App Gestione File

Indice

Obiettivi:

- Il programma deve sincronizzare due cartelle, una di origine e una di destinazione
- Il processo di aggiornamento della cartella di destinazione deve avvenire ciclicamente dopo un delay impostato dall'utente
- Vengono controllati data di modifica e data di creazione del file, se rientrano nel range di tempo che va dall'ora attuale a un delay indietro allora la cartella di destinazione viene aggiornata

Funzioni:

- L'utente può scegliere se il programma deve effettivamente aggiornare la cartella di destinazione oppure se deve solo visualizzare le modifiche in una lista
- I pulsanti "Avvia" e "Stop" gestiscono lo stato di funzionamento
- Il percorso di origine e di destinazione viene scelto dall'utente da una lista salvata su txt che può essere aggiornata dall'interfaccia del programma
- L'attesa non blocca l'interfaccia, ma viene divisa in tante attese da 5 secondi in modo che l'utente possa fermare il ciclo senza dover attendere la fine dell'attesa totale, ma invece al massimo 5 secondi.
- Il programma capisce se un file è stato modificato oppure è stato creato nel range di tempo
- Ogni operazione viene registrata in un file di log .csv, visualizzabile con Excel
- Ogni volta che il programma esegue il controllo per la sincronizzazione delle due cartelle, se il file log.csv non esiste, viene creato e nella prima riga vengono impostati i titoli delle colonne
- Il programma tiene sincronizzati i file nella cartella di origine con quelli nella cartella di destinazione, anche se viene inserito un file con data di creazione precedente al filtro, in quel caso viene comunque copiato nella cartella di destinazione
- La gestione dei file rinominati, visto che rinominare un file non modifica né la data di creazione né quella di ultima modifica, funziona grazie al confronto dei nomi dei file nella cartella di destinazione con quelli nella cartella di origine. Quando nella cartella di destinazione viene trovato un file che non è presente nella cartella di destinazione, questo viene eliminato. Poi si esegue l'operazione inversa tra la cartella di origine e quella di destinazione, in cui vengono copiati gli eventuali file mancanti
- Per ottenere la lista dei file nella cartella di destinazione, viene usato un file di testo che funge da database. Ad ogni operazione di aggiornamento della cartella di destinazione, anche quel file va allineato. Questa funzione permette di non sovraccaricare la cartella di destinazione nel caso in cui non fosse locale.

Mancanze e problemi noti:

- ~~Se un file viene rinominato, il programma non rileva nessuno modifica, se l'opzione di sincronizzazione è attiva, nel processo di confronto tra la due cartelle, la lista dei file nella cartella di destinazione contiene un file con nome diverso (il vecchio nome del file) e viene interpretato come un file eliminato perché non è presente nessun file con quel nome nella cartella di origine~~

  Risolto: ora quando un file viene rinominato, viene prima eliminato dalla cartella di destinazione ma poi il programma esegue un confronto con la cartella di origine da cui copia i file mancanti, tra cui anche quello rinominato
- ~~È stata inserita una ProgressBar che aumenta dopo ogni attesa parziale (5 secondi), dovrebbe aggiornarsi alla percentuale di tempo atteso rispetto all'attesa totale. È una funzione asincrona che implementa anche un'animazione ad ogni aggiornamento. Ma non funziona.~~

  Risolto: la ProgressBar ora funziona e aumenta dopo ogni attesa parziale, quando arriva all'ultima attesa rimane per un istante al 100% perché poi inizia la nuova verifica e quindi riparte da 0 anche la ProgressBar.
- ~~Se mentre il programma è in esecuzione si ripreme il tasto "Avvia" il programma ricomincia il ciclo da capo~~

  Risolto: ora la funzione che viene richiamata quando viene premuto il bottone "Avvia", come avviene per il tasto "Stop", si effettua la verifica sulla variabile "run" per rilevare se il ciclo di sincronizzazione è attualmente in esecuzione

Migliorie possibili:

- Aggiungere la possibilità di caricare i file su un servizio cloud (es. OneDrive, Google Drive, Dropbox, …). Utilizzare un'API per autenticarsi al servizio e leggere e scrivere i file sul server.
- La pagina di login al servizio cloud potrebbe essere incorporata in una nuova finestra dell'app, in modo da poter gestire ogni tipo di autenticazione (anche a Due Fattori o con codice App Authenticator)
- Attualmente il confronto tra la cartella di origine e quella di destinazione avviene controllando in entrambi i percorsi tutti i file e le sottocartelle. Nel caso in cui la destinazione dovesse essere un cloud, questo metodo richiederebbe una gran quantità di lavoro al server. Sarebbe meglio implementare un sistema basato su database locale che mantenga una lista dei file attualmente presenti sulla destinazione, in questo modo basterebbe accedere a questo database per ottenere la lista dei file nel percorso di destinazione.
- Impostare una dimensione iniziale della finestra e una dimensione minima per mantenere sempre la leggibilità dell'interfaccia
- Aggiungere una pagina di impostazioni da cui poter inserire i settaggi prima dell'avvio (origine, destinazione, delay, funzione copia, funzione elimina sulla destinazione, aggiorna tutto alla prima sincronizzazione)

