POC REPORTING

- MedHead Consortium -

David EVAN

23/06/2023

Version 1.0

Emergency Responder System – MedHead

Historique des révisions

Numéro de version	Auteur	Description	Date de modification
1.0	EVAN David *(Architecte logiciel)*	Livraison initiale	23/06/2023

Tableau 1 - Historique des révisions

Objectif du document

Dans le cadre du PoC développé pour le système Medheard ERS, Ce document de reporting a pour double objectifs de présenter les justifications pour des choix (technologiques ou d'architecture) retenus pour la PoC et de synthétiser les résultats obtenus.

Il peut servir de base pour la rédaction des spécifications techniques complètes du futur produit.

TABLE DES MATIERES

CONTEXTE DU PROJET

Rappel du contexte

Dans le cadre du projet de développement d'une plateforme de traitement des urgences médicale pour le consortium MedHead, une PoC a été réclamée afin de valider la faisabilité technique et de définir les briques technologiques de référence pour la future plateforme.

La PoC présentée vise à démontrer la fiabilité et la rapidité de traitement de la future brique de « dispatching » des urgences médicale. Cette PoC est désignée sous le nom « MedHead ERS » tout au long de ce document.

L'ensemble des code sources et des ressources documentaires peuvent être consulté depuis le répertoire : https://github.com/medhead-ers.

Principes d'architecture

Les principes d'architectures auxquels se conforme la PoC sont décrit dans le document de définition des principes d'architecture.

Il est à noter que la déclaration de travaux précise que « ces principes sont assouplis pour les validations de principes et les mises en œuvre destinées à l'apprentissage. »

Exigences fonctionnelles et non fonctionnelles

Les exigences fonctionnelles et non fonctionnelles de la PoC sont décrites dans la déclaration de travaux et le document de définition de l'architecture.

IMPLÉMENTATION DE L'ARCHITECTURE

Architecture de l'application

Terminologie

Afin de simplifier et de standardiser le nom des composants, une terminologie commune a été utilisée. **Signification des abréviation utilisées** :

- **WEB** : Application Web - Application accessible depuis un navigateur web (depuis mobile ou desktop) dans le cas présent).
- **BSNS** : Business Service - Micro-service métier. Regroupe les règles métiers d'un service spécifique (ex : Gestion des patients ou des hôpitaux).
- **TRAN** : Transverse Service - Micro-service transverse (technique ou business). Peut être amené à exploiter plusieurs services métiers.

Vision d'architecture

Le schéma ci-après (*Figure 1*) présente la vision d'architecture sur laquelle s'appuie le développement du système MedHead ERS. *La description des différentes applications est fournie dans la section suivante.*

