CESI
ÉCOLE D'INGÉNIEURS

EasySave v1.0 – Documentation Technique de l'application

Tuteur : M. EL FALOU Mohamad

Rayane HASSANI

Bastien LEDOUX

Léo MARTIN

Février 2026

Table des matières

1. Présentation du produit 3
2. Configuration requise 4
3. Installation 4
4. Emplacement des fichiers 5
5. Architecture 6
6. Fonctionnalités 8
6.1 Gestion des travaux de sauvegarde 8
6.2 Types de sauvegarde 8
6.3 Support linguistique 8
7. Interface en ligne de commande (CLI) 9
8. Fichiers de configuration 10
9. Fichiers de logs 11
10. Fichier d'état 12
11. Dépannage 13
12. Diagrammes UML 14
12.1 Diagramme de cas d'utilisation 14
12.2 Diagramme de classes 16
12.3 Diagramme de séquence - Créer un travail de sauvegarde 18
12.4 Diagramme de séquence - Exécuter une sauvegarde complète 20
12.5 Diagramme d'activité 22
13. Glossaire 24
14. Annexe 25
..... 25

1. Présentation du produit

Description

EasySave est un logiciel de sauvegarde développé par **ProSoft**, permettant aux utilisateurs de créer et d'exécuter des travaux de sauvegarde. L'application prend en charge les stratégies de sauvegarde complète et différentielle.

Informations de version :

Attribut	**Valeur**
Version	1.0
Date de sortie	06/02/2025
Framework	.NET 10.0
Interface	Application Console

Fonctionnalités principales :

- Création jusqu'à 5 travaux de sauvegarde
- Sauvegarde complète (copie tous les fichiers)
- Sauvegarde différentielle (copie uniquement les fichiers modifiés)
- Support multilingue (Anglais / Français)
- Exécution des travaux de sauvegarde en ligne de commande
- Journalisation en temps réel
- Suivi d'état en temps réel

2. Configuration requise

Configuration minimale

Composant	Exigence
Système d'exploitation	Windows 10 / Windows 11 / Windows Server 2019+
Framework	.NET 10.0 Runtime
RAM	512 Mo minimum
Espace disque	50 Mo pour l'installation
Processeur	Compatible x64

Types de stockage supportés

- Disques locaux (C:, D:, etc.)
- Disques externes (USB, HDD, SSD)
- Lecteurs réseau (chemins UNC : \serveur\partage)

---

3. Installation

Étapes d'installation

1. S'assurer que .NET 10.0 Runtime est installé
2. Extraire le package EasySave à l'emplacement souhaité
3. Exécuter EasySave.exe pour démarrer l'application

Vérifier l'installation

```
EasySave.exe --version
```

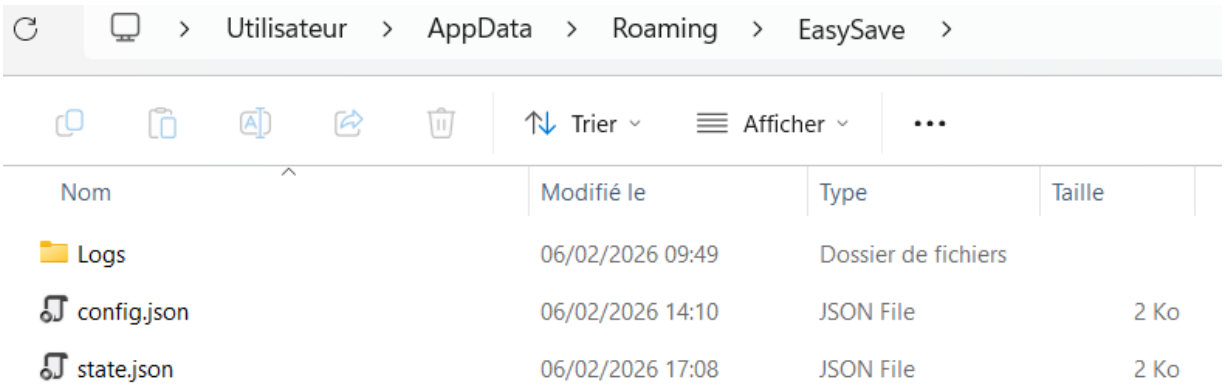
4. Emplacement des fichiers

Fichiers de l'application

Fichier	Emplacement	Description
EasySave.exe	Dossier d'installation	Exécutable principal de l'application
EasySave.dll	Dossier d'installation	Bibliothèque principale de l'application
EasyLog.dll	Dossier d'installation	Bibliothèque de journalisation (logs)
EasySave.deps.json	Dossier d'installation	Fichier de dépendances (.NET)
EasySave.runtimeconfig.json	Dossier d'installation	Configuration du runtime .NET
EasySave.pdb	Dossier d'installation	Fichier de débogage (symboles)
EasyLog.pdb	Dossier d'installation	Fichier de débogage de la bibliothèque EasyLog

Fichiers de données

Tous les fichiers de données sont stockés dans le dossier AppData de l'utilisateur :



Exemple de chemin complet

C:\Users\{NomUtilisateur}\AppData\Roaming\EasySave\

Comment y accéder

1. Appuyer sur Win + R
2. Taper %APPDATA%\EasySave
3. Appuyer sur Entrée

5. Architecture

Patron de conception

L'application suit le patron d'architecture **MVVM (Model-View-ViewModel)**, préparant une future migration vers une interface graphique.

