Architecture Decision Document

Application : Suivi FTI EMGI (suivi facturation / échéancier / paiement – EMGI)

This document builds collaboratively through step-by-step discovery. Sections are appended as we work through each architectural decision together.

Project Context Analysis

Requirements Overview

Functional Requirements: 13 FRs organisés en 6 catégories : Client Management, Invoice Tracking, Payment Management, Financial Calculations, Reporting & Analytics, Data Management. Application web MPA avec calculs financiers automatiques (encours, échéances) et export de données.

Non-Functional Requirements: Security (accès restreint équipe commerciale), Performance (< 3s chargement, < 1s interactions).

Scale & Complexity:

- Primary domain: Full-stack Web App
- Complexity level: Low-Medium (règles métier simples, pas de contraintes techniques complexes)
- Estimated architectural components: 3 composants principaux (frontend, backend API, base de données)
- Mode: POC (Proof of Concept) puis migration vers MODX en production

Technical Constraints & Dependencies

- **Facturation :** la facturation (document facture, édition) provient de **Pennylane** ; l'application assure uniquement le suivi (saisie manuelle des factures pour le suivi).
- Budget ~20k€ → architecture minimale et simple
- Stack POC: PHP (API REST) + JavaScript + Tailwind CSS + HTML + MariaDB/MySQL
- Stack Production: MODX (CMS/PHP)
- Technologies simples et éprouvées
- Pas de temps réel requis
- Application interne uniquement
- Développement en mode POC collaboratif

Cross-Cutting Concerns Identified

- Calculs financiers automatiques (encours, échéances) - règles métier simples
- Gestion des données centralisée (factures, paiements, clients)
- Authentification/autorisation simple (équipe commerciale uniquement) - pas de focus particulier pour le POC
- Export de données
- Migration POC → MODX en production

Starter Template Evaluation

Primary Technology Domain

Full-stack Web App - POC en PHP/JavaScript, Production en MODX.

Technical Stack Selection

POC Stack:

- **Backend:** PHP (API REST)
- **Frontend:** JavaScript + Tailwind CSS + HTML
- **Database:** MariaDB/MySQL
- **Authentication:** Simple (pas de focus particulier pour le POC)

Production Stack:

- **MODX** (CMS/PHP) pour base de données et frontend

Rationale for POC Stack Selection:

- PHP pour API REST simple et rapide à développer
- JavaScript vanilla + Tailwind CSS pour frontend léger et rapide
- MariaDB/MySQL pour base de données relationnelle simple
- Stack simple permettant validation rapide du concept avant migration vers MODX

Architectural Decisions:

Language & Runtime:

- PHP pour backend API REST
- JavaScript pour frontend interactif

Styling Solution:

- Tailwind CSS pour styling rapide et moderne
- HTML pour structure

Database:

- MariaDB/MySQL pour stockage relationnel (factures, paiements, clients)

API Architecture:

- API REST en PHP pour séparation frontend/backend
- Endpoints simples pour chaque capability (clients, factures, paiements, calculs)

Development Approach:

- POC développé directement avec cette stack simple
- Migration vers MODX prévue pour la production
- Focus sur validation fonctionnelle plutôt que stack complexe

Core Architectural Decisions

Decision Priority Analysis

Critical Decisions (Block Implementation):

- Stack POC : PHP (API REST) + JavaScript + Tailwind CSS + HTML + MariaDB/MySQL
- Structure base de données : 4 tables principales (clients, factures, paiements, echeanciers)
- Architecture API : REST avec endpoints simples

Important Decisions (Shape Architecture):

- Frontend : Pages HTML séparées (MPA), JavaScript vanilla, Tailwind CSS
- Backend : API REST PHP avec endpoints JSON
- Base de données : MariaDB/MySQL avec schéma simple

Deferred Decisions (Post-MVP):

- Authentification avancée (simple pour POC)
- CI/CD et monitoring (pas nécessaire pour POC)
- Optimisations de performance avancées

Data Architecture

Database Schema:

- Tables : `clients`, `factures`, `paiements`, `echeanciers`
- Relations : factures → client (foreign key), paiements → facture (foreign key), echeanciers → facture (foreign key)
- Champs factures : n_facture, n_vente, date_facture, type_produit (alevins/larves), cycle (ex. 24-25), quantites, qte_livree, montant_ht
- Champs echeanciers : facture_id, nb_echeances, date_debut_echeancier
- État payé/en attente dérivé des paiements

Data Validation:

- Validation côté API PHP (simple)
- Contraintes DB basiques (NOT NULL, clés étrangères)

Migration:

