Knesh - Architecture

Architecture de l'application

> Nous proposons une architecture **3-tiers**, découpée en plusieurs couches logiques (Architecture en couches).

Pour répondre au besoin de **disponibilité**, nous recommandons une architecture avec des services déployés dans des environnements **Docker**, orchestrés par **Kubernetes**, ce qui permet un déploiement tolérant aux pannes. En effet, ces technologies permettrons un backup permanent et une remise en route rapide en cas d'erreur fatale.

L'architecture en 3 niveaux:

- Présentation (Front-end)
- Logic (Back-end)
- Data (Base de donnée)

L'avantage principal de cette architecture est la maintenabilité des services. Chaque couche peut être maintenu, améliorée, modifié, évoluée sans avoir d'impact sur les autres couches.

A lire

→ Architecture à 3 niveaux

Presentation tier

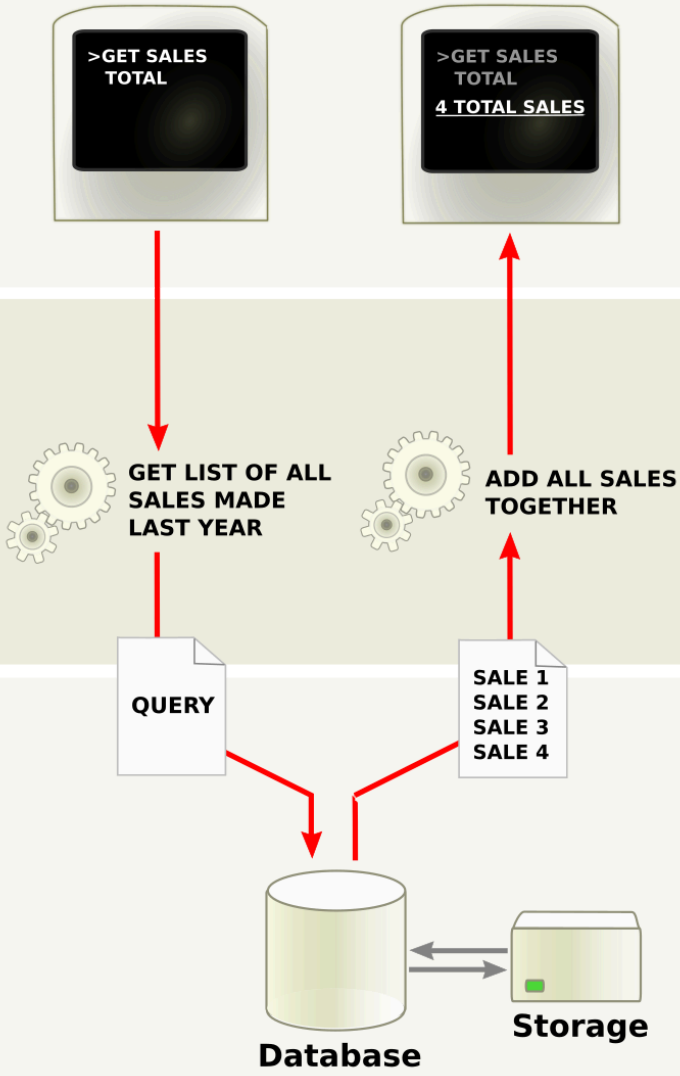
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.



Front-end

> Développement avec **Angular**.

Ce framework est un choix pertinent pour sa robustesse et sa large communauté, ce qui répond au besoin d'évolutivité et de maintenance. Angular est également conçu pour des applications à forte interactivité, garantissant ainsi une **rapidité** optimale dans l'affichage des interfaces.

> **A lire**
> → Benefices d'Angular

Back-end

> Développement avec **Java Spring Boot**.

Ce framework est un choix pertinent pour les applications robustes et évolutives. Spring Boot permet de construire des services *REST* rapidement. Sa forte communauté offre aussi Java Spring Boot comme une stack riche et stable pour un développement à long termes. C'est aussi la technologie principale maîtrisée par nos équipes internes.

> **A lire**
> → Bénéfices de Java - Spring Boot

Base de donnée

Une base de données relationnelle comme **PostgreSQL**. Ce *SGBD (Système de Gestion de Base de Données)* est fiable et performant, garantissant la gestion efficace des transactions tout en permettant une évolutivité à long terme. C'est aussi le cas des autres type de base de donnée mais **PostgreSQL** a quelques avantages comme un planificateur de requêtes performant et une license MIT plus permissive que la license GPL de **Mysql** par exemple.

