?? Gestione offline — Dati locali, coda e sincronizzazione

Questa pagina descrive come utilizzare `IndexedDbLocalDataStoreService`, `CodaOfflineService` e `OfflineCrudServiceBase<TEntita>` per gestire dati e operazioni offline nelle applicazioni Soluziona.

Il flusso prevede il salvataggio sul dispositivo e il successivo invio al gestionale attraverso una coda persistente.

> □□ **Salvataggio locale e salvataggio sul server sono distinti**
> Un'operazione accodata è disponibile sul dispositivo, ma deve ancora essere confermata dal server.

? Responsabilità dei servizi

Componente	Responsabilità
`IndexedDbLocalDataStoreService`	Salva, legge ed elimina dati nel database locale del browser.
`CodaOfflineService`	Memorizza le operazioni da inviare, il loro collegamento alle entità e gli errori di sincronizzazione.
`OfflineCrudServiceBase<TEntita>`	Fornisce lettura e salvataggio locali e gestisce i diversi casi di eliminazione offline.
`SyncService`	Esegue le chiamate HTTP presenti in coda.
`SyncMonitorService`	Richiede la sincronizzazione quando cambia la connessione o viene modificata la coda, se il dispositivo risulta online.
`ISyncSuccessHandler`	Aggiorna i dati locali dopo una risposta positiva del server, secondo le regole del dominio.

? Flusso generale

```
L’utente esegue un’operazione
            ↓
Il servizio applicativo salva l’entità in IndexedDB
            ↓
Accoda il comando da inviare al server
            ↓
La UI può mostrare il dato locale come “Da sincronizzare”
            ↓
SyncService invia il comando quando viene avviata la sincronizzazione
            ↓
Il server risponde positivamente
            ↓
L’handler aggiorna l’entità locale
            ↓
L’operazione viene marcata come sincronizzata e rimossa dalla coda
```

Questo modello può essere utilizzato anche quando il dispositivo è già online: il servizio applicativo salva e accoda, mentre il monitor può avviare subito l’invio.

> **i L’offline va implementato per ciascuna funzionalità**
> La presenza dei servizi non rende automaticamente offline tutte le chiamate API dell’app.

? IndexedDbLocalDataStoreService

È il servizio di accesso ai dati locali. Serializza gli oggetti in JSON e richiama le funzioni JavaScript di `soluzionaIndexedDb`.

Metodi disponibili

Metodo	Utilizzo
`SalvaAsync<T>(storeName, key, value)`	Inserisce o sostituisce il valore associato alla chiave.
`RicercaByKeyAsync<T>(storeName, key)`	Recupera un oggetto; restituisce il valore predefinito se assente.
`ElencoAsync<T>(storeName)`	Legge gli oggetti dello store logico.

Metodo	Utilizzo
`EliminaAsync(storeName, key)`	Elimina un singolo oggetto.
`PulisciStoreAsync(storeName)`	Elimina tutti gli oggetti dello store logico.

Store e chiavi

Nell’implementazione JavaScript di Timbratura:

Elemento	Valore
Database	`SoluzionaAppsDb`
Object store fisico	`local_data`
Identificativo del record	`storeName::key`

Il parametro `storeName` rappresenta quindi un **raggruppamento logico** all’interno dello stesso object store.

Esempio di utilizzo

```
string store = TimbraturaLibrary.STORE_RICHIESTA;
string key = richiesta.Id.ToString();

// Salvataggio locale
await localDataStore.SalvaAsync(store, key, richiesta, cancellationToken);

// Lettura per chiave
CustomRichiesta? salvata = await localDataStore.RicercaByKeyAsync<CustomRichiesta>(store, key,
cancellationToken);

// Lettura elenco
List<CustomRichiesta> elenco = await localDataStore.ElencoAsync<CustomRichiesta>(store,
cancellationToken);
```

> □□ **Salvare in IndexedDB non accoda alcuna chiamata**
> L’invio al server deve essere predisposto esplicitamente tramite `CodaOfflineService`.

? CodaOfflineService

La coda viene memorizzata nello store logico **`offline_queue`**.

Ogni elemento descrive un’operazione da eseguire verso il server.