Structure du projet

```
EasySave Solution
├── EasyLog (Class Library - DLL)
│   ├── Models/
│   │   └── ModelLogEntry.cs
│   └── Services/
│       ├── ILogService.cs
│       └── LogService.cs
│
└── EasySave (Console Application)
    ├── Models/
    │   ├── Enums/
    │   │   ├── BackupType.cs
    │   │   ├── BackupStatus.cs
    │   │   └── Language.cs
    │   ├── ModelBackupJob.cs
    │   └── ModelBackupState.cs
    ├── Services/
    │   ├── Interfaces/
    │   │   ├── IConfigurationService.cs
    │   │   └── IStateService.cs
    │   ├── ConfigurationService.cs
    │   ├── LanguageManager.cs
    │   ├── ServiceBackupExecution.cs
    │   ├── ServiceBackupScheduler.cs
    │   ├── ServiceCommandLineParser.cs
    │   └── StateService.cs
    ├── Strategies/
    │   ├── IBackupStrategy.cs
    │   ├── FullBackupStrategy.cs
    │   └── DifferentialBackupStrategy.cs
    ├── Utils/
    │   └── FileUtils.cs
    ├── ViewModels/
    │   └── MainViewModel.cs
    ├── Views/
    │   └── ConsoleView.cs
    └── Program.cs
```

Principes architecturaux

L'architecture du projet sépare la vue console (**ConsoleView**) de la logique métier (**MainViewModel**). Le ViewModel gère les travaux de sauvegarde, leur persistance et leur exécution ; la vue ne fait qu'afficher et relayer les actions utilisateur. Cette séparation permet de remplacer plus tard la console par une interface WPF en ne modifiant que la couche présentation, sans changer le code métier.

Le **pattern Strategy** est utilisé pour les algorithmes de sauvegarde : l'interface IBackupStrategy définit un contrat commun implémenté par FullBackupStrategy et DifferentialBackupStrategy, permettant d'ajouter de nouveaux types de sauvegarde sans impacter le code existant.

Le pattern Singleton est appliqué au LogService pour garantir une instance unique de journalisation à travers l'application. Enfin, les services techniques (Log, State, Configuration) sont isolés derrière des interfaces, facilitant les tests et la maintenance.

Responsabilités des composants principaux :

Composant	**Responsabilité**
ConsoleView	Affichage de l'interface utilisateur et capture des entrées
MainViewModel	Orchestration de la logique métier
ServiceBackupExecution	Moteur d'exécution des sauvegardes
ServiceBackupScheduler	Exécution séquentielle des travaux
ServiceCommandLineParser	Analyse des arguments CLI
FullBackupStrategy	Copie tous les fichiers
DifferentialBackupStrategy	Copie uniquement les fichiers modifiés
LogService	Écriture des fichiers de logs journaliers (EasyLog.dll)
StateService	Écriture du fichier d'état en temps réel
LanguageManager	Support multilingue
FileUtils	Utilitaires système de fichiers

6. Fonctionnalités

6.1 Gestion des travaux de sauvegarde

Créer un travail de sauvegarde

- Maximum 5 travaux autorisés
- Chaque travail nécessite :
    - **Nom** : Identifiant unique
    - **Répertoire source** : Doit exister
    - **Répertoire cible** : Créé s'il n'existe pas
    - **Type** : Complet ou Différentiel

Lister les travaux de sauvegarde

- Affiche tous les travaux configurés
- Montre le nom, le type, la source et la cible

Supprimer un travail de sauvegarde

- Supprime le travail de la configuration
- Ne supprime pas les fichiers sauvegardés

6.2 Types de sauvegarde

Type	Comportement	Cas d'utilisation
Complet	Copie TOUS les fichiers de la source vers la cible	Première sauvegarde, récupération complète nécessaire
Différentiel	Copie uniquement les fichiers NOUVEAUX ou MODIFIÉS	Sauvegardes régulières, économise temps et espace

Logique de sauvegarde différentielle

Un fichier est copié si :

- Le fichier cible N'EXISTE PAS
- La date de modification source > Date de modification cible

6.3 Support linguistique

Langue	Code
Anglais	EN
Français	FR

La langue peut être changée depuis le menu principal (Option 5).

7. Interface en ligne de commande (CLI)

Pour utiliser le mode CLI, ouvrir un **Invite de commandes (CMD)** et naviguer vers le répertoire de l'exécutable :

```
cd  "C:\chemin\vers\projet\EasySave\EasySave\bin\Debug\net10.0"
```

Syntaxe :

```
EasySave.exe [indices_travaux]
```

Modes d'exécution

Mode	Commande	Description
Interactif	EasySave.exe	Lance le menu de l'application
CLI	EasySave.exe <indices>	Exécute les travaux spécifiés

Syntaxe des indices

Syntaxe	Description	Exemple	Résultat
Simple	Un seul travail	EasySave.exe 1	Exécute le travail 1
Plage	Plage de travaux	EasySave.exe 1-3	Exécute les travaux 1, 2, 3
Liste	Travaux spécifiques	EasySave.exe 1;3	Exécute les travaux 1 et 3

Exemples d'utilisation :

```
# Exécuter uniquement le travail 1
EasySave.exe 1

# Exécuter les travaux 1 à 3 (séquentiel)
EasySave.exe 1-3

# Exécuter uniquement les travaux 1 et 3
EasySave.exe 1;3

# Exécuter tous les 5 travaux
EasySave.exe 1-5
```

8. Fichiers de configuration

config.json

Emplacement

%APPDATA%\EasySave\config.json

Structure


```
{
  "Name": "Backup_5",
  "SourceDirectory": "C:\\Users\\Utilisateur\\Documents\\Diagrammes_V1",
  "TargetDirectory": "C:\\Users\\Utilisateur\\Documents\\3\u00E8me Ann\u00E9e\\BLOC g\u00E9nie logiciel\\Projet\\V1",
  "Type": 0
}
```


Description des champs

Champ	Type	Description
Name	string	Nom unique du travail
SourceDirectory	string	Chemin du dossier source
TargetDirectory	string	Chemin du dossier cible
Type	int	0 = Complet, 1 = Différentiel

Valeurs des types de sauvegarde

Valeur	Type
0	Complet
1	Différentiel

9. Fichiers de logs

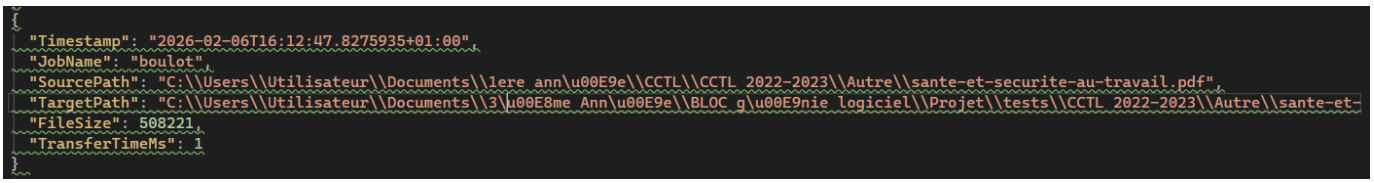
Emplacement

%APPDATA%\EasySave\Logs\YYYY-MM-DD.json

Nommage des fichiers

- Un fichier par jour
- Format : YYYY-MM-DD.json
- Exemple : 2025-02-06.json

Structure d'une entrée de log



Description des champs

Champ	Type	Description
Timestamp	DateTime	Horodatage de l'action (ISO 8601)
JobName	string	Nom du travail de sauvegarde
SourcePath	string	Chemin du fichier source (format UNC)
TargetPath	string	Chemin du fichier cible (format UNC)
FileSize	long	Taille du fichier en octets
TransferTimeMs	long	Temps de transfert en millisecondes (-1 si erreur)

