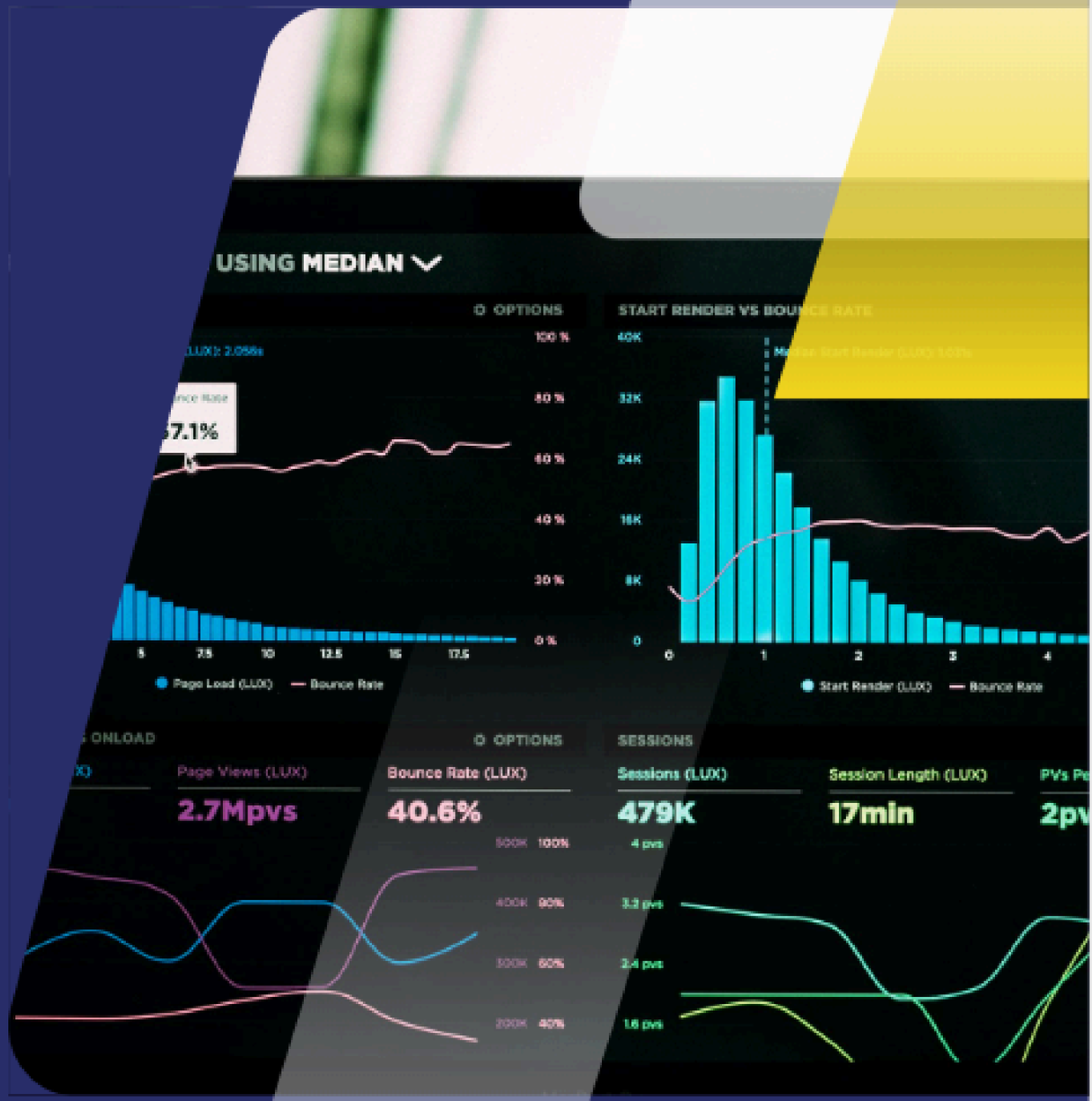
Gestion Client moderne avec **ASP.NET** Full Stack

Présentateurs: Ayoub Elmardi & Zayd Bentalha



Presenter Name
Presenter Designation

Contexte & Objectifs du projet

Centraliser opérations clients, ventes et banque

01	**Problématique: gestion centralisée des opérations clients, ventes et banque**	Réduire silos et doublons de données pour un reporting fiable
02	**Objectif: interface simple et responsive**	Expérience utilisateur fluide sur tous appareils
03	**Objectif: API RESTful modulaire**	Modules indépendants pour intégration et tests
04	**Objectif: architecture scalable et maintenable**	Croissance sans refonte et facilitation de maintenance

01 **Centralisation réduit duplication des données**

Moins d'erreurs et source de vérité unique

02 **Facilite reporting et intégration**

Rapports cohérents et connexions tierces simplifiées

03 **Modularité permet évolutivité et tests indépendants**

Déploiements isolés et évolutions rapides

04 **Visuel suggéré: diagramme conceptuel d'usage**

Inclure le diagramme initial mentionné dans la diapo originale

Architecture technique: vue d'ensemble 3 couches

Séparation UI, logique métier et persistance pour tests et scalabilité

Présentation – UI

Frontend: HTML5, CSS3, JavaScript vanilla. Expose l'interface utilisateur et appels API.

Logique métier – Services

Backend: ASP.NET Core Web API en C#. Gère règles métier, validation des entrées, CORS.

Persistance – BD / ORM

Base de données via ORM. Stockage durable et accès transactionnel.

Serveur et réseau

Kestrel en production, reverse proxy optionnel, HTTPS obligatoire.

Sécurité et bonnes pratiques

HTTPS, CORS configuré, validation des entrées pour prévenir injections.

Structure des modules : rôles et entités clés

Quatre domaines fonctionnels et recommandations d'architecture

Client

- Contacts et historique de relations
- Marketing et segmentation
- Téléphonie et interactions
- Propriétés de Sales ownership properties

Ventes

- Enregistrement des transactions
- Règles métiers de validation
- Gestion des opportunités et pipeline
- Statuts commerciaux et reporting

Trade

- Gestion des deals et exécutions
- Conformité et contrôles réglementaires
- Lifecycle des positions
- Logs et audit des opérations

Banque

- Monitoring des flux financiers
- Rapprochements et cash management
- Interfaces de paiement et settlement
- Surveillance des risques bancaires

Backend : conception d'API REST efficaces

Patterns, endpoints clés, technologies et bonnes pratiques

Pattern Controller-Service-Repository

- Controller: orchestration des requêtes et réponses
- Service: logique métier testable et réutilisable
- Repository: accès aux données via Entity Framework Core
- Dependency Injection pour découplage et testabilité

Endpoints principaux

- GET /api/clients: liste et filtrage
- POST /api/sales: création de vente
- GET /api/bank/monitoring: état du monitoring bancaire

Technologies utilisées

- Entity Framework Core pour ORM
- Dependency Injection native .NET
- Swagger/OpenAPI pour documentation

Bonnes pratiques et tests

- Versionner APIs (versioning) et gérer compatibilité
- Validation et gestion d'erreurs centralisée
- Documenter avec OpenAPI et Swagger UI
- Tests unitaires pour services et tests d'intégration pour controllers

Frontend : **Architecture** et UI légères

SPA vanilla, responsive, accessible et maintenable

Structure HTML/CSS

Layout avec Flexbox et Grid, composants réutilisables, tokens de thème dark and light

Composants JavaScript

SPA légère en Vanilla JS, composants isolés et accessibles

Appels asynchrones

Fetch API pour données dynamiques et gestion d'erreurs

Gestion d'état

Pattern événementiel ou store léger pour synchroniser UI

Rendering

Mise à jour DOM basée sur composants et données asynchrones

Workflow de développement et outils clés

Séquence pragmatique pour passer de la conception au déploiement

Conception des modèles de données

Définir schémas, relations et validations; préparer API contracts

Création des interfaces frontend

Construire vues, appels API et interactions utilisateur

Pipelines CI/CD

Automatisation des builds, tests et déploiements

Développement des contrôleurs API

Implémenter endpoints, logique métier et gestion des erreurs

Tests et intégration

Unitaires et d'intégration automatisés; revues de code

Déploiement

Options: IIS pour Windows ou Docker container pour portabilité

Défis techniques rencontrés et solutions pragmatiques

Clarifier entités, stabiliser API, maîtriser états frontend

Défis

- Interprétation du diagramme initial : clarification des entités manquante
- Communication frontend et backend : incohérences d'API
- Gestion des états frontend : complexité et propagation d'événements

Vs

Solutions et recommandations

- Définir contracts de données et schémas partagés
- Valider payloads avec JSON Schema ou utiliser DTOs
- Adopter conventions d'erreur standardisées et codes clairs
- Surveiller latence API et métriques de performance
- Maintenir approche minimaliste sans frameworks lourds pour réduire complexité

Roadmap et conclusion: trajectoire vers une solution extensible

Phases clés, jalons et actions recommandées pour la mise en production

Phase 1

- MVP avec fonctionnalités de base
- Livrable: version minimale viable prête pour tests fonctionnels

Phase 2

- Ajout d'Entity Framework
- Intégration base de données et mapping ORM

Phase 3

- Authentification avec ASP.NET Identity
- Sécurisation des accès et gestion des utilisateurs

Phase 4

- Tableaux de bord et reporting avancé
- Visualisations, KPIs et exports pour le pilotage métier

Modèles de données: principes et mapping EF Core

Bonnes pratiques pour structure, relations et migrations

01 **Normalisation raisonnable pour éviter la redondance excessive**

équilibrer normalisation et performance

02 **Désambiguïsation des clés: clés naturelles vs clés substituts**

préférer clés substituts si ambiguïté ou évolution

03 **Utiliser DTOs à la frontière API**

protéger le modèle domaine et contrôler le contrat API

04 **EF Core: configurations via Fluent API**

centraliser mapping et contraintes

05 **Migrations contrôlées et seeds pour tests**

migrations versionnées + seeds pour environnement de test

06 **Relations: gérer 1:N et N:N explicitement**

prévoir tables d'association pour N:N et contraintes claires

07 **Stratégies de chargement: eager vs lazy selon cas**

préférer eager pour requêtes prédictibles, lazy pour mémoire

08 **Tests et seeds: environnement de test reproductible**

seed contrôlé pour scénarios et CI

09 **Résumé: clarté des clés, mapping explicite, migrations sûres**

garantir maintenabilité et performance

Design API: contrats et sécurité

Bonnes pratiques pour DTO versionnés, validations et documentation OpenAPI

Contrats: versioning et DTO

- Utiliser **DTOs versionnés** pour compatibilité ascendante
- Définir schémas clairs et stables pour chaque version
- Valider côté serveur et renvoyer erreurs explicites
- Automatiser tests de contrat contre les schémas

Sécurité: authentification et autorisation

- Prévoir **authentification** selon roadmap
- Appliquer autorisation par endpoint et rôle
- Limiter privilèges et audit des accès
- Utiliser tokens sécurisés et rotation

Validation et protection contre attaques

- Valider strictement tous les inputs
- Protéger contre injection et payload malveillant
- Sanitiser et normaliser avant traitement
- Mettre des limites de taille et rate limiting

Documentation: OpenAPI et tests automatisés

- Documenter chaque version avec **OpenAPI**
- Générer clients et mocks à partir du spec
- Intégrer tests contractuels CI/CD
- Tenir la doc synchronisée avec implémentation

Frontend: patterns de **réutilisabilité** et accessibilité

Composants modulaires, checklist d'accessibilité et optimisations de performance

01 **Composants modulaires réutilisables**

Isolation, props clairs, tests unitaires

02 **Conventions CSS: BEM ou CSS Modules**

Nomination prévisible, styles scoping

03 **Documentation de composants**

Storybook ou catalogue de composants

04 **Tests contraste pour dark et light**

Vérifier ratios de contraste

05 **Navigation au clavier et attributs ARIA**

Focus visible, rôles et labels ARIA

06 **Lazy loading des données**

Charger on demand pour réduire la charge initiale

07 **Rendu conditionnel pour optimiser initial load**

Afficher seulement ce qui est nécessaire

08 **Checklist rapide d'accessibilité**

Contraste, clavier, ARIA, tests utilisateurs

Tests: stratégie et types clés

Où exécuter chaque test et comment automatiser efficacement

01 **Unitaires: services et logique métier**

Rapides, isolés; exécuter en local et dans CI

02 **Intégration: controllers plus base de données en mémoire ou test DB**

Vérifie interactions composantes; exécuter en CI

03 **End-to-end: parcours critiques avec navigateur headless**

Tests utilisateurs complets; exécuter en CI pour releases

04 **Automatisation via pipeline CI**

Orchestrer suites unitaires, d'intégration et E2E

05 **Outils typiques: frameworks .NET pour unitaires et intégration**

Headless browser pour E2E (qualitatif)

06 **Recommandation pratique: prioriser parcours critiques en E2E**

Limiter E2E au noyau pour rapidité et fiabilité

Déploiement et opérations : **IIS vs Docker**

Choisir selon portabilité, exploitation et sécurité

IIS (Internet Information Services)

- Intégration native Windows et support .NET
- Déploiement direct sur VM/serveur avec état local
- Pipelines CI/CD : build puis publication (Web Deploy / MSBuild)
- Monitoring : PerfMon et Event Logs ; nécessite centralisation externe
- Sécurité : TLS, certificats Windows ; attention aux secrets locaux
- Stratégies de déploiement : blue-green et canary possibles mais plus lourdes

Docker container

- Portabilité entre environnements et isolation par container
- Déploiement immutable via images versionnées et registries
- Pipelines CI/CD : build image, push registry, déploiement via orchestrateur
- Monitoring : métriques container, logs centralisés et traces distribuées
- Sécurité : TLS, gestion des secrets via vault/secret store, scan d'images
- Stratégies de déploiement : blue-green et canary facilités par orchestrateur

Résumé technique & **Q&A**

Architecture modulaire ASP.NET Core + Vanilla JS

Architecture

Solution modulaire ASP.NET Core avec frontend Vanilla JS

API et données

API REST bien définies, modèles gérés via EF Core

CI/CD et conteneurs

Workflow CI/CD recommandé et conteneurisation

Points d'action

Clarifier diagramme initial, définir contracts d'API, planifier tests et monitoring

Q&A

Ouvrir la discussion technique