Informazioni principali

Campo	Significato
`Id`	Identificativo dell’elemento in coda.
`IdOperazioneClient`	Identificativo dell’operazione, utile per riconoscere i tentativi ripetuti.
`TipoOperazione`	Identifica la funzione applicativa e gli handler da eseguire.
`AzioneCoda`	Classifica l’azione: creazione, modifica, eliminazione o personalizzata.
`Endpoint`	Percorso API da chiamare.
`MetodoHttp`	Metodo HTTP da utilizzare.
`PayloadJson`	Corpo della richiesta serializzato.
`StoreNameEntitaLocale`	Store dell’entità collegata.
`KeyEntitaLocale`	Chiave dell’entità collegata.
`NumeroTentativi`	Numero di tentativi falliti registrati.
`UltimoErrore`	Dettaglio dell’ultimo errore.
`IsSincronizzato`	Indica se l’operazione è stata completata.

Accodare un’operazione

Estratto del flusso utilizzato per le richieste:

```
await CodaOfflineService.AccodaAsync(
    tipoOperazione: TimbraturaOperazioniOffline.RichiestaCreate,
    endpoint: "api/Richiesta/Nuova",
    payload: request,
    storeNameEntitaLocale: TimbraturaLibrary.STORE_RICHIESTA,
    keyEntitaLocale: richiestaLocale.Id.ToString(),
    azioneCoda: AzioniCodaOffline.Create,
    cancellationToken: cancellationToken);
```

Il collegamento tramite **store e chiave** permette di recuperare tutte le operazioni relative alla stessa entità.

> ⚠ **Azione logica e metodo HTTP sono diversi**
> `AccodaAsync` imposta attualmente `MetodoHttp = "POST"`. Passare `AzioniCodaOffline.Delete` classifica l'operazione come eliminazione, ma non la trasforma in una richiesta HTTP `DELETE`.

Metodi di gestione

Metodo	Utilizzo
`GetPendentiAsync()`	Restituisce le operazioni non sincronizzate, ordinate per creazione.
`GetPendentiPerEntitaAsync(...)`	Recupera le operazioni collegate a un'entità.
`EsisteCreatePendentePerEntitaAsync(...)`	Verifica la presenza di una creazione pendente.
`RimuoviPendentiPerEntitaAsync(...)`	Rimuove tutte o alcune azioni pendenti dell'entità.
`SetAsSincronizzatoAsync(...)`	Registra il completamento dell'operazione.
`SetAsFallitoAsync(...)`	Incrementa i tentativi e memorizza l'errore.
`PulisciSincronizzatiAsync()`	Elimina dalla coda le operazioni completate.

Le modifiche alla coda generano l'evento **`CodaModificata`**, utilizzabile dal monitor e dalla UI.

? OfflineCrudServiceBase

La classe base richiede un'entità che implementi **`IEntitaOffline`**.

```
public interface IEntitaOffline
{
    Guid Id { get; set; }
    Guid? IdServer { get; set; }
    EStatoSyncLocale StatoSyncLocale { get; set; }
    bool IsEliminataLocalmente { get; set; }
}
```

Proprietà	Significato
`Id`	Identificativo usato localmente.
`IdServer`	Identificativo del gestionale, se già disponibile.
`StatoSyncLocale`	Stato della sincronizzazione dell'entità.
`IsEliminataLocalmente`	Segnala che l'entità deve essere esclusa dalla visualizzazione.

Stati disponibili

Stato	Significato
`Sincronizzata`	Entità allineata al server.
`DaCreare`	Creazione ancora da sincronizzare.
`DaModificare`	Modifica ancora da sincronizzare.
`DaEliminare`	Eliminazione ancora da sincronizzare.
`ErroreSync`	Stato disponibile per rappresentare un errore di sincronizzazione.

> **i Gli stati non vengono tutti gestiti automaticamente**
> Il servizio concreto e gli handler devono mantenere aggiornato lo stato. Ad esempio, un errore della coda non imposta automaticamente `ErroreSync` sull'entità.

Cosa implementa la classe base

- `RicercaLocaleByIdAsync`: cerca l'entità nello store configurato.
- `SalvaLocaleAsync`: salva l'entità usando, per impostazione predefinita, `Id.ToString()` come chiave.
- `EliminaAsync`: applica le regole di eliminazione offline.

