C# Moderne et Migration WPF vers .NET 9

m2iformation.fr

Sommaire

Nouveautés C#

C# 9.0 (.NET 5) - Révolution Fonctionnelle :

- Records pour l'immutabilité
- Init-only properties
- Pattern matching étendu
- Top-level programs

C# 10.0 (.NET 6) - Simplification Syntaxique :

- Global using directives
- File-scoped namespace
- Constant interpolated strings
- Record structs

C# 11.0 (.NET 7) - Robustesse :

- Required members
- Raw string literals
- List patterns avancés
- Static virtual members

C# 12.0 (.NET 8) - Constructeurs Primaires :

- Primary constructors pour classes
- Collection expressions
- Alias any type

C# 13.0 (.NET 9) - Optimisations :

- Field keyword
- Lock improvements
- Escape sequences étendues

Global Usings et File-Scoped Namespace

Global Usings - Organisation Centralisée :

- **Un seul fichier** `GlobalUsings.cs` pour toute l'application
- **Déclaration au niveau projet** dans le `.csproj`
- **Réduction drastique** du code répétitif
- **Maintenance facilitée** des imports

```
// GlobalUsings.cs
global using System;
global using System.Windows;
global using Microsoft.Extensions.Logging;
```

Global Usings et File-Scoped Namespace

File-Scoped Namespace - Syntaxe Allégée :

- **Suppression de l'indentation** inutile
- **Lisibilité améliorée** des fichiers
- **Standard moderne** recommandé Microsoft

```
namespace MyWpfApp.ViewModels;

public class MainViewModel { }
```

Impact WPF :

- Fichiers ViewModels plus lisibles
- Organisation cohérente des Services
- Maintenance simplifiée des contrôles utilisateur

Records et Immutabilité

Records - Révolution pour les Données :

- **Immutabilité par défaut** avec syntaxe concise
- **Value semantics** automatiques (égalité, hash)
- **Déconstruction** native
- **With expressions** pour mutations contrôlées

```
public record Customer(int Id, string Name, string Email);

// Usage avec with expression
var updatedCustomer = customer with { Email = "new@email.com" };
```

Records et Immutabilité

Records Struct - Performance Optimale :

- **Stack allocation** pour petites structures
- **Zero copy** dans certains scénarios
- **Parfait** pour coordonnées, points, configurations

Applications WPF :

- **ViewModels immutables** plus prévisibles
- **État application** plus robuste
- **Debugging facilité** grâce au pattern matching
- **Thread-safety** naturelle

Primary Constructors

Révolution Syntaxique C# 12 :

- **Constructeurs inline** dans la déclaration de classe
- **Paramètres accessibles** dans tout le corps de classe
- **Injection de dépendances** simplifiée
- **Code boilerplate** drastiquement réduit

```
public class CustomerService(ILogger<CustomerService> logger, IRepository repo)
{
    public async Task<Customer> GetCustomerAsync(int id)
    {
        logger.LogInformation("Getting customer {Id}", id);
        return await repo.GetByIdAsync(id);
    }
}
```

Primary Constructors

Parfait pour WPF :

- **ViewModels** avec services injectés
- **Services** avec repositories
- **Contrôles** avec propriétés de configuration
- **Factories** avec paramètres

Bonnes Pratiques :

- Validation des paramètres dans le corps
- Combinaison avec required members
- Documentation des paramètres

Required Members

Garantie d'Initialisation C# 11 :

- **Compilation-time safety** pour propriétés critiques
- **Alternative moderne** aux constructeurs complexes
- **Flexibilité** d'initialisation maintenue
- **Documentation** des dépendances obligatoires

```
public class CustomerViewModel
{
    public required string Name { get; init; }
    public required string Email { get; init; }
    public string? Phone { get; init; }
}
```

Required Members

Applications WPF :

- **Propriétés de dépendance** critiques
- **Configuration** de contrôles
- **Injection** de services obligatoires
- **Initialisation** de ViewModels

Avantages :

- Bugs détectés à la compilation
- API plus claire et documentée
- Maintenance facilitée des contrats

Pattern Matching Avancé

List Patterns - Navigation Collections :

- **Déconstruction** de listes et arrays
- **Pattern matching** sur séquences
- **Conditions complexes** simplifiées

```
return items switch
{
    [] => "Empty",
    [var single] => $"Single: {single}",
    [var first, .. var rest] => $"First: {first}, Count: {rest.Length + 1}"
};
```

Pattern Matching Avancé

Switch Expressions Étendues :

- **Syntaxe concise** pour branchements complexes
- **Type inference** améliorée
- **Exhaustiveness checking** renforcé

Applications WPF :

