santai SOFTWARE

L'ARCHITECTURE HEXAGONALE DÉMYTHIFIÉE :

PROTÉGEZ VOTRE CŒUR MÉTIER, PAS VOTRE BASE DE DONNÉES

SOMMAIRE

santai SOFTWARE

APRÈS LA RECETTE, L'ORGANISATION : LE VRAI DÉFI DU CHEF

santai SOFTWARE

INTRODUCTION
DE LA RECETTE À LA CUISINE CHAOTIQUE

Dans notre précédent livre blanc, nous avons forgé notre trésor : la recette parfaite. Grâce à l'Event Sourcing, nous avons appris à capturer l'essence de notre métier dans des **Agrégats** et des **Événements**, garantissant que chaque étape de notre logique est respectée à la lettre.

Mais une recette, aussi parfaite soit-elle, ne suffit pas à faire tourner un grand restaurant.

Imaginez la scène : c'est le "coup de feu". Les commandes affluent. Sans une cuisine bien organisée, c'est le chaos assuré. Un serveur (l'interface utilisateur) fait irruption pour demander si un plat est bientôt prêt, un fournisseur (un service externe) sonne à la porte de derrière, et le Chef (notre domaine) passe plus de temps à s'inquiéter de la marque du frigo (la base de données) qu'à cuisiner.

Face à ce chaos, notre premier réflexe est souvent de "structurer" en créant des zones : la salle pour les clients (**Présentation**), la cuisine pour la préparation (**Logique Métier**), et la grande réserve pour les stocks (**Accès aux Données**).

INTRODUCTION
DE LA RECETTE À LA CUISINE CHAOTIQUE

C'est là que réside l'erreur fondamentale.

Dans cette vision, tout est pensé en fonction de la réserve. On dessine les étagères et on catalogue les tiroirs (**le schéma de la base de données**) avant même de savoir quelles recettes le Chef voudra créer. La cuisine est alors contrainte par la structure de la réserve, et non l'inverse.

Et si l'on inversait la logique ? Si l'on construisait d'abord un **sanctuaire pour le Chef et ses recettes**, un espace inviolable et parfaitement isolé ?

C'est la promesse de l'**Architecture Hexagonale**. Ce n'est pas une complication, c'est une libération. Ce guide va vous montrer comment devenir l'architecte de cette cuisine parfaite.

BÂTIR LA CUISINE PARFAITE : LE SANCTUAIRE DU CHEF

santai SOFTWARE

LA MÉTAPHORE DE LA **CUISINE BIEN ORGANISÉE**

Oublions les couches superposées et pensons en termes de zones de responsabilité. Une cuisine d'excellence est un sanctuaire organisé autour de son élément le plus précieux : le Chef.

- **Le Sanctuaire (Le Domaine) :** Au cœur de tout, il y a le poste de travail du Chef et ses fiches de recettes. C'est ici que la valeur est créée. Le Chef est entièrement concentré sur son art : la validation des règles (la recette) et la création du plat (l'événement). Il ignore tout du monde extérieur.
- **Les Points de Contact (Les Ports) :** Le sanctuaire n'est pas une prison. Il communique avec l'extérieur via des points de contact standardisés et non négociables : le passe-plat pour recevoir les commandes et un carnet de besoins pour demander des ingrédients. Ce sont des contrats, des "interfaces".
- **Le Monde Extérieur (Les Adaptateurs) :** Tout le reste. Les serveurs, les applications de livraison, les fournisseurs, le plongeur, la réserve... Chacun a un rôle spécifique et interagit avec la cuisine UNIQUEMENT via les points de contact définis.

santai SOFTWARE

LES PORTS :
LES PORTES DE LA CUISINE

Les ports sont les contrats formels qui régissent toute communication avec le domaine. Ils se divisent en deux types, aussi simples que fondamentaux.

Port d'Entrée (Primaire) :
"Comment le monde extérieur parle au Chef ?"

C'est le passe-plat où arrivent les bons de commande. Le Chef ne voit qu'une intention claire : `PréparerPlatPourTable7`. Il ne se préoccupe pas de savoir si la commande vient d'un serveur, d'un site web ou d'une application. C'est l'interface par laquelle le monde extérieur **pilote** le domaine.

Exemple de contrat (code) :

```
interface ServiceDeCuisine {
  executerCommande(CommandePlat);
}
```

santai SOFTWARE

LES PORTS :
LES PORTES DE LA CUISINE

Port de Sortie (Secondaire) :
"De quoi le Chef a-t-il besoin du monde extérieur ?"

C'est la liste de besoins du Chef. Il n'écrit pas "Acheter la viande chez Mr. Dupont", mais "J'ai besoin de 200g de filet de bœuf". C'est une interface que le domaine **utilise** pour obtenir des services, mais dont l'implémentation concrète lui est inconnue. Le domaine est **piloté** par ce besoin.