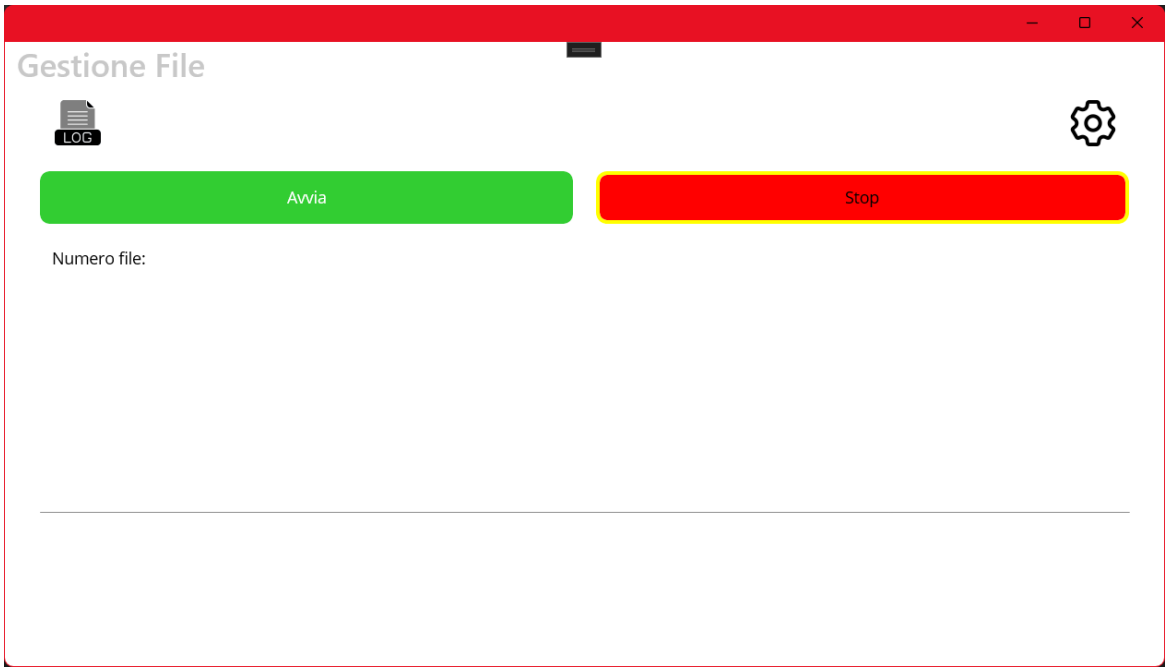
Funzionamento:

Il framework .NET MAUI permette la programmazione multipiattaforma scrivendo il codice C# una sola volta, ma a causa della differenza di dimensioni dello schermo tra i vari sistemi operativi, è spesso necessario adattare l'interfaccia grafica alle diverse esigenze di visualizzazione. Per questo programma sono stati creati quindi più file XAML con i relativi file contenenti il codice che ovviamente invece è identico per tutte le piattaforme.

Schermata principale:

Nell'interfaccia iniziale dell'applicazione è possibile gestire lo stato di funzionamento del sistema di sincronizzazione trami i bottoni appositi e accedere alla schermata delle impostazioni cliccando sull'icone dell'ingranaggio.

Windows:



Android:

Schermata impostazioni:

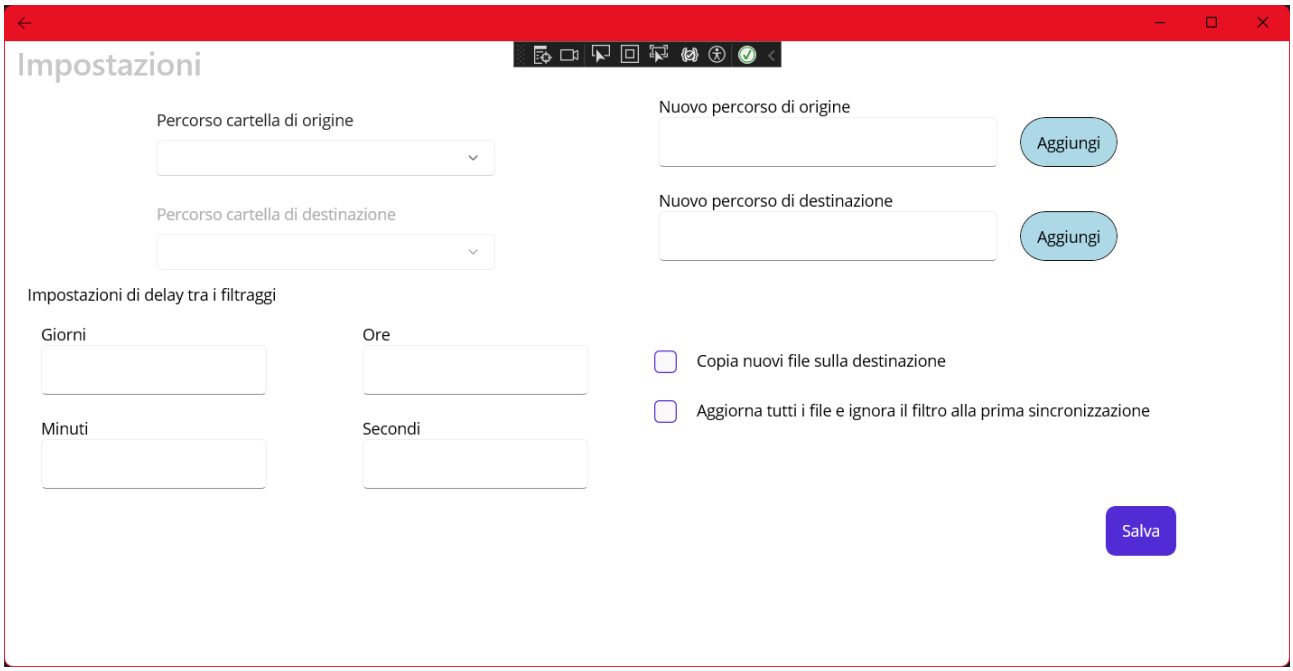
Impostare un percorso di origine selezionandolo dal menù a cascata contenente i percorsi precedentemente impostati

- Attivare o disattivare la funzione di copia nella cartella di destinazione
- Se si attiva la copia nella cartella di destinazione, impostare un percorso di destinazione selezionandolo dal menù a cascata contenente i percorsi precedentemente impostati
- Abilitare o disabilitare la funzione di copia di tutti i file alla prima esecuzione della sincronizzazione indipendentemente dal filtro
- Impostare il delay tramite le caselle di testo (ogni quanto tempo deve avvenire la sincronizzazione)

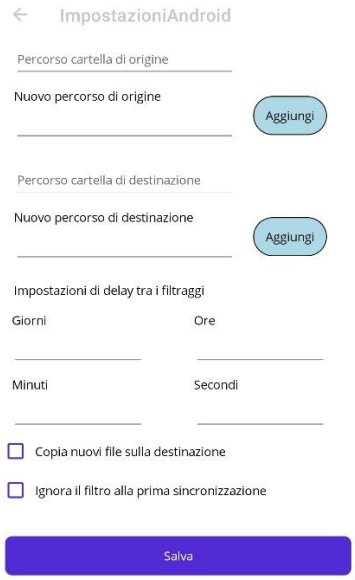
Nel caso in cui si volesse aggiungere un nuovo percorso di origine o destinazione, lo si può fare tramite le apposite caselle di testo e i pulsanti.

Una volta terminati tutti i settaggi, salvare le impostazioni con il tasto “Salva” e tornare alla Schermata Principale.

Windows:



Android:



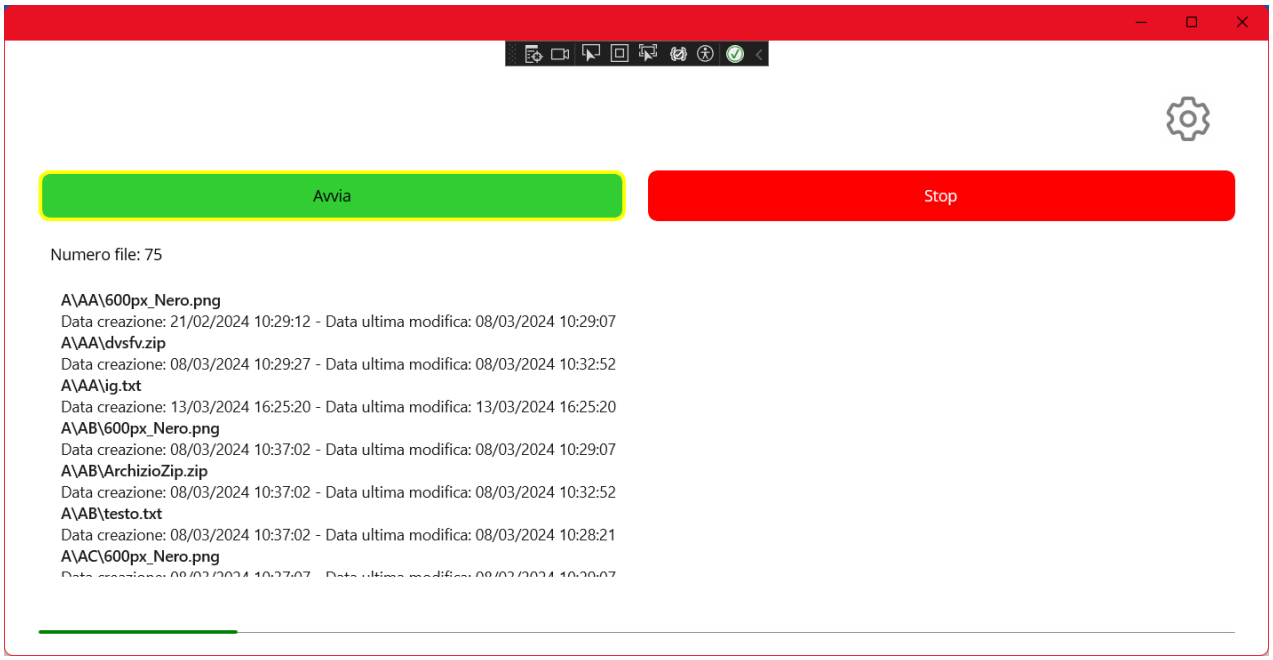
Schermata principale in esecuzione:

Non appena si clicca il tasto “Avvia”, il programma carica le impostazioni e inizia la sincronizzazione. Alla prima esecuzione del ciclo, se l’impostazione apposita è stata abilitata, il filtro viene impostato al massimo possibile, senza tenere conto del delay impostato, in modo tale da copiare tutti i file presenti nella cartella di origine. Dalla seconda sincronizzazione in poi, viene impostato il filtro come la sottrazione del delay dalla data e ora attuale, in modo da non caricare ogni volta tutti i file.

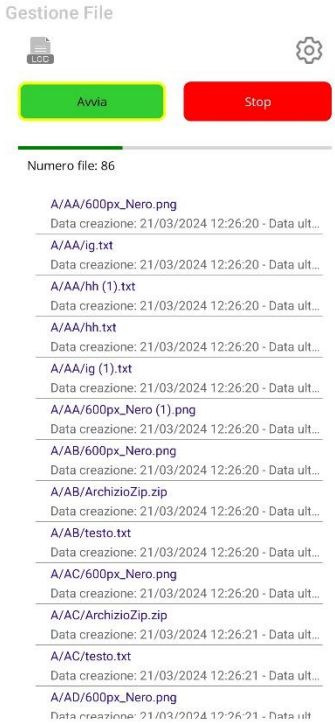
I nuovi file trovati nella cartella di origine, se l’impostazione è stata abilitata, vengono copiati nella cartella di destinazione e aggiunti alla lista, che viene aggiornata assieme alla Label con il numero di nuovi file trovati.

Lasciando il programma avviato, questo rimarrebbe in esecuzione all’infinito. Per fermarne il funzionamento basta cliccare il tasto “Stop”, che tramite un messaggio a schermo comunica l’effettivo cambiamento di stato del programma. A causa dell’attesa tra una sincronizzazione e l’altra, lo stop del programma non è istantaneo ma avviene dopo al massimo 5 secondi.

Windows:



Android:



XAML:

Struttura grafica dell'interfaccia di MainPage:

Windows:

La finestra principale dell'applicazione è formata da un bottone "Impostazioni" (BUT_Settings) e uno "Log" (BUT_LogFile) e una griglia con 2 colonne e 4 righe, in modo da mantenere un'organizzazione degli elementi ordinata.

Riga 1)
- Colonna 1) Bottone "Avvia" (BUT_Avvia)
- Colonna 2) Bottone "Stop" (BUT_Stop)

Riga 2) Lista file (LST_Output)
Riga 3) Progress Bar (PGB_Timer)

Android:

La finestra principale è formata da una griglia formata da 2 colonne e 4 righe, per una disposizione ordinata degli elementi.

Struttura grafica dell'interfaccia di Impostazioni:

Windows:

La finestra delle Impostazioni è formata da una griglia con 4 colonne e 4 righe, in modo da mantenere un'organizzazione degli elementi ordinata.