- **Navigation** conditionnelle
- **Validation** de données utilisateur
- **État application** et transitions
- **Mapping** de données complexes

Collection Expressions

Syntaxe Unifiée C# 12 :

- **Création** simplifiée de collections
- **Spread operator** pour concaténation
- **Type inference** automatique
- **Performance** optimisée

```
// Ancienne syntaxe
var items = new List<string> { "a", "b" };
items.AddRange(otherItems);

// Nouvelle syntaxe
string[] items = ["a", "b", ..otherItems];
```

Collection Expressions

Impact WPF :

- **Binding** de collections simplifié
- **Initialisation** de listes de données
- **Combinaison** de sources multiples
- **Configuration** de contrôles

Exemples d'Usage :

- Listes de ViewModels pour ComboBox
- Collections d'éléments de menu
- Données de graphiques et tableaux

Asynchronous Streams (IAsyncEnumerable)

Streaming de Données Moderne :

- **IAsyncEnumerable** pour flux de données
- **yield return async** pour production paresseuse
- **await foreach** pour consommation asynchrone
- **Cancellation** intégrée avec CancellationToken

```
public async IAsyncEnumerable<Customer> LoadCustomersAsync()
{
    await foreach (var customer in dataService.GetCustomersStreamAsync())
        yield return customer;
}
```

Asynchronous Streams (IAsyncEnumerable)

Cas d'Usage WPF :

- **Chargement progressif** de grandes collections
- **Mise à jour temps réel** d'interfaces
- **Import/Export** de fichiers volumineux
- **Streaming** de données depuis API

Avantages :

- **Mémoire optimisée** - pas de chargement complet
- **Réactivité** - UI reste responsive
- **Scalabilité** - gère de gros volumes
- **Cancellation** - arrêt propre des opérations

Task et Async/Await Améliorés

ValueTask - Performance Optimale :

- **Allocation réduite** pour opérations synchrones rapides
- **Pooling** automatique des tâches
- **Interopérabilité** avec Task classique
- **Recommandé** pour méthodes fréquemment appelées

```
public ValueTask<bool> IsDataCachedAsync(string key)
    => cache.ContainsKey(key) ? new(true) : LoadFromDatabaseAsync(key);
```

Task et Async/Await Améliorés

ConfigureAwait(false) - Bonnes Pratiques :

- **Performance** - évite captures de contexte inutiles
- **Deadlock prevention** - réduit les risques de blocage
- **Recommandé** dans les couches service
- **Éviter** dans le code UI WPF

Task.Run vs Async :

- **Task.Run** pour CPU-bound operations
- **Async/await** pour I/O-bound operations
- **Dispatching** correct vers UI thread

Nullable Reference Types

Sécurité à la Compilation :

- **Activation** progressive par projet ou fichier
- **Annotations** explicites des intentions nullability
- **Warnings** pour usages potentiellement dangereux
- **Flow analysis** sophistiquée du compilateur

```
public class CustomerService(ILogger<CustomerService> logger)
{
    public Customer? FindCustomer(string? email) =>
        string.IsNullOrEmpty(email) ? null : repository.GetByEmail(email);
}
```

Nullable Reference Types

Impact WPF :

- **Binding** plus sûr avec null checks
- **ViewModels** robustes
- **Validation** automatique des propriétés
- **Debugging** facilité

Migration Progressive :

- Activation par fichier avec `#nullable enable`
- Suppression progressive des warnings
- Tests automatisés pour validation

Source Generators

Génération de Code à la Compilation :

- **Métaprogrammation** moderne et type-safe
- **Performance** - pas de réflexion runtime
- **Code généré** intégré à l'assembly
- **Debugging** possible du code généré

```
[AutoNotify] // Source generator attribute
public partial class CustomerViewModel
{
    private string _name; // Génère Name property avec INotifyPropertyChanged
}
```

Source Generators

Cas d'Usage WPF :

- **ViewModels** générés automatiquement
- **DependencyProperty** scaffolding
- **Converters** type-safe
- **Serialization** optimisée

Avantages :

- **Productivité** - moins de code boilerplate
- **Performance** - optimisations compile-time
- **Type Safety** - erreurs détectées tôt
- **Maintenance** - code généré cohérent

Architecture .NET Core vs .NET Framework

Philosophie .NET Core/5+ :

- **Cross-platform** par design
- **Performance** optimisée (GC, JIT)
- **Modularité** - packages granulaires
- **Open source** et développement rapide

Changements Architecturaux Majeurs :

- **SDK-style projects** plus simples
- **Global.json** pour versioning SDK
- **Runtime identifier (RID)** pour ciblage plateforme
- **Self-contained deployments** possible

Impact sur WPF :