Exemple de contrat (code) :

```
interface ReserveIngredients {
  trouverIngredient(nom, quantite);
}
```

santai SOFTWARE

LES ADAPTATEURS : LE MONDE RÉEL ET INTERCHANGEABLE

Si les ports sont les prises, les adaptateurs sont tout ce que l'on peut y brancher. Un adaptateur est un morceau de code qui fait le pont entre le monde extérieur et un port spécifique.

L'Adaptateur Primaire prend une requête du monde extérieur (une requête HTTP, un clic de bouton) et la **traduit** en un appel sur un port d'entrée. Le contrôleur de votre API web est un adaptateur.

L'Adaptateur Secondaire prend une demande d'un port de sortie et la **concrétise**. Votre classe qui dialogue avec la base de données SQLite pour trouver un ingrédient est un adaptateur. Une autre classe qui utilise une liste en mémoire pour les tests en est un autre.

La beauté de ce modèle ? **Les adaptateurs sont interchangeables.** Vous pouvez changer de fournisseur d'ingrédients (passer de SQLite à PostgreSQL) sans que le Chef n'ait à changer sa recette d'un iota.

LA RÈGLE D'OR :
TOUT MÈNE AU CHEF

Si vous ne deviez retenir qu'une seule chose de l'Architecture Hexagonale, ce serait celle-ci :

LA RÈGLE DE DÉPENDANCE

Le code du domaine ne dépend de RIEN d'extérieur. Tout le code extérieur (les adaptateurs) dépend du domaine.

En termes techniques, cela signifie qu'aucun fichier dans votre dossier "Domaine" ne doit jamais contenir une instruction `import` ou `using` qui fait référence à un framework web, une librairie de base de données, ou tout autre détail d'infrastructure.

Le domaine est le cœur stable, autonome et intemporel de votre application. L'infrastructure est la partie volatile et remplaçable qui gravite autour. Cette inversion du flux de dépendance est le secret d'une application robuste, flexible et infiniment plus simple à maintenir et à tester.

MISE EN PRATIQUE SIMPLIFIÉE : LE DOMAINE D'ABORD, LE RESTE ENSUITE

santai SOFTWARE

LE POINT DE DÉPART, **LA VRAIE AGILITÉ**

Oubliez tout ce que vous savez sur le démarrage d'un projet.

Pas de configuration de base de données. Pas de choix de framework web. Pas de serveur à lancer.

Notre point de départ est bien plus simple et puissant : un test.

Notre objectif est de prouver que la logique fondamentale de notre application fonctionne parfaitement, en totale isolation. Pour ce faire, nous avons besoin d'un contrat clair pour la persistance de nos événements. Ce contrat, c'est notre Port de Sortie, que nous avions déjà esquissé : l'interface `EventStore`.

Commençons par écrire ce contrat noir sur blanc.

```
// Le contrat que notre Chef utilisera
// pour enregistrer ce qui s'est passé.
interface EventStore {
  sauvegarder(evenements: DomainEvent[]): Promise<void>;
  recuperer(aggregateId: string): Promise<DomainEvent[]>;
}
```

C'est tout. C'est notre seule concession au monde extérieur pour l'instant.

L'ADAPTATEUR “INMEMORY” : **NOTRE OUTIL DE LIBERTÉ**

Maintenant que nous avons le contrat (`EventStore`), nous avons besoin de quelqu'un pour le remplir. Mais au lieu de faire appel à un "vrai" fournisseur (une base de données), nous allons créer notre propre commis de cuisine ultra-efficace, qui travaille directement depuis sa mémoire.

Voici l'adaptateur `InMemoryEventStore`. Son rôle est simple : il prétend être un système de stockage, mais se contente de garder les événements dans une simple liste en mémoire.

Extrait de code :

```
class InMemoryEventStore implements EventStore {
  private events: { [aggregateId: string]: DomainEvent[] } = {};

  async sauvegarder(evenements: DomainEvent[]): Promise<void> {
    const aggregateId = evenements[0].aggregateId;
    if (!this.events[aggregateId]) {
      this.events[aggregateId] = [];
    }
    this.events[aggregateId].push(...evenements);
  }

  async recuperer(aggregateId: string): Promise<DomainEvent[]> {
    return this.events[aggregateId] || [];
  }
}
```

Cet outil simple est la clé de notre agilité. Il nous libère de toute dépendance externe et nous permet de tester notre logique métier à la vitesse de l'éclair.

santai SOFTWARE

LE "AHA!" MOMENT : **DÉVELOPPER EN ISOLATION TOTALE**

Le décor est planté. Nous avons un contrat (`EventStore`) et une implémentation fictive (`InMemoryEventStore`). Nous pouvons maintenant faire ce qui compte vraiment : développer et valider la logique de notre Chef.

