**FUNZIONE SINCRONA E FUNZIONE ASINCRONA CON AWAIT**

La differenza principale tra chiamare una funzione **sincrona** e una **asincrona** con **await** riguarda il modo in cui il programma gestisce l’attesa del risultato e l’uso delle risorse:

**Funzione sincrona:**

* Il codice si ferma e aspetta che la funzione finisca prima di proseguire.
* Blocca il thread corrente (ad esempio l’interfaccia utente si blocca durante una chiamata di rete lunga).

**Funzione asincrona con await:**

* Il codice “mette in pausa” solo il metodo corrente, ma il thread corrente viene rilasciato e può continuare a fare altro (ad esempio rispondere all’utente).
* Non blocca l’interfaccia utente: il programma resta reattivo.

**Come si comportano async e await in WPF?**

Distinguiamo il thread principale dai thread del thread pool. Un evento (click, caricamento, ecc.) viene gestito nel **thread principale**, detto **UI thread**. Questo è il thread che **aggiorna l’interfaccia** e **risponde all’utente**. Tipicamente, il gestore di quell’evento possiede al suo interno una chiamata ad una funzione asincrona con **await**. Quando viene eseguita quell’istruzione il codice:

* **si sospende all’await**.
* Rilascia **temporaneamente il thread UI**.
* La parte Async viene **eseguita in un thread secondario del thread pool** (es. per operazioni I/O come chiamate HTTP, lettura da DB, ecc.).

Mentre l’operazione await è in corso, **il thread della UI può continuare** a gestire altri eventi/interazioni. Questo evita **blocchi dell’interfaccia**.

Quando l’operazione asincrona finisce il codice **riprende automaticamente dal punto dopo l’await**. Questo accade **sul thread UI**, grazie al meccanismo di **SynchronizationContext** (WPF/WinForms). **Non avviene alcuna "biforcazione di thread" tipo fork-processi.** È semplicemente una sospensione e una ripresa nel giusto contesto.

**Importante**: async/await ≠ multithreading.  
Significa invece **“non bloccare il thread corrente mentre aspetti”**.

L'await rilascia il thread corrente, che può fare altro (es. gestire altri eventi UI), e poi riprende quando il lavoro è pronto. **Tutto gestito automaticamente.**

async/await **può usare** più thread oppure solo uno e l’operazione asincrona può essere eseguita in background da un meccanismo del SO, ma comunque **non è pensato per eseguire codice in parallelo**. È più che altro un **modello cooperativo**, più vicino a **"sospensione e ripresa"** che a "esecuzione parallela". L’obiettivo è **non bloccare** thread preziosi (come il thread UI), non fare le cose contemporaneamente.

**TIPO TASK<T> NELLA PROGRAMMAZIONE ASINCRONA IN WPF**

**Task<T>** è un tipo della libreria .NET che rappresenta un'operazione asincrona che restituirà un risultato di tipo **T** in futuro.  
È usato per la programmazione asincrona, cioè quando vuoi eseguire operazioni che potrebbero richiedere tempo (come chiamate HTTP, accesso a file, database, ecc.) senza bloccare l'interfaccia utente o il thread principale.

Quando usare **Task<T>** come tipo di ritorno?

* Quando il metodo esegue un'operazione **asincrona** e deve restituire un risultato.
* Tipico per metodi che fanno chiamate a servizi web, I/O, attese temporizzate, ecc.
* Se il metodo non restituisce un valore, si usa **Task** (senza **<T>**).

Quando NON usarlo

* Per metodi che eseguono solo operazioni sincrone e veloci (es: semplici calcoli, accesso a proprietà in memoria).
* Se non hai bisogno di eseguire codice in modo asincrono, usa un tipo di ritorno normale (int, string, ecc.).

**FRAME WPF**

Un Frame in WPF è un contenitore che gestisce la navigazione tra pagine (Page).

Pensa al frame come a una “finestra interna” che può caricare e mostrare diverse pagine, una alla volta, mantenendo una cronologia di navigazione (come un browser web).

Perché potresti avere più frame nella stessa pagina?

• Se vuoi aree indipendenti che navigano tra contenuti diversi.

Esempio: una pagina con un menu laterale (in un frame) e un’area principale (in un altro frame), ognuna con la propria cronologia.

• Se vuoi dialoghi o pannelli che cambiano contenuto senza cambiare la pagina principale.

• In applicazioni complesse, puoi avere frame annidati per gestire sezioni diverse in modo indipendente.

Significato pratico di frame

• Un frame ospita e visualizza una pagina (Page).

• Puoi navigare tra pagine all’interno di un frame senza cambiare la finestra principale.

• Ogni frame ha la sua cronologia di navigazione (puoi tornare indietro/avanti solo nel frame specifico).