- **Windows-only** mais .NET moderne
- **Performance** significativement améliorée
- **Tooling** Visual Studio optimisé
- **Packages** NuGet plus légers

Bénéfices Concrets :

- Temps de démarrage réduits
- Consommation mémoire optimisée
- Hot reload pendant développement
- Déploiement simplifié

Migration WPF vers .NET 9 - Stratégie

Outils de Migration :

- **Upgrade Assistant** Microsoft
- **Portability Analyzer** pour audit préalable
- **Try-Convert** pour conversion automatique
- **Manual migration** pour contrôle total

Points d'Attention :

- **Packages tiers** - vérifier compatibilité .NET 9
- **App.config** - migration vers appsettings.json
- **References** - nettoyage et modernisation
- **Breaking changes** - adaptation du code

Étapes Recommandées :

1. **Audit** - analyse compatibilité packages
2. **Tests** - couverture maximale avant migration
3. **Migration** - conversion progressive
4. **Validation** - tests exhaustifs post-migration

```
<!-- Migration de projet -->
<TargetFramework>net9.0-windows</TargetFramework>
<UseWPF>true</UseWPF>
<OutputType>WinExe</OutputType>
```

Gestion des Packages Tiers

Stratégie de Compatibilité :

- **NuGet Package Manager** - analyse des dépendances
- **Package Reference** moderne vs packages.config
- **Floating versions** vs versions fixes
- **Transitive dependencies** - gestion automatique

Packages Problématiques Courants :

- **Enterprise Library** → Microsoft.Extensions
- **Unity Container** → Microsoft.Extensions.DependencyInjection
- **Log4Net** → Microsoft.Extensions.Logging
- **Newtonsoft.Json** → System.Text.Json

Solutions de Migration :

- **Wrapper classes** pour transition graduelle
- **Adapter pattern** pour API incompatibles
- **Feature flags** pour rollback sécurisé
- **Testing** intensif des remplacements

```
<PackageReference Include="Microsoft.Extensions.Hosting" Version="9.0.0" />
<PackageReference Include="Microsoft.Extensions.Logging" Version="9.0.0" />
```

SDK-Style Projects

Nouveau Format de Projet :

- **Syntaxe simplifiée** et plus lisible
- **Auto-inclusion** des fichiers source
- **Multi-targeting** natif
- **NuGet** intégré sans packages.config

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net9.0-windows</TargetFramework>
    <UseWPF>true</UseWPF>
  </PropertyGroup>
</Project>
```

Avantages :

- **Fichiers plus courts** - réduction 80% en moyenne
- **Performance** build améliorée
- **Tooling** Visual Studio optimisé
- **MSBuild** moderne avec targets optimisés

Migration :

- Conversion automatique possible
- Validation manuelle recommandée
- Nettoyage des références obsolètes
- Tests post-migration essentiels

Configuration Moderne : App.config → appsettings.json

Évolution de Configuration :

- **JSON** plus lisible que XML
- **Hierarchical** structure intuitive
- **Environment-specific** configurations
- **Strong typing** avec IOptions pattern

Avantages :

- **IntelliSense** dans Visual Studio
- **Validation** schema JSON
- **Hot reload** sans redémarrage
- **Configuration providers** multiple

s

Integration WPF :

- Microsoft.Extensions.Configuration
- IOptions pour binding type-safe
- User Secrets pour développement
- Environment variables support

```
{
  "ConnectionStrings": {
    "Default": "Server=.;Database=MyApp;Integrated Security=true"
  },
  "Logging": {
    "LogLevel": { "Default": "Information" }
  }
}
```

Injection de Dépendances en WPF

Microsoft.Extensions.DependencyInjection :

- **Container** léger et performant
- **Lifetime management** (Singleton, Scoped, Transient)
- **Integration** native avec Logging/Configuration
- **Standards** industry pour DI

```
public partial class App : Application
{
    private ServiceProvider _serviceProvider;

    protected override void OnStartup(StartupEventArgs e)
    {
        var services = ConfigureServices();
        _serviceProvider = services.BuildServiceProvider();
    }
}
```

Injection de Dépendances en WPF

Registration Pattern :

- **Services** layer registration
- **ViewModels** registration
- **Repository** pattern integration
- **Factory** pattern support

Résolution :

- Constructor injection recommandé
- Service Locator anti-pattern à éviter
- Lazy initialization support

Microsoft.Extensions.Logging en WPF

Framework de Logging Unifié :

- **Abstraction** - changement de provider transparent
- **Performance** - lazy evaluation et filtering
- **Structured logging** - données searchables
- **Multiple providers** simultanés