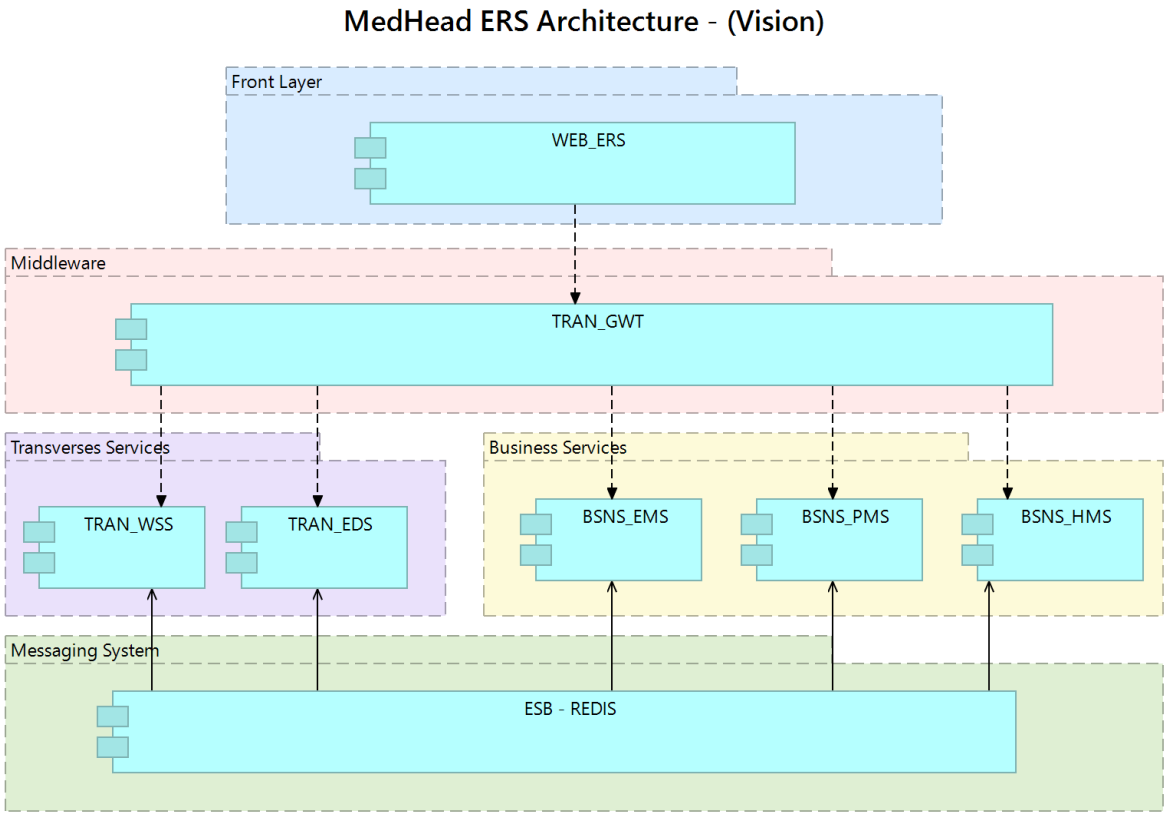

Figure 1 - MedHead ERS Architecture

Le système de dispatching d'urgence étant au centre de l'application, le schéma ci-après (*Figure 2*) présente un aperçu du processus métier de « déclaration » d'une urgence et de son fonctionnement.