- Scripts SQL simples pour création des tables
- Pas de migration complexe pour le POC

Authentication & Security

Authentication:

- Sessions PHP simples ou authentification basique
- Pas de JWT/OAuth pour le POC

- Focus minimal sur l'authentification

API & Communication Patterns

API REST Design:

- Endpoints simples : `/api/clients`, `/api/factures`, `/api/paiements`, `/api/echeanciers`, `/api/calculs`
- Format JSON pour requêtes et réponses
- Gestion d'erreurs simple (codes HTTP standards : 200, 400, 404, 500)

API Documentation:

- Commentaires dans le code ou README simple

Frontend Architecture

State Management:

- Pas de framework d'état - JavaScript vanilla avec variables globales ou localStorage si besoin

Routing:

- Pages HTML séparées (MPA) : `index.html`, `client.html`, `dashboard.html`
- Navigation simple entre pages

Performance:

- Pas d'optimisation complexe pour le POC
- Tailwind CSS en CDN ou build simple

Infrastructure & Deployment

Hosting:

- Serveur PHP simple (local ou hébergement PHP basique)
- Pas de CI/CD complexe pour le POC

Monitoring:

- Pas de monitoring avancé pour le POC

Decision Impact Analysis

Implementation Sequence:

1. Création base de données (tables clients, factures, paiements)
2. Développement API REST PHP (endpoints de base)
3. Développement frontend HTML/JS/Tailwind (pages principales)
4. Intégration frontend/backend
5. Tests fonctionnels basiques

Cross-Component Dependencies:

- Frontend dépend de l'API REST pour toutes les données
- API dépend de la structure de base de données
- Calculs financiers (encours, échéances) dans l'API PHP

Implementation Patterns & Consistency Rules

Pattern Categories Defined

Critical Conflict Points Identified: Naming (DB, API, code), structure des fichiers, formats API.

Naming Patterns

Database Naming Conventions:

- Tables : minuscules pluriel (`clients` , `factures` , `paiements` , `echeanciers`)
- Colonnes : snake_case (`client_id` , `n_facture` , `date_facture`)
- Clés étrangères : `client_id` , `facture_id`

API Naming Conventions:

- Endpoints : pluriel (`/api/clients` , `/api/factures` , `/api/paiements` , `/api/echeanciers`)
- Paramètres query : snake_case (`?client_id=1`)
- Route params : `:id` ou `{id}` selon le routeur

Code Naming Conventions:

- PHP : fonctions camelCase (`getClientData()`), variables `$camelCase`
- JavaScript : fonctions camelCase (`getClientData()`), variables camelCase
- Fichiers : minuscules avec tirets si besoin (`client-detail.js`)

Structure Patterns

Project Organization:

- API PHP : `/api/clients.php` , `/api/factures.php` , `/api/paiements.php` , `/api/echeanciers.php`
- Frontend : `/index.html` , `/client.html` , `/dashboard.html`
- Utilitaires JS : `/js/utils.js`
- CSS : Tailwind CDN ou `/css/` si custom

File Structure Patterns:

- Config : `.env` à la racine, pas de config complexe pour POC
- Scripts SQL : `/sql/schema.sql` ou à la racine

Format Patterns

API Response Formats:

- Succès : réponse directe `{...}` (pas de wrapper `data`)

- Erreur : `{error: "message"}` avec code HTTP approprié (400, 404, 500)
- Dates : ISO strings (`2026-02-09T10:00:00Z`)

Data Exchange Formats:

- JSON : champs en snake_case pour cohérence avec la DB
- Booléens : `true` / `false`

Process Patterns

Error Handling:

- API : retourner code HTTP + `{error: "message"}`
- Frontend : afficher message utilisateur simple

Loading States:

- Gestion simple (pas de pattern imposé pour POC)

Enforcement Guidelines

All AI Agents MUST:

- Utiliser snake_case en base de données et dans les réponses JSON API
- Utiliser camelCase en PHP et JavaScript pour variables/fonctions
- Respecter la structure des endpoints (`/api/clients` , etc.)
- Retourner les erreurs au format `{error: "message"}`

Anti-Patterns:

- Ne pas mélanger camelCase et snake_case dans le même contexte (ex. JSON API en camelCase alors que DB en snake_case)
- Ne pas créer de wrapper `{data: ...}` pour les réponses API (réponse directe)