```
services.AddLogging(builder =>
{
    builder.AddConsole()
           .AddDebug()
           .AddFile("logs/app-{Date}.txt")
           .SetMinimumLevel(LogLevel.Information);
});
```

Microsoft.Extensions.Logging en WPF

Providers Disponibles :

- **Console** - développement
- **Debug** - Visual Studio output
- **File** - persistance locale
- **EventLog** - Windows Event Log
- **Serilog** - structured logging avancé

vs Enterprise Library :

- Configuration plus simple
- Performance supérieure
- Écosystème plus riche
- Support cross-platform

Logging Structuré - Bonnes Pratiques

Structured Logging :

- **Template messages** avec paramètres nommés
- **Searchable data** dans les logs
- **Performance** - pas de string interpolation
- **Tooling** - analyseurs de logs modernes

```
logger.LogInformation("User {UserId} performed action {Action} at {Timestamp}",
    userId, actionName, DateTime.UtcNow);
```

Logging Structuré - Bonnes Pratiques

Niveaux de Log Appropriés :

- **Trace** - debugging détaillé
- **Information** - flux applicatif normal
- **Warning** - situations récupérables
- **Error** - erreurs gérées mais significatives
- **Critical** - erreurs fatales

Contexte et Correlation :

- **Scoped logging** pour tracking requests
- **Correlation IDs** pour diagnostics
- **User context** pour audit trails
- **Performance counters** intégrés

Export et Persistance des Logs

Stratégies de Stockage :

- **Files locaux** - rolling files avec archivage
- **Base de données** - requêtes complexes possibles
- **Elastic Stack** - search et analytics avancés
- **Cloud providers** - Azure Monitor, AWS CloudWatch

```
services.AddLogging(builder =>
{
    builder.AddFile("logs/app-{Date}.log", options =>
    {
        options.RetainedFileCountLimit = 30;
        options.FileSizeLimitBytes = 10_000_000;
    });
});
```

Export et Persistance des Logs

Filtrage et Performance :

- **Log levels** par category
- **Sampling** pour high-volume scenarios
- **Async logging** pour performance
- **Buffering** et batch writes

Monitoring et Alerting :

- **Health checks** integration
- **Metrics** exposition
- **Real-time monitoring** capabilities
- **Automated alerting** sur erreurs critiques

Gestion Centralisée des Exceptions

Handlers Globaux WPF :

- **DispatcherUnhandledException** - UI thread exceptions
- **AppDomain.UnhandledException** - background threads
- **TaskScheduler.UnobservedTaskException** - task exceptions
- **Custom exception boundaries** dans ViewModels

```
private void Application_DispatcherUnhandledException(object sender,
    DispatcherUnhandledExceptionEventArgs e)
{
    logger.LogError(e.Exception, "Unhandled UI exception");
    e.Handled = true; // Prevent app crash
}
```

Gestion Centralisée des Exceptions

Exception Propagation MVVM :

- **Command exception handling** centralisé
- **AsyncCommand** avec error reporting
- **UI feedback** pour erreurs utilisateur
- **Retry mechanisms** automatiques

Patterns Recommandés :

- **Result** pattern pour operations fallibles
- **Exception aggregation** pour batch operations
- **Circuit breaker** pour services externes
- **Graceful degradation** strategies

MVVM et Propagation d'Erreur

Error Handling dans ViewModels :

- **INotifyDataErrorInfo** pour validation
- **Error collections** pour feedback utilisateur
- **Command state** réflétant disponibilité
- **Loading states** avec indicateurs visuels

```
public class CustomerViewModel : ViewModelBase, INotifyDataErrorInfo
{
    public bool HasErrors => _errors.Count > 0;

    private void ValidateProperty(string value, [CallerMemberName] string propertyName = "")
    {
        // Validation logic with error collection update
    }
}
```

MVVM et Propagation d'Erreur

AsyncCommand Pattern :

- **Exception capture** et reporting
- **Loading state** management
- **Cancellation** support
- **Progress reporting** pour long operations

User Experience :

- **Error messages** contextuels et actionables
- **Retry buttons** pour operations récupérables
- **Offline mode** pour résilience
- **Progress indicators** pour feedback

Configuration Moderne Avancée

appsettings.json Hiérarchique :

- **Structure** intuitive et maintenable
- **Environment overrides** automatiques
- **IntelliSense** et validation schema
- **Hot reload** sans redémarrage application