Format de chemin UNC

Tous les chemins sont convertis au format UNC :

- Local : C:\Dossier\fichier.txt → \\NOM-PC\C$\Dossier\fichier.txt
- Réseau : Déjà au format UNC

Valeurs du temps de transfert

Valeur	Signification
>= 0	Transfert réussi (temps en ms)
-1	Erreur lors du transfert

10. Fichier d'état

Emplacement

%APPDATA%\EasySave\state.json

Objectif

Suit la progression en temps réel des travaux de sauvegarde.

Structure

```
{
  "JobName": "Backup_5",
  "Timestamp": "2026-02-06T17:08:02.1726732+01:00",
  "Status": 2,
  "TotalFilesToCopy": 7,
  "TotalFilesSize": 2552918,
  "Progression": 100,
  "RemainingFiles": 0,
  "RemainingFilesSize": 0,
  "CurrentSourceFile": null,
  "CurrentDestinationFile": null
}
```

Description des champs

Champ	Type	Description
JobName	string	Nom du travail de sauvegarde
Timestamp	DateTime	Horodatage de la dernière action
Status	int	État actuel (voir ci-dessous)
TotalFilesToCopy	int	Nombre total de fichiers à copier
TotalFilesSize	long	Taille totale en octets
Progression	int	Pourcentage de progression (0-100)
RemainingFiles	int	Fichiers restants à copier
RemainingFilesSize	long	Taille restante en octets
CurrentSourceFile	string	Chemin du fichier source actuel
CurrentDestinationFile	string	Chemin du fichier cible actuel

Valeurs d'état (« Status »)

Valeur	État	Description
0	Inactif	Travail non en cours
1	Actif	Travail en cours d'exécution
2	Terminé	Travail terminé avec succès
3	Erreur	Le travail a rencontré une erreur
4	En pause	Travail en pause (fonctionnalité v3.0)

11. Dépannage

Problèmes courants

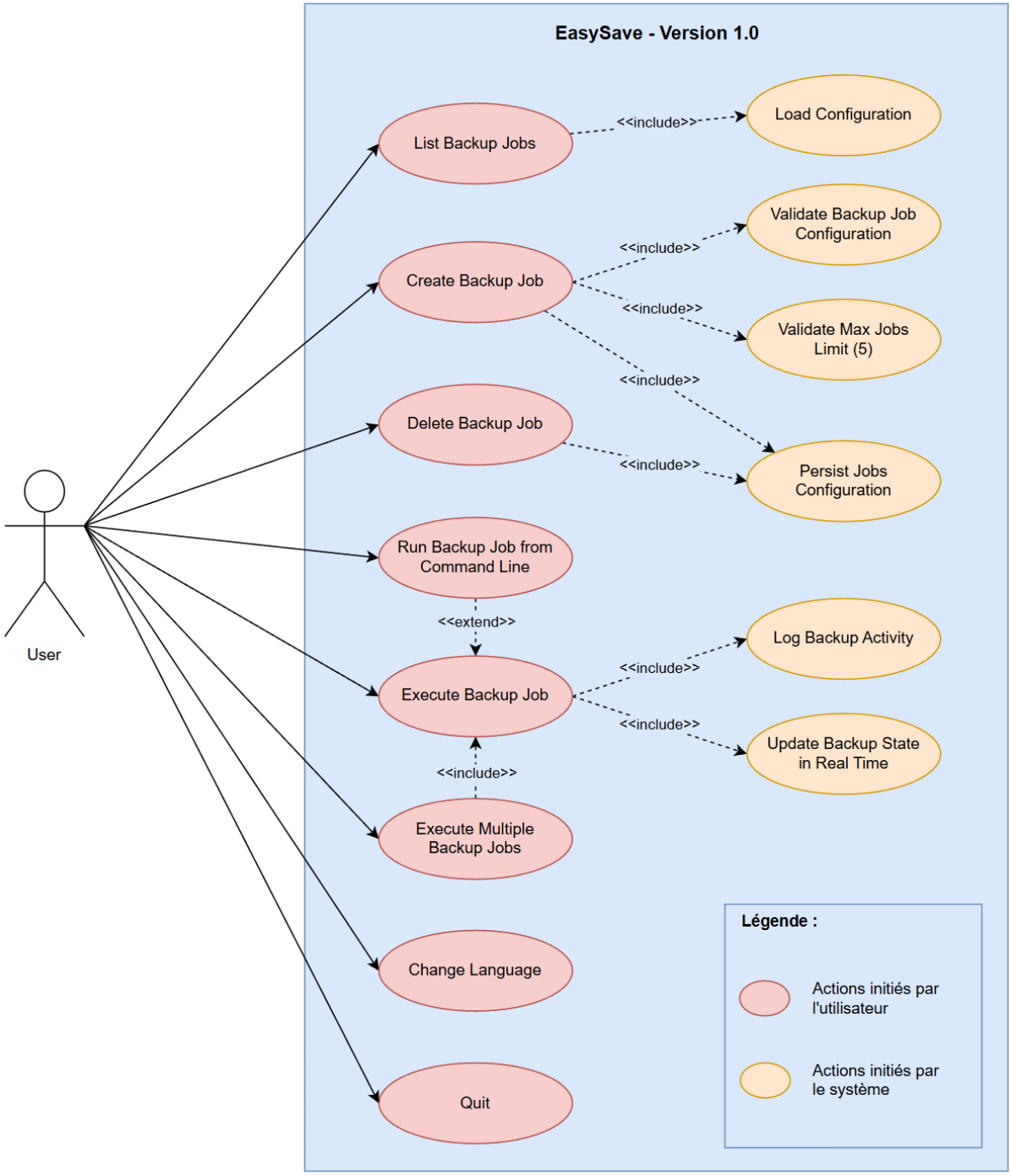
Message d'erreur	Cause	Solution
"Maximum 5 travaux autorisés"	Limite atteinte	Supprimer un travail existant
"Chemin invalide"	Répertoire source inexistant	Vérifier le chemin source
"Le nom existe déjà"	Nom en double	Utiliser un nom unique
"Accès refusé"	Fichier verrouillé ou permissions	Fermer les applications ou exécuter en Administrateur
Application plante au démarrage	config.json corrompu	Supprimer %APPDATA%\EasySave\config.json
Commande CLI non reconnue	Mauvais répertoire	Se placer dans le dossier d'installation (.\EasySave\bin\Debug\net10.0\)

