Background jobs – Tema 2 DATC | Mihaela TUTU | semigrupa 5.2

Definitie:

Un *background job* este un proces neinteractiv care ruleaza in spatele operatiilor interactive. *Background jobs* ruleaza in paralel si nu afecteaza procesele interative (foreground jobs).

Avantajele folosirii *background jobs:*

- Automatizeaza sarcini si reduc efortul introducerii manuale a datelor;
- Pot fi declansate sau programate sa se declanseze in functie de dorinta utilizatorului;
- Reduc interactiunea cu utilizatorul;
- Sunt ideale pentru sarcinile consumatoare de timp si pentru programele care folosesc multe resurse, putand fi programate sa ruleze peste noapte (cand sistemul nu este incarcat).

Criteriu de implementare a unei sarcini (*task*) drept *background job:*

Inainte de a implementa o sarcina (*task*) ca un *background job*, trebuie mai intai sa determinam daca acea sarcina va rula fara a fi nevoie de interactiunea cu utilizatorul si fara ca interfata Web sa trebuiasca sa astepte ca sarcina sa se temine. Sarcinile (*task-urile*) care au nevoie ca utilizatorul sa introduca date sau ca interfata sa astepte terminarea executiei lor, nu sunt potrivite spre a se implementa ca *background jobs*.

Tipuri de *background jobs*:

- *Job-uri* care solicita CPU, cum ar fi calcule matematice complexe;
- *I/O jobs*, cum ar fi schimbul de continut sau cautarea in fisiere;
- *Batch jobs*, cum ar fi actualizari de date peste noapte;
- Fluxuri de lucru care ruleaza o perioada indelungata, cum ar fi completarea unui formular;
- Procesari de date confidentiale.

Background jobs pot fi declansate **ca raspuns la un eveniment** sau **pot fi programate sa se declanseze pe baza unui cronometru**.

Exemplu de *background job* declansat ca raspuns la un eveniment:

Un job trimite un mesaj intr-o coada. Mesajul contine date despre actiuni care au avut loc (plasarea unei comenzi).Task-ul care asculta coada detecteaza sosirea unui mesaj, il citeste si foloseste informatia ca date de intrare pentru *background job*.

Alte sarcini (*task-uri*) care se pot implementa ca *background jobs* declansate ca un raspuns la un eveniment sunt cele care includ procesarea de imagini sau trimiterea de email-uri.

Exemplu de *background job* declansat ca pe baza unui cronometru:

Un proces sau o aplicatie porneste un cronometru care declanseaza apelarea *background job-ului* la un anumit moment. Alte sarcini (task-uri) de acest fel include *batch routines*, analiza de date pentru rapoarte zilnice, etc.

Background jobs se executa asincron intr-un proces separat sau chiar intr-o locatie separata de interfata utilizator (UI) sau de procesul care le-a apelat. In mod ideal, *background jobs* sunt operatii *"fire and forget"* iar executia lor nu are niciun impact asupra UI-ului ori asupra procesului care le-a apelat. Asta inseamna ca procesul care a apelat *background jobs* nu asteapta terminarea executiei lor si, in consecinta, nu pot detecta in mod automat cand ele se incheie.

Daca este nevoie ca un *background job* sa comunice cu task-ul care l-a apelat pentru a indica un progres sau terminarea executiei, trebuie implementat un mecanism care sa faca asta.

Spre **exemplu**, se poate stabili o coada care va da raspunsul catre UI-ul sau catre task-ul care o asculta. *Background job-ul* va trimite atunci mesaje cozii care sa indice progresul, starea sau terminarea executiei. Datele pe care *background job-ul* trebuie sa le returneze sunt plasate in aceste mesaje.

Conflicte

Daca exista mai multe instante ale unui *background job*, este posibil ca ele sa concureze pentru accesul la resurse, cum ar fi bazele de date. Accesul in mod concurent la resurse poate conduce la o lupta pentru ele, care poate cauza conflicte privind disponibilitatea de servicii si integritatea datelor stocate in baza de date. Aceasta lupta poate fi evitata prin blocarea accesului in mod concurent sau prin implementarea *background task-ului* ca *singleton*, asa incat doar o singura instanta va rula (**dezavantaj**: se elimina beneficiile legate de performanta pe care le furnizeaza un un sistem cu mai multe instante).