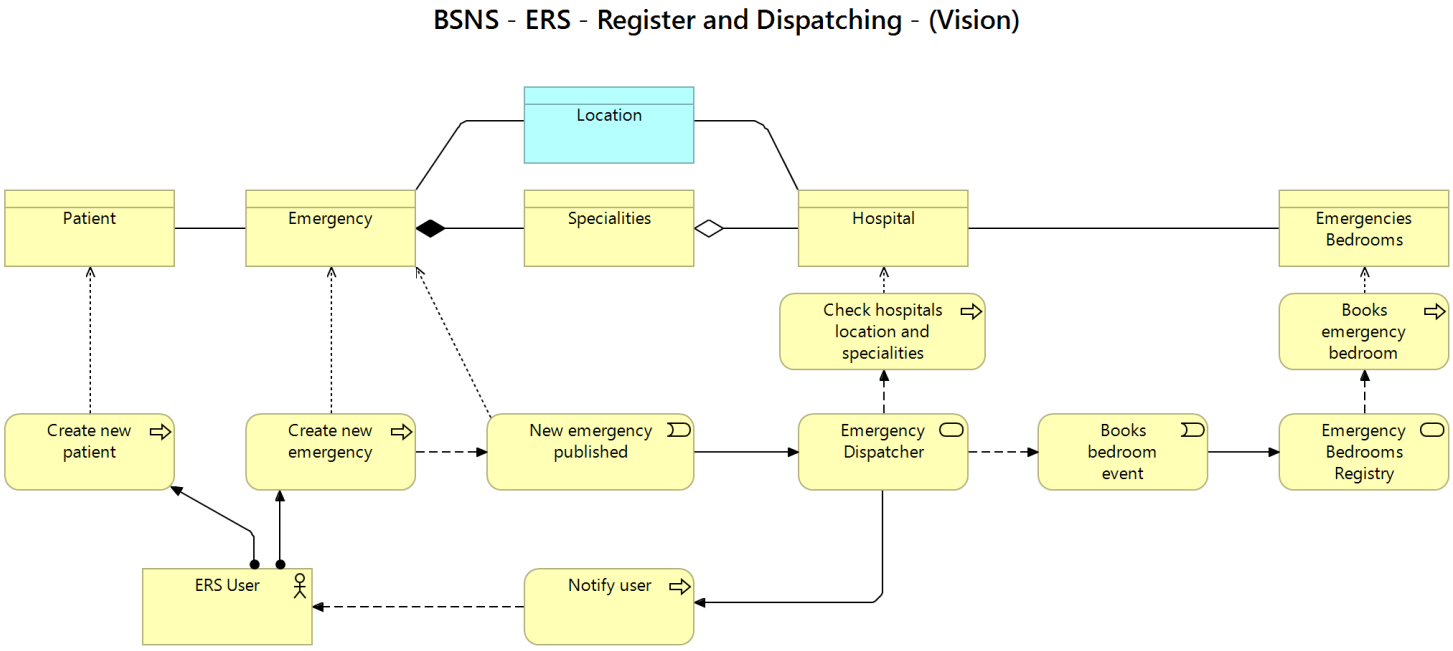

Figure 2 : Business Process - Register and dispatching an emergency.

Event Driven Micro-Service Architecture

Afin de satisfaire aux principes d'architecture, de réduire le couplage entre les services, d'améliorer la tolérance aux erreurs et de permettre une évolutivité rapide du système, une implémentation d'une architecture Event Driven a été choisie pour la réalisation du PoC MedHead ERS.

De manière simplifiée, chaque composant logiciel émet des évènements à chaque changement d'état (ex : modification d'une donnée). Ces événements peuvent être écoutés par d'autres composant logiciels qui vont réagir aux messages en effectuant les actions correspondantes. Dans le cadre du PoC, une implémentation « from scratch » a été décidé, basé sur quelques concepts standards :

- **Les messages** : sont formés sur une structure « commune » et sont publiés par les différentes applications sous une forme interopérable (JSON).
- **Les événements** : sont la traduction des « messages » lorsqu'ils sont attrapés par une application.
- **Les jobs Processor** : sont les fonctions qui vont traiter les événements (équivalent d'un contrôleur en Http).

Afin de ne conserver un code simple pour la PoC, il n'a pas été retenue l'utilisation des Event Spring et le pattern Observer pour le déclenchement des messages. Toutefois, cette approche semble justifiée pour le système final.

Implémentation retenue

Le tableau ci-après présente la liste des composants logiciels développés ou implémentés dans le cadre de la PoC :

ID Application	Nom Application	Description	Technologie
WEB_ERS	WebApp - ERS	IHM sous forme de WebApp SPA permettant d'accéder aux fonctionnalités de l'ERS.	Vue JS (SPA)
BSNS_PMS	Patient Management Service	Business service - Création et suivi des informations sur les patients.	Java / Spring
BSNS_EMS	Emergency Management Service	Business service - Création et suivi des urgences médicales.	Java / Spring
BSNS_HMS	Hospital Management Service	Business service - Suivi des informations sur les hôpitaux.	Java / Spring
TRAN_EDS	Emergency Dispatcher Service	Service Transverse - Dispatching et affectations des urgences médicales en fonction du contexte (localisation, spécialité médicale, disponibilités des hôpitaux ...).	Java / Spring
TRAN_WSS	WebSocket Server	Server WebSocket - Maintient une connexion ouverte avec l'application web en vue de la rendre "reactive" aux événements events	Javascript
TRAN_GWT	API Gateway - ERS (Traefik based)	API Gateway - Centralise, réécrit et transfère les requêtes aux différents micro-services. Assure l'authentification et les protections CORS.	Traefik
ESB_REDIS	Event Bus Redis	Composant Mock basé sur Redis qui permet de pub/sub de message.	Redis

Tableau 2 : Liste des composants applicatifs MedHead ERS

Technologies

Back-end

Micro-services

Afin de satisfaire aux exigences, les services back-end (micro-services) ont été construits sur une base **Java** (v19). Le Framework **Spring** a été retenu pour sa forte popularité et sa capacité à répondre aux besoins des différents composants logiciels (**Spring Boot** principalement, mais d'autre dépendance à Spring peut exister en fonction des besoins de chaque micro-services).