Réinitialisation complète

1. Fermer l'application
2. Supprimer le dossier %APPDATA%\EasySave\
3. Redémarrer l'application

12. Diagrammes UML

12.1 Diagramme de cas d'utilisation



Acteur et Interactions

- **Acteur Principal (User) :** Représente l'utilisateur final pilotant l'application. Il initie l'ensemble des fonctionnalités via l'interface interactive ou par des commandes directes.
- **Système (EasySave - Version 1.0) :** Englobe l'ensemble des fonctionnalités internes et des garanties de persistance offertes par la solution.

Cas d'Utilisation Initiés par l'Utilisateur

Ces cas représentent les fonctionnalités directement accessibles depuis l'interface utilisateur :

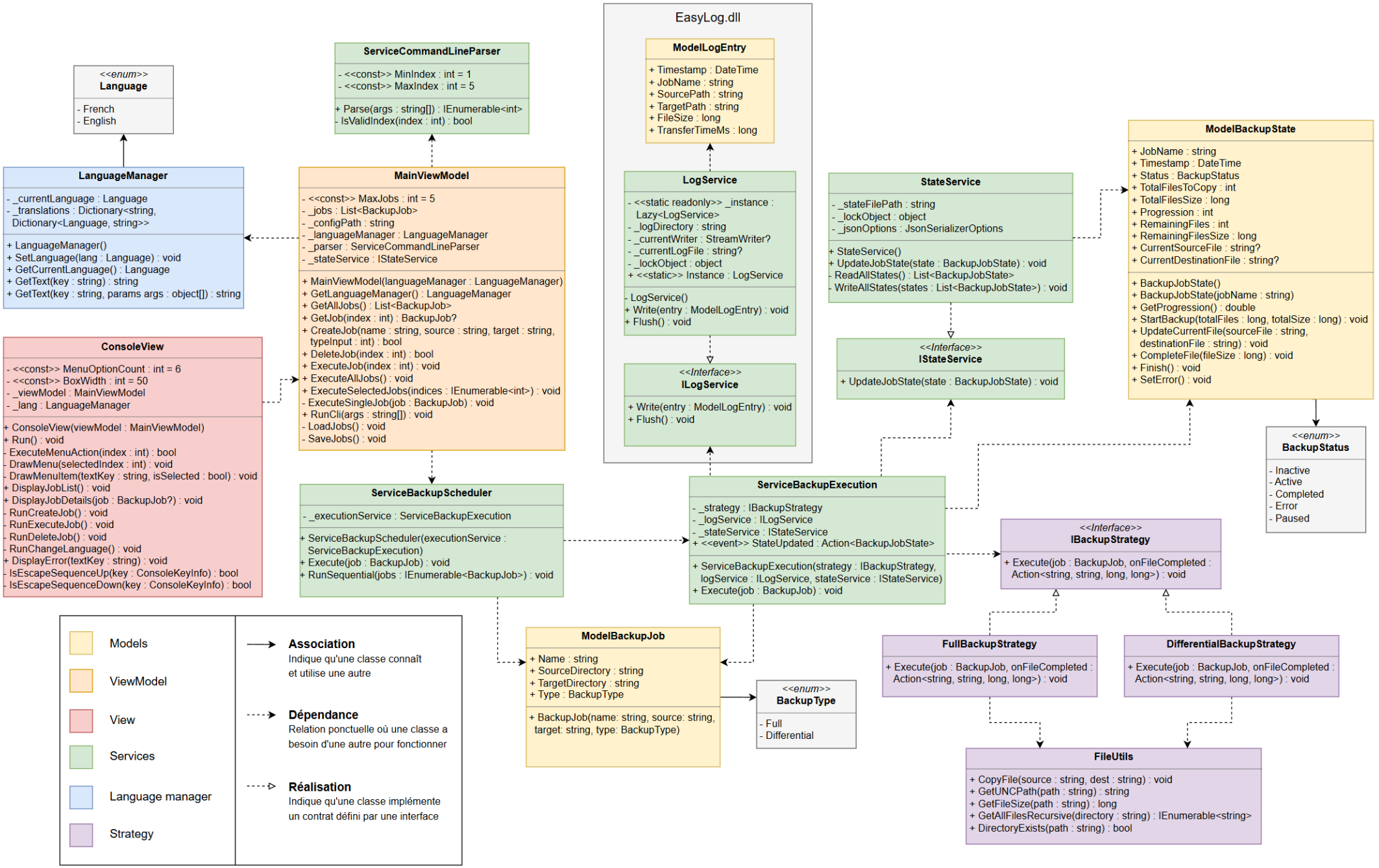
- **Configuration des travaux :** L'utilisateur peut lister (**List**), créer (**Create**) ou supprimer (**Delete**) des travaux de sauvegarde.
- **Modes d'Exécution :**
    - **Execute Backup Job :** L'action de base pour lancer une sauvegarde.
    - **Execute Multiple Backup Jobs :** Permet le lancement séquentiel d'une sélection de travaux (via une relation d'inclusion <<include>> vers l'exécution unitaire).
    - **Run Backup Job from Command Line :** Une extension optionnelle <<extend>> permettant de déclencher les sauvegardes via des arguments de console sans passer par le menu interactif.
- **Administration :** Gestion de la localisation (**Change Language**) et fermeture sécurisée du programme (**Quit**).

