutions (SMS, Whatsapp, Discord, Slack, Mattermost, etc).

Migration possibilities

Different migration possibilities for the refactorization are possible as for now :

Blue-green migration

(replacing an old stack to a new one but changing endpoints)

We construct, from scratch, the new stack and get it up and running alongside the existing one. It is only after a round of test with fake data, charge load, security attacks etc that the endpoints (so basically the FQDN of ShopNow) will be switched to not point to the reverse-proxy anymore but to a load-balancer inside a scalable cluster (either a docker-swarm solution on a single powerful instance for local shopping, or a kubernetes cluster for international ambitions).

Please note that the cost would be doubled for the time of the migration, and should only be applicable if the old stack is too much technically in-debt to do otherwise.

In-place migration

(updating a stack step by step without any service interruption)

Element by element, the old stack is migrated to the new one, letting an IaC project deploy new resources and permissions one by one, until the job is done. This is by far the most painful and lasting solution, as changing configuration while keeping the current resources up and running requires a lot of nerves to not break anything, the testing environment being scarce unless you do this on hours where shopping doesn’t happen. But the costs are the most controlled as no other team is required, and the resources deployed are not additional but updated.