Le service EDS (Emergency Dispatcher Service) s'appuie sur une API tierce pour le calcul de la proximité d'une urgence par rapport aux hôpitaux : GraphHopper Matrix API. Afin de garantir une continuité dans le fonctionnement du dispatcher en cas d'indisponibilité de l'API tierce, un fallback basé sur un calcul trigonométrique a été implémenté.

Redis

Dans le cadre du PoC, une implémentation simplifiée de l'Event Driven a été retenue en se basant sur le système pub/sub de Redis.

Bien que totalement opérationnel dans un contexte « normal », il est fortement recommandé que ce logiciel soit remplacé par un outil de type Apache Kafka (ou éventuellement Rabbit MQ) qui permet d'ajouter des mécanismes de reprise aux messages non distribués (arrêt d'un service) et facilite la scalabilité des micro-services.

Traefik

Le logiciel Traefik a été retenu pour la construction du PoC afin d'assurer la fédération des différents micro-services (API Gateway) et sécuriser les flux http.

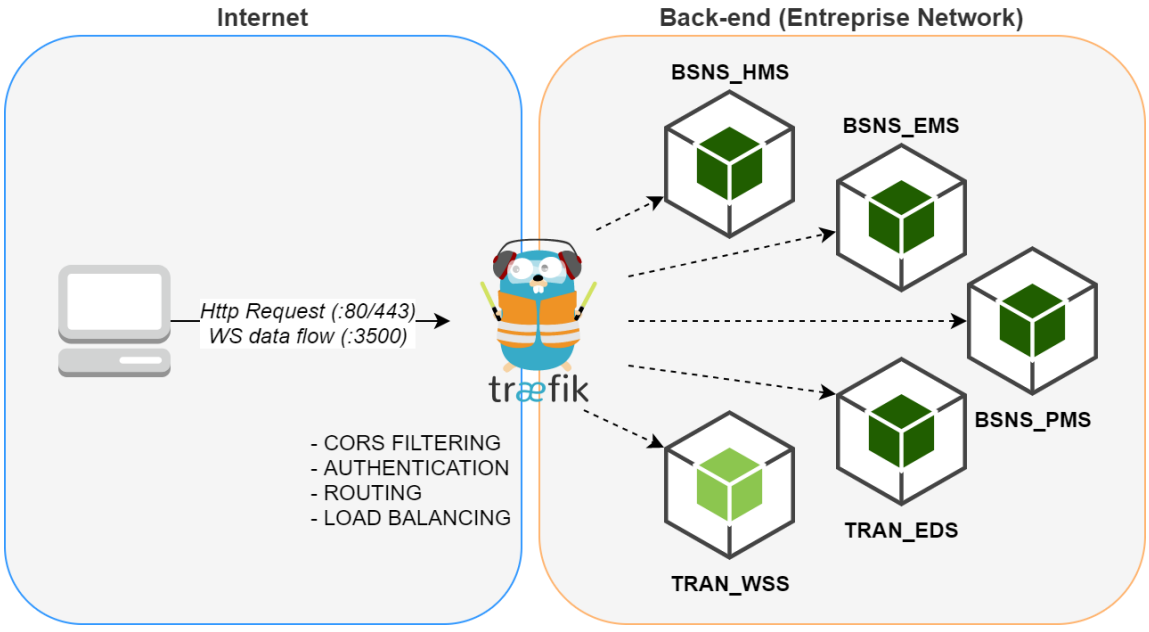

Figure 3 : Traefik

Front-end

La Web App (SPA) a été construite sur une base Javascript / Vue.Js. Ce Framework a été retenu pour sa rapidité de mise en œuvre et son écosystème riche. L'application est construite sous la forme d'un SPA connecté à un serveur de WebSocket (lui-même à l'écoute des évènements transitant dans les files de message) pour rendre l'affichage « réactif » et recharger les modules en fonction des événements transmis.