> **A lire**
> → Comparaison PostgreSQL vs MySQL

Librairies

Tests

Front-end

> ⭐ En front-end, nous pouvons tester deux aspects.
>
> 1. Le end-to-end (E2E) qui aide tester le *métier* de l'application en *mockant*, entre autres, les dépendances, routes et endpoints de l'api. Par exemple, nous pouvons tester la sécurité de l'affichage dépendant des droits de l'utilisateur connecté.
> 2. L'affichage pur, pour vérifier, entre autres, que le *theming* et la présence et l'absence des éléments fonctionnent correctement.

Pour les tests unitaire, nous utiliserons **Jest**. Grâce à sa librairie de méthodes performantes, nous pourrons tester les services de l'application pour s'assurer du bon fonctionnement de l'authentification ou du traitement des données récupérées, par exemple.

Aussi, nous proposons d'utiliser **Cypress** pour les tests end-to-end. Cypress est une librairie reconnue pour sa simplicité et sa rapidité dans les tests automatisés sur les interfaces Angular.

Il permet de simuler des interactions utilisateurs, de vérifier les flux et de garantir que l'expérience client soit optimisée, répondant ainsi au besoin de **rapidité** et de **disponibilité**. De plus, sa compatibilité avec Angular est bien documentée, garantissant un environnement facilement maintenable.

> ℹ **A lire**
>
> → Courte présentation Cypress

Back-end

Pour les tests sur le back-end Java, **JUnit** est la solution de choix.

Utilisé massivement dans l'écosystème **Java**, **JUnit** est mature, bien documenté, et largement adopté, ce qui facilite le travail des développeurs futurs.

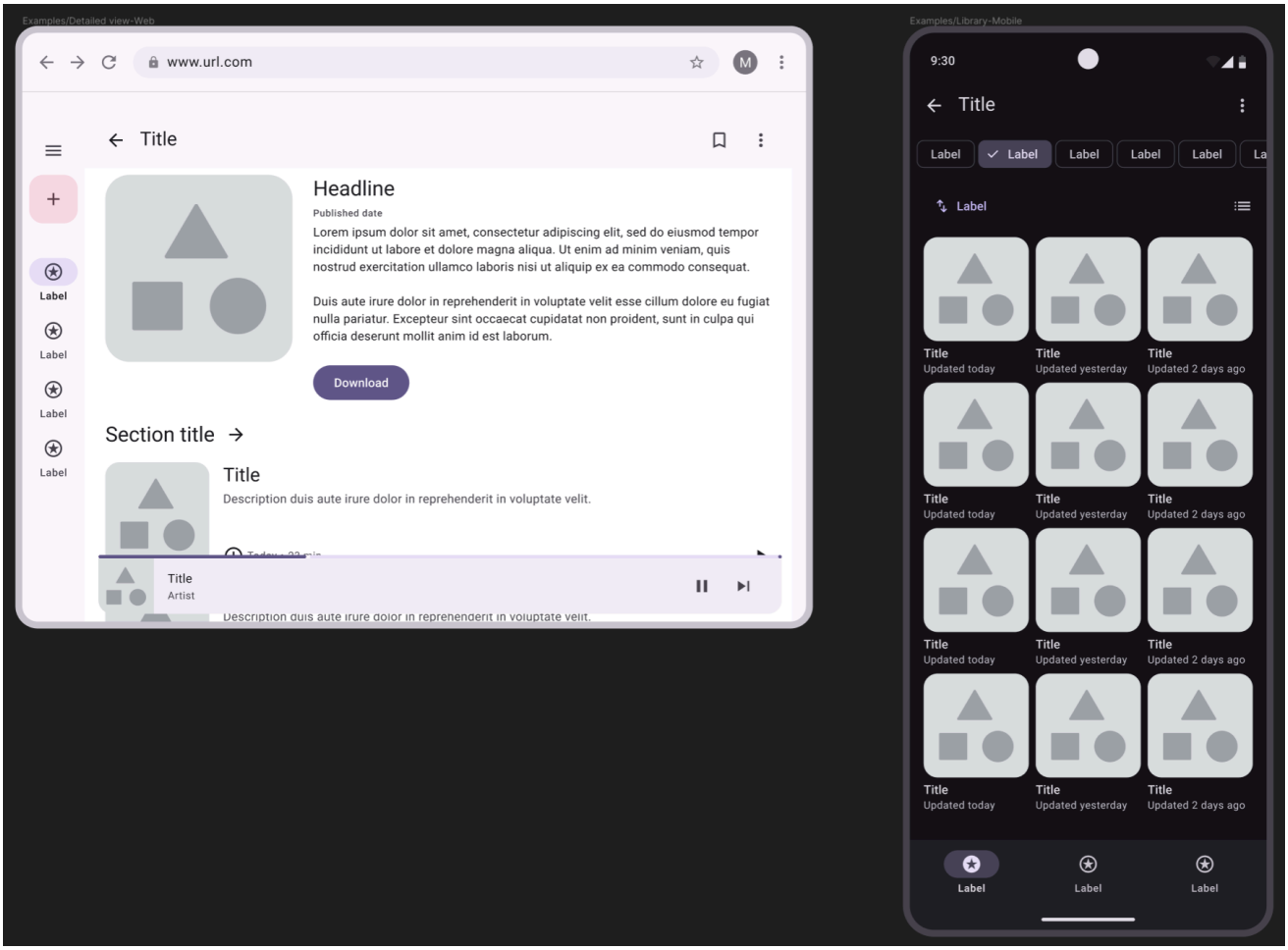
Mockito sera utilisé en parallèle pour le mock des dépendances externes, permettant de réduire le temps des tests unitaires tout en isolant les composants testés.