Riga 1)
- Colonna 1) Picker "Percorso cartella di origine" (PCK_Path)
- Colonna 2) Picker "Percorso cartella di origine" (PCK_Path)
- Colonna 3) Entry "Nuovo percorso di origine" (TXT_NewPath) e (BUT_AddPath)
- Colonna 4) Entry "Nuovo percorso di origine" (TXT_NewPath) e (BUT_AddPath)

Riga 2)
- Colonna 1) Picker "Percorso cartella di destinazione" (PCK_DestPath)
- Colonna 2) Picker "Percorso cartella di destinazione" (PCK_DestPath)
- Colonna 3) Entry "Nuovo percorso di origine" (TXT_NewDestPath) e (BUT_AddDestPath)
- Colonna 4) Entry "Nuovo percorso di origine" (TXT_NewDestPath) e (BUT_AddDestPath)

Riga 3)
- Colonna 1) Impostazioni Delay (Giorni)
- Colonna 2) Impostazioni Delay (Ore)
- Colonna 3) Impostazioni aggiuntive
- Colonna 4) Impostazioni aggiuntive

Riga 4)
- Colonna 1) Impostazioni Delay (Minuti)
- Colonna 2) Impostazioni Delay (Secondi)
- Colonna 3) Impostazioni aggiuntive

Colonna 4) Impostazioni aggiuntive

Android:

La finestra delle Impostazioni è formata da una griglia con 2 colonne e 8 righe che permettono di disporre gli elementi in una maniera ordinata

Struttura grafica dell'interfaccia di Log:

Windows e Android:

La finestra di visualizzazione del log è formata da una Label con il titolo della pagina, una Label con il percorso del file CSV con i dati di log e una CollectionView contenente i dati, ogni riga ha come schema una Grid con 3 colonne

C#:

Classi:

Il programma è formato da otto classi:

- **MainPage**, la classe standard creata all'avvio dell'applicazione che contiene tutti gli elementi grafici
- **Impostazioni**, classe della schermata impostazioni
- **Log**, classe della schermata dei Log
- **VisualizzaLog**, classe della schermata di visualizzazione dettagliata del log selezionato
- **DatiFile**, questa classe contiene i dettagli sui nuovi file trovati nella cartella e che verranno copiati nella cartella di destinazione (nome, data_creazione, data_modifica, data_completo) *Nota: l'attributo data_completo serve esclusivamente alla formattazione delle proprietà dei file per una corretta visualizzazione tramite Binding nella TextCell all'interno della lista (LST_Output)*
- **DatiLog**, questa classe costituisce il template di ogni riga della pagina di log delle operazioni effettuate. Contiene quindi i dati di ogni file su cui è stata eseguita un'operazione, questi servono ad avere un resoconto di quello che ha fatto il programma.
- **Settings**, questa classe contiene le impostazioni del sistema di sincronizzazione (percorso di origine, di destinazione, delay, stato di attivazione della funzione di copia nella cartella di destinazione, stato di attivazione della funzione di ignoro del filtro alla prima sincronizzazione). I dati vengono caricati dopo aver premuto il bottone "Salva" nella finestra delle Impostazioni.
- **Sync**, questa classe contiene tutti gli attributi e i metodi necessari alla sincronizzazione delle cartelle di origine e destinazione. Viene creato un oggetto di questa classe ad ogni avvio del ciclo una volta premuto il bottone "Avvia", al costruttore vengono passati il delay impostato, la cartella di origine e di destinazione, lo stato di attivazione della funzione "Copia nella cartella di destinazione", la variabile "run" che gestisce lo stato di funzionamento del ciclo di

sincronizzazione e tutti i nomi dei file che servono al funzionamento interno dell'applicazione.

MainPage:

Questa è la classe della pagina iniziale dell'applicazione, contiene le funzioni richiamate dall'interazione con l'interfaccia utente. La funzione BUT_Settings_Clicked(), richiamata al click dell'omonimo bottone, apre la finestra delle Impostazioni.

DatiFile:

```
8 riferimenti
public class DatiFile
{
    4 riferimenti
    public string nome { get; set; }
    4 riferimenti
    public string data_creazione { get; set; }
    4 riferimenti
    public string data_modifica { get; set; }
    0 riferimenti
    public string data_completo { get { return $"Data creazione: {data_creazione} - Data ultima modifica: {data_modifica}"; } }
}
```

Questa classe serve solo a contenere i dati in modo da poterci accedere tramite Binding da XAML.

DatiLog

```
14 riferimenti
public class DatiLog
{
    3 riferimenti
    public string file { get; set; }
    3 riferimenti
    public string data { get; set; }
    3 riferimenti
    public string azione { get; set; }
    3 riferimenti
    public string data_creazione { get; set; }
    3 riferimenti
    public string data_modifica { get; set; }
}
```

Questa classe serve solo a contenere i dati in modo da poterci accedere tramite Binding da XAML.