Regardez ce test. Il valide un scénario métier complet, sans jamais toucher à une base de données ou un réseau.

Extrait de code (un test unitaire/d'intégration) :

```
// TEST : Un client commande un plat qui existe.
test('devrait créer un plat lorsque la commande est valide', async () => {
  const eventStore = new InMemoryEventStore();
  const serviceCuisine = new CuisineService(eventStore);

  const commande = new CommanderPlatCommand({ platId: 'carbonara', tableId: 7 });
  await serviceCuisine.executer(commande);

  //    que le bon événement a été enregistré.
  const events = await eventStore.recuperer('carbonara-7');
  expect(events[0]).toBeInstanceOf(PlatPrepareEvent);
});
```

À cet instant précis, notre application est fonctionnellement correcte. Nous avons prouvé que nos règles métier sont justes, que notre logique est saine. Tout cela en quelques millisecondes, directement dans notre éditeur de code. C'est ça, la vraie agilité.

santai SOFTWARE

BRANCHER LE MONDE RÉEL **(SANS EFFORT)**

Notre cœur métier est sain, testé et validé. Il est temps de le connecter au monde réel. Grâce à nos ports, cette étape devient presque triviale.

Brancher la persistance réelle

Il suffit de créer un nouvel adaptateur, `SQLiteEventStore`, qui implémente la même interface `EventStore`. Le Chef ne verra aucune différence.

Extrait de code (Adaptateur SQLite) :

```
// Le "vrai" fournisseur pour la production.
class SQLiteEventStore implements EventStore {
  // ... logique SQL pour insérer et
  //     récupérer des événements ...
}
```

BRANCHER LE MONDE RÉEL **(SANS EFFORT)**

Brancher l'interface utilisateur

De la même manière, on crée un adaptateur pour notre API web. Il reçoit une requête HTTP, la transforme en objet `Commande` et la passe à notre `CuisineService`.

Extrait de code (Contrôleur API) :

```
// Port d'entrée
interface CommandeApplication {
  executer(commande: CommanderPlatCommand): void;
}

// Implémentation côté application
class CuisineService implements CommandeApplication {
  executer(commande: CommanderPlatCommand): void {
    const plat = new Plat(commande.details);
    this.repository.sauvegarder(plat);
  }
}

// Adaptateur Express
app.post('/commandes', (req, res) => {
  const commande = new CommanderPlatCommand(req.body);
  application.executer(commande); // On appelle le port
  res.status(202).send();
});
```

Le domaine, lui, n'a pas changé d'une seule ligne. Nous avons simplement branché de nouveaux périphériques.

SCHÉMA
"AVANT / APRÈS"

Le résultat : une clarté retrouvée.

En passant d'une architecture en couches classiques à une architecture hexagonale, nous ne faisons pas que réorganiser des dossiers. Nous transformons fondamentalement la manière dont les différentes parties de notre application interagissent, passant du chaos à l'ordre.

AVANT : La Cuisine Chaotique
Les dépendances vont dans tous les sens. Le Chef est interrompu en permanence. Changer un fournisseur implique de réorganiser toute la cuisine.

APRÈS : La Cuisine Parfaite
Le Chef est dans son sanctuaire. Les dépendances pointent toutes vers lui. Changer de fournisseur ou de système de prise de commande est aussi simple que de brancher un nouvel appareil.

PLUS QU'UNE ARCHITECTURE UN CHANGEMENT DE PRIORITÉS

santai SOFTWARE

LES BÉNÉFICES CONCRETS : **POURQUOI TOUT CE TRAVAIL ?**

Adopter l'Architecture Hexagonale n'est pas un exercice académique. C'est un investissement qui rapporte des bénéfices concrets, tangibles et durables à chaque étape de la vie d'un projet.

Testabilité Ultime

- Testez 90% de votre logique métier sans jamais lancer une base de données ou un serveur web.
- Des suites de tests qui s'exécutent en millisecondes, pas en minutes.
- Identifiez les bugs dans la logique pure, sans le bruit de l'infrastructure.

Flexibilité Radicale (À l'épreuve du futur)

- Votre technologie de base de données devient un détail. Passez de SQLite à PostgreSQL en écrivant un seul adaptateur.
- Changez de framework web sans réécrire votre domaine. Exposez votre logique via une API REST aujourd'hui, GraphQL demain, et un client lourd après-demain.