In sintesi:

Un frame è come un browser dentro la tua app: puoi averne uno solo (caso più comune) o più di uno se vuoi gestire più “aree navigabili” indipendenti nella stessa finestra o pagina.

Nella maggior parte delle app, un solo frame principale è sufficiente.

**GORM E L’INSERIMENTO IN TABELLE**

Quando crei una nuova categoria e invii un JSON **senza campo**[id](vscode-file://vscode-app/c:/Users/signo/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) (o con [id: 0](vscode-file://vscode-app/c:/Users/signo/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)), GORM (e MySQL) si comportano così:

* In Go, il campo ID della struct sarà 0 (valore di default per uint).
* Quando chiami [database.DB.Create(&categoria)](vscode-file://vscode-app/c:/Users/signo/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o "), GORM **ignora il valore 0** e lascia che sia il database (MySQL) a generare automaticamente l’ID grazie all’attributo AUTO\_INCREMENT della colonna [id](vscode-file://vscode-app/c:/Users/signo/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
* Dopo l’inserimento, GORM aggiorna il campo ID della struct con il valore generato dal database.

**Quindi:**

* Non serve specificare l’ID nel JSON in fase di creazione.
* Non viene generato errore se l’ID è 0 o assente.
* L’inserimento funziona e l’ID viene assegnato automaticamente dal database.

**AGGIORNARE LE VIEW DI UNA PAGE**

Vediamo la differenza tra aggiornare direttamente le collezioni e usare il metodo FiltraCategorie.

1. Aggiornare direttamente le collezioni

Se aggiungi o rimuovi una categoria da entrambe le liste (Categorie e TutteLeCategorie), la DataGrid si aggiorna **solo se** la categoria è presente nella vista filtrata corrente.

**Limite:**  
Se c’è un filtro attivo (es. ricerca o checkbox), la nuova categoria potrebbe non apparire (o potresti vedere una categoria che non dovrebbe esserci) perché non rispetta il filtro corrente.

2. Usare FiltraCategorie

Quando chiami:

TutteLeCategorie.Add(finestra.CategoriaCreata);

FiltraCategorie((SearchBox?.Text) ?? string.Empty);

* Aggiorni la lista completa (TutteLeCategorie).
* Poi **rigeneri la vista filtrata** (Categorie) in base al filtro attuale (testo di ricerca, pubblica/privata, ecc).

**Vantaggio:**  
La DataGrid mostra sempre le categorie corrette in base ai filtri attivi, senza rischi di desincronizzazione tra la vista e la lista completa.

Cosa conviene fare?

**Conviene sempre aggiornare la lista completa e poi chiamare FiltraCategorie.**  
Così sei sicuro che la vista sia coerente con i filtri attivi e non rischi di mostrare dati non aggiornati o incoerenti.

**BEST PRACTICES PROPRIETA’ IN C#**

Ecco le best practices principali per implementare le proprietà in una classe C#:

1. \*\*Usa proprietà auto-implementate quando non serve logica aggiuntiva:\*\*

public string Nome { get; set; }

2. \*\*Rendi le proprietà il più possibile immutabili:\*\*

- Se il valore non deve cambiare dopo la creazione, usa solo `get` o `private set`:

public int Id { get; }

public string Nome { get; private set; }

```

3. \*\*Aggiungi logica di validazione nel `set` se necessario:\*\*

private int \_eta;

public int Eta

{

get => \_eta;

set

{

if (value < 0) throw new ArgumentException("L'età non può essere negativa");

\_eta = value;

}

}

```

4. \*\*Usa nomi PascalCase per le proprietà:\*\*

- Es: `public string Cognome { get; set; }`

5. \*\*Non esporre campi pubblici, preferisci sempre le proprietà:\*\*

- Favorisce l’incapsulamento e la futura estendibilità.

6. \*\*Per il binding (es. in WPF), implementa `INotifyPropertyChanged` se la proprietà può cambiare:\*\*

public event PropertyChangedEventHandler? PropertyChanged;

private string \_nome;

public string Nome

{

get => \_nome;

set

{

if (\_nome != value)

{

\_nome = value;

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(nameof(Nome)));