Project Structure & Boundaries

Complete Project Directory Structure

```
bmad/
├── README.md
├── .env.example
├── .env
├── .gitignore
├── index.html
├── client.html
├── dashboard.html
├── top-clients.html
├── quantites-clients.html
├── api/
│   ├── config.php
│   ├── clients.php
│   ├── factures.php
│   ├── paiements.php
│   ├── calculs.php
│   ├── reporting.php
│   ├── top-clients.php
│   └── quantites-clients.php
├── js/
│   ├── utils.js
│   ├── client-detail.js
│   ├── paiements.js
│   ├── dashboard.js
│   ├── facture-form.js
│   ├── top-clients.js
│   └── quantites-clients.js
├── css/
│   └── (optional custom styles, Tailwind via CDN by default)
└── sql/
    └── schema.sql
```

Architectural Boundaries

API Boundaries:

- **Externe :** Tous les endpoints sous `/api/*.php` (clients, factures, paiements, calculs, reporting)
- **Interne :** `api/config.php` fournit la connexion DB (requise par les autres scripts API)
- **Autorisation :** Pas de couche auth dédiée pour le POC ; accès restreint géré par l'hébergement / le serveur

Component Boundaries:

- **Frontend :** Pages HTML indépendantes ; chaque page charge les scripts JS nécessaires (pas de SPA, pas de state partagé entre pages)
- **Backend :** Chaque fichier API gère un domaine (clients, factures, etc.) ; pas de couche service partagée pour le POC

Service Boundaries:

- Frontend ↔ API : appels `fetch()` vers `/api/*.php`, réponses JSON
- API ↔ DB : PDO dans chaque script API, config centralisée dans `config.php`

Data Boundaries:

- **DB :** Tables `clients`, `factures`, `paiements`, `echeanciers` ; schéma dans `sql/schema.sql`
- **API :** Lecture/écriture via PDO ; pas de cache pour le POC
- **Export :** Géré par `api/reporting.php` (CSV/JSON selon besoin)

Requirements to Structure Mapping

Feature / FR category mapping:

- **Client Management (FR1, FR2) :** `api/clients.php`, `client.html`, `js/client-detail.js`
- **Invoice Tracking (FR3, FR3b) :** `api/factures.php` (GET par client, POST ajout), `client.html`, formulaire ajout facture (`js/facture-form.js`)
- **Payment Management (FR4, FR5) :** `api/paiements.php`, `client.html`, `js/paiements.js`
- **Écheancier (FR4b, FR5b) :** `api/echeanciers.php` (GET par client_id ou facture_id, POST création), `client.html`, formulaire et affichage écheancier
- **Financial Calculations (FR6–FR9) :** `api/calculs.php`, `client.html`, `dashboard.html`
- **Reporting & Analytics (FR10, FR10b, FR10c, FR11) :** `api/reporting.php`, `api/top-clients.php`, `api/quantites-clients.php`, `dashboard.html`, `top-clients.html`, `quantites-clients.html`, `js/dashboard.js`, `js/top-clients.js`, `js/quantites-clients.js`
- **Data Management (FR12, FR13) :** Base de données + tous les endpoints ; pas de fichier dédié

Cross-cutting:

- **Config / DB :** `api/config.php`, `sql/schema.sql`
- **Utilitaires frontend :** `js/utils.js` (helpers, appels API communs)

Integration Points

Document détaillé : *architecture-generale-fonctionnelle-et-interfaces.md* — architecture générale (utilisateurs, extérieur, entrées/sorties), décomposition fonctionnelle (modules), **interfaces logicielles** détaillées (I1 à I6 : utilisateur↔appli, front↔API, API↔DB, API↔mock, export CSV, futur Pennylane).

Internal communication:

- Frontend → API : `fetch()` vers `/api/clients.php`, `/api/factures.php`, etc. (paramètres query ou body selon besoin)
- API → DB : PDO via `config.php`
- Pas d'appels API-to-API pour le POC

External integrations:

- Aucune pour le MVP (pas d'ERP, pas de compta externe)

Data flow:

- Liste clients : `index.html` → `api/clients.php` → DB
- Détail client : `client.html` → `api/clients.php`, `api/factures.php`, `api/paiements.php`, `api/calculs.php` → DB
- Dashboard : `dashboard.html` → `api/calculs.php`, `api/reporting.php` → DB
- Enregistrement paiement : `client.html` + `js/paiements.js` → `api/paiements.php` → DB → recalcul côté API

File Organization Patterns

Configuration:

- `.env` à la racine (DB, options éventuelles) ; `.env.example` sans secrets
- `api/config.php` lit `.env` ou des constantes et expose la connexion PDO

Source:

- API : un fichier par ressource (clients, factures, paiements, calculs, reporting)
- Frontend : une page HTML par écran principal ; scripts JS par page ou par feature (client-detail, paiements, dashboard)

Tests:

- Pas de structure de tests formelle pour le POC ; tests manuels ou scripts ad hoc si besoin

Assets:

- Tailwind via CDN dans les HTML ; `css/` optionnel pour overrides
- Pas de build d'assets pour le POC

Development Workflow Integration

Development server:

- Serveur PHP intégré (`php -S`) ou Apache/Nginx pointant sur la racine du projet
- Accès : `http://localhost[:port]/index.html`, `http://localhost[:port]/api/clients.php`, etc.