Processus Système Automatisés (Actions "Include")

Le diagramme illustre la robustesse du système grâce à des actions de contrôle et de suivi qui sont systématiquement incluses lors des opérations majeures :

- **Lors de la Création/Suppression :** Le système procède à la validation de la configuration et de la limite légale des 5 travaux. Toute modification entraîne l'action **Persist Jobs Configuration**, garantissant la sauvegarde des données sur disque.
- **Lors de l'Exécution :** Toute sauvegarde inclut obligatoirement deux services critiques :
    - **Log Backup Activity :** Génération des fichiers de logs journaliers.
    - **Update Backup State in Real Time :** Mise à jour continue du fichier d'état (progression, fichiers restants) pour le monitoring.
- **Au Démarrage :** L'action **Load Configuration** est incluse dès la demande de listing des travaux, assurant la synchronisation avec les données persistées.

12.2 Diagramme de classes



Architecture MVVM (Model-View-ViewModel)

L'application adopte le patron d'architecture **MVVM**, favorisant une séparation nette entre l'interface utilisateur et la logique métier :

- **View (ConsoleView)** : Responsable du rendu visuel en console et de la capture des entrées utilisateur. Elle communique exclusivement avec le ViewModel.
- **ViewModel (MainViewModel)** : Pivot central de l'application. Il contient la logique de présentation, gère la liste des travaux de sauvegarde (List<BackupJob>) et orchestre les services de persistance et d'exécution.
- **Models** : Classes de données pures comme ModelBackupJob (configuration) et ModelBackupState (état dynamique).

Design Pattern Strategy

Pour répondre à l'exigence de flexibilité des types de sauvegarde, le système utilise le pattern **Strategy** :

- **Interface IBackupStrategy** : Définit un contrat unique via la méthode Execute. Elle permet au moteur d'exécution d'appeler n'importe quel algorithme de manière interchangeable.
- **Stratégies Concrètes** : FullBackupStrategy et DifferentialBackupStrategy implémentent les algorithmes spécifiques. Ce découplage permet l'ajout futur de nouveaux types de sauvegarde sans modifier le code existant.

Modularité et Services Techniques

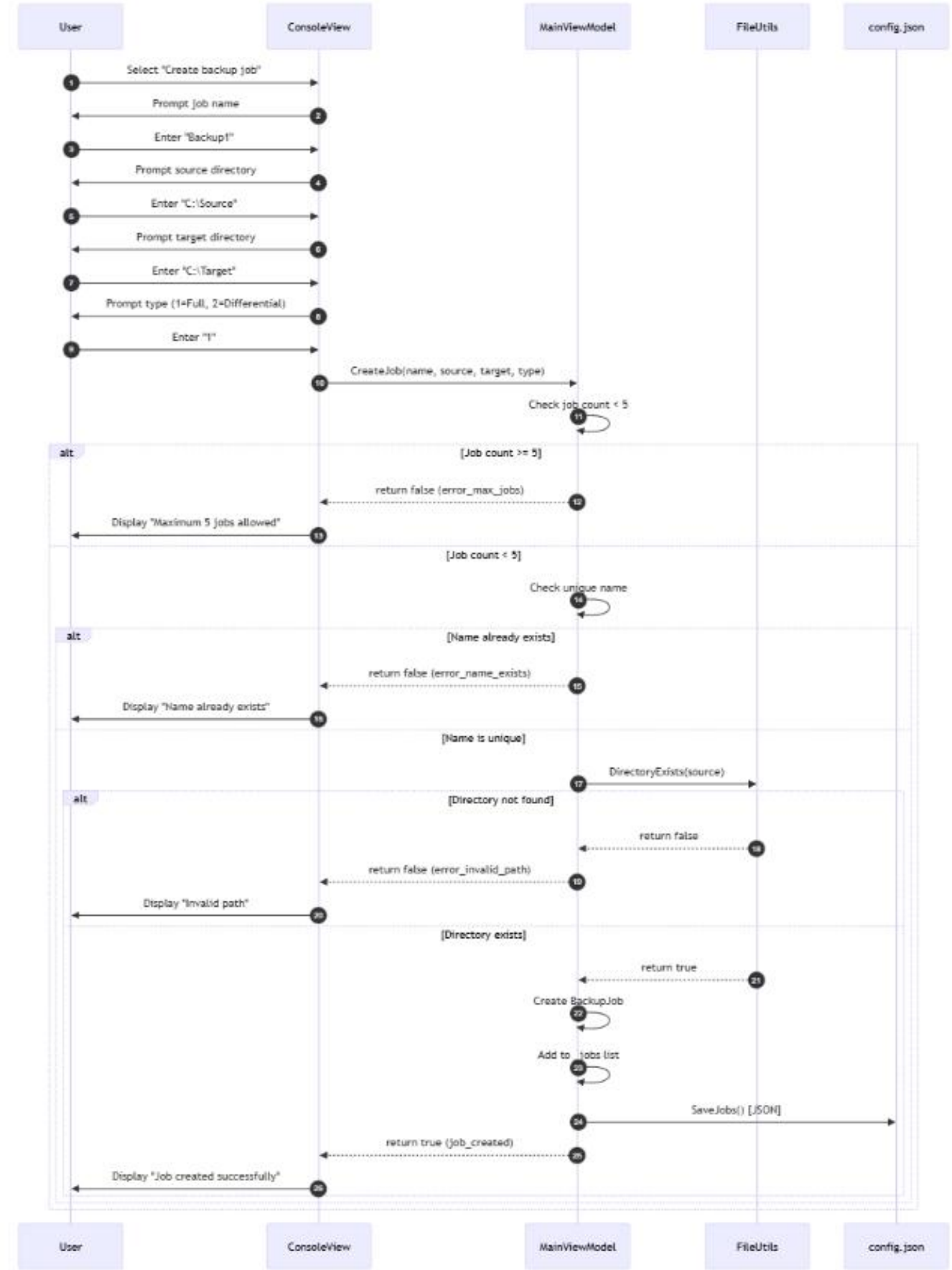
L'architecture s'appuie sur des services spécialisés pour garantir la maintenabilité :

- **Log Management (EasyLog.dll)** : Bibliothèque externe isolée contenant le LogService (implémenté en Singleton) et ModelLogEntry. Elle assure la journalisation des transferts indépendamment du reste du système.
- **State Management (StateService)** : Gère la persistance de l'état en temps réel. Il implémente l'interface IStateService et utilise la sérialisation JSON pour sauvegarder l'avancement des travaux.
- **Execution & Scheduling** : ServiceBackupExecution pilote le processus de sauvegarde unitaire, tandis que ServiceBackupScheduler orchestre les exécutions séquentielles ou multiples.
- **Utility Helper (FileUtils)** : Centralise les opérations de bas niveau sur le système de fichiers (copie, calcul de taille, vérifications) afin d'assurer une gestion d'erreurs uniforme.