Clarté et Maintenance Simplifiée

- Les nouvelles règles métier s'ajoutent au cœur du domaine, à leur place logique.
- La nouvelle technologie s'intègre via un nouvel adaptateur, sans perturber le cœur existant.
- Accueillir un nouveau développeur devient plus simple : "Commence par comprendre le domaine. Le reste n'est que de la plomberie."

santai SOFTWARE

VAINCRE LA PEUR DE "L'OVER-ENGINEERING"

"C'est de l'over-engineering ! On n'a pas besoin de toute cette complexité pour une simple application CRUD."

C'est l'objection la plus courante, et elle est légitime. Mais elle repose sur une méprise. L'Architecture Hexagonale n'ajoute pas de la complexité, elle déplace la complexité là où elle est maîtrisable.

Comment convaincre votre équipe (ou vous-même) ?

1. **Commencez Petit.** N'essayez pas de refondre une application existante. Appliquez ce principe sur un **nouveau module**, un nouveau service. L'effort initial est faible et les bénéfices deviennent vite évidents.

2. **Montrez, ne Dites Pas.** Le meilleur argument est une démonstration. Montrez à quel point il est rapide de lancer les tests du domaine en isolation. Chronométrez-les. Montrez ensuite comment vous pouvez "brancher" un adaptateur en mémoire puis un adaptateur de base de données sans changer le code métier.

3. **Parlez le Langage du Risque.** L'argument final n'est pas technique, il est stratégique. "Cette approche nous **découple des choix technologiques**. Si notre base de données actuelle ne convient plus dans 2 ans, le coût du changement sera 10 fois moindre." C'est un argument de **pérennité et de réduction des risques** que tout décideur peut comprendre.

La vraie complexité n'est pas dans les quelques interfaces que l'on crée au départ. Elle est dans le code "spaghetti" d'une application de 3 ans où la logique métier, les requêtes SQL et le formatage HTML sont inextricablement liés.

santai SOFTWARE

OUVERTURE :
LA VUE D'ENSEMBLE

Notre cuisine est désormais un modèle d'organisation pour le **"coup de feu"**. Nous maîtrisons parfaitement le chemin de l'**Écriture (les Commandes)** : recevoir une commande, la valider contre nos recettes sacrées, et produire un plat.

Mais une question demeure...

Comment le client choisit-il son plat sur le menu ? Comment le gérant du restaurant visualise-t-il les ventes de la soirée sur son tableau de bord ? On ne va pas leur donner le grand livre des événements de la cuisine, ce serait illisible.

Pour répondre à ce besoin de **Lecture (les Queries)**, nous avons besoin de vues optimisées, de **projections** créées sur mesure à partir des événements.

C'est là qu'intervient le partenaire naturel de l'Architecture Hexagonale et de l'Event Sourcing : **CQRS (Command Query Responsibility Segregation)**. Le principe de séparer radicalement la manière dont on écrit les données de la manière dont on les lit.

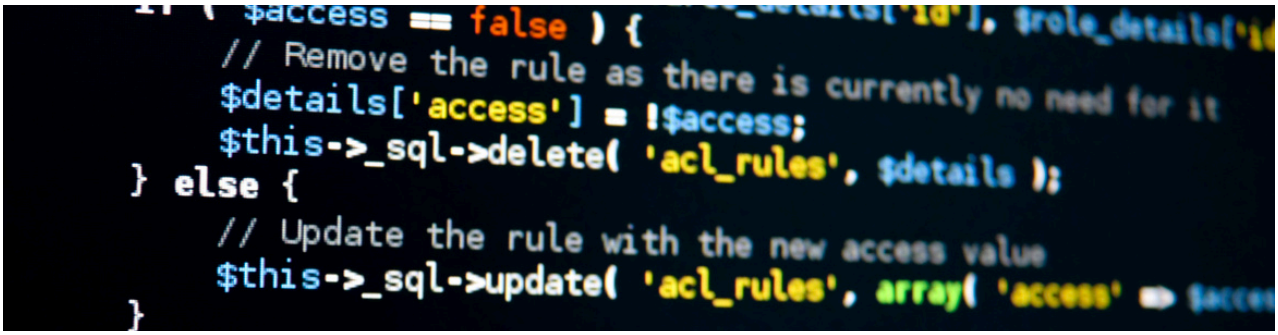
Dans notre prochain guide, nous ajouterons les tableaux de bord à notre restaurant et achèverons notre chef-d'œuvre architectural.

ET MAINTENANT ?

Merci d'avoir lu ce guide.

J'espère qu'il vous a convaincu que l'Architecture Hexagonale n'est pas un "délire d'informaticien", mais une approche pragmatique et puissante pour construire des logiciels de qualité.

Mon objectif est de déconstruire les mythes autour du Software Craftsmanship et de rendre ces pratiques accessibles à tous.



Nous sommes convaincus que le software craftmanship peut transformer votre entreprise.

Contactez-nous, et explorons ensemble les possibilités !

santai SOFTWARE