Build:

- Aucun build obligatoire ; déploiement = copie des fichiers (PHP, HTML, JS, SQL)

Deployment:

- Fichiers déployés sur un hébergement PHP avec MariaDB/MySQL ; exécution de `sql/schema.sql` une fois ; `.env` configuré en production

Architecture Validation Results

Coherence Validation ✅

Decision Compatibility: Les choix techniques POC (PHP, JavaScript, Tailwind, HTML, MariaDB) sont cohérents entre eux. MPA, API REST et structure un fichier par ressource sont alignés. Aucun conflit de

versions identifié.

Pattern Consistency: Les conventions (snake_case DB/API, camelCase code, réponses directes, erreurs `{error: "message"}`) sont cohérentes avec la stack et la structure définie.

Structure Alignment: La structure du projet (api/, js/, sql/, pages à la racine) supporte les décisions et les limites (API, frontend, DB) sont clairement définies.

Requirements Coverage Validation ✅

Epic/Feature Coverage: N/A (pas d'epics). Les catégories FR sont couvertes.

Functional Requirements Coverage: Chaque catégorie FR (Client Management, Invoice Tracking, Payment Management, Financial Calculations, Reporting & Analytics, Data Management) dispose de composants et endpoints dédiés ; le mapping exigences → structure est documenté.

Non-Functional Requirements Coverage: NFR1 (accès restreint) est assumé par l'hébergement/serveur pour le POC. Cibles de performance respectées par une architecture simple.

Implementation Readiness Validation ✅

Decision Completeness: Décisions critiques documentées (stack, schéma DB, API, frontend, patterns, structure). Versions et règles d'usage précisées où pertinent.

Structure Completeness: Arborescence complète avec fichiers et dossiers nécessaires ; limites et points d'intégration décrits.

Pattern Completeness: Conflits de nommage adressés ; conventions et formats API définis ; gestion d'erreurs et flux de données décrits.

Gap Analysis Results

- **Critical:** Aucun.
- **Important:** Détail du schéma SQL (colonnes exactes) et format d'export (CSV/JSON) à préciser en implémentation si besoin.
- **Nice-to-have:** Auth détaillée et déploiement MODX à documenter en phase production.

Validation Issues Addressed

Aucun point bloquant ; quelques précisions laissées à l'implémentation (schéma SQL détaillé, export, auth).

Architecture Completeness Checklist

✅ **Requirements Analysis**

- ☑ Contexte projet analysé
- ☑ Échelle et complexité évaluées
- ☑ Contraintes techniques identifiées
- ☑ Préoccupations transversales cartographiées

✅ **Architectural Decisions**

- ☑ Décisions critiques documentées
- ☑ Stack technique précisée
- ☑ Patterns d'intégration définis
- ☑ Performance prise en compte

✅ Implementation Patterns

- ☑ Conventions de nommage établies
- ☑ Patterns de structure définis
- ☑ Patterns de communication précisés
- ☑ Patterns de processus documentés

✅ Project Structure

- ☑ Structure de répertoires complète
- ☑ Limites des composants établies
- ☑ Points d'intégration cartographiés
- ☑ Mapping exigences → structure complété

Architecture Readiness Assessment

Overall Status: READY FOR IMPLEMENTATION

Confidence Level: Élevé, au vu de la cohérence, de la couverture des exigences et de la clarté des patterns et de la structure.

Key Strengths: Stack simple et maintenable ; séparation nette API / frontend / DB ; patterns explicites pour les agents ; structure prête pour le POC.

Areas for Future Enhancement: Auth détaillée, schéma SQL fin, export, migration MODX en production.

Implementation Handoff

AI Agent Guidelines:

- Suivre les décisions et patterns du document d'architecture.
- Respecter la structure du projet et les limites des composants.
- Utiliser ce document comme référence pour toute question d'architecture.

First Implementation Priority: Créer le schéma DB (`sql/schema.sql`), puis `api/config.php` , puis les endpoints de base (`api/clients.php` , `api/factures.php` , `api/paiements.php` , `api/echeanciers.php`).