Impostazioni:

Questa è la classe della schermata impostazioni dell'applicazione. La creazione di questa schermata viene chiamata al click del bottone "Impostazioni" sulla schermata iniziale e può essere richiamata solo quando lo stato di esecuzione della sincronizzazione è disattivato. Viene richiamata tramite una delle seguenti istruzioni, in base alla piattaforma su cui il programma è attualmente in esecuzione:

```
await Navigation.PushAsync(new ImpostazioniWinUI(pathsfilename, destpathsfilename, settings));
```

```
await Navigation.PushAsync(new ImpostazioniAndroid(pathsfilename, destpathsfilename, settings));
```

Al costruttore vengono passati i nomi dei file che contengono i percorsi di origine e di destinazione e un'oggetto della classe "Settings", questo oggetto è vuoto se la sincronizzazione non è stata ancora mai avviata, mentre contiene i dati se questi erano già stati impostati nella pagina Impostazioni.

Nel costruttore della classe vengono salvati i parametri passati e vengono poi chiamate le funzioni di caricamento dei Picker del percorso di origine e destinazione: queste funzioni leggono i file di testo contenenti i percorsi aggiunti in precedenza e li aggiungono alle possibili scelte di percorso che si possono fare tramite “PCK_Path” e “PCK_DestPath”.

Poi viene chiamata la funzione di caricamento dei campi nel caso in cui l’oggetto “settings” dovesse contenere già dei dati, infatti questa funzione permette all’utente di modificare queste impostazioni già presenti senza doverle riscrivere tutte da capo.

Infine, viene chiamata la funzione che sincronizza lo stato di attivazione del Picker “PCK_DestPath” con la CheckBox “CHB_Copia”, infatti quando la funzione copia è disattivata è inutile impostare un percorso di destinazione.

Nelle funzioni di aggiunta di nuovi percorsi tramite i pulsanti, viene verificato che il percorso sia valido e che non sia nullo, in modo da evitare opzioni non funzionanti.

Costruttore:

```
public Impostazioni(string pathsfilename, string destpathsfilename, Settings settings)
{
    InitializeComponent();
    this.pathsfilename = pathsfilename;
    this.destpathsfilename = destpathsfilename;
    this.settings = settings;
    UpdatePathsPicker(Path.Combine(FileSystem.AppDataDirectory, this.pathsfilename),
PCK_Paths);
    UpdatePathsPicker(Path.Combine(FileSystem.AppDataDirectory,
this.destpathsfilename), PCK_DestPaths);
    LoadSettings();
    SyncCopia_DestPaths();
}
```

Funzione UpdatePathsPicker():

Passati i parametri del file con la lista dei percorsi aggiunti e il Picker di riferimento, la funzione carica tutti i file in una lista di tipo string tramite un ciclo while che legge tutte le righe del file fino alla fine. Infine viene impostata la sorgente degli elementi della Picker come la lista di percorsi appena creata.

Funzione UpdatePaths():

Passati i parametri percorso da aggiungere, file contenente i percorsi e Picker di riferimento, la funzione verifica che il percorso da aggiungere non sia già nella lista dei percorsi aggiunti.

- Apre un’istanza FileStream e una StreamReader
- Inizializza la flag del percorso trovato a false
- Legge tutto l’elenco di percorsi sul file tramite un ciclo while
- Verifica per ogni riga se è uguale al percorso da aggiungere, nel caso in cui lo fosse imposta la flag a true
- Una volta uscito dal ciclo, se la flag è false allora aggiunge una riga con il nuovo percorso al file

- Poi chiama la funzione UpdatePathsPicker() per aggiornare il Picker del percorso, passandole tutti gli appositi attributi

Funzione LoadSettings():

Questa funzione riempie i campi con le impostazioni attualmente presenti nell'applicazione, ovvero quelle aggiunte in precedenza dall'utente

Controlla prima di tutto se esiste l'impostazione del percorso di origine, che è una di quelle obbligatorie; quindi, controllando quella si può determinare conseguentemente se sono già presenti le impostazioni. Se l'esito del controllo dovesse essere positivo, tutti i campi vengono compilati con le rispettive impostazioni.

Funzione SyncCopia_DestPaths():

Imposta semplicemente lo stato di attivazione del Picker "PCK_DestPaths" uguale allo stato della CheckBox "CHB_Copia".

Log:

Questa classe gestisce la finestra di visualizzazione del Log direttamente all'interno dell'applicazione senza dover utilizzare un software di visualizzazione di file CSV (ad es. Excel).

La schermata può essere aperta soltanto quando lo stato di esecuzione della sincronizzazione è disattivato. Viene richiamata tramite una delle seguenti istruzioni, in base alla piattaforma su cui il programma è attualmente in esecuzione:

```
await Navigation.PushAsync(new LogWinUI(logfilename, settings.path));
```

```
await Navigation.PushAsync(new LogAndroid(logfilename, settings.path));
```

Costruttore:

```
public LogWinUI(string logfilename, string path)
{
    InitializeComponent();
    this.logfile = Path.Combine(FileSystem.AppDataDirectory, logfilename);
    this.path = path;
    LBL_Path.Text = this.path;
    LoadList();
}
```

Funzione LoadList():

La funzione carica i dati dal file CSV di log delle operazioni ad un una CollectionView per la visualizzazione delle informazioni di un file per riga.