```
{
  "App": {
    "Title": "Mon Application WPF",
    "Version": "2.1.0",
    "Features": {
      "EnableAnalytics": true,
      "MaxConcurrentUsers": 100
    }
  }
}
```

Configuration Moderne Avancée

Multiple Sources :

- **appsettings.json** - configuration de base
- **appsettings.{Environment}.json** - overrides par environnement
- **User Secrets** - développement local sécurisé
- **Environment variables** - déploiement cloud
- **Command line** - overrides runtime

Strong Typing avec IOptions :

- **Binding** automatique vers POCO
- **Validation** à l'injection
- **Hot reload** avec IOptionsMonitor
- **Snapshot** avec IOptionsSnapshot

Environnements et User Secrets

Gestion Multi-Environnement :

- **ASPNETCORE_ENVIRONMENT** variable pour WPF
- **Development/Staging/Production** configurations
- **Override hierarchy** bien définie
- **Configuration validation** au démarrage

User Secrets - Sécurité Développement :

```
dotnet user-secrets init
dotnet user-secrets set "ConnectionStrings:DefaultConnection" "Server=dev;..."
```

Environnements et User Secrets

Avantages User Secrets :

- **Stockage sécurisé** hors repository
- **Par utilisateur** et par projet
- **Integration** Visual Studio native
- **No accidental commits** de données sensibles

Best Practices :

- **Jamais** de secrets dans appsettings.json
- **Environment variables** en production
- **Key Vault** pour secrets partagés
- **Rotation** régulière des secrets

Gestion Dépendances NuGet Entreprise

Stratégie Versioning :

- **Semantic versioning** stricte (Major.Minor.Patch)
- **Lock files** pour builds reproductibles
- **Central Package Management** pour solutions multi-projets
- **Vulnerability scanning** automatisé

```
<PackageReference Include="Newtonsoft.Json" Version="13.0.3" />
<!-- Central Package Management -->
<GlobalPackageReference Include="Microsoft.Extensions.Logging" Version="9.0.0" />
```

Gestion Dépendances NuGet Entreprise

Private Feeds :

- **Azure Artifacts** pour packages internes
- **NuGet.Server** on-premise
- **Package source mapping** pour sécurité
- **Authentication** et authorization

Policies Entreprise :

- **Approved packages list** centralisée
- **License compliance** checking
- **Security vulnerability** alerts
- **Update policies** graduelles

Création de Packages NuGet Professionnels

Structure de Projet :

- **Class library** avec metadata complètes
- **Multi-targeting** pour compatibilité étendue
- **Dependencies** minimales et justifiées
- **Documentation** intégrée

```
<PropertyGroup>
  <PackageId>MyCompany.WPF.Controls</PackageId>
  <Version>1.2.0</Version>
  <Authors>Équipe Architecture</Authors>
  <Description>Contrôles WPF standardisés pour applications entreprise</Description>
  <PackageTags>wpf;controls;enterprise</PackageTags>
  <RepositoryUrl>https://github.com/mycompany/wpf-controls</RepositoryUrl>
</PropertyGroup>
```

Création de Packages NuGet Professionnels

Contenu Package :

- **Assemblies** principales
- **XML Documentation** pour IntelliSense
- **Symbols packages** pour debugging
- **README** et changelog
- **Samples** et documentation

Build et Distribution :

- **CI/CD pipeline** automatisé
- **Quality gates** (tests, analysis)
- **Staging feeds** pour validation
- **Production release** semi-automatique

Publication et Maintenance Packages

Publication Multi-Feed :

- **NuGet.org** pour packages open source
- **Private feeds** pour packages internes
- **Symbolic linking** entre feeds
- **Automated publishing** avec GitHub Actions

```
dotnet pack --configuration Release
dotnet nuget push MyPackage.1.2.0.nupkg --api-key [key] --source https://api.nuget.org/v3/index.json
```

Publication et Maintenance Packages

Versioning Strategy :

- **Breaking changes** = Major version
- **New features** = Minor version
- **Bug fixes** = Patch version
- **Preview releases** avec suffixes

Maintenance Long-terme :

- **Support matrix** clairement définie
- **Backward compatibility** guarantees
- **Migration guides** pour breaking changes
- **End-of-life** policies communiquées

Quality Assurance :

- **Automated testing** sur multiple frameworks
- **Performance benchmarks** regression detection
- **Documentation** toujours à jour
- **Community feedback** integration

Types de Déploiement .NET 9 WPF

Self-Contained Deployment :

- **Runtime inclus** - aucune dépendance système
- **Taille importante** - ~100MB pour WPF app simple
- **Isolation complète** - pas de conflits version
- **Target specific** - un build par OS/architecture

Framework-Dependent Deployment :

- **Runtime séparé** - installation .NET 9 requise
- **Taille réduite** - quelques MB seulement
- **Updates automatiques** - sécurité runtime centralisée
- **Shared dependencies** - optimisation mémoire

Single File Publishing :

- **Un seul exécutable** - distribution simplifiée
- **Extraction automatique** au premier lancement
- **Performance** - léger overhead startup
- **Debugging** - symbols séparés possibles