}

}

}

```

7. \*\*Documenta le proprietà pubbliche con commenti XML, se fanno parte di una API o libreria.\*\*

\*\*In sintesi:\*\*

Usa proprietà per incapsulare i dati, mantieni la logica semplice dove possibile, proteggi lo stato interno della classe e segui le convenzioni di naming di C#.

**FILE ‘\_\_init\_\_.py’ IN PYTHON**

A cosa serve?

• Trasforma directory in package Python: Senza \_\_init\_\_.py, Python non riconosce una cartella come package

• Controlla gli import: Definisce cosa è disponibile quando importi il package

• Codice di inizializzazione: Può contenere codice che si esegue al primo import

• Può essere vuoto: Spesso è vuoto, serve solo a "marcare" la directory come package

**ALTRI MODI PER DEFINIRE PACKAGE IN PYTHON**

• Namespace Packages (Python 3.3+): Puoi creare package SENZA \_\_init\_\_.py

• Differenze:

- Regular Package (con \_\_init\_\_.py): Più controllo, compatibilità, codice di inizializzazione

- Namespace Package (senza \_\_init\_\_.py): Più semplice, per package distribuiti

Il file \_\_init\_\_.py è il modo tradizionale e raccomandato per creare package Python. Anche se da Python 3.3+ puoi farne a meno, averlo ti dà più controllo e flessibilità.

**INTERFACCIA COL DB IN PYTHON: SESSIONMAKER(), SESSION E DEPEND() IN FASTAPI + SQLALCHEMY**

Una session **è un oggetto che rappresenta una connessione temporanea al database.**

Serve per:

* **Eseguire query**
* **Inserire/aggiornare/cancellare** dati
* **Tenere traccia** delle modifiche agli oggetti (ORM)
* Gestire **transazioni** (cioè blocchi di operazioni che vanno fatte tutte insieme o nessuna)

In pratica, è l'interfaccia principale che usi per “parlare” con il DB.

Sessionmaker() è una functions factory: crea una funzione che genera oggetti Session.

*Es. SessionLocal = sessionmaker(bind=engine)*

SessionLocal non è una sessione, è una funzione che crea nuove sessioni: quando chiami SessionLocal() ottieni una nuova Session.

**Ma perché non creo direttamente una Session?**

Perché:

* Hai bisogno di **una sessione nuova per ogni richiesta** (soprattutto in web app).
* Vuoi che tutte le sessioni siano **configurate nello stesso modo** (autocommit, autoflush, bind=engine, ecc.)
* Una factory ti permette di **centralizzare** questa configurazione e generare sessioni consistenti.

**Perché servono più sessioni, anche sullo stesso DB?**

Perché ogni sessione è **isolata**, come una mini-connessione con:

* La sua transazione attiva
* Le sue modifiche in corso (non visibili alle altre sessioni finché non fai commit)
* La sua cache degli oggetti (Unit of Work)

**In un'app web:**

Ogni **richiesta HTTP** (es: login, register, ecc.) viene gestita in parallelo da thread/processi diversi → ciascuno ha bisogno della **propria sessione**.

Condividere una sessione tra richieste è **pericoloso e sbagliato**: porta a corruzione dati, bug e race condition.

**yield — Cosa significa e perché si usa**

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

**✅ In breve:**

yield trasforma la funzione in un **generatore**. In senso tradizionale, un **generatore** in Python è una funzione **speciale** che restituisce un **oggetto iterabile (oggetto generatore)**, ma invece di restituire tutti i valori subito (come una lista), li produce **uno alla volta**, **solo quando servono**, usando la parola chiave yield.

*Es*.

*def conta\_fino\_a\_3():*

*yield 1*

*yield 2*

*yield 3*

*g = conta\_fino\_a\_3()*

*print(next(g)) # ➜ 1*

*print(next(g)) # ➜ 2*

*print(next(g)) # ➜ 3*

I generatori vengono anche usati per gestire in modo elegante e sicuro risorse che vanno **aperte e poi chiuse correttamente**.

**🧠 Come funziona in FastAPI:**

1. Quando una route richiede una Session, FastAPI chiama get\_db().
2. Il yield db **consegna la sessione al chiamante**.
3. Dopo che l'endpoint ha finito, FastAPI **riprende la funzione**, esegue il finally, e chiude la connessione (db.close()).

**Depends() — Cos'è e perché si usa**

python

CopiaModifica

def register\_user(user: UserCreate, db: Session = Depends(get\_db)):

**✅ In breve:**

Depends() è il **sistema di dependency injection** di FastAPI.

**🔍 Cosa fa:**

* Dice a FastAPI: “Per favore, **chiama get\_db() e metti il risultato nel parametro db**”.
* Se get\_db() è un generatore (con yield), FastAPI **gestisce l'intero ciclo di vita** della risorsa:
  + crea la sessione
  + la passa alla funzione
  + la chiude automaticamente dopo

**Senza Depends, dovresti gestire tutto a mano: apertura e chiusura della sessione inclusa.**

**Differenza tra db = get\_db() e db: Session = Depends(get\_db)**

**❌ db = get\_db()**

Se tu scrivessi così nel tuo endpoint:

*def register\_user(...):*

*db = get\_db()*