Il servizio derivato deve definire:

```
protected override string StoreName => TimbraturaLibrary.STORE_RICHIESTA;


protected override Task AccodaEliminazioneAsync( entita, cancellationToken = default)
{
    // Accodamento del comando specifico del dominio.
```

```
}
```

Creazione, modifica, lettura dal server e gestione dei conflitti restano responsabilità del servizio applicativo.

?? Come funziona l’eliminazione offline

Caso 1 — Entità non ancora sincronizzata

La classe base considera questo caso quando si verifica almeno una condizione:

- Esiste una `Create` pendente.
- Lo stato è `DaCreare`.
- `IdServer` è `null`.

L’eliminazione:

1. Rimuove tutte le operazioni pendenti collegate.
2. Elimina fisicamente l’entità locale.
3. Non accoda un comando di eliminazione.

```
Creazione locale → CREATE in coda → Eliminazione prima dell’invio
                                          ↓
                                Rimozione del dato e della coda
```

Caso 2 — Entità già presente sul server

L’eliminazione:

1. Rimuove le eventuali `Update` pendenti.
2. Imposta `StatoSyncLocale = DaEliminare`.
3. Imposta `IsEliminataLocalmente = true`.
4. Salva l’entità locale.
5. Chiama `AccodaEliminazioneAsync`.

Il servizio concreto prepara il comando per il gestionale. Dopo la conferma, l’handler può eliminare definitivamente il dato locale.

> **□□ La UI deve filtrare le entità eliminate localmente**
> La classe base imposta il flag, ma non modifica direttamente le liste visualizzate.

```
var visibili = elenco
    .Where(x => !x.IsEliminataLocalmente)
    .ToList();
```

? Invio e gestione degli esiti

`SyncService.SincronizzaAsync()`:

1. Verifica che non sia già in corso una sincronizzazione nella stessa istanza.
2. Controlla la connessione.
3. Recupera le operazioni pendenti.
4. Seleziona quelle con meno di **5 tentativi falliti**.
5. Le elabora in ordine di creazione.
6. Invia le richieste tramite il client autenticato **`SoluzionaApi`**.
7. Esegue gli handler compatibili quando la risposta HTTP è positiva.
8. Marca le operazioni riuscite e le rimuove dalla coda.

In caso di errore, l'operazione rimane pendente con `NumeroTentativi` e `UltimoErrore` aggiornati.

> ⚠ **Dopo 5 fallimenti l'operazione resta in coda, ma viene esclusa dai successivi invii**
> Occorre prevedere una gestione applicativa per diagnosi e ripristino. Il servizio non applica automaticamente un nuovo ciclo illimitato di tentativi.

L'evento **`SyncCompletata`** indica che il ciclo non ha prodotto errori tra gli elementi elaborati: non garantisce che la coda sia vuota, perché possono restare elementi esclusi per numero di tentativi.

? Handler dopo la sincronizzazione

Gli handler implementano **`ISyncSuccessHandler`** e vengono selezionati in base a `TipoOperazione`.

Sono il punto in cui gestire:

- L'identificativo restituito dal server.

- L’allineamento dei dati locali alla risposta.
- Il passaggio a `Sincronizzata`.
- La rimozione locale dopo un’eliminazione confermata.

Nel progetto Timbratura sono registrati:

```
services.AddScoped<ISyncSuccessHandler,RichiestaCreateSyncSuccessHandler>();
services.AddScoped<ISyncSuccessHandler,RichiestaDeleteSyncSuccessHandler>();
```

> ◫ **Registrare gli handler è parte dell’integrazione**
> Senza un handler compatibile, `SyncService` può comunque considerare riuscita la chiamata HTTP e rimuoverla dalla coda, lasciando il dato locale non allineato.

?? Come integrare una nuova entità

1. **Definisci il modello locale** implementando `IEntitaOffline`.
2. **Scegli uno store logico** e una chiave coerente.
3. **Crea il servizio applicativo** derivando da `OfflineCrudServiceBase<TEntita>`.
4. **Implementa la creazione locale** con ID locale, `IdServer = null` e stato `DaCreare`.
5. **Salva e accoda** il DTO destinato all’API, collegandolo a store e chiave.
6. **Implementa l’accodamento dell’eliminazione** per le entità già sincronizzate.
7. **Implementa e registra gli handler** di successo.
8. **Integra la UI** con dati locali, indicatori di sincronizzazione e filtro delle eliminazioni.
9. **Avvia il monitor** nel ciclo di vita dell’app e fermalo quando non serve più.