```
<PropertyGroup>
  <SelfContained>true</SelfContained>
  <RuntimeIdentifier>win-x64</RuntimeIdentifier>
  <PublishSingleFile>true</PublishSingleFile>
</PropertyGroup>
```

Comparaison avec Méthodes Traditionnelles

ClickOnce Legacy vs Moderne :

- **Sécurité** améliorée avec signing moderne
- **Update mechanism** plus robuste
- **Offline capability** native
- **Browser deployment** toujours possible

Setup.exe Traditional :

- **MSI** reste gold standard entreprise
- **Group Policy** deployment supporté
- **Uninstall** propre et complet
- **Registry** et system integration

Avantages .NET 9 Deployment :

- **Xcopy deployment** simple et fiable
- **Container** ready si nécessaire
- **Cloud** deployment facilité
- **Rollback** instantané possible

Choix Strategy :

- **Entreprise** → MSI/MSIX pour contrôle total
- **Distribution** → Single file self-contained
- **Internal tools** → Framework-dependent
- **Store apps** → MSIX obligatoire

MSIX - Packaging Moderne

Microsoft Store Integration :

- **Sandboxing** pour sécurité renforcée
- **Automatic updates** centralisées
- **Digital signing** obligatoire
- **App isolation** - pas d'impact système

MSIX vs MSI :

- **Installation** plus rapide et sûre
- **Uninstall** complet garanti
- **Streaming** - lancement avant download complet
- **Delta updates** - only changed files

Challenges MSIX : - **Legacy dependencies** parfois problématiques - **File system** redirection complexe - **Registry** virtualization limitations - **Debugging** plus complexe

Tools Ecosystem :

- **MSIX Packaging Tool** pour conversion
- **Visual Studio** integration native
- **Advanced Installer** support MSIX
- **WiX Toolset** v4+ avec MSIX

```
<PropertyGroup>
  <WindowsPackageType>MSIX</WindowsPackageType>
  <WindowsAppSDKSelfContained>true</WindowsAppSDKSelfContained>
  <Platform>x64</Platform>
</PropertyGroup>
```

Setup Personnalisé et Ressources

Inclusion Ressources :

- **Native DLLs** - packaging et loading
- **Configuration files** - defaults et overrides
- **Static content** - images, templates, help
- **Database scripts** - migration automatique

```
<ItemGroup>
  <Content Include="Resources\**\*" CopyToOutputDirectory="PreserveNewest" />
  <None Include="NativeDlls\**\*" CopyToOutputDirectory="PreserveNewest" />
</ItemGroup>
```

Setup Personnalisé et Ressources

Custom Actions :

- **Post-install** configuration
- **Registry** entries si nécessaire
- **Service** installation/configuration
- **Prerequisites** checking et installation

Advanced Scenarios :

- **Multi-language** support avec resource satellites
- **Plugin architecture** avec MEF ou custom loading
- **Auto-updater** intégré à l'application
- **Crash reporting** et telemetry setup

Security Considerations :

- **Code signing** obligatoire pour distribution
- **Certificate** management et renewal
- **Trust levels** et user prompts
- **Antivirus** whitelisting strategies

ADO.NET en .NET 9 - Modernisation

Évolutions Majeures :

- **Performance** améliorée - moins d'allocations
- **Async** everywhere - SqlCommand, DataReader
- **Modern patterns** - using statements, ConfigureAwait
- **Security** renforcée - Always Encrypted support

```
await using var connection = new SqlConnection(connectionString);
await connection.OpenAsync();
await using var command = new SqlCommand(sql, connection);
await using var reader = await command.ExecuteReaderAsync();
```

ADO.NET en .NET 9 - Modernisation

Connection Management :

- **Connection pooling** optimisé automatiquement
- **Connection resiliency** avec retry policies
- **Health checks** intégrés
- **Monitoring** via Activity et Tracing

vs Entity Framework :

- **Performance** - ADO.NET plus rapide pour bulk operations
- **Control** - SQL queries explicites
- **Legacy** - compatibilité avec stored procedures
- **Complexity** - EF pour développement plus rapide

Patterns Modernes ADO.NET

Repository Pattern Async :

- **Interface contracts** bien définis
- **Dependency injection** friendly
- **Unit testing** facilité
- **Error handling** centralisé

