User Manual and Use Cases for ntJobApp

1. Introduction and Purpose

An ntJobApp is an application designed to function in **batch mode**[1]. The architecture uses the ncJobApp class and the jData instance to orchestrate these micro-batch applications[2].
The goal of an ntJobApp is threefold:

1. Read an .ini file passed as a parameter[3333].
2. Execute one or more commands requested by the caller[4].
3. Return a result .ini file of the processing[555]. The output .ini file will be renamed with the .end extension[6].

2. Execution Logic for the Operator

The execution of an ntJobApp follows a standard sequential logic[7]:

1. **Instance Creation:** The ncJobApp instance is created in the global variable jData[8].
2. **Startup (Start()):** self.Start() is executed, which reads the .ini file passed as a parameter (or creates it) and saves it in self.dictJobs[9].
    - If sResult (the operation result) is different from "", the application exits with errorlevel **1**[101010].
3. **Execution (Run()):** self.Run() is executed, which sequentially runs every single job defined in self.dictJobs[11].
4. **Finalization (End()):** self.End(sResult) is executed[12].

3. Execution Instructions

To run an ntJobApp, you must launch the application by passing the full path of the configuration .ini file as the **first parameter**[13].

3.1. The Input File (.ini)

The .ini file must contain at least the [CONFIG] section and one or more *job* sections[141414].

Example of ntjobsapp.ini file read at startup:

```
Ini, TOML
[CONFIG]
TYPE=NTJOBS.2.0
EXIT=TRUE       ; If "TRUE", the application exits upon a job error. [cite: 48, 97]
LOG="FILE.LOG"  ; Optional Log file name. [cite: 49, 95]
NAME=ID_APPLICAZIONE [cite: 50, 96]
[JOB1_ID]
ACTION=NOME_AZIONE
FILE.ID1=PathFile1
PARAM.ID1=Valore
```

Crucial Validations in Start():

- **CONFIG Section:** It must be present in the dictionary read into self.dictJobs[15], otherwise the application exits with the error "Sezione CONFIG non trovata" (Section CONFIG not found).
- **COMMAND Key:** Every job section (except CONFIG) must contain the COMMAND key[16], otherwise the error is appended to sResult.
- **File Check:** If there are keys starting with FILE. (e.g., FILE.ID1), the application verifies that the file exists[17]. If a required file is missing, the error is appended to sResult, but the execution **is not** immediately halted[18].

3.2. The Output File (.end)

Upon completion, the End(sResult) method saves the content of self.dictJobs as an .ini file at the self.sJobEnd path[19]. self.sJobEnd is the name of self.sJobIni with the extension changed to .end[20].
The .end file will contain the results of each job, including:

- RETURN.TYPE (e.g., E=Error, ""=OK)[21212121].
- RETURN.VALUE (Optional return message)[22222222].

- TS.START and TS.END (Start and end timestamps)[23].
- Keys with prefix FILE.RETURN. (references to files returned by the job)[24242424].

3.3. Result Codes (Errorlevel)

Upon completion, the ntJobApp returns a result code (errorlevel) indicating the final status[25]:

Code	Description	Meaning
0	All clear	The application terminated correctly. [26]
1	Failed to reach job execution	Critical error during startup (Start()), and the finished .ini file is not generated. [27]
2	One or more jobs terminated with an error	One or more jobs within the .ini file ended with an error. [28]

4. Use Cases and Advantages vs. Web Automation (Clicks and Waits)

The ntJobApp architecture is specifically designed for **batch automation** processes [29] and offers significant advantages over solutions that require graphical interaction (like clicking and waiting on a website).

Typical ntJobApp Use Cases	Key Advantages Over Web Automation (GUI)
Asynchronous Processes: Complex or massive processing, such as overnight report generation, that should not block the user.	**Efficiency and Speed:** As a *backend* process, it has no graphical overhead, requires no waiting for page rendering or human interaction, drastically reducing idle time.
Data Integration: Mass movement, conversion, or processing of files (e.g., from FTP to a database), processes working "under the hood."	**Robustness and Reliability:** It is not subject to errors due to changes in the user interface (like a button changing its ID) or network timeout issues caused by page loading.
Structured Orchestration: Calling different applications sequentially (jobs), where the results and return files of one action are managed as input for the next[30].	**Standardized Traceability:** The .ini format for input and the .end output ensure a uniform and readable status format for the caller (code 0, 1, or 2)[31].
Scheduling: Execution of automatic tasks at predefined times (like Cron or Task Scheduler).	**Ease of Automation:** Command-line execution with a parameter file makes it ideal for any *job scheduler* system, requiring only the path to the .ini file.