Gestion de l'Environnement

- **LanguageManager** : Gère la localisation de l'interface en français ou en anglais via un dictionnaire de traductions piloté par l'énumération Language.
- **ServiceCommandLineParser** : Analyse les arguments fournis au démarrage (CLI) pour permettre le pilotage automatique de l'application sans interface interactive.

12.3 Diagramme de séquence - Créer un travail de sauvegarde



Participants

Participant	Rôle
User	Utilisateur final initiant l'action
ConsoleView	Interface console capturant les entrées
MainViewModel	Orchestrateur de la logique métier
FileUtils	Utilitaire de validation des chemins
config.json	Fichier de persistance des travaux

Flux d'exécution

L'utilisateur initie la création d'un travail depuis le menu principal. La ConsoleView collecte séquentiellement les informations nécessaires : nom du travail, répertoire source, répertoire cible et type de sauvegarde (1=Complet, 2=Différentiel).

Une fois les données saisies, la ConsoleView délègue la création au MainViewModel via CreateJob(). Le ViewModel exécute alors une chaîne de validations :

1. Validation de la limite : Vérification que le nombre de travaux est inférieur à 5
2. Validation de l'unicité : Contrôle que le nom n'existe pas déjà
3. Validation du chemin : Appel à FileUtils pour confirmer l'existence du répertoire source

Gestion des cas alternatifs

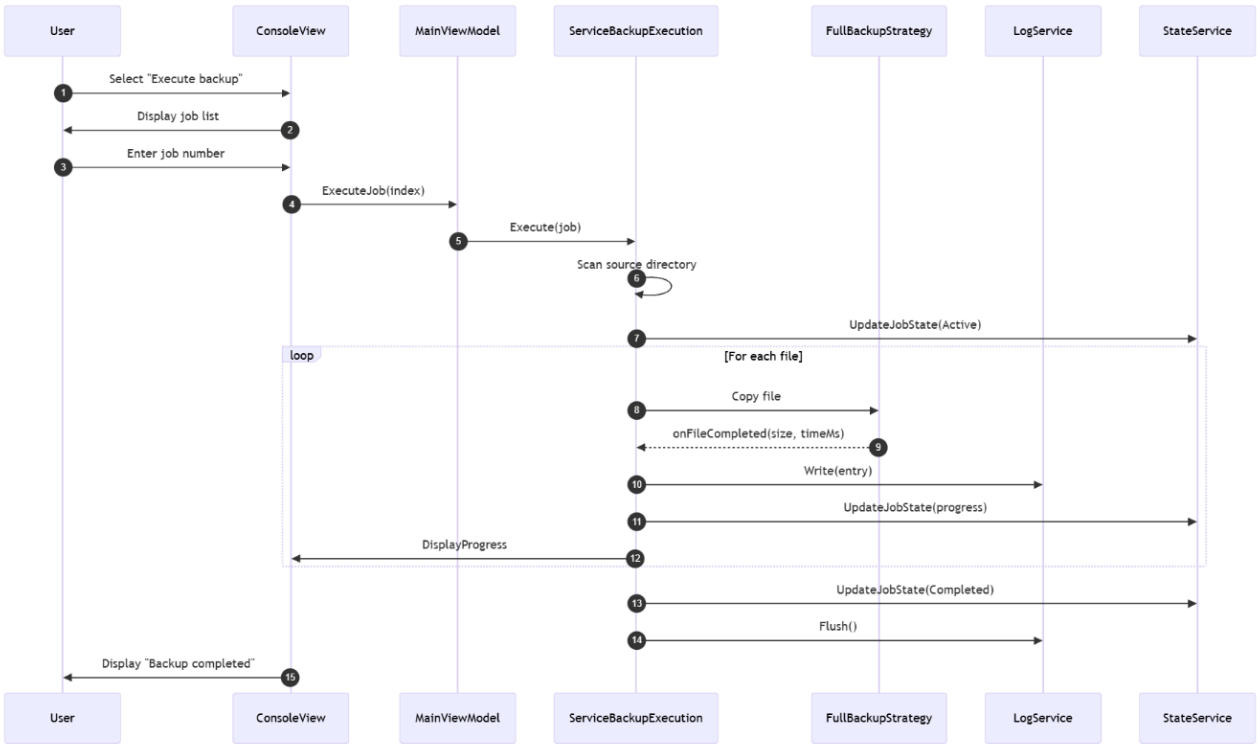
Le diagramme illustre trois chemins d'erreur possibles (blocs alt) :

- Limite atteinte : Retourne immédiatement une erreur si 5 travaux existent
- Nom dupliqué : Rejette la création si le nom est déjà utilisé
- Chemin invalide : Refuse la création si le répertoire source n'existe pas

Persistance

En cas de succès, le ViewModel crée l'objet BackupJob, l'ajoute à sa liste interne, puis déclenche SaveJobs() pour persister la configuration dans config.json. La ConsoleView affiche ensuite le message de confirmation à l'utilisateur.

12.4 Diagramme de séquence - Exécuter une sauvegarde complète



Participants

Participant	Rôle
User	Utilisateur déclenchant l'exécution
ConsoleView	Interface affichant la progression
MainViewModel	Point d'entrée de l'exécution
ServiceBackupExecution	Moteur d'exécution des sauvegardes
FullBackupStrategy	Algorithme de copie complète
LogService	Service de journalisation (EasyLog.dll)
StateService	Service de suivi d'état temps réel

Phase d'initialisation

L'utilisateur sélectionne un travail à exécuter. Le MainViewModel récupère le job correspondant et instancie le ServiceBackupExecution avec la stratégie appropriée (FullBackupStrategy pour une sauvegarde complète).

Le moteur d'exécution commence par scanner le répertoire source pour recenser tous les fichiers et calculer la taille totale. Ces informations initialisent l'état du travail, qui passe en statut **Active** via le StateService.

Boucle de copie (bloc loop)

Pour chaque fichier du répertoire source, le diagramme montre le cycle suivant :

1. **Copie** : La FullBackupStrategy copie le fichier vers la cible
2. **Callback** : La stratégie notifie la fin du transfert via onFileCompleted(size, timeMs)
3. **Journalisation** : Le LogService écrit l'entrée dans le log journalier
4. **Mise à jour état** : Le StateService actualise la progression dans state.json
5. **Affichage** : La ConsoleView rafraîchit la barre de progression

Ce cycle garantit une visibilité temps réel de l'avancement, conformément aux exigences du cahier des charges.

Phase de finalisation

Une fois tous les fichiers traités, le ServiceBackupExecution :

- Met à jour l'état final à **Completed**
- Appelle Flush() sur le LogService pour garantir la persistance des dernières écritures
- Notifie la ConsoleView qui affiche le message de succès

Observabilité

À tout moment pendant l'exécution, un outil externe peut consulter :

- **state.json** : Pour connaître la progression en cours (fichiers restants, pourcentage)
- **Logs/YYYY-MM-DD.json** : Pour voir l'historique des fichiers déjà copiés

12.5 Diagramme d'activité

Lecture du diagramme d'activité EasySave / EasyLog (cf 14. Annexe)

Vue d'ensemble

EasySave est une application console qui exécute des travaux de sauvegarde (jusqu'à 5 dans la version 1.0). Un travail correspond à un objet BackupJob : un nom, un répertoire source, un répertoire cible et un type de sauvegarde (Full ou Differential). EasySave s'appuie sur deux briques transverses pendant l'exécution : la bibliothèque EasyLog pour écrire le log journalier, et un service d'état pour écrire un fichier d'état temps réel unique.

La logique métier est organisée pour rester proche de l'exécution réelle : le MainViewModel gère la liste des travaux, leur persistance (fichier de configuration) et le déclenchement d'une exécution. La console (ConsoleView) est principalement une interface utilisateur qui appelle le ViewModel. Le cœur "runtime" d'une sauvegarde est porté par ServiceBackupExecution, qui orchestre la stratégie de copie choisie (IBackupStrategy) et synchronise tout ce qui doit être visible "en temps réel" : progression, état, logs.

En pratique, les fichiers se retrouvent par défaut dans le profil utilisateur (AppData), ce qui évite les emplacements non adaptés en environnement client/serveur : le fichier de configuration des jobs (config.json), le fichier d'état (state.json) et les logs journaliers (un fichier par jour dans un dossier Logs).

1. Exécution d'un travail de sauvegarde (du lancement jusqu'à la copie)

Au démarrage, EasySave choisit son mode de fonctionnement selon la présence d'arguments. Sans argument, l'utilisateur passe par le menu console (création, liste, exécution, suppression, langue). Avec un argument, l'application interprète une sélection de jobs à exécuter (par exemple une plage ou une liste), puis déclenche l'exécution de chaque job demandé.

Quand un job est exécuté, la première décision importante est le choix du comportement de sauvegarde. EasySave sélectionne une stratégie conforme au type déclaré dans le job : FullBackupStrategy pour une copie complète, ou DifferentialBackupStrategy pour une copie différentielle. Cette stratégie est injectée dans ServiceBackupExecution, qui devient ensuite le point central du diagramme d'activité.

Avant de copier quoi que ce soit, ServiceBackupExecution prépare une "photo" de départ de l'exécution : il crée un BackupJobState associé au nom du job, recense les fichiers présents dans le répertoire source (récursivement) et en déduit une taille totale. Ces informations servent à initialiser l'état en mode Active avec un nombre de fichiers et une taille à transférer, puis à publier immédiatement un premier état (événement + écriture dans state.json). À ce moment, un observateur externe peut déjà voir qu'une sauvegarde a démarré, combien de fichiers sont attendus et quelle quantité de données est prévue.

Ensuite, ServiceBackupExecution délègue la boucle de copie à la stratégie choisie. Dans les deux cas (Full ou Differential), la stratégie parcourt les fichiers du répertoire source de manière

récursive, reconstruit le chemin cible en conservant la structure relative, puis copie le fichier. Le temps de transfert est mesuré au moment de la copie afin d'alimenter le log et l'état.

La différence entre Full et Differential se situe juste avant la copie : en Full, chaque fichier est copié (écrasement autorisé). En Differential, un fichier peut être ignoré si sa version cible existe déjà et est au moins aussi récente que la source ; dans ce cas, la stratégie passe au fichier suivant sans déclencher de mise à jour "fichier terminé".

2. *État et log "temps réel" (ce qui est produit à chaque fichier*)

Le diagramme d'activité est plus lisible si l'on garde en tête une règle simple : la stratégie s'occupe de "faire la copie", tandis que ServiceBackupExecution s'occupe de "rendre la copie observable".

Concrètement, après chaque tentative de copie, la stratégie appelle un callback fourni par ServiceBackupExecution (souvent nommé "fin de fichier"). Ce callback reçoit le chemin source, le chemin destination, la taille du fichier et le temps de transfert en millisecondes (valeur négative en cas d'erreur). À partir de ces informations, ServiceBackupExecution met à jour l'état :

- si le temps est négatif, l'état passe en Error ;
- sinon, l'état mémorise brièvement le fichier courant, décrémente les compteurs "restants" et recalcul la progression.

Dans les deux cas (succès ou erreur), un enregistrement est écrit immédiatement dans le log journalier via EasyLog (ILogService / LogService). Le log contient l'horodatage, le nom du job, les chemins source et cible, la taille et le temps de transfert ; il est écrit au fil de l'eau, ce qui permet d'examiner une exécution pendant qu'elle est en cours. Après l'écriture du log, ServiceBackupExecution publie à nouveau l'état (événement) et réécrit state.json, ce qui matérialise la notion "temps réel" demandée au livrable 1.