```
public interface ICustomerRepository
{
    Task<Customer?> GetByIdAsync(int id, CancellationToken cancellationToken = default);
    Task<IEnumerable<Customer>> GetAllAsync(CancellationToken cancellationToken = default);
}
```

Patterns Modernes ADO.NET

DataReader vs DataTable :

- **Streaming** avec IAsyncEnumerable pour gros volumes
- **Memory efficient** - pas de matérialisation complète
- **Real-time** processing possible
- **Cancellation** support natif

Parameterized Queries :

- **SQL injection** prevention absolue
- **Performance** - plan caching automatique
- **Type safety** avec parameters
- **Stored procedures** vs dynamic SQL

Transaction Management :

- **TransactionScope** pour distributed transactions
- **Database transactions** pour performance locale
- **Ambient transactions** avec async/await
- **Rollback strategies** robustes

Dapper et EF Core Compléments

Dapper - Micro ORM :

- **Performance** proche ADO.NET natif
- **Simplicity** - mapping automatique POCO
- **SQL control** - queries explicites maintenues
- **Learning curve** minimale

```
var customers = await connection.QueryAsync<Customer>(
    "SELECT * FROM Customers WHERE Region = @region",
    new { region = "Europe" });
```

Dapper et EF Core Compléments

Entity Framework Core :

- **Code First** - database schema depuis models
- **Migrations** - évolution schema automatisée
- **Change tracking** - update automatique
- **LINQ** - queries type-safe et IntelliSense

Hybrid Approach :

- **EF Core** pour CRUD et relations complexes
- **Dapper** pour queries performance-critiques
- **ADO.NET** pour bulk operations et stored procedures
- **Consistency** - same connection string et transaction

Performance Tuning :

- **Connection strings** optimisées
- **Command timeout** appropriés
- **Batch operations** pour multiple inserts/updates
- **Read-only** connections pour reporting

Architecture .NET MAUI

Multi-platform Strategy :

- **Single codebase** - Windows, macOS, iOS, Android
- **Native performance** - compilation native
- **Platform-specific** customizations possibles
- **Shared business logic** maximisée

MAUI vs WPF Migration :

- **ViewModels** - réutilisation directe possible
- **Services** - migration straightforward
- **UI** - redesign avec MAUI controls
- **Platform features** - handlers customs si nécessaire

Architecture .NET MAUI

```
public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder.UseMauiApp<App>()
               .ConfigureFonts(fonts => fonts.AddFont("OpenSans-Regular.ttf"));
        return builder.Build();
    }
}
```

Shared Components :

- **Business logic** - .NET Standard libraries
- **Data access** - Entity Framework Core
- **Services** - HTTP clients, caching
- **Models** - POCOs et DTOs

Platform Specifics :

- **Native APIs** via platform handlers
- **File system** access différentié
- **Permissions** model mobile
- **Deployment** stores vs enterprise

Blazor Hybrid - WPF + Web

Architecture Hybride :

- **WebView2** intégré dans WPF
- **Blazor components** dans WebView
- **Interop** bidirectionnel C# ↔️ JavaScript
- **State sharing** entre WPF et Blazor

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        blazorWebView.HostPage = "wwwroot/index.html";
        blazorWebView.Services = serviceProvider;
        blazorWebView.RootComponents.Add(new RootComponent
        {
            Selector = "#app",
            ComponentType = typeof(App)
        });
    }
}
```

Blazor Hybrid - WPF + Web

Use Cases Perfects :

- **Dashboards** avec visualisations complexes
- **Reports** avec layouts flexibles
- **Forms** dynamiques configurables
- **Content** riche avec HTML/CSS

Avantages :

- **Web skills** réutilisées en desktop
- **Component libraries** web disponibles
- **Responsive** design natif
- **Rapid prototyping** possible

Considérations :

- **Performance** - overhead WebView2
- **Deployment** - WebView2 runtime requis
- **Debugging** - tools web + desktop
- **Styling** - cohérence avec WPF themes

Réutilisation de Composants

Strategy Layers :

- **Business Logic** - .NET Standard class libraries
- **Data Models** - shared POCOs et contracts
- **Services** - HTTP, caching, validation
- **ViewModels** - adapters pour différentes UI

Shared Libraries Architecture :

```
├── MyApp.Core (Business Logic)
├── MyApp.Data (Data Access)
├── MyApp.Services (External APIs)
├── MyApp.WPF (Desktop UI)
├── MyApp.MAUI (Mobile/Cross-platform)
└── MyApp.Blazor (Web UI)
```

Réutilisation de Composants

Abstraction Layers :

- **INavigationService** - navigation abstraite
- **IDialogService** - dialogs cross-platform
- **IFileService** - file access unifié
- **IPlatformService** - services spécifiques plateforme

Code Sharing Strategy :

- **90%** business logic partagé
- **70%** ViewModels réutilisables
- **50%** Views adaptées par plateforme
- **30%** platform-specific features

Migration Path :

- Phase 1 : Extraction business logic
- Phase 2 : ViewModels abstraction
- Phase 3 : Services uniformisation
- Phase 4 : Multi-platform deployment