* **Ottieni un generatore**, non una sessione vera.
* get\_db() è una funzione **generatore**: non restituisce la sessione db direttamente, ma **restituisce un oggetto generatore**.
* Per ottenere la sessione, dovresti fare: *db = next(get\_db())*

Ma così **non eseguirai mai il finally: db.close()**, quindi **non chiuderesti mai la connessione al DB**, e nel tempo la tua app **collasserebbe**.

**✅ db: Session = Depends(get\_db)**

Con questo approccio: *def register\_user(user: UserCreate, db: Session = Depends(get\_db)):*

* FastAPI **esegue correttamente il generatore get\_db()**:
  1. Lo avvia → arriva al yield → prende la db
  2. Usa db nella funzione
  3. Quando la request finisce, riprende il generatore → esegue finally → **chiude la connessione**
* Questo è **dependency injection automatica**: FastAPI **inietta** nella funzione tutto ciò che serve, usando Depends(...).

Una **dependency** è **qualcosa di esterno di cui una funzione o un oggetto ha bisogno per funzionare**. **Injection** significa che quella dipendenza **non viene creata "dentro" la funzione**, ma le viene **fornita (iniettata)** da qualcun altro (di solito il framework). Quindi, **Dependency Injection** = "iniettare le dipendenze dall’esterno".

**POSTGENERATEEXAMS E PIPE IN GO**

È un endpoint HTTP del server (in Go con Gin) che, quando un docente invia una richiesta con i quesiti e il numero di schede, genera un archivio ZIP contenente:

* i file di testo di tutte le schede d’esame (con domande e opzioni mescolate);
* un file CSV con le soluzioni.

Lo ZIP viene **inviato direttamente al client** come download, senza essere salvato su disco.

Come?

Tramite una pipe. Una *pipe* in Go è un canale di comunicazione in memoria tra un *writer* (pw) e un *reader* (pr). Qualsiasi cosa scritta su pw può essere letta da pr **in streaming**, senza bufferizzare tutto quanto l’output in RAM.

io.Pipe **non** alloca un grande buffer per contenere l’intero output. Ha un **buffer piccolo** (implementato internamente o, in certi casi, nessun buffer reale) e un meccanismo di **blocco**:

* Se il lettore (pr) non legge, il writer (pw.Write) si **blocca** fino a quando non c’è spazio (proprietà di **backpressure**).
* Se il writer non scrive, il lettore (pr.Read) si **blocca** in attesa.

Questo significa che i dati “vivono” in RAM **solo il tempo necessario** a passare dal writer al reader, e in porzioni piccole (chunk).

Successivamente nel codice ci si serve di una goroutine per avviare la **generazione dello ZIP** in background mentre la funzione principale inizia a inviare dati al client.

Dopodichè si fa: io.Copy(c.Writer, pr)

* c.Writer è la connessione HTTP verso il client.
* pr è la pipe reader che riceve i dati generati dalla goroutine.
* io.Copy legge dalla pipe e scrive nella risposta **pezzo per pezzo** (chunked transfer encoding).
* Così il client inizia a ricevere lo ZIP **mentre viene creato**, senza aspettare la fine.

**Perché questa architettura**

* **Streaming immediato**: il download parte subito; non si aspetta di avere tutto il file pronto.
* **Efficienza memoria**: nessun file temporaneo su disco, nessun buffer gigante in RAM.
* **Scalabilità**: adatto anche a ZIP molto grandi.
* **Gestione errori pulita**: se la generazione fallisce, CloseWithError fa interrompere il download evitando di consegnare file corrotti.

Cosa dice il commento di io.Pipe() nella sua libreria:

“Pipe crea una pipe sincrona in-memory. Può essere utilizzata per collegare codice che si aspetta un io.Reader con codice che si aspetta un io.Writer.

"In-memory" significa che tutto avviene direttamente nella **memoria RAM** del programma, senza passare per il disco o per altre forme di archiviazione persistente.

Nel contesto della tua descrizione:

* La *pipe* creata da Pipe non scrive i dati su un file o un socket.
* I dati vengono trasferiti **da scrittore a lettore direttamente in RAM**.
* È veloce, ma i dati esistono solo finché il programma è in esecuzione — appena terminano la lettura o il programma, spariscono.

Le operazioni di lettura e scrittura sulla pipe vengono abbinate una a una, tranne nei casi in cui siano necessarie più letture per consumare una singola scrittura. In altre parole, ogni scrittura sul [PipeWriter] rimane bloccata finché non viene soddisfatta da una o più letture dal [PipeReader] che consumino completamente i dati scritti. I dati vengono copiati direttamente dalla scrittura alla corrispondente lettura (o letture); non è presente alcun buffering interno.

È sicuro chiamare Read e Write in parallelo tra loro o insieme a Close. Anche le chiamate parallele multiple a Read e a Write sono sicure: le singole chiamate verranno comunque eseguite in sequenza.”