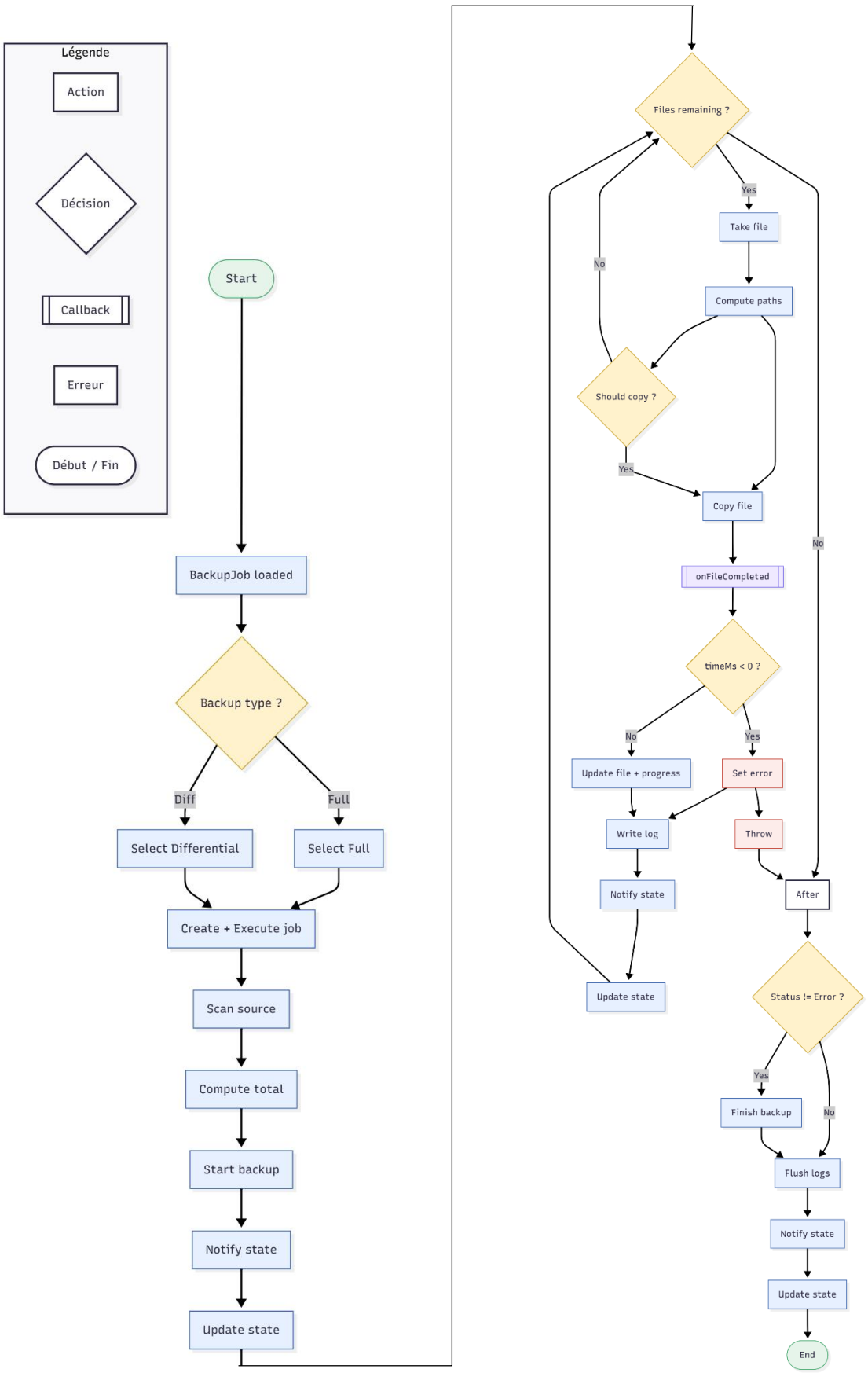
Le chemin d'erreur est volontairement "court" au runtime : si une exception se produit pendant la copie, la stratégie notifie l'échec via le callback (temps négatif), puis relance l'exception. Cela a deux effets visibles dans le diagramme : l'état passe en erreur, un log d'échec est écrit, puis la sauvegarde s'arrête immédiatement (les fichiers suivants ne sont pas traités). Ce comportement facilite le diagnostic et évite de produire un résultat partiel silencieux.

Quand la stratégie se termine normalement (tous les fichiers traités), ServiceBackupExecution finalise l'exécution : si aucune erreur n'a été enregistrée, l'état bascule en Completed. Le service de log est ensuite "flush" pour s'assurer que les dernières écritures sont bien persistées, puis l'état final est publié et sauvegardé une dernière fois. À ce stade, un utilisateur (ou un outil externe) peut s'appuyer sur state.json pour connaître l'issue finale du job, et sur le log du jour pour reconstruire exactement ce qui a été fait, fichier par fichier.

13. Glossaire

Terme	Définition
Travail de sauvegarde	Une tâche de sauvegarde configurée avec source, cible et type
Sauvegarde complète	Copie tous les fichiers indépendamment de leur état de modification
Sauvegarde différentielle	Copie uniquement les fichiers nouveaux ou modifiés
Chemin UNC	Convention de nommage universel (\serveur\partage\chemin)
AppData	Dossier de données d'application utilisateur Windows
CLI	Interface en ligne de commande
MVVM	Patron de conception Model-View-ViewModel
DLL	Bibliothèque de liens dynamiques
Fichier d'état	Fichier JSON suivant la progression de sauvegarde en temps réel
Fichier de log	Fichier JSON enregistrant toutes les actions de sauvegarde

14. Annexe


Légende
Action
Décision
Callback
Erreur
Début / Fin
Start
BackupJob loaded
Backup type ?
Diff
Full
Select Differential
Select Full
Create + Execute job
Scan source
Compute total
Start backup
Notify state
Update state
Files remaining ?
Yes
Take file
Compute paths
No
Should copy ?
Yes
Copy file
onFileCompleted
timeMs < 0 ?
No
Yes
Update file + progress
Set error
Write log
Throw
Notify state
After
Update state
Status != Error ?
Yes
Finish backup
No
Flush logs
Notify state
Update state
End