

EasySave

Technical documentation (V3.0)

Table of Contents

<i>Introduction</i>	2
Project Overview	2
Key Features	2
<i>Software Architecture</i>	4
Architectural pattern: MVVM (Model-View-ViewModel)	4
UML Diagrams.....	4
<i>Technical Design & Patterns</i>	5
Strategy pattern	5
<i>Data Persistence</i>	8
Daily Logging System.....	8
Real Time State Logging System	9
Json Job Saving System.....	10

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 3.0 builds upon the Avalonia 11.3 graphical interface introduced in V2.0, adding parallel backup execution, real-time job controls (Play, Pause, Stop), priority file management, large file transfer restrictions, a CryptoSoft mono-instance enforcement, and a centralized Docker-based log server. The internal architecture continues to follow the MVVM pattern and supports both the legacy CLI and the GUI.

Key Features

- **Backup job management:** Create, delete, and manage jobs (Name, Source, Target, Type).
- **Two backup modes:**
 - o Full back up: Copies all files from the source to the target.
 - o Differential backup: Copies only files modified since the last backup.
- **Flexible execution:** Support for a single job execution or parallel execution of all jobs simultaneously.
- **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 copying.
- **Business software detection:** All running backup jobs are automatically paused when a configured business software process is detected. Jobs resume automatically once the software is no longer running.
- **Parallel execution:** All backup jobs run concurrently using async tasks and CancellationTokens, replacing the previous sequential model.
- **Job controls:** Each job (or all jobs at once) can be individually paused, resumed, or stopped in real time via the GUI. Progress is tracked per job with a live percentage indicator.

- **Priority file management:** Files matching user-defined priority extensions are transferred first. Non-priority file transfers are held until all priority files across all active jobs have been processed.
- **Large file transfer restriction:** To prevent bandwidth saturation, only one file exceeding the configured size threshold (n KB) may be transferred at a time across all jobs. Smaller files are not affected.
- **CryptoSoft mono-instance:** CryptoSoft.exe is enforced as a single instance using a named Mutex. Concurrent encryption requests are serialized to avoid conflicts during parallel backup execution.
- **Centralized log server:** An optional Docker-based HTTP log server (EasySaveLogServer) centralizes daily log entries from multiple machines into a single file. Three modes are available: local only, Docker only, or both simultaneously.

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:**
 - o MainWindow.axaml (Avalonia GUI): The graphical interface with a sidebar menu, dynamic content panels, and header controls for language and log format selection.
 - o 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:**
 - o MainWindowViewModel: Handles GUI commands, navigation between pages, localization, and data validation. Uses CommunityToolkit.Mvvm (ObservableProperty, RelayCommand).
 - o BackupManager: Acts as a bridge between the View and the Model. Holds the list of BackupJobs, handles parallel execution commands, and coordinates with BusinessSoftwareService, StateTracker, JobController, PriorityFileManager, and LargeFileTransferManager.

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 pauses all active backup jobs when a configured business software process is detected. Jobs automatically resume once the process stops.

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 the business software is active, the job is queued and waits until the software is closed before starting.

- During backup: Both FullBackupStrategy and DifferentialBackupStrategy check IsRunning() between each file copy. If the business software starts mid-backup, all jobs are paused after the current file completes and automatically resume when the software exits.

Configuration:

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

Parallel Execution :

Starting from V3.0, all backup jobs execute concurrently using Task.Run() with individual CancellationTokenSource instances. The BackupManager launches each job on its own thread and monitors their states through the StateTracker and the JobController.

Job Controls (Play / Pause / Stop) :

The JobController class manages the lifecycle state of each backup job. It exposes Play, Pause, and Stop commands bound to the GUI via RelayCommand. Pausing is cooperative: the running strategy checks a shared SemaphoreSlim at each file iteration and blocks until the job is resumed. Stopping triggers the CancellationToken, which causes the strategy to exit cleanly after the current file transfer completes.

Priority File Management :

The PriorityFileManager maintains a shared counter of pending priority-extension files across all jobs. Before transferring a non-priority file, each strategy checks whether the global priority counter is greater than zero. If so, the transfer is deferred using a wait loop until all priority files have been processed. Priority extensions are defined by the user in settings.json.

Large File Transfer Restriction :

The LargeFileTransferManager uses a SemaphoreSlim(1) to ensure that only one file exceeding the configured size threshold (LargeFileSizeKB in settings.json) is transferred at any given time across all parallel jobs. Files below the threshold are not affected and can transfer freely in parallel. This prevents bandwidth saturation on the client's network.

CryptoSoft Mono-Instance :

CryptoSoft.exe is restricted to a single running instance at a time using a named system Mutex ("CryptoSoftMutex"). When a backup strategy needs to encrypt a file, it acquires the mutex before launching the CryptoSoft process and releases it upon completion. If the

mutex is already held (by another job running concurrently), the requesting strategy waits. This guarantees correct serialization of all encryption operations across parallel jobs.

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:**

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

- **Data Logged** (Contained in LogEntry objects):

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

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

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

- **Data Logged:**

- o Job Name
- o Time Stamp
- o State
- o Total Files Number
- o Total Size Number
- o Files Remaining
- o Size Remaining
- o Current Source File
- o 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:**

- o `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:**
 - o Windows: \AppData\Roaming\EasySave
 - o Linux: \$HOME/.config/EasySave
 - o Unix: \$HOME/.config/EasySave
- **File Name:** jobs.json
- **Data Logged:**
 - o Job Name
 - o Source Directory
 - o Target Directory
 - o Back Up Type

Docker Log Server

EasySaveLogServer is an optional ASP.NET HTTP server deployed via Docker that centralizes daily log entries from multiple EasySave instances. When enabled, each client sends log entries as JSON payloads over HTTP to the server, which merges them into a single shared daily log file, identified by machine name and user.

- **Three log modes:** configurable in settings.json via the LogMode property.
 - o Local: logs are written only to the local machine (default behavior, backward compatible).
 - o Docker: logs are sent exclusively to the centralized server; nothing is written locally.
 - o Both: log entries are written locally and simultaneously forwarded to the Docker server.
- **Server endpoint:** configured via DockerServerUrl in settings.json (e.g. <http://localhost:5000/log>).
- **Centralized file format:** The server maintains a single daily log file (JSON or XML, matching the client's configured format) regardless of the number of connected instances. Each entry includes a MachineName field to differentiate the source.

- **Thread safety:** The server uses a file-level lock to serialize concurrent write requests from multiple clients, preventing log corruption.