Roadmap et Conseils Futurs

Tendances Technologiques :

- **.NET** unified platform consolidation
- **Cloud-first** development paradigm
- **Microservices** et containerization
- **AI/ML** integration native

WPF Future :

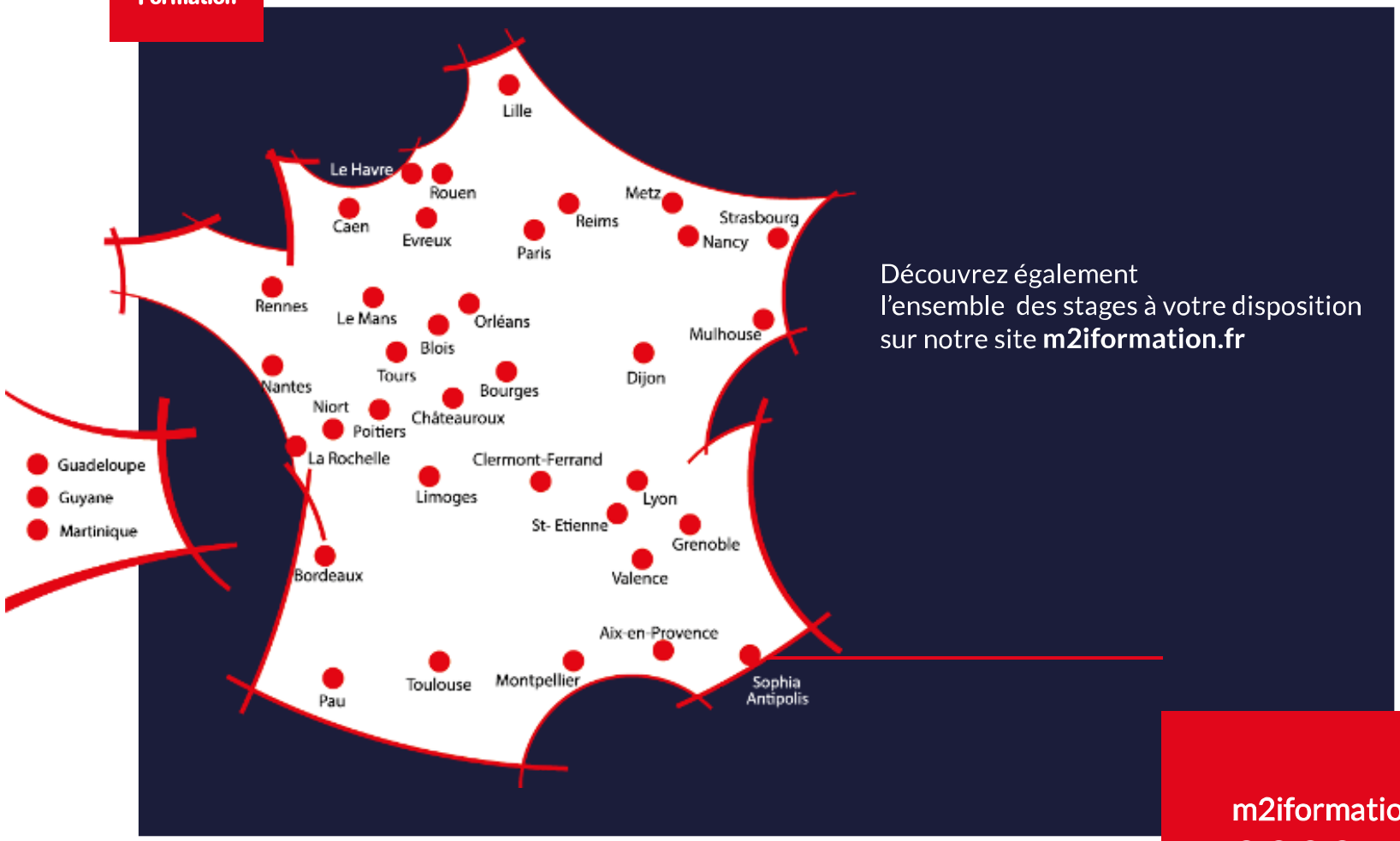
- **Maintenance mode** - pas de nouvelles fonctionnalités majeures
- **Security updates** et bug fixes continuent
- **Performance** optimizations occasionnelles
- **Migration path** vers technologies modernes

Préparation Migration :

- **Clean Architecture** dès maintenant
- **Dependency Injection** systématique
- **MVVM** strict sans code-behind
- **Service layers** bien abstraites

Technologies Émergentes :

- **WebAssembly** pour applications web
- **Progressive Web Apps** pour mobile
- **.NET MAUI** pour cross-platform native
- **Blazor Hybrid** pour transition graduelle



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr