EasySave

Technical documentation (V2.0)

Table of Contents

Introduction

Project Overview

EasySave is a backup management software developed in C# .NET 10.0. Its primary goal is to allow users to secure and efficiently back up sensitive data. Version 2.0 introduces a graphical user interface built with Avalonia 11.3, replacing the previous command-line interface. The internal architecture follows the MVVM pattern and supports both the legacy CLI and the new GUI.

Key Features

- **Backup job management**: Create, delete, and manage jobs (Name, Source, Target, Type).
- **Two backup modes**:
    - Full back up: Copies all files from the source to the target.
    - Differential backup: Copies only files modified since the last backup.
- **Flexible execution**: Support for a single job execution, sequential range of execution, or executing all jobs.
- **Real-time monitoring**: Updates a state.json file instantly to track progress (activity status, percentage).
- **History & Logging**: Generates daily logs in JSON and XML format using a custom DLL (EasyLog).
- **Location**: Built-in support for English and French.
- **File encryption**: Integration with CryptoSoft, an external XOR encryption tool. Files matching configured extensions are automatically encrypted after copy.
- **Business software detection**: Backup execution is blocked if a configured business software process is running. Detection occurs before launch and between each file copy.

Software Architecture

Architectural pattern: MVVM (Model-View-ViewModel)

To ensure maintainability and scalability, the project follows the **MVVM pattern**. This strictly separates the user interface from the business logic.

- **View** :
    - MainWindow.axaml (Avalonia GUI): The graphical interface with a sidebar menu, dynamic content panels, and header controls for language and log format selection.
    - ConsoleView (Legacy CLI): The console-based presentation layer, still available via EasySave.exe.
- **Model** (BackupJob, StateEntry, Strategies...): Represents the data and the core business logic (file copying algorithms).
- **ViewModel** :
    - MainWindowViewModel: Handles GUI commands, navigation between pages, localization, and data validation. Uses CommunityToolkit.Mvvm (ObservableProperty, RelayCommand).
    - BackupManager: Acts as a bridge between the View and the Model. Holds the list of BackupJobs, handles execution commands, and coordinates with BusinessSoftwareService and StateTracker.

UML Diagrams

To find the UML diagrams, refer to the UML folder.

Technical Design & Patterns

Strategy pattern

The program needs to support multiple backup algorithms (full & differential) that share the same context but perform the work differently. That is why we decided to implement the Strategy pattern:

- IBackupStrategy (Interface): Defines the contract that all strategies must follow.
- FullBackupStrategy: Implements a recursive algorithm to copy the entire directory structure.
- DifferentialBackupStrategy: Implements a logic check to only copy modified files.

This approach respects the Open/Closed Principle (SOLID). We can add a new backup type in the future without modifying the existing code in BackupManager.

CryptoSoft :

CryptoSoft is a standalone console application that performs XOR encryption on files.

Flow: After each file is copied to the target directory, the strategy checks if the file extension is in the configured EncryptedExtensions list. If so, CryptoSoft.exe is launched as an external process with the target file path and the encryption key as arguments.

Encryption time: CryptoSoft returns the encryption duration (in milliseconds) as its process exit code. This value is recorded in the log as EncryptionTimeMs.

Configuration: CryptoSoftPath, EncryptionKey, and EncryptedExtensions are defined in settings.json.

Business Software Detection :

The BusinessSoftwareService prevents backups from running when a configured business software is active.

Detection method: Uses System.Diagnostics.Process.GetProcessesByName() to check if the configured process is running.

Two-level check:

- Before backup launch: BackupManager.ExecuteJob() checks IsRunning() before starting the job. If blocked, a log entry is created with TransferTimeMs = -1 and the job is not executed.

- During backup: Both FullBackupStrategy and DifferentialBackupStrategy check IsRunning() between each file copy. If the business software starts mid-backup, execution is interrupted immediately.

Configuration:

The process name is defined in BusinessSoftwareName in settings.json. The .exe extension is automatically stripped.

Data Persistence

Daily Logging System

Logging is handled by an external library (EasyLog.dll). Every file transfer is recorded in a JSON file or, an XML one, specific to the current date.

- **Location**: Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData)\EasySave\DailyLog
    - Windows: \AppData\Roaming\EasySave\DailyLog
    - Linux: $HOME/.config/EasySave\DailyLog
    - Unix: $HOME/.config/EasySave\DailyLog

- **Data Logged** (Contained in LogEntry objects):
    - Timestamp
    - Job name
    - Source path (UNC format)
    - Target path (UNC format)
    - File size
    - Transfer time (ms).

- **File Name**: Log file per day with the current format --> Year-Month-Day.json or .xml

- **Usage**: The DLL has a static Logger Class. So, there is no need to create Objects for every Logger call. (e.g.s. Logger.Log(entry) with entry an LogEntry object)

- **Pagination**:
    - JSON:
        - JsonSerializerOptions to indent the json content using the WriteIndented = true option
    - XML:
        - For the xml indentation: using Formatting.Indented.

    - For better displayed daily logs, using Environment.NewLine to insert a backrow between two records (\n for Unix platforms and \r\n for non-Unix platforms)

- Errors: Thread Safe logic using lock (_lock) [a read-only object] to save logs one by one. If the Transfer Time is negative --> an error occurred during the back up

Real Time State Logging System

Logging into a static path Json the executions state in real time. It makes following the jobs execution easier, returning important data. This allows external applications to monitor the active backup process (e.g. a Dashboard).

Location: Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData)\EasySave

  - Windows: \AppData\Roaming\EasySave\
  - Linux: $HOME/.config/EasySave\
  - Unix: $HOME/.config/EasySave\

- **Data Logged**:
  - Job Name
  - Time Stamp
  - State
  - Total Files Number
  - Total Size Number
  - Files Remaining
  - Size Remaining
  - Current Source File
  - Current Target File.

- **File Name**: state.json

- **Usage**: The DLL has a static Logger Class. So, there is no need to create Objects for every Logger call. (e.g.s. Logger.Log(entry) with entry an LogEntry object)

- **Pagination**:
  - JsonSerializerOptions to indent the json content using the WriteIndented = true option

Json Job Saving System

To save the jobs, and indeed don’t delete them when closing the application, and enable the CLI version, the jobs are Created, Read and Deleted in a Json file.

- **Location**:
  - Windows: \AppData\Roaming\EasySave
  - Linux: $HOME/.config/EasySave
  - Unix: $HOME/.config/EasySave

- **File Name**: jobs.json

- **Data Logged**:
  - Job Name
  - Source Directory
  - Target Directory
  - Back Up Type