Cette approche permet de disposer d'un affichage temps réel sans consommer de bande passante inutile (à contrario d'un système de polling par exemple).

Sécurité

Authentification

Afin de garantir la confidentialité des données personnelles et de prémunir le système contre toute utilisation / consultation non autorisée, l'ensemble des services backend sont placés derrière une barrière d'authentification gérée par Traefik.

Pour les besoins de la PoC, l'authentification repose sur le schéma « Basic Auth » supporté par le protocole HTTP. Les couples login / mot de passe utilisés pour se connecter à la Web App sont permettent de passer la barrière d'authentification sur Traefik (et donc d'accéder aux API).

Bien que fonctionnelle et totalement sécurisé, cette approche ne permet pas de gérer de système de droit et limite l'évolutivité. Il est fortement recommandé d'utiliser un système basé sur un standard de type OAuth / SAML pour gérer le contrôle d'accès des utilisateurs.

Dans l'approche retenue, aucune authentification au niveau des services n'est implémentée. Ce choix permet d'éviter un couplage fort entre le système d'authentification / d'autorisation et l'API en elle-même en déléguant ces contrôles à un composant externe (Traefik dans le cas présent). Les API n'étant pas disponibles sans passer par Traefik, la sécurité est garantie.

CORS

Le composant Traefik assure aussi les contrôles d'origine dans le cadre du standard CORS. Cette approche permet d'éviter d'implémenter des listes de domaines autorisés coté backend et réduit le couplage front / back.

Autres sécurités

Les composants développés sont exempts de failles de sécurité identifiées et plus particulièrement de celles relevées dans le Top 10 OWASP (Injection SQL, XSS …)

Plan de test

Généralité

Afin de renforcer la qualité et de garantir le bon fonctionnement des logiciels, divers niveaux de tests sont implémentés au sein des micro-services et / ou de la web app :

- **Tests unitaires**, qui visent à tester principalement des « unités » de code.
- **Tests d'architecture**, qui visent à garantir que le code respect les conventions d'architecture (implémentation DDD, nommage …)
- **Tests d'intégration et tests fonctionnels**, qui visent à vérifier le comportement face à des use-case définis (erreur ou succès). Ces tests peuvent embarquer la construction à la volée de composant (ex : container Redis).
- **Tests de performance / de charge**, qui visent à s'assurer que les services répondent bien aux exigences de performances définis.
- **Tests E2E,** qui visent à tester l'application dans sa globalité, au plus proche des uses cases définis.

Hormis les tests de performance qui s'exécute sur un profil Maven spécifique, l'ensemble des tests livrés s'exécute sur les pipelines CI lors des déclencheurs spécifiques (*pull request*, *commit* sur *develop*, *commit* sur *main*).

Lors de l'exécution des pipelines, les résultats sont collectés par la plateforme SonarCloud pour génération des rapports et des métriques de qualité / sécurité.

Lors des phases de développement, le plug-in JaCoCo (Java) ou Istanbul (Javascript) permet de générer des rapports de couvertures de code au format HMTL. (Voir readme.md).

ArchUnit

Les tests d'architectures ont été implémentés à l'aide de la bibliothèque ArchUnit. Cette bibliothèque fournie une syntaxe simple pour l'écriture de test d'architecture. Ces tests sont généralement communs à l'ensemble des projets de l'entreprise et devrait s'appuyer sur un référentiel commun.

Il est fortement recommandé, le cas échéant, que le consortium MedHead se dote d'un tel référentiel et que ces tests soient externalisés dans un sous-projet afin d'être rapidement implémentés dans chaque produit développé par la société.

JMeter

Les tests de performance ont été développés avec l'aide de l'outil open source JMeter. Son utilisation est relativement simple et ne nécessitent pas de connaissance avancée en développement logiciel.

Le profil Maven « performanceTest » permet de charger les diverses dépendances pour lancer en automatique les tests avec JMeter. Il est cependant tout à fait possible de télécharger l'outil en externe et de charger les fichiers. jmx pour édition ou exécution dans des contextes différents.