Per richiedere esplicitamente un ciclo di sincronizzazione:

```
var esiti = await syncService.SincronizzaAsync(cancellationToken);
```

`SyncMonitorService.Avvia()` sottoscrive gli eventi; non esegue da solo un invio iniziale e non introduce un timer periodico.

? Aspetti da gestire nell’applicazione

Aspetto	Indicazione
Idempotenza	Il server deve riconoscere `IdOperazioneClient` per evitare duplicazioni in caso di reinvio. La sola presenza dell’ID nel client non basta.

Aspetto	Indicazione
Atomicità	Salvataggio dell’entità e accodamento sono operazioni separate: gestisci il caso in cui riesca soltanto la prima.
Operazioni già in invio	Rimuovere un elemento dalla coda non annulla una richiesta HTTP già partita. Coordinare eliminazione e sincronizzazione richiede attenzione.
Aggiornamento delle liste	Unisci i dati server con quelli locali pendenti, evitando di perdere creazioni offline o ripristinare elementi marcati per eliminazione.
Cambio utente	La coda viene inviata con la sessione corrente: gestisci l’appartenenza dei dati a utente e azienda prima di consentire cambi di account.
Persistenza locale	IndexedDB conserva i dati nel browser; la cancellazione dei dati del sito può rimuovere anche le operazioni non inviate.
Disponibilità offline dell’app	IndexedDB conserva i dati. Il caricamento dell’app senza rete dipende anche dalla configurazione PWA e dal service worker.

> □□ **Messaggi corretti per l’utente**
> Dopo l’accodamento usa una conferma come **“Salvato sul dispositivo, in attesa di sincronizzazione”**. Riserva **“Sincronizzato”** alla conferma effettiva del server.

??? Guida per lo sviluppatore — Esempio generico

L’esempio seguente mostra come gestire offline una semplice **nota**, dalla creazione locale all’eliminazione.

I nomi degli endpoint e i DTO sono illustrativi: devono essere adattati alle API del progetto.

1. Definisci il modello locale

Il modello locale contiene i dati applicativi e le informazioni necessarie alla sincronizzazione.

```
public sealed class CustomNotaLocale : IEntitaOffline
{
    public Guid Id { get; set; }
    public Guid? IdServer { get; set; }

    public EStatoSyncLocale StatoSyncLocale { get; set; }
    public bool IsEliminataLocalmente { get; set; }

    public string Testo { get; set; } = string.Empty;
}
```

Definisci separatamente i DTO scambiati con il server:

```
public sealed class ReqCreaNota : ReqOperazioneOfflineBase
{
    public string Testo { get; set; } = string.Empty;
}

public sealed class ReqEliminaNota : ReqOperazioneOfflineBase
{
    public Guid IdNota { get; set; }
}

public sealed class ResNota
{
    public Guid Id { get; set; }
    public string Testo { get; set; } = string.Empty;
}
```

> **□□ Modello locale e DTO hanno scopi diversi**
> Il modello locale serve alla UI e alla gestione offline. Il DTO contiene i dati richiesti dall'API. Gli stati di sincronizzazione non devono necessariamente essere inviati al server.

2. Definisci store e tipi di operazione

Usa costanti condivise tra servizio e handler.

```
public static class {NomeApp}Library
{
    public const string Store = "note";

    public const string Create = "NOTA_CREATE";
    public const string Delete = "NOTA_DELETE";
}
```

`Store` identifica il raggruppamento dei dati locali. `Create` e `Delete` permettono al sincronizzatore di individuare gli handler appropriati.