Este vital sa se asigure faptul ca *background job-ul* se va restarta automat si ca are capacitatea de a faca fata unui numar mare de cereri. Acest lucru poate fi realizat prin alocarea de reusurse suficiente sau prin implementarea unui mecanism de tip coada, care va stoca cererile atunci cand numarul lor creste.

Studiu de caz – Multithreading cu BackgroundWorker

In mod normal, de fiecare data cand o aplicatie executa o portiune de cod, acest cod este rulat pe acelasi thread ca si aplicatia, ceea ce presupune ca nimic altceva nu se intampla in interiorul aplicatiei, nici macar actualizarea UI-ului.

Acest lucru este surprinzator pentru incepatorii in programarea Windows, in special atunci cand scriu pentru prima data un program care ruleaza mai mult de o secunda si realizeaza ca de fapt, aplicatia lor nu face nimic altceva in timpul asta. Deseori solutia este actualizarea unui *progress*

bar in timpul rularii proceselor costisitoare ca timp, desi acest *progress bar* nu se va actualiza pana ce procesul se incheie.

Solutia adevarata este folosirea mai multor thread-uri, iar C# faciliteaza acest lucru punand la dispozitie *BackgroundWorker-ii*.

Cum functioneaza BackgroundWorker?

Cel mai dificil concept despre multithreading intr-o aplicatie Windows este faptul ca nu ai permisiunea de a face schimbari asupra UI-ului dintr-un alt thread deoarece aplicatia se va bloca. Ceea ce se poate face in schimb, este apelarea unei metode prin intermediul thread-ului principal din UI care sa faca schimbarile dorite.

Cand un *task* este apelat printr-un *thread* diferit, comunicarea cu restul aplicatiei are loc in doua cazuri:

1) Cand aplicatia trebuie actualizata, ca sa se vada starea la care a ajuns procesul;
2) Cand *task-ul* s-a incheiat si trebuie afisate rezultatele.

BakgroundWorker-ul este construit in jurul acestor idei si vine cu 3 evenimente:

- *PragressChanged*;
- *RunWorkerCompleted;*
- *DoWork.*

Regula generala in privinta lui DoWork este ca nu avem voie sa ne atingem de nimic din UI prin acest eveniment. In schimb, se apeleaza metoda ReportProgress(), care declaseaza evenimentul ProgressChanged, de unde se poate actualiza *UI-ul*. Odata incheiat procesul, un rezultat este atribuit *worker-ului* si evenimentul *RunWorkerCompleted* este declansat.

Cu alte cuvinte, evenimentul *DoWork* face treaba cea mai grea. Tot codul scris in cadul lui este executat printr-un alt thread si din acest motiv nu putem schimba ceva in UI din cadrul lui. In schimb, acest eveniment primeste date (din UI sau dintr-un alt loc al aplicatiei) apeland metoda *RunWorkerAsync()* si atribuind datele rezultate unei proprietati *e.Result.*

Evenimentele *ProgressChanged* si *RunWorkerCompleted* sunt executate pe acelasi *thread* in care este creat *BackgroundWorker-ul,* care de obicei este si thread-ul principal (*UI thread*) si astfel se pot face schimbari in UI prin intermediul lor. Comunicarea intre *task-ul* din *background* si *UI* este prin metoda *ReportProgress().*

https://www.wpf-tutorial.com/misc/multi-threading-with-the-backgroundworker/

https://www.guru99.com/background-job-processing.html

https://docs.microsoft.com/en-us/azure/architecture/best-practices/background-jobs?fbclid=IwAR2Y0JgxAJdrOFSn-9ZnpjWOpnaFFIiA-W91tsYZGYn4jkldymqUDMGoY-U