ARCHITECTURE DOCUMENT

Search Sensei: Multi Tenant SaaS Customer Portal

CLIENT **Search Sensei Pty Ltd**

STACK ASP.NET Core 8, React 18 TypeScript, Keycloak 26, Stripe, BookStack, Zammad, Docker, Azure App Service

AUTHOR Akash Ramesh

STATUS v1.0.0 in production for a paying client

1. System Overview

Search Sensei is a multi tenant SaaS customer portal wrapping an existing enterprise search API. Each tenant organisation self registers, selects a subscription plan, and receives an isolated workspace covering authentication, billing, a knowledge base, and support ticketing. Zero cross tenant visibility is enforced at the identity layer.

The backend is the single trust boundary for the entire system. The browser holds only a short lived JWT. Every downstream credential (BookStack PAT, Zammad PAT, Stripe secret key) is held server side and never exposed to the client.

2. High Level Architecture

Component map

The system is composed of six layers, each with a distinct responsibility:

Layer	Component	Responsibility
Frontend	React 18 SPA (TypeScript)	User interface, auth code flow, JWT handling

Layer	Component	Responsibility
Identity	Keycloak 26	Multi tenant org management, OIDC provider, SSO broker
Backend	ASP.NET Core 8	API gateway, proxy layer, billing pipeline, business logic
Billing	Stripe	Checkout, subscriptions, metered usage, webhooks
Knowledge base	BookStack	Tenant documentation, proxied via backend
Support	Zammad	Support ticketing, proxied via backend with On Behalf Of
Search	Search API (Cosmos DB)	Core enterprise search, proxied per tenant

Request flow

All browser traffic flows through two hops: Keycloak for authentication, then the ASP.NET Core backend for everything else. No downstream service is reachable directly from the browser.

```
Browser
  |
  |-- Auth Code + PKCE ----------> Keycloak 26
  |<-- JWT (short lived) ----------|
  |
  |-- Bearer JWT ---------------> ASP.NET Core 8 Backend
                                  |
                                  |-- Admin API -----------> Keycloak
                                  |-- Stripe.net + webhooks -> Stripe
                                  |-- PAT proxy -----------> BookStack
                                  |-- PAT + On Behalf Of --> Zammad
                                  |-- Proxy --------------> Search API
```

3. Identity and Authentication

Keycloak 26 Organisations for multi tenancy

Keycloak was the existing identity provider for the Search Sensei platform. The architectural decision was how to model tenant isolation on top of it. Rather than building isolation into application code, the system uses Keycloak 26 native Organisations to scope users, roles, and sessions to an org boundary.

The backend resolves the current tenant from an active_tenant claim on every inbound request. This keeps isolation in a battle tested layer rather than bespoke application logic that would need to be maintained and audited separately.

Auth flow: Authorization Code with PKCE

The React SPA uses the Authorization Code flow with PKCE for all browser sessions. No client secret is exposed to the frontend. The flow:

- User lands on the portal and is redirected to Keycloak
- Keycloak authenticates the user and issues a short lived JWT
- The SPA sends the JWT as a Bearer token on every API request to the backend
- The backend validates the JWT, extracts the active_tenant claim, and routes accordingly
- Token refresh is handled silently by the SPA using the refresh token

Stripe customer identity colocation

Subscription and customer IDs are stored as attributes on the Keycloak organisation itself rather than duplicated into a first party database. This gives a single source of truth for the mapping between a tenant org and a Stripe customer, eliminating the risk of the two systems drifting out of sync.

4. Billing Architecture

Stripe Checkout and subscription lifecycle

New tenants complete plan selection and payment via Stripe Checkout during onboarding. The backend creates a Stripe customer and subscription, then stores the resulting IDs on the Keycloak organisation. Subsequent subscription lifecycle events arrive via signed Stripe webhooks and are handled by the backend.

Metered billing pipeline

Three consumption dimensions are tracked and reported to Stripe as meter events:

- Search query volume
- Document pages indexed
- Additional sites crawled

An internal usage reporting endpoint accepts events from backend platform services. Each event requires a caller supplied idempotency key before the backend forwards it to Stripe as a meter event. This prevents double counting on retries, which is the standard pattern for financial grade event pipelines.