3. Implementa il servizio applicativo

Il servizio eredita la gestione dell'eliminazione da `OfflineCrudServiceBase`.

```
public sealed class NotaService : OfflineCrudServiceBase<NotaLocale>
{
    protected override string StoreName => NotaOffline.Store;

    public NotaService(IIndexedDbLocalDataStoreService localDataStore, ICodaOfflineService
codaOfflineService) : base(localDataStore, codaOfflineService)
    {
    }

    public async Task<CustomNotaLocale> CreaAsync(string testo, CancellationToken
cancellationToken = default)
    {
        if (string.IsNullOrWhiteSpace(testo))
            throw new ArgumentException("Inserisci il testo della nota.");

        var nota = new CustomNotaLocale
        {
            Id = Guid.NewGuid(),
            IdServer = null,
            Testo = testo.Trim(),
            StatoSyncLocale = EStatoSyncLocale.DaCreare,
            IsEliminataLocalmente = false
        };

        var request = new ReqCreaNota
```

```
        {
            IdOperazioneClient = Guid.NewGuid(),
            Testo = nota.Testo
        };

        // 1. Rende il dato disponibile sul dispositivo.
        await SalvaLocaleAsync(nota, cancellationToken);

        // 2. Predispone l'invio al server.
        await CodaOfflineService.AccodaAsync(
            tipoOperazione: NotaOffline.Create,
            endpoint: "api/Note/Crea",
            payload: request,
            storeNameEntitaLocale: StoreName,
            keyEntitaLocale: GetKey(nota.Id),
            azioneCoda: AzioniCodaOffline.Create,
            cancellationToken: cancellationToken);

        return nota;
    }

    public async Task<List<CustomNotaLocale>> ElencoLocaleAsync(CancellationToken
cancellationToken = default)
    {
        var elenco = await LocalDataStore.ElencoAsync<NotaLocale>(StoreName,
cancellationToken);

        return elenco.Where(x => !x.IsEliminataLocalmente).ToList();
    }

    protected override async Task AccodaEliminazioneAsync(NotaLocale entita, CancellationToken
cancellationToken = default)
    {
        if (!entita.IdServer.HasValue || entita.IdServer.Value == Guid.Empty)
        {
            throw new InvalidOperationException( "Identificativo server della nota non
disponibile.");
        }

        var request = new ReqEliminaNota
```

```
        {
            IdOperazioneClient = Guid.NewGuid(),
            IdNota = entita.IdServer.Value
        };

        await CodaOfflineService.AccodaAsync(
            tipoOperazione: NotaOffline.Delete,
            endpoint: "api/Note/Elimina",
            payload: request,
            storeNameEntitaLocale: StoreName,
            keyEntitaLocale: GetKey(entita.Id),
            azioneCoda: AzioniCodaOffline.Delete,
            cancellationToken: cancellationToken);
    }
}
```

In questo esempio, entrambi gli endpoint accettano richieste **HTTP POST**, coerentemente con il comportamento attuale di `AccodaAsync`.

Per eliminare una nota si utilizza il metodo ereditato:

```
await notaService.EliminaAsync(idNota, cancellationToken);
```

La classe base distingue automaticamente tra nota ancora locale e nota già sincronizzata.

> □□ **L’esempio mantiene stabile l’ID locale**
> Dopo la creazione sul server, `Id` resta invariato e `IdServer` viene valorizzato con l’identificativo ricevuto. In questo modo il collegamento tra UI, record locale e coda rimane stabile.

4. Gestisci la conferma del server

L’handler aggiorna il record locale dopo una creazione riuscita e lo rimuove dopo un’eliminazione confermata.