> **A lire**
>
> → Présentation JUnit

Ui - Composants visuels

Pour accélérer le développement et réduire les coûts, nous préconisons l'utilisation de **Angular Material**, une librairie de composants UI réutilisables.

Elle fournit des composants visuels modernes et optimisés qui suivent les pratiques de **Material Design** de Google, ce qui garantit des interfaces intuitives et faciles à utiliser (répondant ainsi au besoin de **rapidité**). **Angular Material** est également bien intégré dans l'écosystème Angular, ce qui simplifie la gestion des mises à jour et garantit une **évolutivité**.



> **A lire**
>
> → Démonstration des composant

Paradigmes de programmation

Front-end

Le paradigme principal utilisé dans **Angular** est la **programmation réactive** avec **RxJS**.

Ce paradigme est essentiel pour gérer des flux de données asynchrones de manière efficace, réduisant ainsi les temps de latence et garantissant la **rapidité** des interactions. En pratique, cela permet de réagir aux événements utilisateurs de manière fluide et sans dégradation de performance, améliorant ainsi l'expérience utilisateur.

De plus, **Angular** suit un paradigme de **composants modulaires**, favorisant la réutilisation et la maintenance du code. Chaque composant est indépendant et peut être réutilisé dans plusieurs parties du projet, ce qui permet de minimiser les efforts de développement futurs et favorise une **évolutivité**.

L'évolution facilité et favorisé par cette technologie voit donc son coût s'amoindrir. Ses composant modulaire font en sorte que l'évolutivité de l'un n'impact pas forcément l'autre. Faire évoluer quelques chose ne casse donc pas autre chose. On peut donc en conclure que la rapidité de développement est gagnante pour les mêmes raisons.

Sur un plan plus technique, les technologies comme le **HMR** (Hot Module Reload), qui recharge automatiquement la compilation du code, permet une aisance et un environnement de travail qui favorise la rapidité de développement pour les développeurs.

> **A lire**
>
> → Pourquoi utiliser RxJs

Back-end

Du côté back-end, **Java Spring Boot** suit un paradigme de **programmation orientée objet** (POO) avec une forte utilisation des **injections de dépendances**. Cela permet une bonne séparation des responsabilités et améliore la lisibilité et la maintenance du code. Ce paradigme garantit également une structure modulaire, facilitant les ajouts futurs.

En parallèle, le paradigme **RESTful** est utilisé pour la conception des API, favorisant l'interopérabilité avec d'autres services et systèmes. Grâce à cette approche, nous garantissons une bonne évolutivité, une facilité de communication, ainsi qu'une intégration simple avec des services tiers.

> **A lire**
>
> → Présentation RESTful

Conclusion

Avec cette proposition et le choix des technologies, nous répondons aux 4 besoins exprimés:

- Disponibilité
- Rapidité
- Coûts
- Évolutivité

En capitalisant sur des technologies éprouvées et largement adoptées telles qu'**Angular**, **Java Spring Boot**, et **PostgreSQL**, tout en intégrant des outils modernes pour le développement, nous assurons un projet stable, performant, et facile à maintenir sur le long terme.