- Crea una lista di tipo DatiLog
- Inizializza il flusso per la lettura del file con StreamReader
- Utilizza l'istanza StreamReader e un ciclo per leggere il file riga per riga fino alla fine
- Su ogni riga viene eseguita la funzione di Split per salvare i campi contenuti tra i ";" in ogni oggetto di tipo DatiLog della lista
- Una volta concluso il ciclo viene impostata la posizione del file all'inizio
- La CollectionView viene popolata con la lista che contiene i dati di tutto il Log

Funzione CLT_Log_SelectionChanged():

Questa procedura viene chiamata ogni volta che il programma rileva che l'oggetto attualmente selezionato sulla CollectionView viene cambiato. Controlla che sia diverso da *null* e crea un oggetto di tipo DatiLog uguale a quello selezionato dalla lista, in modo da essere passato come parametro durante la creazione della finestra di visualizzazione Log dettagliata.

VisualizzaLog:

Questa classe gestisce la schermata che permette la visualizzazione del log selezionato nella precedente schermata con l'elenco delle operazioni effettuate. La finestra viene chiamata tramite questa istruzione:

```
await Navigation.PushAsync(new VisualizzaLog(selected, this.path));
```

Costruttore:

```
public VisualizzaLog(DatiLog log, string path)
{
        InitializeComponent();
        this.log = log;
        this.path = path;
        Load();
}
```

Funzione Load():

Questa funzione inserisce i dati memorizzati nell'oggetto di tipo DatiLog nelle Label della pagina per la visualizzazione delle informazioni sul file.

Settings:

```
6 riferimenti
public class Settings
{
    8 riferimenti
    public TimeSpan delay { get; set; }
    12 riferimenti
    public string destpath {  get; set; }
    19 riferimenti
    public string path { get; set; }
    5 riferimenti
    public bool copy { get; set; }
    3 riferimenti
    public bool firstrun { get; set; }

}
```

Questa classe contiene le impostazioni per la sincronizzazione che vengono mantenute tra le attivazioni del ciclo di sincronizzazione.

Sync:

```
Sync sync;
```

Viene creato un oggetto di questa classe al lancio dell'applicazione, ma senza passare parametri.

```
sync = new Sync(run, settings, LBL_NumeroFile, LST_Output, PGB_Timer, dbfilename, logfilename, logtitles);
```

L'oggetto viene creato completamente, passando i parametri al costruttore, all'interno della funzione chiamata dalla pressione del pulsante, ovvero quando l'utente ha già inserito le impostazioni e vuole avviare il ciclo di sincronizzazione.

Costruttore:

```
public Sync(bool run, Settings settings, Label LBL_NumeroFile, ListView LST_Output,
ProgressBar PGB_Timer, string dbfilename, string logfilename, string logtitles)
{
    this.run = run;
    this.settings = settings;
    this.LBL_NumeroFile = LBL_NumeroFile;
    this.LST_Output = LST_Output;
    this.PGB_Timer = PGB_Timer;
    this.dbfilename = dbfilename;
    this.logfilename = logfilename;
    this.logtitles = logtitles;
    this.firstrun = settings.firstrun;
}
```

Funzione Start():

Questa funzione contiene il ciclo while su condizione "run == true", ad ogni esecuzione del ciclo viene eseguita la sincronizzazione delle cartelle:

- Crea la variabile filtro di tipo DateTime come l'ora attuale meno il delay impostato
- Se il file di log non esiste lo crea e scrive i titoli delle colonne nella prima riga
- Scrive la lista di tipo DatiFile "lista_filtrati" tramite la funzione Filter()
- Crea la lista "file_origine" leggendo la cartella di origine con la funzione FileList()
- Crea la lista "file_destinazione" leggendo il database dei file salvato su un file i testo
- Confronta la due liste di file verificando se qualche file è stato eliminato o rinominato tramite le funzioni LogDeleted() e LogRenamed()

- Imposta “lista_filtrati” come sorgente dati per la ListView “LST_Output”
- Scrive il numero di file nella Label corrispondente
- Avvia la fase di attesa fino alla nuova sincronizzazione

```
int n_semidelay = (int)Math.Round(delay.TotalSeconds / 5.0);
int i = 0;
while (i < n_semidelay && run)
{
    PGB_Timer.Progress = i / (double)(n_semidelay);
    await Task.Delay(TimeSpan.FromSeconds(5));
    i++;
}
```

n_semidelay è il numero di attese parziali da 5 secondi

Il ciclo while ripete le istruzioni fino ad aver atteso “n_semidelay” volte o se la variabile “run” cambia stato

Viene aggiornato lo stato della ProgressBar uguale alla percentuale raggiunta da “i” rispetto a “n_semidelay”

Attesa di 5 secondi tramite una funzione asincrona, in modo da non bloccare l’interfaccia grafica

Nota: l’attesa totale viene divisa in tante attese da 5 secondi in modo da dare all’utente la possibilità di fermare il processo di sincronizzazione senza dover attendere tutto il delay impostato. In questo modo il programma impiegherà al massimo 5 secondi a fermarsi dopo il click del tasto “Stop” da parte dell’utente.