> Key design rule: idempotency keys are caller supplied, not generated by the reporting endpoint. This means the caller controls the deduplication boundary: if a service retries due to a network failure, the same key is reused and Stripe discards the duplicate. The endpoint never generates its own key, which would defeat the purpose.

Webhook handling

Stripe webhooks are verified using the webhook signing secret before any processing occurs. Handled events include subscription created, subscription updated, subscription deleted, and payment failed. Each event updates the corresponding Keycloak organisation attributes to keep identity and billing state in sync.

5. Backend Proxy Pattern

Why a proxy rather than direct browser access

BookStack and Zammad both require long lived Personal Access Tokens for API access. Sending those tokens to the browser would expose them in DevTools network requests, localStorage, and any JavaScript bundle analysis. The backend proxy pattern eliminates that surface entirely.

The ASP.NET Core backend holds all PATs as server side configuration. The browser sends only its short lived JWT. The backend validates the JWT, resolves the tenant, selects the correct PAT, calls the downstream service, and returns only what the authenticated user is permitted to see.

BookStack proxy

- Staff and admin views are embedded in the portal as iframes or component views
- All requests pass through the backend, which attaches the BookStack PAT
- Read only and admin editor access are separated by the backend based on the user role claim
- The BookStack PAT never appears in any browser request

Zammad proxy

- Support ticket creation and viewing are embedded in the portal
- The backend forwards requests to Zammad using its API Token
- On Behalf Of identity forwarding is used so tickets are attributed to the correct end user in Zammad, not to the service account
- The Zammad API Token never appears in any browser request

Search API proxy

- Tenant search requests are forwarded to the Search API with tenant scoping applied by the backend
- The backend enforces that each tenant can only query their own indexed data
- No Search API credentials are exposed to the browser

6. Key Data Model Decisions

Tenant identity: single source of truth

A core decision was avoiding a first party tenants table. Instead:

- Keycloak Organisation: canonical tenant record. Holds org name, members, roles, and session scoping
- Keycloak Organisation attributes: Stripe customer ID and subscription ID stored here, not in a separate database
- Result: one place to look up the full tenant state. No sync jobs, no drift between systems

Usage event design

Usage events are fire and forget with idempotency, not transactional. The pipeline is:

- Platform service emits a usage event with a stable idempotency key (e.g. a hash of the operation ID and dimension)
- The reporting endpoint validates the key format, then forwards to Stripe as a meter event
- Stripe handles deduplication. The backend does not store usage events in its own database
- Billing reconciliation is done entirely through the Stripe dashboard and Stripe webhooks

7. Security Design

Concern	Approach	Where enforced
Tenant isolation	Keycloak Organisations with active_tenant claim	Backend on every request
Credential exposure	All PATs and secret keys held server side only	Backend proxy pattern
Token lifetime	Short lived JWTs, silent refresh via refresh token	Keycloak + SPA

Concern	Approach	Where enforced
Webhook integrity	Stripe webhook signature verification before processing	Backend webhook handler
Auth flow	Authorization Code + PKCE, no implicit flow	Keycloak + SPA
Rate limiting	Stripe metered billing enforces usage limits by plan tier	Stripe + backend

8. Deployment

All services are containerised with Docker. The backend and frontend are deployed to Azure App Service. Keycloak, BookStack, and Zammad are self hosted on containerised infrastructure.

```
Azure App Service
  |-- ASP.NET Core 8 backend (containerised)
  |-- React 18 SPA (static, served via CDN)

Self hosted containers
  |-- Keycloak 26
  |-- BookStack
  |-- Zammad

External SaaS
  |-- Stripe
  |-- Search API (Cosmos DB, existing Search Sensei infrastructure)
```

9. My Contributions

Within the 5 person capstone team, my specific ownership:

- Full Stripe integration: Checkout, subscription lifecycle, metered billing pipeline, webhook handling, idempotency design
- Full Keycloak integration: multi tenant organisation setup, PKCE auth flow, role based access, Stripe ID colocation on org attributes
- ASP.NET Core proxy layer: design and implementation for BookStack, Zammad, and the Search API

- Sprint planning and client demos within the Agile delivery process
- AI assisted code review using Claude Code for vulnerability detection and edge case analysis ahead of every merge

---

Source code is in a private client repository. Akash Ramesh is available to walk through any aspect of this architecture in detail during interview or technical review.