Cypress

Des tests E2E ont été mise en œuvre pour la web app à l'aide de la solution Cypress. Cypress permet de définir facilement des scénarios de tests en imitant le comportement naturel d'un utilisateur. Ces tests permettent principalement de s'assurer que l'application fonctionne comme prévu.

Il devient facilement possible de tester, par exemple, l'authentification utilisateur ou le clic sur un bouton et vérifier le résultat par rapport à l'attendu.

RÉSULTATS DE LA POC

Évaluation de la conformité à la demande

Le tableau ci-après rappel les objectifs définis pour la PoC et indique leur niveau de complétion :

ID	Objectif	Statut	Commentaire
O1	La PoC est conforme au scénario de test décrit par les exigences.	Atteint	Implémentation d'une API tierce pour trouver le meilleur hôpital + Service de secours pour garantir le fonctionnement en cas de défaillance.
O2	La PoC utilise la technologie Java pour la création d'API Restful	Atteint	Le back-end est développé à l'aide de la technologie Java et du Framework Spring Boot. Les API sont conforme au modèle de maturité de Richardson (Level 2)
O3	La PoC dispose d'une interface graphique basé sur l'un des Frameworks Javascript/Typescript courant du marché : Angular, React, VueJS.	Atteint	WepApp SPA basée sur VueJS.
O4	Les APIs de la PoC peuvent s'intégrer dans une architecture micro-service.	Atteint	La PoC est conforme à la description d'une MSA event driven.
O5	La PoC est entièrement validée avec des tests reflétant la pyramide des tests. Des tests de stress sont implémentés.	Atteint	Des tests de plusieurs niveaux sont présents. Les tests de performances sont définis. Notons toutefois que ces tests devront être rejoués sur une architecture technique cible.
O6	La PoC embarque des pipelines CI/CD.	Atteint	Des pipelines CI ont été exploitées durant toute la phase de développement. Les pipelines permettent de produire des images docker prêtes à être déployées.
O7	Les données sont correctement protégées dans la PoC.	Atteint	Les API / interfaces graphique dispose d'un mécanisme d'authentification. Aucune données sensible (clef d'API, login / mot de passe) ne sont disponible sur les répertoires Git ou à l'intérieur des images docker.
O8	Le code est versionné à l'aide d'un workflow Git adapté.	Atteint	Le code est gitté et le workflow est décrit au sein des fichiers readme.md.

Tableau 3 : Évaluation de la conformité à la demande

Recommandation pour la mise en œuvre de la solution finale

La PoC a permis d'obtenir un modèle de micro-service découplée et facilement extensible. L'architecture de la solution et l'architecture de code retenue peuvent servir de modèle pour la construction et l'industrialisation de micro-services pour la solution finale.

Au-delà de la revue des exigences fonctionnelles, il est toutefois recommandé de procéder aux ajustements suivants pour la mise en œuvre du produit final :

- Modification du système d'authentification actuel (basé sur BasicAuth) par un système basé sur OAuth / OIDC embarquant une gestion d'autorisation en plus de l'authentification.
- Utilisation d'un système de messaging avec persistance, distribution de message améliorée (ex : gestion du multi-instance d'un même service) et mécanisme de reprise en cas d'échec d'un service (ex : Apache Kafka, RabbitMQ).
- Amélioration des pipelines CI/CD pour aller jusqu'au déploiement des images docker (à minima dans les environnements : dev & qualif).
- Amélioration des tests front-end.
- Remplacement de la base de données H2 par une base de données de production. (SQL ou no-SQL).
- Publication de la documentation des différentes API au format SWAGGER. (Open API). Ajout d'exemple (type : bibliothèque Postman) pour favoriser la collaboration.

Ces recommandations ne constituent pas une liste exhaustive et devront être revue en fonction des exigences fonctionnelles et non fonctionnelles définies pour la solution finale.

TABLES DES RÉFÉRENCES

Figures

Tableaux