L’esempio assume che l’endpoint di creazione restituisca direttamente un oggetto `NotaResponse`.

```
public sealed class NotaSyncSuccessHandler : ISyncSuccessHandler
{
    private readonly IIndexedDbLocalDataStoreService _localDataStore;
```

```
    public NotaSyncSuccessHandler(IIndexedDbLocalDataStoreService localDataStore)
    {
        _localDataStore = localDataStore;
    }

    public bool CanHandle(string tipoOperazione)
    {
        return tipoOperazione == NotaOffline.Create || tipoOperazione == NotaOffline.Delete;
    }

    public async Task GestisciSuccessoAsync( CustomOggettoInCoda item, string? responseJson,
CancellationToken cancellationToken = default)
    {
        string store = item.StoreNameEntitaLocale ?? throw new
InvalidOperationException("Store locale mancante.");

        string key = item.KeyEntitaLocale ?? throw new InvalidOperationException("Chiave
locale mancante.");

        if (item.TipoOperazione == NotaOffline.Delete)
        {
            await _localDataStore.EliminaAsync( store, key, cancellationToken);

            return;
        }

        var risposta = JsonSerializer.Deserialize<NotaResponse>( responseJson ?? throw new
InvalidOperationException("Risposta vuota."),
            new JsonSerializerOptions
            {
                PropertyNameCaseInsensitive = true
            });

        if (risposta is null || risposta.Id == Guid.Empty)
        {
            throw new InvalidOperationException( "La risposta non contiene un identificativo
valido.");
        }
```

```
        var nota = await _localDataStore.RicercaByKeyAsync<CustomNotaLocale>( store, key,
cancellationToken);

        if (nota is null)
        {
            throw new InvalidOperationException( "La nota locale da sincronizzare non è
disponibile.");
        }

        nota.IdServer = risposta.Id;
        nota.Testo = risposta.Testo;
        nota.StatoSyncLocale = EStatoSyncLocale.Sincronizzata;

        await _localDataStore.SalvaAsync( store, key, nota, cancellationToken);
    }
}
```

Non è necessario chiamare `SetAsSincronizzatoAsync` dentro l'handler: lo esegue `SyncService` dopo che gli handler sono terminati correttamente.

> ⚠ **Gestisci esplicitamente le anomalie**
> Se il server ha creato la nota ma l'handler fallisce, l'operazione può essere ritentata. L'API deve riconoscere lo stesso `IdOperazioneClient` e restituire il risultato già prodotto senza creare duplicati.

5. Registra servizio e handler

Aggiungi le registrazioni specifiche dell'app:

```
services.AddScoped<NotaService>();

services.AddScoped<ISyncSuccessHandler, NotaSyncSuccessHandler>();
```

I servizi condivisi di storage, coda, connettività e sincronizzazione devono essere già registrati tramite la configurazione Core del progetto.

6. Utilizza il servizio dalla pagina

La pagina lavora con il servizio applicativo e rilegge l'elenco locale dopo le operazioni.

```
private List<CustomNotaLocale> Note = new();

private async Task CreaNotaAsync(string testo)
{
    await NotaService.CreaAsync(testo);

    Note = await NotaService.ElencoLocaleAsync();

    // Mostrare una conferma di salvataggio locale.
}

private async Task EliminaNotaAsync(Guid id)
{
    await NotaService.EliminaAsync(id);

    Note = await NotaService.ElencoLocaleAsync();
}
```

Per un comando manuale di sincronizzazione:

```
private async Task SincronizzaAsync()
{
    var esiti = await SyncService.SincronizzaAsync();

    Note = await NotaService.ElencoLocaleAsync();

    // Esaminare gli esiti e gli eventuali elementi ancora pendenti.
}
```

Quando è attivo `SyncMonitorService`, la sincronizzazione può partire anche automaticamente. La pagina deve quindi aggiornarsi agli eventi appropriati, come `SyncCompletata` e `SyncFallita`, rimuovendo le sottoscrizioni alla propria chiusura.

7. Verifica i casi principali

Prova	Risultato atteso
Crea una nota offline	Record locale con stato `DaCreare` e operazione `Create` pendente.
Elimina la nota prima della sincronizzazione	Rimozione del record locale e delle operazioni pendenti collegate.

Prova	Risultato atteso
Sincronizza una creazione	`IdServer` valorizzato, stato `Sincronizzata` e operazione rimossa dalla coda.
Elimina una nota sincronizzata	Nota nascosta localmente e operazione `Delete` pendente.
Sincronizza l’eliminazione	Rimozione definitiva del record locale.
Simula un errore API	Operazione ancora pendente, errore memorizzato e contatore incrementato.

> **□□ Prima dell’utilizzo in produzione**
> Completa l’esempio con la gestione del fallimento tra salvataggio e accodamento, il coordinamento delle eliminazioni mentre una sincronizzazione è in corso e l’isolamento dei dati per utente e azienda. I tre servizi non risolvono automaticamente questi casi.

Revisione #10
Creato 2026-09-29 14:43:00 UTC da Admin
Aggiornato 2026-09-29 15:10:43 UTC da Admin