Funzione Filter():

Si tratta di una funzione ricorsiva che verifica la presenza di sottocartelle e nel caso in cui ci fossero entra sempre più in profondità filtrando ogni volta per data di ultima modifica e data di creazione i file contenuti in ogni cartella.

- Per ogni file nella cartella selezionata viene controllato se la data di creazione o di modifica rispetta il filtro
- Se il filtro è rispettato, viene creato un oggetto DatiFile con i dati del file
- L’oggetto creato viene aggiunto alla lista “lista_filtrati”
- Se la data di creazione è maggiore del filtro significa che il file è stato creato, altrimenti che è stato modificato
- Se la variabile “copy” è vera allora il file viene copiato nella cartella di destinazione nella stessa sottocartella della directory di origine ma partendo dalla cartella di destinazione e viene aggiunto il file al database dei file nella cartella di destinazione tramite la funzione AddToDB()
- Viene aggiunta al file la riga di log contenente l’azione eseguita e le informazioni sul file
- Verifica se ci sono altre sottocartelle, se ci sono la funzione richiama sé stessa per filtrare ogni cartella trovata

Funzione FileList():

Questa funzione viene chiamata per creare una lista in formato string dei file presenti dentro una cartella e nelle sue sottocartelle. Usa lo stessa logica della funzione Filter() ma ovviamente non filtra

i file ma li aggiunge tutti indistintamente alla lista. Viene utilizzato come nome del file il percorso a partire da una cartella impostata alla chiamata della funzione.

- Scorre tutti i file
- Aggiunge il percorso relativo alla directory iniziale alla lista
- Scorre tutte le cartelle
- Richiama su ognuna la sua stessa funzione per entrare in ogni cartella

Funzione LogDeleted():

Passata come parametro la lista "lista_eliminati", il programma elimina questi file anche dalla cartella di destinazione, aggiunge una riga al file di log ed aggiorna il file di database con la funzione DeleteFromDB()

- Scorre tutti gli elementi stringa della lista "lista_eliminati"
- Elimina il file corrispondente dalla cartella di destinazione
- Aggiorna il database tramite la funzione DeleteFromDB() passandogli il file eliminato
- Aggiunge una riga al file di log con l'azione di eliminazione del file ma senza dati sul file perché non sono disponibili essendo stato eliminato

Funzione LogRenamed():

Passata come parametro la lista "lista_rinominati", il programma copia questi file nella cartella di destinazione come se fossero stati appena creati, visto che quelli con il nome precedente sono già stati eliminati dalla funzione LogDeleted(), aggiunge una riga di log come "file creato" e aggiorna il database con la funzione AddToDB()

- Scorre tutti gli elementi stringa della lista "lista_rinominati"
- Copi il file corrispondente nella cartella di destinazione
- Aggiorna il database tramite la funzione AddFromDB() passandogli il file copiato
  Aggiunge una riga al file di log con l'azione di creazione di un file e i dati sul file copiato

Funzione AddToDB():

Controlla se il file passato come parametro è presente nel database, se non è presente lo aggiunge in fondo al file

- Apre un'istanza FileStream e una StreamReader
- Legge ogni riga del file e se trova una riga uguale al file da aggiungere imposta una flag a "true" ed esce dal ciclo
- Se la flag è "false" allora il file non era presente nel database
- Perciò aggiunge il file come ultima riga del database

Funzione DeleteFromDB():

Controlla se il file è presente nel database, se è presente lo rimuove eliminando e ricreando tutto il database senza la riga del file eliminato

- Apre un'istanza FileStream e una StreamReader

- Legge ogni riga del file e se trova una riga uguale al file da aggiungere imposta una flag a “true” ed esce dal ciclo
- Se la flag è “true” allora il file era presente nel database
- Perciò salva tutte le righe del database in una lista di tipo string
- Elimina il file del database e ne ricrea uno nuovo
- Scrive riga per riga il nuovo file verificando che la riga non sia quella contenente il file eliminato

Funzione NowFormatted():

Questa funzione serve per comodità in modo da avere una stringa con data e ora formattati semplicemente chiamando questa funzione

- Viene restituita la stringa formattata dall’attributo “Now” della classe “DateTime”, con la data secondo il formato della cultura italiana e l’ora fino ai secondi