# Sincronizzazione

## Indice generale

# Sincronizzazione

Comunica in tempo reale. Semplifica l'architettura client-cloud. Gestisci il funzionamento in assenza di connessione.

# Introduzione

I sistemi di trasformazione digitale presentano frequentemente architetture complesse in cui i vari componenti devono comunicare fra di loro, spesso in tempo reale.

Alcuni esempi di questo tipo di comunicazioni sono:

- Messaggistica integrata con l'applicazione, stile WhatsApp.
- Notifica di eventi rilevati da sistemi IoT come allarmi, stato degli impianti, eccetera.
- Comunicazione del cambio di stato di un documento da una sessione alle altre.
- Acquisizione di informazioni dal backend da parte di un device.
- Allineamento di un database offline con la controparte cloud.

Instant Developer Cloud include un sistema di sincronizzazione e di scambio di messaggi fra sessioni applicative e device in grado di risolvere i casi sopra indicati. Le caratteristiche principali sono le seguenti:

- Gestione automatica della connessione con la controparte nel cloud.
- Scambio di messaggi in tempo reale fra sessioni e applicazioni.
- Gestione di messaggi in modalità offline.
- Gestione di operazioni remote sui documenti.
- Sincronizzazione dello stato dei documenti fra database offline e backend cloud.

## Architettura delle sessioni

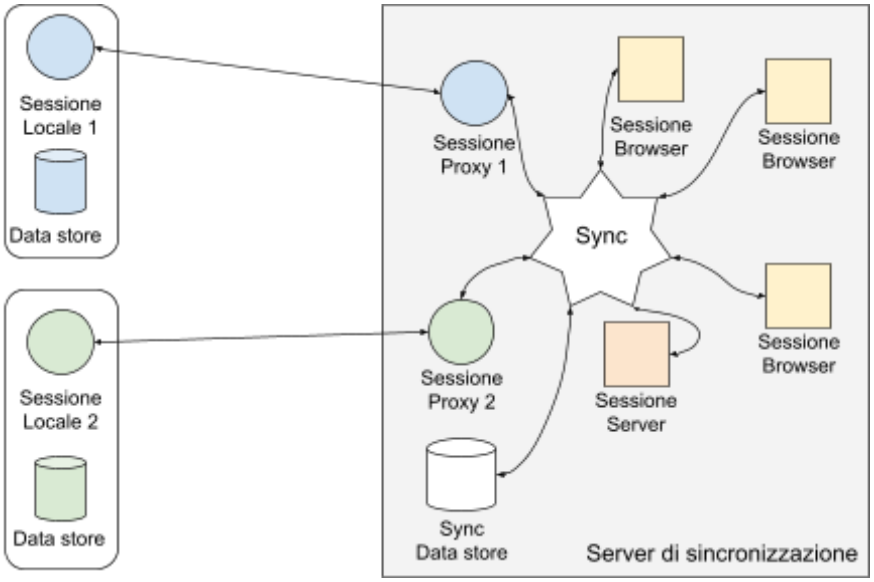
La sincronizzazione mette in comunicazione le varie sessioni applicative, sia quelle in esecuzione nel server nel cloud, che quelle in esecuzione nei device degli utenti. Le prime vengono chiamate sessioni online, le seconde sessioni offline o sessioni locali.

È importante notare che dal termine sessione offline deriva il concetto di *applicazione offline* utilizzato in questa documentazione. Si definisce infatti *applicazione offline* un'applicazione in grado di funzionare *anche* in assenza di connessione internet e che può essere sincronizzata con il server ogni volta che il device in cui è installata si riconnette a internet.

Per far sì che una sessione locale possa comunicare con le altre sessioni, sia quelle online che quelle locali nei dispositivi, la sincronizzazione crea per ogni dispositivo connesso una sessione *proxy* nel cloud. Si chiama *proxy* perché essa rappresenta il device all'interno del server per quanto riguarda le operazioni di sincronizzazione.

Lo schema alla pagina seguente illustra la connessione tra le sessioni di vario tipo: se, ad esempio, la sessione server riceve un evento IoT come l'attivazione di un allarme, essa può comunicarlo a tutte le sessioni interessate, sia quelle online che quelle locali, in esecuzione nei device.

L’invio dei messaggi ai device potrà avvenire anche se sono momentaneamente offline, in quanto sia il server nel cloud che i device possiedono un data store della sincronizzazione che memorizza i messaggi non volatili in modo che possano essere consegnati anche se la connessione non era attiva al momento dell’invio.




Le sessioni locali e online possono appartenere anche ad applicazioni diverse. Ad esempio l’applicazione nel server può essere quella di back office, mentre quella sui device viene usata dagli utenti finali. Nel server, inoltre, possono essere presenti più applicazioni diverse e la sincronizzazione può essere configurata per inviare messaggi anche tra sessioni che non appartengono alla stessa applicazione. Questo può essere utile se le varie applicazioni modificano gli stessi dati o appartengono allo stesso sistema informativo.

# Configurazione della sincronizzazione

Il sistema di sincronizzazione deve essere configurato per ogni tipo di sessione, sia nel server che per quelle locali ai device. In questo paragrafo viene analizzata la configurazione base per attivare la sincronizzazione; nei paragrafi successivi verranno aggiunti ulteriori parametri in funzione dei servizi che si desidera utilizzare.

## Sessioni online

Per le sessioni online, la configurazione della sincronizzazione deve avvenire all’inizio della sessione, quindi:

- Nell’evento *app.onStart* per le sessioni di tipo browser o di tipo server.
- Nell’evento *app.onCommand* per le sessione di tipo REST.
- Nell’evento *app.sync.onConnect* per le sessioni di tipo proxy.

La prima proprietà da configurare negli eventi di inizio sessione è *app.sync.dataStore*, che serve per comunicare alla sincronizzazione il database da utilizzare per memorizzare i messaggi permanenti, cioè quelli che devono essere inviati ai device anche se sono momentaneamente disconnessi. Ad esempio:

```
app.sync.dataStore = App.NwindDB;
```

La seconda proprietà è *relatedApps*, che rappresenta il nome dell'installazione delle altre applicazioni che devono ricevere i messaggi di sincronizzazione, impostate come stringa o array di stringhe. Questa proprietà deve essere impostata solo se nel server sono installate due o più applicazioni che trattano gli stessi dati da sincronizzare. Ad esempio:

```
app.sync.relatedApps = "backoffice";
```

La terza proprietà è *topics,* un oggetto JavaScript che rappresenta i messaggi ai quali è interessata la sessione. Ogni sessione, infatti, non deve ricevere tutti i messaggi delle altre sessioni, ma solo quelli che la riguardano direttamente. I topics verranno spiegati nei paragrafi successivi. Ecco un esempio:

```
app.sync.topics = ["utente0", "utente10", "utente2"];
```

## Sessioni locali (offline)

In aggiunta alle proprietà delle sessioni online, quelle locali devono specificare *serverUrl* e *appName,* di solito nell'evento *onStart*. La prima rappresenta il server nel cloud a cui ci si deve collegare, la seconda il nome dell'installazione dell'applicazione che gestisce la sincronizzazione. Ad esempio:

```
if (app.runsLocally()) {
  app.sync.serverUrl = app.sync.serverUrl ||
                       "https://myserver.instantdevelopercloud.com";
  app.sync.appName = "backoffice";
}
```

Il metodo *app.runsLocally* restituisce *true* se la sessione è locale, così da poter inserire il codice aggiuntivo solo in questo caso.

Si noti che *serverUrl* viene impostata solo se è vuota, perché il framework di Instant Developer Cloud può preimpostare questa proprietà quando l'applicazione viene lanciata dall'IDE in anteprima FEBE (Front End - Back End).

Infine è possibile impostare la proprietà *autoConnect*, che rappresenta la modalità di gestione della connessione. Il valore di default è *App.Sync.autoConnectTypes.automatic*, che crea una connessione solo al bisogno e poi la disattiva dopo un tempo prestabilito di inattività.

Nella pratica comune, però, è preferibile lasciare la sessione sempre connessa al server di sincronizzazione in modo che possa rimanere aggiornata in tempo reale. Per ottenere questo risultato, inserire il seguente codice:

```
app.sync.autoConnect = App.Sync.autoConnectTypes.alwaysConnected;
```

# Attivazione della sincronizzazione

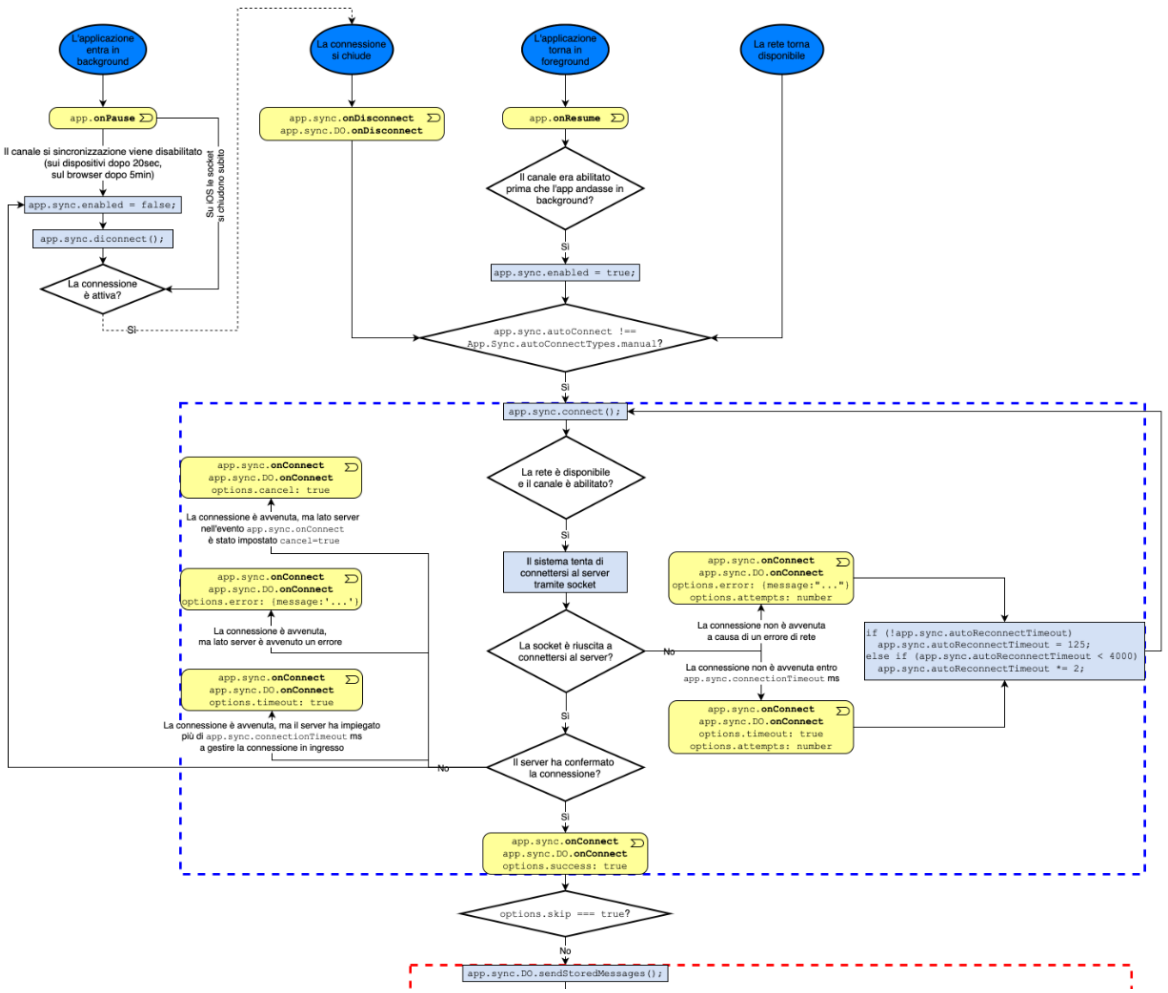
Mentre le sessioni online sono sempre connesse al proprio sistema di sincronizzazione, per quelle locali la sincronizzazione deve essere esplicitamente attivata tramite la proprietà *enabled*. Quindi, dopo aver impostato le proprietà di configurazione, quando si deve attivare la sincronizzazione occorre scrivere il seguente codice:

```
app.sync.enabled = true;
```

Fintanto che questa proprietà rimane attiva, la sessione locale gestirà la sincronizzazione. Questo non significa che il device rimarrà sempre connesso al server. Esso si può disconnettere e poi riconnettere in automatico, a seconda del valore della proprietà *autoConnect*, dello stato della connessione e dello stato del device.

Si segnala che se la proprietà *autoConnect* è impostata a *App.Sync.autoConnectTypes. manual* l'attivazione della proprietà *enabled* non apre mai la connessione; sarà necessario usare i metodi *connect* e *disconnect* dell'oggetto *sync* per gestirla.

Nell'immagine seguente viene riportato il ciclo di vita della connessione all'interno di una sessione locale.



## Attivazione della connessione

Quando il framework tenta di aprire la connessione, viene contattata l'applicazione specificata del server di sincronizzazione e, lato server, viene creata una sessione proxy relativa al device che sta tentando di aprire la connessione.

Nella sessione proxy, cioè lato server, viene notificato l'evento *app.sync.onConnect* che permette alla sessione di determinare se il device ha il diritto di connettersi o meno. Se nel codice dell'evento viene attivata la proprietà *cancel* del parametro *options*, la connessione lato device viene chiusa e la proprietà *app.sync.enabled* della sessione locale viene disattivata per evitare inutili tentativi di riconnessione. In questo caso anche la sessione proxy viene terminata immediatamente.

Si noti che viene aspettato il completamento dell'evento *onConnect* prima di dare il consenso al device, quindi è necessario che il codice interno all'evento non richieda troppo tempo per essere eseguito. In caso contrario, può scattare il timeout di connessione della sessione locale che per default è 5000 ms, valore modificabile tramite la proprietà *app.sync.connectionTimeout*.

Se si verifica un timeout di connessione mentre la connessione al server è già attiva, nella sessione locale viene resettata la proprietà *app.sync.enabled*. Infatti, il verificarsi del timeout è determinato dal fatto che l'evento *onConnect* lato server è stato troppo lento e questo può essere segno di un sovraccarico. Tentando di nuovo la connessione, il sovraccarico potrebbe aumentare.

La stessa cosa avviene se l'evento *onConnect* lato server genera un'eccezione. Anche in questo caso è bene che la sessione locale non tenti di nuovo la connessione in maniera automatica.

In ogni caso, al termine del tentativo di connessione, la sincronizzazione notifica l'evento *app.sync.onConnect* anche alla sessione locale passando come parametri lo stato della connessione e l'eventuale motivo della mancata connessione da parte del server.

Vediamo un esempio di codice di *onConnect* nelle sessioni online.

```
app.sync.onConnect = function (options)
{
  // Solo online!
  if (!app.runsLocally()) {
    app.sync.dataStore = "ToBuyDB";
    // Carico l'utente in base ai dati passati come topics
    if (app.sync.topics)
      app.account = yield App.TBBE.Account.loadByKey(app, app.sync.topics);
    //
    if (!app.account) {
      options.cancel = true;
      options.cancelReason = "User not found";
    }
  }
};
```

### Chiusura della connessione

Dopo che la connessione è stata instaurata, essa viene gestita in automatico dal sistema di sincronizzazione. Ci sono alcune situazioni in cui la connessione viene chiusa:

- Mancanza di segnale di rete del device.
- Applicazione in background.
- Device bloccato.
- *app.sync.enabled* impostata a false.
- se *autoConnect* non è “automatic”, dopo il timeout di inutilizzo della sincronizzazione, che per default è 30 secondi.

In tutti questi casi, viene lanciato l’evento *onDisconnect* sia nella sessione locale che in quella online e poi quest’ultima viene terminata. Se il device effettua un altro tentativo di connessione, esso avverrà su una sessione proxy diversa dalla precedente.

L’evento *onDisconnect* della sessione locale può essere utilizzato per avvertire l’utente che la sincronizzazione non è disponibile. Questo può essere utile soprattutto se l’applicazione non ha dati locali ma si basa su chiamate remote al server.

# Scambio di messaggi in tempo reale

Dopo aver visto come connettere le varie sessioni al sistema, illustriamo il comportamento di base della sincronizzazione: lo scambio di messaggi tra sessioni.

Per mostrare i vari passaggi, utilizzeremo il seguente caso: si hanno un'applicazione che ammette una chiamata REST per acquisire lo stato del segnale di allarme di una centralina, e delle sessioni di visualizzazione che mostrano lo stato di una specifica centralina o di un insieme di centraline collocate nel medesimo sito.

In questo caso, la sessione che gestisce la chiamata REST deve avvisare del cambiamento le sessioni di visualizzazione. Questo deve avvenire solo per le sessioni che visualizzano lo stato della centralina in cui è avvenuto l’evento.

## Inviare e ricevere messaggi

Per inviare un messaggio è sufficiente utilizzare il metodo *sendMessage* dell’oggetto *app.sync*. Ad esempio la sessione REST che riceve i dati dell’allarme potrebbe inviarli alle altre sessioni tramite questa riga di codice:

```
app.sync.sendMessage({
  topics:”1212”,
  body:{sensor:”W1”, status:”ON”},
  ttl:0
});
```

Il primo parametro sono i *topics* del messaggio, che spiegheremo nel paragrafo successivo, il secondo è il contenuto (*body*) del messaggio e l’ultimo (*ttl*) è il tempo di vita del messaggio.

Come contenuto del messaggio possiamo usare un qualunque oggetto JavaScript, mentre il tempo di vita è un numero intero che rappresenta il numero di secondi per cui il messaggio deve essere memorizzato nel caso non sia possibile consegnarlo subito alle sessioni offline. Se è pari a 0 o viene omesso, il messaggio viene inviato solo alle sessioni attualmente collegate.

Il messaggio viene notificato tramite l'evento *app.sync.onMessage*, che ne contiene i dati. Per intercettare il messaggio ed aggiornare l'interfaccia utente si può usare la seguente riga di codice:

```
app.sync.onMessage = function (message)
{
  App.Pages.postMessage(app, message);
};
```

Come abbiamo visto nel capitolo relativo al framework IonicUI, il metodo *postMessage* invia un messaggio alla videata attiva del page controller. A questo punto la videata può intercettare l'evento *onMessage* ed aggiornare gli elementi visuali di conseguenza. Si ricorda che impostando *message.bc* = *true*, il messaggio verrà consegnato a tutte le videate aperte e non solo a quella attiva.

## I topics

Il sistema di sincronizzazione non deve inviare tutti i messaggi ad ogni sessione collegata, ma occorre effettuare una selezione. Prima di poter scambiare i messaggi, è necessario comprendere come avviene il filtro del sistema di sincronizzazione, in modo da poter inviare ad ogni sessione solo i messaggi che la riguardano.

Per poter effettuare questa selezione, è necessario familiarizzare con il concetto di *topics* di un messaggio e di *topics* della sessione.

I *topics* di un messaggio rappresentano i suoi argomenti e sono rappresentati come oggetto JavaScript. I *topics* della sessione sono gli argomenti a cui una determinata sessione dichiara di essere interessata, anch'essi espressi come oggetto JavaScript.

In questo modo un messaggio viene inviato ad una sessione solo se i *topics* del messaggio sono compatibili con i *topics* della sessione.

Siccome i *topics* sono oggetti JavaScript, essi possono essere composti a più livelli, sia tramite array, che con valori singoli.

Nell'esempio delle centraline, i *topics* del messaggio di cambio stato dell'allarme devono contenere almeno il codice della centralina. Ogni sessione può dichiarare a quale centralina è interessata usando il suo codice come *topics* della sessione di sincronizzazione. Se quindi i *topics* del messaggio sono *"1212",* i topics delle sessioni che intercettano il messaggio possono essere definiti come segue:

```
app.sync.topics = "1212";
```

Siccome le centraline sono raggruppate in diversi siti, una sessione può visualizzare lo stato di tutte le centraline di un sito. Per farlo essa può impostare come *topics* della sessione un array che contiene tutti i codici delle centraline del sito, come mostrato nella riga di codice seguente:

```
app.sync.topics = [“1212”, “1225”, “1201”, ...];
```

Una soluzione alternativa è quella di usare come *topics* del messaggio sia il codice della centralina che quello del sito, ad esempio:

```
app.sync.sendMessage({topics:{box:”1212”, site:”east”}, ...});
```

In questo modo le sessioni di visualizzazione potrebbero “sottoscrivere” uno o più “canali” di interesse: alcune potrebbero essere interessate ai box, altre ai siti. In questo caso una sessione deve definire a cosa è interessata, inserendo un asterisco nelle proprietà rimanenti. Vediamo alcuni esempi:

```
// Mi interessa solo una centralina 1212, senza considerare il sito
app.sync.topics = {box:”1212”, site:”*”};

// Mi interessano tutte le centraline di un sito
app.sync.topics = {box:”*”, site:”east”};

// Mi interessano varie centraline (anche di siti diversi)
app.sync.topics = {box:[”1212”,“2519”], site:”*”};

// Mi interessano varie centraline, di un unico sito
app.sync.topics = {box:[”1212”,“1201”], site:”east”};

// Mi interessano più siti
app.sync.topics = {box:”*”, site:[”east”, “west”]};
```

È possibile che una sessione sia interessata a tutti i messaggi; in tal caso può usare come *topics* il carattere asterisco. In generale questo non deve avvenire perché inviare tutti i messaggi a tutte le sessioni può peggiorare le performance e minare la privacy del sistema.

## Topics come canali di interesse

L’uso di un oggetto JavaScript con diverse proprietà come *topics* dei messaggi e delle sessioni può essere interpretato come la dichiarazione dei possibili “canali” a cui le varie sessioni possono dichiarare di essere interessate.

In un sistema informativo complesso, infatti, si potrebbero scambiare messaggi relativi a diverse entità, anche completamente indipendenti. Ad esempio, oltre al sistema degli allarmi, nello stesso sistema informativo può essere presente lo scambio di messaggi relativi al decollo o all’atterraggio di aerei negli aeroporti.

In questo caso si possono definire due canali di interesse: le centraline e i voli. Per quanto riguarda le centraline, i dati importanti sono il codice e il sito, mentre per i voli il codice dell'aereo e quello dell'aeroporto.

Se si invia un messaggio relativo ad una centralina verranno usati i seguenti *topics*:

```
app.sync.sendMessage({topics:{alarm:{box:"1212", site:"east"}, ...});
```

Mentre per un volo:

```
app.sync.sendMessage({topics:{flight:{plane:"UK1234", airport:"BLQ"},...});
```

Le sessioni interessate agli allarmi sottoscriveranno solo il canale allarmi. Ad esempio:

```
app.sync.topics = {alarm:{box:"*", site:["east", "west"]}};
```

Mentre quelle interessate agli aeroporti sottoscriveranno solo il canale voli. Ad esempio:

```
app.sync.topics = {flight:{plane:"*", airport:"BLQ"}};
```

Se una sessione fosse interessata sia ad aeroporti che ad allarmi potrebbe avere come *topics* quanto segue:

```
app.sync.topics = {flight:{plane:"*", airport:"BLQ"},
                   alarm:{box:"*", site:"east"}};
```

## *Topics* delle sessioni locali

Le sessioni locali definiscono i *topics* come tutte le sessioni nel cloud, tuttavia esse entrano nel sistema di sincronizzazione per mezzo della sessione proxy ad esse collegata. Quando una sessione locale apre la connessione, essa passa i suoi *topics* alla sessione proxy, che può cambiarli a suo piacimento. La sessione locale quindi riceverà i messaggi relativi ai *topics* della sessione proxy e non ai suoi.

Questo funzionamento serve come validazione dei *topics* della sessione locale rispetto al sistema cloud. Se ad esempio viene aperta una connessione che utilizza come *topics* l'ID dell'utente della sessione locale, può essere utile modificare i *topics* della sessione proxy in funzione dei dati che tale utente deve vedere. Nell'evento *onConnect* della sessione proxy, dopo aver verificato che l'utente sia valido, si possono modificare i *topics* in modo da inviare alla sessione locale non solo i messaggi che riguardano direttamente l'utente, ma anche quelli che riguardano altre entità che l'utente può vedere.

Immaginiamo, ad esempio di voler gestire un sistema di messaggistica in cui è possibile inviare messaggi fra due utenti oppure agli utenti di un gruppo, come avviene ad esempio in WhatsApp.

Il messaggio tra due utenti può essere inviato con i seguenti *topics*:

```
app.sync.sendMessage({topics:{sender:"ute0", receiver:"ute1"}, ...});
```

Mentre il messaggio relativo inviato in gruppo potrebbe avere i seguenti *topics*:

```
app.sync.sendMessage({topics:{sender:"ute0", receiver:"grp3"}, ...});
```

Quando una sessione locale apre la connessione, essa si può presentare con i seguenti *topics*:

```
app.sync.topics = "ute0";
```

A questo punto, in funzione dei gruppi a cui l'utente è iscritto, la sessione proxy può modificare i *topics* come segue:

```
app.sync.topics = [
  {sender:"ute0", receiver:"*"}, // messaggi inviati da me
  {sender:"*", receiver:"ute0"}, // messaggi inviati a me
  {sender:"*", receiver:"grp3"}, // messaggi inviati al gruppo 3
  {sender:"*", receiver:"grp555"}, // messaggi inviati al gruppo 555
  …
};
```

In questo modo la sessione locale riceve i messaggi inviati dall'utente, quelli inviati all'utente e quelli inviati a tutti i gruppi a cui l'utente è iscritto.

### Variazione dei topics durante una sessione

Abbiamo visto che i *topics* di una sessione locale vengono inviati alla corrispondente proxy all'apertura in modo che l'evento *onConnect* possa gestirli.

È possibile che, in seguito, la sessione locale voglia modificare i suoi *topics* senza dover chiudere e riaprire la connessione.

Per ottenere questo risultato la sessione locale deve modificare i suoi *topics* e poi usare il metodo *app.sync.notifyTopicsChanged* che invia i *topics* alla sessione proxy senza resettare la connessione. Nella sessione proxy viene notificato l'evento *onTopicsChanged* in modo che essa possa gestire la variazione.

## I messaggi permanenti

Abbiamo visto che il terzo parametro della funzione *sendMessage* rappresenta il tempo, misurato in secondi, per cui il messaggio rimane ricevibile da una sessione locale che si collega dopo che il messaggio è stato inviato.

Al contrario, le sessioni collegate direttamente al server nel cloud (sessioni online) non ricevono i messaggi inviati prima dell'inizio della sessione.

Questa differenza nasce dal fatto che una sessione online all'avvio può analizzare l'intero stato del sistema leggendo direttamente il database nel cloud. Inoltre, essendo in esecuzione nel server, per definizione essa rimane connessa alla sincronizzazione fino a

che non termina. Ecco perché nelle sessioni online sono importanti solo i messaggi che arrivano dopo l'avvio.

Invece per una sessione locale è più difficile leggere l'intero stato del sistema perché essa non è connessa al database nel cloud. Inoltre, mentre rimane attiva, la sessione locale può connettersi e disconnettersi dalla sincronizzazione varie volte a causa dello stato della rete cellulare. Per le sessioni locali è quindi essenziale ricevere i messaggi inviati anche quando erano disconnesse, altrimenti potrebbero perderli senza venirne a conoscenza. Da qui la necessità dei cosiddetti *messaggi permanenti*.

### Gestione dei messaggi permanenti

I messaggi permanenti vengono memorizzati in una tabella del database indicato al sistema di sincronizzazione tramite la proprietà *app.sync.dataStore*. La tabella, di nome *z_sync*, viene creata e gestita automaticamente dal sistema.

La tabella *z_sync* viene creata sia nei device che nel database nel cloud. Nei device essa serve per memorizzare i messaggi inviati tramite il dispositivo mentre questo è offline. Essi verranno consegnati all'apertura della connessione e poi cancellati dal database.

Nel cloud, la tabella *z_sync* memorizza i messaggi permanenti di tutte le sessioni non appena essi vengono consegnati, cancellandoli solo quando diventano più vecchi del timeout. In questo modo, quando una sessione locale si connette, la tabella *z_sync* nel cloud viene scansionata per trovare i messaggi a cui la sessione è interessata che devono ancora essere consegnati.

### Messaggi permanenti: quando utilizzarli?

Finora abbiamo visto come configurare ed inviare messaggi di base tramite il sistema di sincronizzazione. Nei paragrafi seguenti vedremo come applicare la sincronizzazione alla gestione dei documenti, in cui i messaggi scambiati non sono più semplici oggetti JavaScript, ma documenti completi.

Non è quindi consigliabile utilizzare la sincronizzazione di base per impostare il proprio sistema di scambio messaggi; il metodo migliore è sempre quello di utilizzare la sincronizzazione applicata ai documenti come illustrato nel paragrafo Document Orientation Remota.

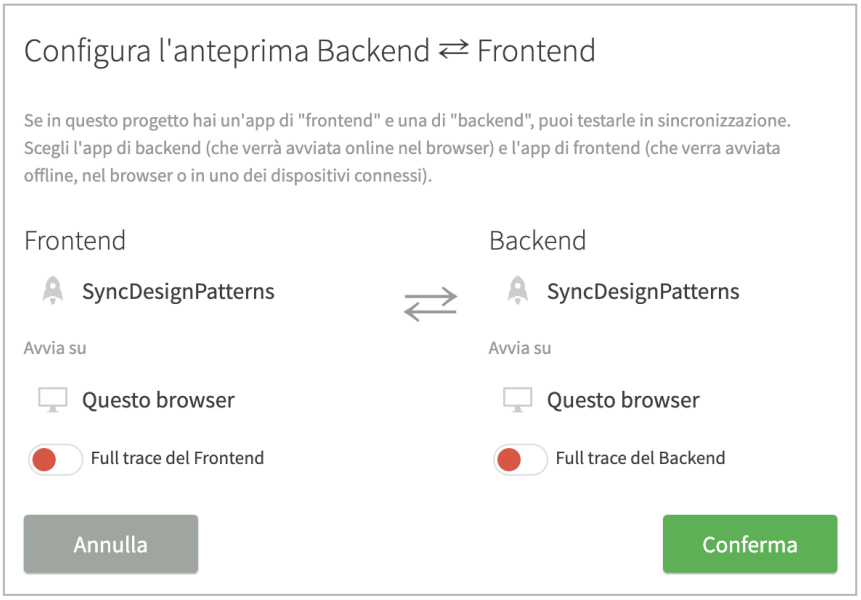
Per questa ragione la gestione dei messaggi permanenti viene disabilitata per default al livello dei messaggi di base. Se si desidera utilizzare i messaggi permanenti di base, è necessario impostare *options.skip = false* nell'evento *onConnect*.

## Test della sincronizzazione

Una volta completate la configurazione e le funzioni di scambio messaggi, il sistema di sincronizzazione deve essere testato. L'IDE possiede una modalità di esecuzione delle applicazioni chiamata *Anteprima Back End - Front End*, abbreviata come FEBE, in grado di

eseguire contemporaneamente due sessioni: una in modalità locale e la seconda in modalità online.

La modalità FEBE si attiva tramite la freccia posta sulla destra del pulsante di anteprima. Una volta attivata, vengono mostrate le opzioni di configurazione, come mostrato nell'immagine seguente:




In questa videata è possibile scegliere l'applicazione del progetto che deve essere usata come front end (sessione locale) e come back end (sessione online). È possibile usare anche la stessa applicazione in entrambi i casi.

È inoltre possibile scegliere se le sessioni devono essere avviate nel browser o in un dispositivo connesso all'IDE tramite l'applicazione InstaLauncher, ed infine se deve essere attivata la modalità di debug *full trace* o meno.

Confermando le opzioni, le sessioni verranno attivate. Se si desidera modificare le opzioni, occorre disattivare e riattivare la modalità FEBE.

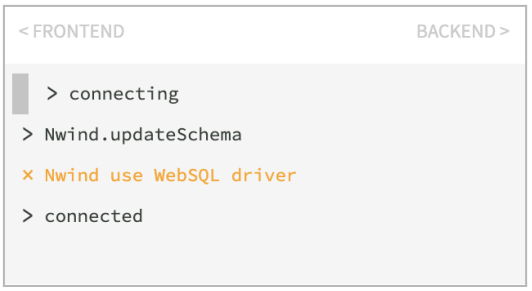
Quando l'applicazione viene lanciata in FEBE, la proprietà *app.sync.serverURL* della sessione locale viene preimpostata dal framework in modo che si possa connettere alla sessione online corrispondente. Per questa ragione, tale proprietà dovrebbe essere impostata da codice solo se viene trovata vuota.

Si noti inoltre che la sessione online funge sia da sessione proxy per la sessione locale che da sessione browser per la visualizzazione dei dati del cloud. Per tale sessione, quindi, vengono notificati sia gli eventi *onStart* che *onConnect*.

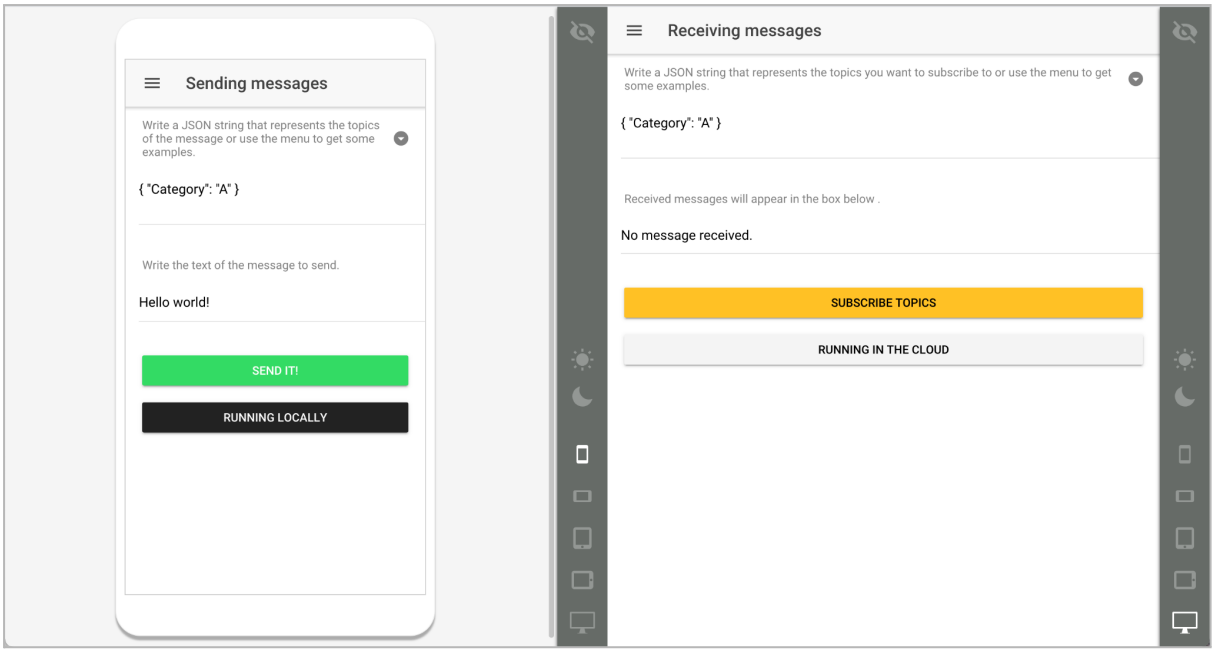
Il vantaggio di questa soluzione consiste nel fatto che la sessione browser/proxy può mostrare a video più facilmente gli effetti delle operazioni di sincronizzazione che avvengono tramite la sessione locale.

Occorre tenere presente, tuttavia, le possibili interazioni nell'avvio dei due tipi di sessioni. Ad esempio, se la sessione browser imposta una proprietà di sessione come *app.loggedUser*, e tale proprietà viene usata nell'evento *onConnect* senza impostarla anche in quel caso, è possibile che l'applicazione funzioni correttamente in modalità FEBE ma fallisca quando viene installata.

Quando l'applicazione è in funzione in modalità FEBE, possiamo notare che la barra della console dell'IDE mostra contemporaneamente i messaggi delle due sessioni. Quella locale (front end) sulla sinistra, quella online (back end) spostata a destra. Nell'immagine seguente il messaggio "connecting" proviene dal back end, mentre gli altri dal front end.




Come esempio del sistema di invio e ricezione messaggi è possibile aprire il progetto [sync-design-patterns] e lanciarlo in anteprima FEBE. Per aprire la videata di test del sistema di scambio messaggi, aprire il menu della sessione locale e scegliere la voce *Sending Messages*.




A questo punto è possibile selezionare i *topics* del messaggio sulla destra e quelli della sessione di sincronizzazione sulla sinistra. Poi occorre premere il pulsante *Subscribe Topics* per aggiornare la sessione di sincronizzazione online ed infine provare ad inviare il messaggio dalla sessione locale a quella online. Se il messaggio viene ricevuto, verrà visualizzato nel campo apposito della sessione online, evidenziato tramite un'animazione.

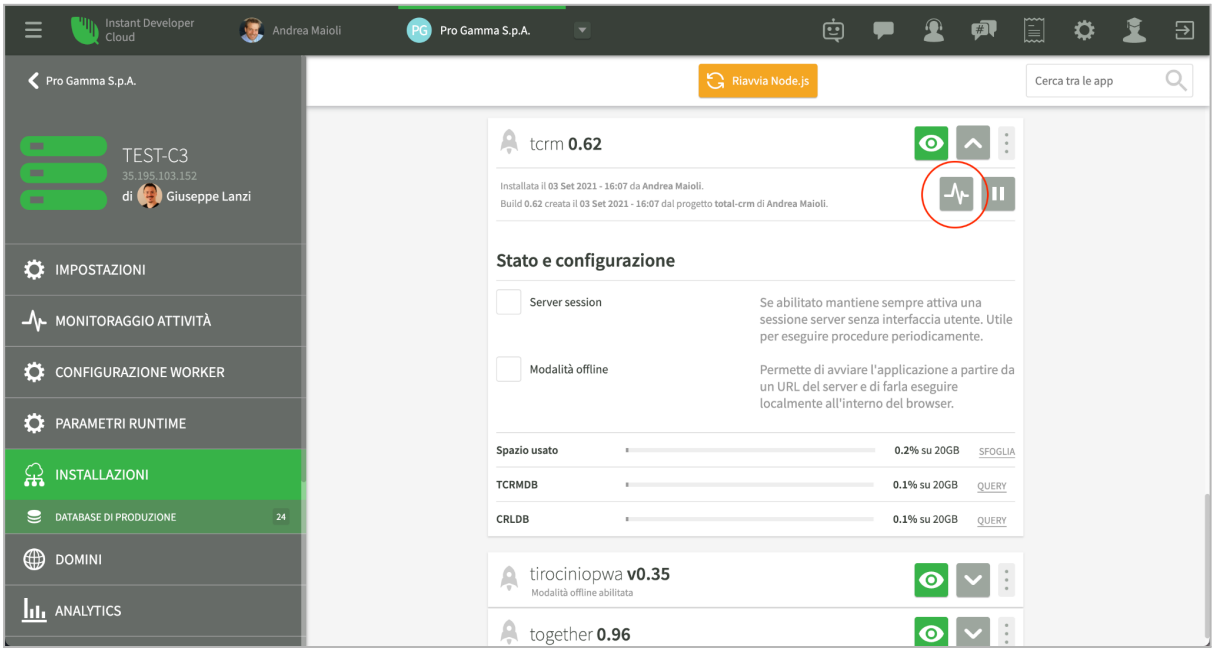
Si possono fare vari esperimenti provando a variare i *topics* del messaggio e quelli della sessione di sincronizzazione.

## Test con applicazione in produzione

Dopo aver testato la sincronizzazione tramite FEBE, è importante provare il funzionamento nell'ambiente di produzione. La modalità di test consigliata è la seguente:

1) Attivare il log strutturato delle sessioni impostando la proprietà *app.sessionName* negli eventi *onStart*, *onConnect* e *onCommand*.
2) Installare in un server di produzione l'applicazione di back end attivando l'opzione di debug a runtime durante la configurazione dell'installazione.
3) Eseguire in anteprima offline nell'IDE l'applicazione di front end. Questa modalità crea una sessione locale nel browser che si connette al back end di sincronizzazione installato nell'ambiente di produzione.

Se vengono evidenziati problemi nel back end non rilevati in modalità FEBE, si consiglia di aprire il debug a runtime della sessione proxy a cui la sessione locale in anteprima nell'IDE si è connessa. Tale sessione viene listata nella pagina *log strutturato* dell'installazione nel server di produzione. Nell'immagine seguente viene evidenziato il pulsante per accedere alla pagina *log strutturato* di un'installazione tramite la console di Instant Developer Cloud.




SI ricorda che per attivare il log strutturato di un'installazione occorre dare un nome alle sessioni che si desidera tracciare impostando da codice la proprietà *app.sessionName*. Inoltre occorre attivare la raccolta dei dati di log tramite gli appositi comandi nella pagina del *log strutturato* della propria installazione.

L'ultima fase di test della sincronizzazione consiste nell'utilizzare il back end nell'ambiente di produzione e l'applicazione di front end installata su un dispositivo, anche tramite

InstaLauncher. Se in questa fase si verificano anomalie non rilevate dalle fasi di test precedenti, si consiglia di utilizzare il debug a runtime della sessione proxy insieme alla raccolta dati di analitica o inviando messaggi di log dal device di front end alla sessione proxy tramite il servizio di log illustrato nel paragrafo successivo.

## Usare il sistema di log della sincronizzazione

Il sistema di sincronizzazione di Instant Developer Cloud include alcuni servizi aggiuntivi fra i quali un servizio che permette di inviare messaggi di log da una sessione locale alla corrispondente proxy. Il servizio viene utilizzato nella sessione locale tramite i seguenti metodi:

```
app.sync.log.log(...);
app.sync.log.warn(...);
app.sync.log.error(...);
```

Nella sessione proxy viene notificato l'evento *onClientMessage* in cui è possibile gestire il messaggio, ad esempio inserendolo in un database o inviandolo ad una applicazione di visualizzazione. Se questo evento non viene gestito, il messaggio appare nel log della sessione proxy ed è quindi leggibile tramite il log strutturato o tramite i log del server.

# Document Orientation Remota

Finora abbiamo visto come la sincronizzazione permette la comunicazione fra sessioni online e locali tramite scambio di messaggi costituiti da oggetti JavaScript.

Tuttavia le applicazioni utilizzano i documenti per le operazioni di trattamento dati, non semplici oggetti JavaScript: per questo motivo la sincronizzazione è stata integrata con il framework di gestione dei documenti.

Gli obiettivi di questa integrazione sono tre:

1) Consentire alle sessioni locali di accedere ai documenti nel cloud come se fossero locali, al fine di semplificare al massimo l'architettura client-cloud.
2) Consentire alle sessioni locali di sincronizzare automaticamente i documenti contenuti nel database offline del dispositivo.
3) Consentire alle sessioni locali e online di reagire in tempo reale alle modifiche confermate di un documento.

In questo paragrafo verrà illustrato il funzionamento della Document Orientation remota, corrispondente al punto 1) dell'elenco precedente. Nel paragrafo successivo, i punti seguenti. Se non si ha familiarità con i concetti relativi alla Document Orientation, si consiglia di leggere il capitolo relativo.

# Leggere documenti dal back end

L'utilizzo della Document Orientation per gestire i documenti del cloud nelle sessioni locali richiede la configurazione a design time dei documenti e dei metodi di accesso.

A livello di progetto i documenti devono essere contenuti in un'apposita libreria così da poter essere condivisi sia dall'applicazione locale che da quella di back end. I documenti, inoltre, devono essere esplicitamente resi remotizzabili attivando la proprietà di design time *Permetti chiamate remote*.

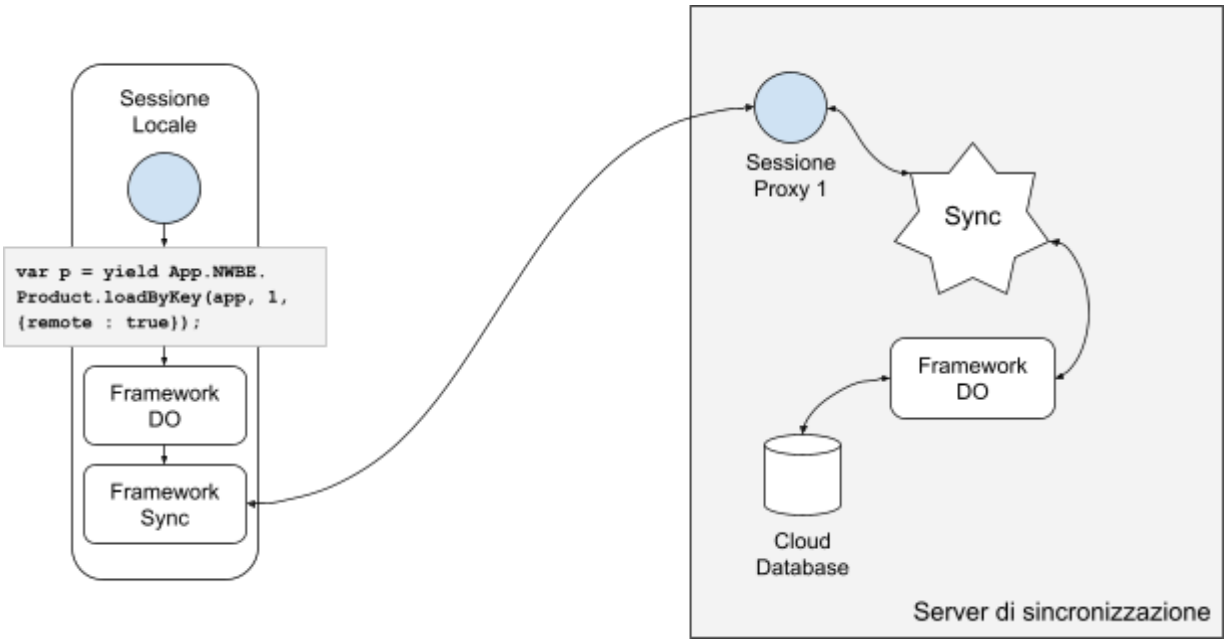
A questo punto la sessione locale può utilizzare i metodi di caricamento e salvataggio del documento attivando l'opzione di remotizzazione. In questo caso l'operazione non verrà effettuata rispetto al database del dispositivo, ma inviando la richiesta alla sessione proxy tramite il canale di sincronizzazione, che a sua volta esegue il comando rispetto al database nel cloud.

La riga di codice seguente carica il documento *Product* con *ID=1* dal cloud tramite il canale di sincronizzazione.

```
var p = yield App.NWBE.Product.loadByKey(app, 1, {remote : true});
```

L'unica differenza è costituita dall'opzione di remotizzazione che è stata attivata.

Le operazioni remotizzate richiedono che nella sessione locale la sincronizzazione sia stata configurata ed attivata. Se al momento dell'invocazione del comando la connessione non è ancora aperta, ne viene gestita l'apertura prima di procedere con l'esecuzione del comando. Se non è possibile aprire la connessione o essa viene rifiutata, il comando restituisce *null*, come nel caso in cui il documento non fosse presente nel database; inoltre viene visualizzato un messaggio di warning nel log della sessione.



Il diagramma precedente mostra i passaggi di esecuzione del comando di caricamento remotizzato:

1) L'applicazione invia il comando di caricamento al documento *Product*.
2) Il documento passa il comando al framework di gestione dei documenti.
3) Essendo stata impostata l'opzione di remotizzazione, la query non può essere eseguita nel database locale, ma viene inviato un messaggio alla sessione proxy tramite il canale di sincronizzazione.
4) La sessione proxy riceve il messaggio e lo gestisce richiedendo al framework DO di caricare il documento richiesto.
5) Il risultato viene restituito alla sessione locale tramite il canale di sincronizzazione.
6) Il framework DO della sessione locale riceve il risultato del comando e lo passa al codice della sessione, in modo trasparente rispetto alla modalità di esecuzione che questa volta è remota invece che locale.

Lo stesso schema di funzionamento avviene per le altre chiamate di caricamento come *loadCollection, getRelated, collection.load* eccetera. Si noti che gli eventi *beforeLoad* e *AfterLoad* vengono eseguiti sia nella sessione locale che nel back end.

Si segnala infine che i metodi di caricamento permettono anche la ricezione di documenti multi-livello impostando la proprietà *childLevel* del parametro *options*.

Per vedere un esempio funzionante di Document Orientation remota è possibile aprire il progetto sync-design-patterns e lanciarlo in anteprima FEBE. Per aprire la videata di test aprire il menu della sessione locale e scegliere la voce *Remote Documents*.

## Scrivere documenti nel back end

Come per le funzioni di caricamento, la remotizzazione delle funzioni di salvataggio di documenti e collection si attiva semplicemente attivando la proprietà *remote* nelle opzioni del metodo. L'esempio di codice seguente legge un documento dal cloud, cambia il valore di una proprietà e poi lo salva nel cloud.

```
var p = yield App.NWBE.Product.loadByKey(app, 1, {remote : true});
p.UnitsOnOrder++;
var b = yield p.save({remote : true});
```

Il principio di funzionamento del salvataggio è il seguente: in primo luogo il documento viene validato localmente, poi, se non ci sono errori, l'intera struttura viene inviata alla sessione proxy tramite la sincronizzazione e poi nuovamente validata e salvata nel cloud.

Se nella sessione proxy vengono apportate delle modifiche alle proprietà del documento, esse vengono restituite alla sessione locale che le applica all'istanza in memoria del documento di cui è stato richiesto il salvataggio.

Se, ad esempio, nell'evento *onSave* notificato nella sessione proxy viene calcolato un progressivo del documento, al termine dell'operazione di salvataggio questo valore è presente anche nell'istanza del documento della sessione locale.

In modo analogo, anche gli errori segnalati sul documento dalla sessione proxy verranno resi disponibili nell'istanza del documento della sessione locale.

Siccome il codice operativo del documento è lo stesso eseguito nella sessione locale e in quella proxy, è possibile che alcune operazioni vengano eseguite solo quando ci si trova lato cloud e non nel dispositivo. Per distinguere i due casi è possibile condizionare l'esecuzione del codice al valore restituito dal metodo *app.runsLocally()*.

Si segnala infine che, per ragioni di performance e di coerenza dei dati, in caso di documenti multi-livello è preferibile remotizzare il salvataggio dell'intera struttura in un'unica operazione di salvataggio piuttosto che con diverse operazioni separate.

## Chiamate remote

Oltre che leggere o scrivere istanze di documenti, la Document Orientation Remota consente di configurare metodi di documento per essere invocati sia localmente che remotamente, in modo da renderne trasparente l'utilizzo.

Per far sì che un metodo sia remotizzabile, occorre attivare a design time l'opzione *permetti chiamate remote*. Per renderlo eseguibile sia in modalità locale che remota, bisogna inserire alcune righe di codice all'inizio del metodo, come mostrato nel template di codice seguente:

```
App.<library>.<document-class>.prototype.<method-name> =
                                                  function (<par1>, <par2>)
{
  if (app.runsLocally()) {
    return yield this.rfc("<method-name>", [<par1>, <par2>]);
  }
  //
  ... method code (online)...
}
```

Un metodo di istanza remotizzabile, quando viene chiamato localmente invoca l'esecuzione remota tramite *this.rfc*, che ammette tre parametri: il primo è il nome del metodo stesso, il secondo è un array di parametri (quelli passati al metodo) ed il terzo è un oggetto di opzioni, che può essere omesso.

Quando il metodo viene chiamato da una sessione locale, esso esegue solamente la chiamata *rfc* e ne restituisce il risultato. A sua volta, il metodo *rfc* chiamato sull'istanza di documento (*this*) utilizza il canale della sincronizzazione per inviare un messaggio alla sessione proxy per richiedere l'esecuzione del metodo su un'istanza di documento corrispondente a quella originaria.

Per questa ragione anche i dati del documento dovranno essere trasferiti alla sessione proxy, in modo che essa possa ricostruire in memoria l'istanza stessa per poi poterne chiamare il metodo, passando il valore dei parametri.

A questo punto verrà richiamato lo stesso codice del metodo, ma nella sessione proxy *app.runsLocally* è *false*, quindi verrà saltata la chiamata a *rfc* e verrà eseguito il codice vero e proprio del metodo. Il valore di ritorno viene poi comunicato dalla sessione proxy a quella locale tramite il canale di sincronizzazione ed infine tale valore viene restituito dal metodo locale al chiamante.

Se avvengono delle eccezioni all'interno della sessione proxy durante la chiamata del metodo, la medesima eccezione verrà generata dal metodo *rfc* chiamato nella sessione locale così che anche in questo caso il metodo si comporti allo stesso modo quando viene chiamato localmente o remotamente.

Si segnala che l'unica opzione supportata da *rfc* è la proprietà *loadRemotely,* attivabile passando *{ loadRemotely:true}* come oggetto di opzioni. In questo caso non vengono passati tutti i dati dell'oggetto alla sessione proxy, ma solo la chiave primaria, in modo che lato cloud il documento venga istanziato caricandolo dal database invece che rigenerato dai dati inviati dalla sessione locale. Questo comportamento può essere utile in due casi:

1) Se i dati del documento sono corposi e si preferisce ricaricarli invece che inviarli.
2) Se si desidera operare sui dati aggiornati presenti nel database del cloud invece che su quelli provenienti dalla sessione locale.

## Chiamate remote a metodi statici

Oltre che poter chiamare remotamente i metodi di istanza, è possibile operare in modo analogo sui metodi statici. In questo caso cambia il template di codice da inserire per rendere trasparente la chiamata del metodo e il fatto che non c'è bisogno di comunicare al proxy i dati del documento, perché non si sta lavorando su un'istanza ma sulla classe stessa.

Per i metodi statici è possibile utilizzare il seguente template di codice. Come si può notare, il metodo *rfc* ha una versione statica che permette la remotizzazione delle chiamate di questo tipo.

```
App.<library>.<document-class>.<method-name> = function (<par1>, <par2>)
{
  if (app.runsLocally()) {
    return yield App.<library>.<document-class>.rfc(app,
                                      "<method-name>", [<par1>, <par2>]);
  }
  //
  ... method code (online)...
}
```

## Tipi gestibili tramite chiamate remote

La chiamata remota avviene passando i parametri e ricevendo il risultato dal metodo tramite il sistema di sincronizzazione. Questo implica che i parametri e il risultato devono essere serializzabili in formato stringa in modo da poter essere passati via socket.

I tipi di parametri che ammettono questo comportamento sono: i tipi base (stringhe, numeri, date, oggetti JavaScript, ecc.), le istanze di documento, di collection e di datamap. Se si devono passare altri tipi di dati ad un metodo remotizzabile, si consiglia di convertirli in stringa o in oggetto per poterli ricostruire nella sessione proxy.

### Test dei metodi remotizzati

Il test delle chiamate ai metodi remoti avviene nello stesso modo delle altre funzioni della sincronizzazione, cominciando dalla modalità FEBE. Un esempio di chiamate remote è presente nel progetto sync-design-patterns, avviando l'anteprima FEBE e scegliendo la voce *Remote Documents* nel menu della sessione locale.

## Remotizzazione dell'intera applicazione

Quando si sviluppa un'applicazione omnichannel, un possibile percorso di implementazione consiste nel progettare l'applicazione inizialmente in modalità online, cioè funzionante tramite browser collegati direttamente al cloud.

Quando l'applicazione è pronta per essere installata nei dispositivi, si deve procedere ad attivare i metodi di sincronizzazione in modo che essa possa funzionare anche con sessioni locali e non solo via browser. È in questo momento che si inizia ad utilizzare la sincronizzazione.

La modalità di utilizzo della sincronizzazione dipende dalle specifiche dell'applicazione, cioè se essa deve funzionare anche in modalità offline, senza connessione ad internet, oppure solo quando la connessione è presente. Nel primo caso è necessario avere i dati in un database locale ed utilizzare la sincronizzazione per allineare i dati del cloud con quelli locali. Questo argomento verrà illustrato in un paragrafo successivo.

Se invece si prevede che la connessione sia sempre presente durante l'utilizzo dell'applicazione, non è necessario configurare l'applicazione per avere una replica locale dei dati, ma è sufficiente automatizzare la remotizzazione delle chiamate ai documenti quando l'applicazione è in esecuzione nei dispositivi.

I passaggi per ottenere questo risultato sono i seguenti:

1) Configurare la sincronizzazione.
2) Configurare la Document Orientation Remota.
3) Rendere remoti i metodi chiamati nella sessione locale, aggiungendo il codice di remotizzazione.
4) Non utilizzare direttamente query sul database se non nel codice online dei metodi remotizzabili. Per l'accesso ai dati usare sempre i metodi dei documenti.
5) Automatizzare la remotizzazione delle chiamate ai documenti attivando la proprietà *remote* del framework quando la sessione è locale.

Impostando a *true* la proprietà *App.Document.remote*, si ottiene che tutte le chiamate di caricamento e salvataggio saranno automaticamente remotizzate, senza dover aggiungere l'opzione relativa nelle chiamate a metodo.

In questo modo è possibile centralizzare in un unico punto l'attivazione della remotizzazione dei documenti; di solito si utilizza il momento del login, dove si hanno disponibili le credenziali dell'utente. Il codice da usare è simile al seguente:

```
$btnLogin.onClick = function(event) {
  if (app.runsLocally()) {
    app.sync.topics = {user: $fldUser.value, pwd: $fldPwd.value}
    app.sync.enabled = true;
    App.Document.remote = true;
  }
  let u = App.BE.User.checkUser($fldUser.value, $fldPwd.value);
  if (u) {
    // ok, procediamo
  }
  else {
    // utente non trovato
  }
}
```

Dove *BE.User.checkUser* è un metodo remotizzabile che restituisce il documento utente date le credenziali dell'utente richieste dalla videata di login, oppure *null* se i dati non corrispondono.

Si segnala che la proprietà *remote* è disponibile anche per una specifica classe di documento oppure per una specifica istanza.

## Gestire le eccezioni

Quando una sessione locale utilizza metodi di documento remotizzati, viene aperta una comunicazione di tipo websocket tra il dispositivo e il back end nel cloud. Siccome i dispositivi utilizzano spesso la rete cellulare, non è detto che si riesca ad instaurare la comunicazione o che la connessione rimanga stabile nel tempo.

Per questa ragione, ogni chiamata remotizzata potrebbe generare eccezioni o restituire valori di errore anche se i dati del database sono corretti. Se un'applicazione utilizza sempre e solo chiamate remote, cioè presuppone che la connessione sia sempre disponibile durante il suo funzionamento, è possibile intercettare l'evento *app.sync.onConnectionStatusChange* dal lato della sessione locale per avvisare l'utente che l'applicazione è utilizzabile o meno a seconda della presenza della connessione.

Un meccanismo di avviso molto semplice ed efficace consiste nell'aprire un *popup* di tipo *loading* se la connessione non risulta attiva, e nascondere tale popup nel momento in cui essa ritorna disponibile. In questo modo l'interfaccia utente dell'applicazione non può essere utilizzata mentre la connessione non è attiva e, di conseguenza, l'utente non potrà fare azioni che richiederebbero l'accesso al server nel cloud.

## Gestire la sicurezza

L'utilizzo del sistema di Document Orientation Remota mette in comunicazione un dispositivo esterno al cloud con il back end. In questi casi è richiesta la verifica delle credenziali del dispositivo per controllare se può accedere e quali dati può vedere.

Per effettuare questa verifica è possibile utilizzare l'evento *app.sync.onConnect* nell'applicazione di back end. Ci sono diverse soluzioni, in funzione dei dati passati all'evento al momento della creazione della connessione. Quella migliore consiste nell'utilizzare le credenziali della sessione locale come *topics* della sincronizzazione, in modo che nell'evento *onConnect* si possa verificare se esse corrispondono ad un utente registrato o meno.

Se l'utente non viene riconosciuto, la connessione deve essere chiusa, altrimenti si può proseguire. Se si deve consentire la lettura solo di determinati dati in funzione dell'utente, durante l'evento *onConnect* è possibile memorizzare il documento che rappresenta l'utente in una proprietà della sessione, che potrà poi essere usata nelle query di caricamento dei documenti per limitare l'accesso.

Un esempio di questa strategia di controllo è visibile nel codice seguente:

```
app.sync.onConnect = function (options)
{
  if (app.sync.topics) {
    if (app.sync.topics.userId) {
      app.account = yield App.TBBE.Account.loadByKey(app,
                                          app.sync.topics.userId);
    }
    if (app.sync.topics.email && app.sync.topics.pwdhash) {
      app.account = yield App.TBBE.Account.loadByKey(app,
          {email:app.sync.topics.email, pwdhash:app.sync.topics.pwdhash);
    }
  }
  //
  if (!app.account) {
    options.cancel = true;
    options.cancelReason = "User not found";
  }
};
```

Per aumentare il grado di sicurezza del sistema è importante ridurre la superficie di attacco, attivando per la remotizzazione il numero minimo di documenti e di metodi. Una buona soluzione in questo senso consiste nella definizione di un documento aggiuntivo, non legato ad una tabella del database, che contiene i metodi necessari ai client per accedere ai dati. A questo punto è possibile rendere remotizzabile solo questo documento e i suoi metodi in modo da concentrare in un unico punto l'accesso esterno ai dati del back end.

# Sincronizzazione del database offline

Affrontiamo ora l'ultimo argomento relativo alla sincronizzazione associata ai documenti: il sistema di aggiornamento automatico del database locale del device.

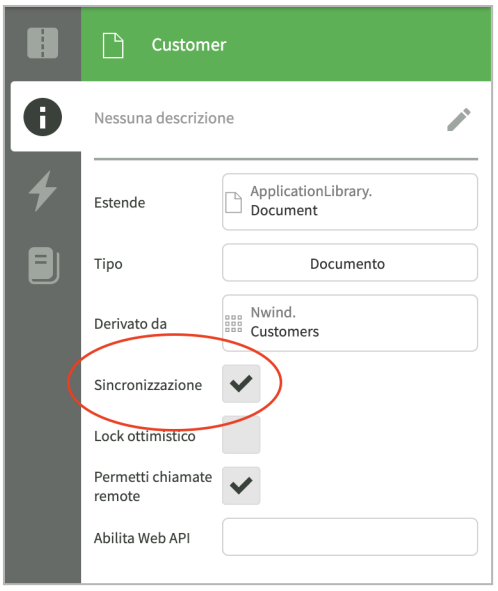
Lo scopo di questa funzione è duplice: da un lato di permettere alle applicazioni mobile di funzionare anche disconnesse da internet. Questa caratteristica può essere anche oggi molto importante perché la qualità delle connessioni è piuttosto variabile da territorio a territorio.

Ma c'è un altro importante motivo di utilizzo di un database locale che si sincronizza con il cloud: *la scalabilità*. Pensiamo infatti ad un'applicazione che utilizza solo web api per funzionare: ad ogni interazione essa deve comunicare con il server nel cloud per reperire nuovi dati o comunicare quelli che l'utente ha inserito. Se il numero dei terminali connessi contemporaneamente è alto, ad esempio nell'ordine del migliaio, il server deve effettuare migliaia di query al secondo. Per ottenere questi risultati sono necessarie architetture cloud complesse e costose.

Utilizzando un database locale si crea un'applicazione in grado di funzionare senza internet, che, quindi, non ha bisogno di interagire con il server ad ogni passaggio. Sarà il servizio di sincronizzazione a garantire, quando la connessione è disponibile, il trasferimento in tempo reale delle modifiche dal database locale al cloud e viceversa. Ciò avviene in background, senza interagire con l'utente, e usando solo una piccola frazione delle risorse cloud rispetto al caso precedente.

## Configurazione

La configurazione della sincronizzazione dei documenti richiede i passaggi visti in precedenza per quella dei messaggi e per l'accesso remoto ai documenti. Inoltre è necessario attivare il servizio di sincronizzazione per i documenti che si desidera utilizzare localmente, impostando il flag relativo nelle proprietà del documento stesso.



Attivando la sincronizzazione per un documento, un flag analogo viene reso disponibile per ogni sua proprietà. Questo flag specifica se la proprietà deve essere sincronizzata o meno ed avrà come valore predefinito *true* per le proprietà che derivano dal database, *false* per quelle derivate o unbound. Se, ad esempio, in un documento sono presenti dati sensibili che non devono essere memorizzati nel database locale, come ad esempio l'hash della password, è sufficiente disattivare il flag per escludere tale proprietà dalla sincronizzazione.

Per ogni documento da sincronizzare dovranno essere definiti due eventi: *onResyncClient* e *onGetTopics*. Tramite questi eventi il framework comprende quali documenti devono essere mandati ai vari device in funzione del loro stato e dell'utente collegato.

Ricordiamo la proprietà *app.sync.relatedApps*, che consente di considerare più applicazioni installate sullo stesso server come un'unica applicazione per quanto riguarda la sincronizzazione. Nel caso della sincronizzazione dei documenti, se ci sono più applicazioni nel cloud che modificano gli stessi dati, è necessario impostare *relatedApps* in modo da inviare le notifiche di variazioni dei documenti a tutte le sessioni dove questo può avvenire.

## Prerequisiti della sincronizzazione dei documenti

Per poter sincronizzare il database locale con il cloud, è necessario che i documenti da sincronizzare abbiano la stessa struttura: tutte le proprietà incluse nella sincronizzazione che derivano dal database devono essere presenti sia localmente che nel cloud e devono avere il medesimo tipo di dati, lunghezza e vincolo di obbligatorietà. Questo non vale per le proprietà unbound o derivate.

Questo requisito è necessario per garantire che il documento possa essere scritto nel database sia in locale che nel cloud, e viene controllato dal framework al momento della connessione del dispositivo alla sessione proxy. Il controllo avviene in questo modo:

1) La sessione locale calcola un valore di hash sommando tutti i dati dei documenti e delle proprietà coinvolte nella sincronizzazione.
2) Questo valore, chiamato *dataStoreHash*, viene comunicato alla sessione proxy al momento della connessione.
3) Il framework della sessione proxy calcola lo stesso hash sulla struttura dei documenti lato cloud.
4) La sessione proxy lancia gli eventi *app.sync.onConnect* (sincronizzazione base) e *app.sync.DO.onConnect* (plugin gestione documenti in sincronizzazione). Nel secondo evento, l'opzione *cancel* viene preimpostata a *true* se i due hash non corrispondono.
5) Se gli hash non corrispondono e l'opzione *cancel* non viene disattivata durante la gestione dell'evento, la connessione viene rifiutata e la ragione del rifiuto sarà la seguente: *SyncDO connection refused by server: Client's dataStore differs from that of the server (server: XXX, client: YYY)*.

Ci si potrebbe chiedere come sia possibile che la struttura dei documenti locali sia diversa da quella del server. Questo può accadere facilmente modificando le tabelle del database e le relative strutture dei documenti e poi installando la nuova versione nel cloud senza aggiornare l'applicazione nei dispositivi locali. In questi casi i dispositivi non aggiornati non potranno più sincronizzarsi per evitare perdite di dati.

# Funzionamento della sincronizzazione dei documenti

Vediamo ora quali sono i passaggi con cui il database locale e quello nel cloud vengono sincronizzati. Entreremo nel dettaglio delle seguenti fasi:

1) Fase di connessione.
2) Fase di resincronizzazione.
3) Sincronizzazione completa.
4) Generazione delle differenze.
5) Sincronizzazione differenziale.

## Fase di connessione

La fase di connessione di un dispositivo al cloud avviene quando nel client viene attivata la sincronizzazione (*app.sync.enabled* = *true*) ed è disponibile la rete internet.

Durante la fase di connessione di un dispositivo al cloud, viene creata la sessione proxy e ad essa viene notificato l'evento *app.sync.onConnect* per gestire la connessione al sistema di sincronizzazione. Successivamente viene notificato l'evento *onConnect* anche al plugin della sincronizzazione dei documenti.

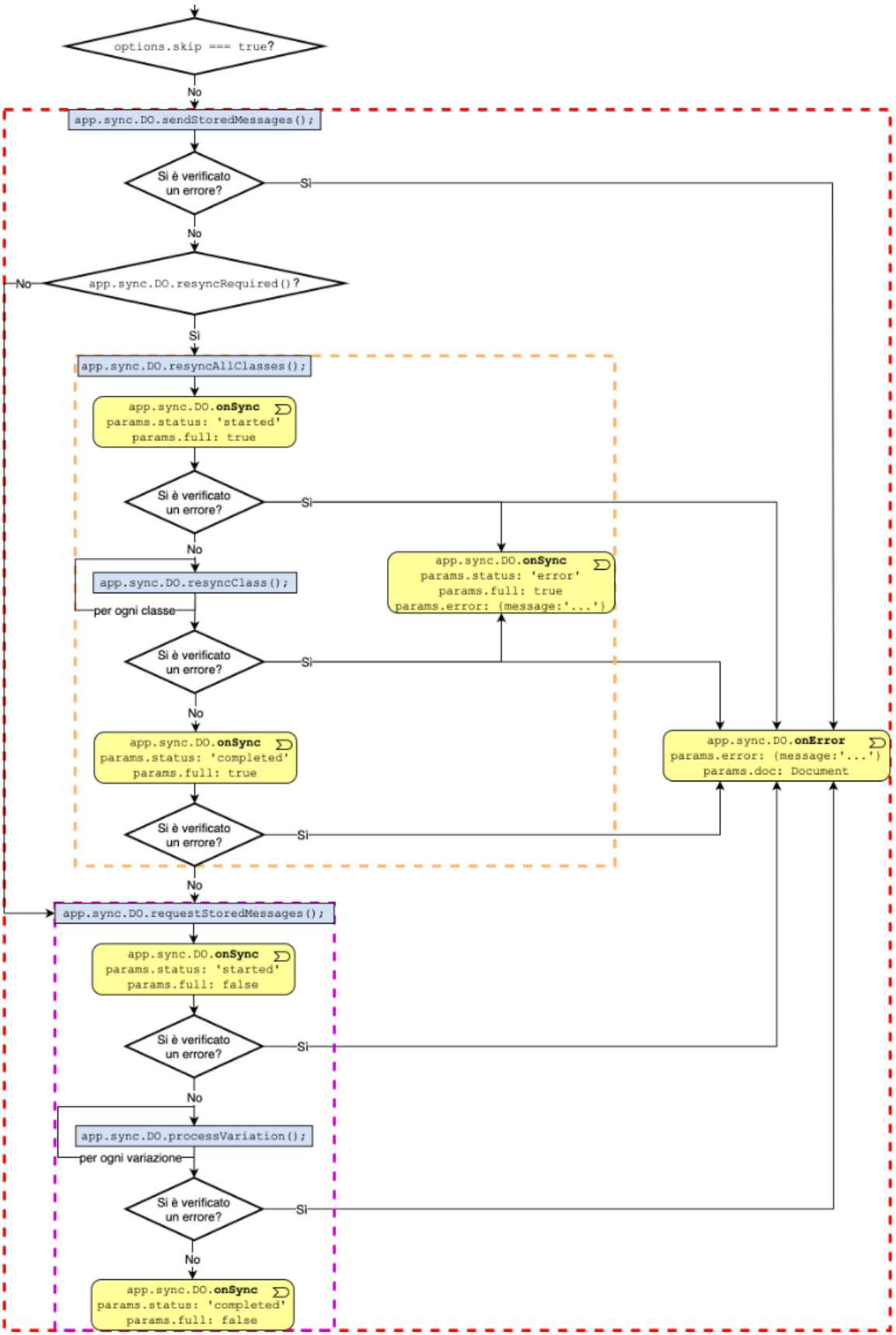
Il diagramma di funzionamento di questa fase è riportato al paragrafo: Attivazione della sincronizzazione.

Abbiamo visto che prima dell'evento *app.sync.DO.onConnect* il framework verifica i requisiti relativi alla struttura dei documenti impostando le seguenti proprietà del parametro *options*:

1) *dataStoreHash*: valore comunicato dal dispositivo come hash della struttura dei suoi documenti.
2) *cancel*: se gli hash non corrispondono e il valore del *dataStoreHash* è diverso da *undefined*, *cancel* viene preimpostato a *true*.
3) *cancelReason*: se *cancel* viene preimpostato a *true*, *cancelReason* viene impostato a *Client's dataStore differs from that of the server.*

Se la connessione al server viene stabilita con successo, gli eventi *onConnect* vengono notificati anche lato client, sia ad *app.sync* che ad *app.sync.DO*, il plugin di sincronizzazione documenti. Durante la gestione di questi eventi è possibile attivare la proprietà *skip* del parametro *options*, ed in questo caso si disabilita completamente la sincronizzazione dei documenti per la sessione in corso senza però chiudere la connessione; in questo modo è ancora possibile accedere in modo remoto ai documenti del cloud.

Nell'immagine seguente vediamo lo schema di funzionamento della sincronizzazione dei documenti nella sessione locale dopo la connessione iniziale, nel caso in cui non venga attivato il parametro *skip*.


options.skip === true?
No
app.sync.DO.sendStoredMessages();
Si è verificato un errore?
Si
No
No
app.sync.DO.resyncRequired()?
Si
app.sync.DO.resyncAllClasses();
app.sync.DO.onSync
params.status: 'started'
params.full: true
Si è verificato un errore?
Si
No
app.sync.DO.onSync
params.status: 'error'
params.full: true
params.error: {message:'...'}
app.sync.DO.resyncClass();
per ogni classe
Si è verificato un errore?
Si
No
app.sync.DO.onSync
params.status: 'completed'
params.full: true
app.sync.DO.onError
params.error: {message:'...'}
params.doc: Document
Si è verificato un errore?
Si
No
app.sync.DO.requestStoredMessages();
app.sync.DO.onSync
params.status: 'started'
params.full: false
Si è verificato un errore?
Si
No
app.sync.DO.processVariation();
per ogni variazione
Si è verificato un errore?
Si
No
app.sync.DO.onSync
params.status: 'completed'
params.full: false

## Fase di resincronizzazione

Se la connessione non viene rifiutata e la sincronizzazione dei documenti non viene disabilitata, il dispositivo invia le variazioni ai documenti generate mentre la connessione non era disponibile (vedi paragrafi successivi) e poi entra nella fase di resincronizzazione, in cui il dispositivo locale deve ricevere tutti i dati aggiornati dall'ultima volta in cui si è collegato.

Questo può avvenire in due modalità: tramite una resincronizzazione completa, oppure in modalità differenziale.

La prima modalità consiste nel leggere tutte le tabelle relative ai documenti da sincronizzare, selezionando i dati che devono essere inviati al dispositivo, per poi trasferirli al dispositivo che infine può memorizzarli localmente. Nella modalità differenziale, invece, viene letta una particolare tabella (*z_syncDO*) che contiene le variazioni avvenute ai documenti sotto forma di messaggi di sincronizzazione. Le variazioni pertinenti al dispositivo verranno inviate e gestite localmente, in modo da aggiornare il database locale.

La sincronizzazione differenziale ha diversi vantaggi rispetto a quella completa: trasferisce solo i dati modificati dall'ultimo collegamento e soprattutto non effettua query sulle tabelle che contengono i dati dell'applicazione, quindi non interferisce con le sessioni browser collegate al server.

Per poter scegliere se effettuare una sincronizzazione completa o differenziale, la sessione locale invia alla proxy il numero dell'ultima variazione che ha ricevuto. La sessione proxy confronta tale valore con il numero della prima variazione contenuta nella tabella *z_syncDO* del server e, se quest'ultimo valore è maggiore del precedente, verrà selezionata la modalità completa perché significa che ci sono variazioni che il dispositivo non ha ricevuto e che il server non ha più nella tabella. Questo solitamente avviene se il dispositivo non si è mai collegato in precedenza, o non si è collegato per più di trenta giorni, periodo dopo il quale le variazioni vengono cancellate dal server. È possibile modificare la durata del periodo tramite la proprietà *app.sync.DO.maxAge*.

Prima di rispondere al dispositivo, il server notifica alla sessione proxy l'evento *app.sync.DO.onResyncRequired* in modo che il codice dell'applicazione possa forzare una resincronizzazione anche se non sarebbe necessaria, oppure effettuare un log delle operazioni di resincronizzazione che vengono richieste.

## Resincronizzazione completa

Se il client riceve l'indicazione di utilizzare la sincronizzazione completa, notifica l'evento *app.sync.DO.onSync* alla sessione locale per indicare che sta per iniziare una fase di resincronizzazione completa e poi chiama il metodo *app.sync.resyncAllClasses*, che esegue la resincronizzazione di ogni documento per cui il servizio è stato attivato.

La resincronizzazione di una classe di documento funziona come segue:

1) La sessione proxy recupera i documenti da inviare al client tramite l'evento *onResyncClient*.

2) La sessione proxy invia i dati al client a blocchi di grandezza pari alla proprietà *app.sync.maxMessages* della sessione client.
3) La sessione client utilizza i dati e salva i record direttamente nel database locale.

Al termine della sincronizzazione di tutti i documenti viene nuovamente notificato l'evento *app.sync.DO.onSync* alla sessione locale per indicare che la fase di resincronizzazione completa è terminata.

L'evento *onSync* può essere usato in due modi: il primo è per bloccare l'interfaccia utente dell'applicazione durante la fase di resincronizzazione completa, utilizzando ad esempio *app.popup* di tipo *loading*. Questo può essere opportuno in quanto nel database sottostante vengono modificati velocemente molti dati: se l'utente potesse usare l'applicazione in questa fase, vedrebbe dati in fase di aggiornamento e l'app presenterebbe una UX degradata dal fatto che il dispositivo è occupato ad aggiornare il database.

Se, al termine della resincronizzazione completa, ci sono videate aperte, è opportuno inviare loro un messaggio per richiedere l'aggiornamento dei dati visto che potrebbero essere cambiati a livello di database.

### Selezione dei documenti da inviare al client: l'evento onResyncClient

Vediamo infine come avviene nella sessione proxy la selezione dei documenti da inviare al client per un particolare tipo di documento.

Quando la sessione proxy riceve la richiesta di resincronizzazione di una classe di documenti, ne crea un'istanza e notifica l'evento onResyncClient. Se questo evento non viene implementato, tutti i documenti di quella classe verranno inviati al client.

All'interno dell'evento, il codice può valorizzare le proprietà del documento (*this*) che verranno utilizzate dal framework come template di filtro per caricare i dati usando il metodo *loadCollection* che verrà chiamato subito dopo l'evento. Che cosa si può usare come filtro? Tutte le proprietà della sessione locale ed in particolare *app.sync.topics*, che di solito contiene le informazioni a cui un determinato utente è interessato.

Se, ad esempio, volessimo inviare al client tutti i documenti *Item* della lista della spesa dell'utente, potremmo utilizzare il seguente codice:

```
App.TBBE.Item.prototype.onResyncClient = function (options)
{
  // topics is the list of the accounts the client is interested in
  this.AccountID = app.sync.topics;
};
```

Le proprietà che si possono utilizzare come filtro sono quelle collegate alla tabella del documento e quelle derivate. Se si imposta la proprietà ad un valore specifico si otterrà un filtro per tale valore; se la si imposta ad un array di valori, si otterrà un filtro tramite una clausola IN sull'array dei valori.

Utilizzando questo metodo, il framework estrarrà i dati un blocco alla volta, senza caricarli tutti in memoria.

Un metodo alternativo per selezionare i documenti da inviare al client è proprio quello di effettuare la query all'interno dell'evento e poi riempire la proprietà *collection* del parametro *options* passato all'evento. Un esempio di questo caso è mostrato nel codice seguente:

```
App.TBBE.ListSharing.prototype.onResyncClient = function (options)
{
  // Get any sharing that relates to one of the two accounts involved
  var ris = yield App.ToBuyDB.query(app,"
      select
        A.ID,
        A.ShoppingListID,
        A.AccountID
      from
        ListSharing A
        inner join ShoppingList B on (A.ShoppingListID = B.ID)
      where
        (B.AccountID in app.sync.topics or A.AccountID in app.sync.topics)
      order by
        A.ID
  ");
  //
  options.collection = yield ris.toCollection(App.TBBE.ListSharing);
  options.skip = true;
};
```

Al termine dell'evento il framework invia la collection dei dati direttamente al client, senza fare ulteriori operazioni.

Se i dati da estrarre sono molti, più di qualche migliaio, è opportuno attivare l'opzione di parzializzazione del caricamento dei dati, impostando a *true* l'opzione *partial*. In questo modo l'evento verrà richiamato più volte fino a che i dati da estrarre saranno esauriti. Per estrarre la pagina di dati giusta, occorre utilizzare l'opzione *lastDoc*, che contiene l'ultimo documento che è stato estratto la volta precedente.

Vediamo come modificare il codice dell'esempio precedente per utilizzare il caricamento parzializzato.

```
App.TBBE.ListSharing.prototype.onResyncClient = function (options)
{
  var last = (options.lastDoc ? options.lastDoc.ID : "");
  //
  // Get any sharing that relates to one of the two accounts involved
  var ris = yield App.ToBuyDB.query(app," \
      select top app.sync.maxMessages
        A.ID,
        A.ShoppingListID,
        A.AccountID
      from
```

```
        ListSharing A
        inner join ShoppingList B on (A.ShoppingListID = B.ID)
      where
        (B.AccountID in app.sync.topics or A.AccountID in app.sync.topics)
        and (A.ID > last or last is null)
      order by
        A.ID
  ");
  //
  options.collection = yield ris.toCollection(App.TBBE.ListSharing);
  //
  options.skip = true;
  options.partial = true;
};
```

Nella prima riga si utilizza il parametro *lastDoc* per comporre un filtro da passare alla query, che, utilizzando una clausola *top*, estrae solo i primi 1000 record.

Impostando il parametro *partial* a *true* nell'ultima riga, l'evento verrà chiamato più volte passando ogni volta un *lastDoc* diverso. Siccome i documenti vengono estratti ordinati per chiave, la tabella verrà effettivamente scansionata tutta, 1000 record alla volta.

## Generazione delle differenze

Prima di affrontare il tema della sincronizzazione differenziale, vediamo come vengono generati i record che rappresentano le variazioni subite dai documenti.

Ogni volta che una sessione salva le modifiche di un documento sottoposto a sincronizzazione, se il salvataggio ha successo, prima di chiudere la transazione viene notificato al documento l'evento *onGetTopics*. Nel codice dell'evento devono essere calcolati i topics della variazione, cioè è necessario definire chi sarà interessato a tale variazione. Al termine dell'evento, viene composto un messaggio permanente di sincronizzazione che rappresenta la variazione. Se la variazione avviene nel dispositivo, il messaggio verrà memorizzato lato client, se avviene nel cloud, verrà memorizzato nel database del server.

Come illustrato nel paragrafo Gestione dei messaggi permanenti, i messaggi permanenti generati nel dispositivo vengono cancellati subito dal database locale appena essi vengono trasferiti. Le variazioni generate nel cloud invece rimangono memorizzate fino al timeout del messaggio, che per la sincronizzazione dei documenti per default è pari a 30 giorni.

L'esempio seguente mostra l'evento *onGetTopics* del documento *Item* di una lista della spesa:

```
App.TBBE.Item.prototype.onGetTopics = function (params)
{
  // An item must synchronize only with the account that knows about it
  params.topics = this.AccountID;
};
```

In questo caso i topics sono rappresentati dall'ID dell'account, cioè dall'ID dell'utente che sta scrivendo la lista della spesa. Quando questo utente utilizza un dispositivo, esso si presenta al server usando come topics proprio l'ID dell'utente; a questo punto la sessione proxy potrà identificare tutte le variazioni da inviare a tale dispositivo.

## Resincronizzazione differenziale

Se il server comunica al client che esso può essere sincronizzato in modo differenziale, la sessione locale notifica l'evento *onSync*, e poi continua richiedendo alla sessione proxy le variazioni da applicare.

La sessione proxy utilizza i *topics* della sincronizzazione per selezionare le variazioni dalla tabella *z_SyncDO* del server, confrontandoli con quelli restituiti dall'evento *onGetTopics* del documento durante il salvataggio.

Per ogni variazione selezionata viene notificato alla sessione proxy l'evento *app.sync.DO.onVariationFiltering*, che permette di decidere se la variazione deve essere inviata al dispositivo o meno. Se l'evento non viene gestito, tutte le variazioni selezionate vengono inviate in quanto rispettano i topics della sessione di sincronizzazione.

Ogni variazione inviata al dispositivo viene gestita in questo modo:

1) Il documento relativo alla variazione viene caricato, a meno che la variazione non sia di inserimento.
2) I valori contenuti nella variazione vengono impostati nel documento.
3) Il documento viene validato e salvato nel database locale.
4) Se richiesto, viene notificato all'applicazione l'evento *onDocUpdate*.

Al termine della gestione di tutte le variazioni, viene nuovamente notificato l'evento *onSync* per comunicare il completamento della fase di resincronizzazione.

Si noti che un meccanismo analogo viene usato dalla sessione proxy per gestire le variazioni ricevute dalla sessione locale. In questo caso però non si verifica la fase di filtro perchè tutte le variazioni locali vengono inviate al server.

### Caricamento delle variazioni lato server

Il server mantiene nella tabella *z_syncDO* le variazioni avvenute nei documenti da parte di tutti i dispositivi e le sessioni browser per un periodo piuttosto lungo, che per default vale 30 giorni. Tale tabella, quindi, è destinata a contenere molte righe, anche nell'ordine dei milioni.

Per questa ragione, quando un dispositivo chiede la sincronizzazione differenziale, non è possibile estrarre tutte le righe della tabella *z_syncDO* per confrontarle con i topics della sessione proxy in modo da verificare se la variazione è pertinente. Occorre aggiungere un filtro alla query per estrarre il numero minimo di variazioni pertinenti.

L'algoritmo di confronto fra topics richiede l'espressione degli stessi come oggetti JavaScript e quindi non è possibile esprimere l'algoritmo esatto come query SQL. Il framework è tuttavia in grado di calcolare un filtro approssimato, esprimibile in linguaggio SQL, che estrae sicuramente tutte le variazioni pertinenti.

A questo punto il framework, dopo che le variazioni sono state estratte, effettua la selezione finale direttamente in JavaScript.

In alcuni casi particolari è possibile ottimizzare ancora di più l'estrazione dei dati dalla tabella *z_syncDO*; per questa ragione prima di eseguire la query il framework notifica l'evento *app.sync.DO.onVariationLoading* alla sessione proxy. Il codice di gestione dell'evento può modificare la query così da applicare le ottimizzazioni desiderate.

### Evento onMissingDocument

Durante la sincronizzazione differenziale è possibile che il dispositivo riceva una variazione di modifica relativa a un documento non presente nel database locale.

Questa eventualità non è comune, infatti il sistema di sincronizzazione garantisce che le variazioni di un documento vengano ricevute nell'ordine in cui sono state generate. Dovrebbe quindi essere impossibile che la sessione locale riceva una variazione di modifica relativa ad un documento di cui prima non ha ricevuto anche la variazione di inserimento.

Questo caso tuttavia può avvenire se un documento cambia i propri topics, o se la stessa cosa avviene per la sessione locale. Se, ad esempio, la sessione locale estende i propri topics, può avvenire che da quel momento in poi riceva variazioni di documenti che prima non riceveva e quindi può capitare che arrivino variazioni di documenti non presenti nel database locale.

In questo caso il framework esegue una chiamata di caricamento remoto del documento in questione dal cloud, lo salva localmente, e poi procede alla gestione della variazione. Prima di effettuare queste azioni viene notificato l'evento *app.sync.DO.onMissingDocument* alla sessione locale in modo che essa possa decidere di risolvere la situazione in modo personalizzato. Perché l'operazione di recupero documenti dal cloud abbia successo, è necessario che il caricamento remoto del documento sia stato attivato.

Si noti che, se si devono variare i topics di una sessione, può essere opportuno forzare una resincronizzazione completa in modo da avere immediatamente la nuova situazione aggiornata nel dispositivo.

Ad esempio, in un'applicazione di tentata vendita, ogni agente deve vedere solo i clienti della sua zona. Se, ad un certo punto, un agente cambia zona, il database locale del suo dispositivo deve essere completamente aggiornato con i dati della nuova zona. Per ottenere questo risultato è possibile forzare una resincronizzazione completa chiamando il metodo *app.sync.DO.resyncAllClasses*, oppure solo un sottoinsieme del database chiamando *app.sync.DO.resyncClass* per le classi di documento da aggiornare.

## Sincronizzazione real time

Al termine della fase di resincronizzazione iniziale, segnalata dalla notifica dell'evento *app.sync.DO.onSync*, la connessione alla sessione proxy rimane attiva e la sincronizzazione dei documenti entra nella fase di real time.

In questa fase, ogni volta che la sessione locale genera una variazione, essa viene inviata direttamente alla sessione proxy, che, a sua volta, la comunica in tempo reale a tutte le altre sessioni interessate a riceverla, in funzione dei topics.

La sessione locale può quindi ricevere in tempo reale le variazioni ai documenti che avvengono nelle altre sessioni, sia locali che nel cloud, sempre che i topics di tali variazioni siano compatibili con i topics della sessione locale stessa.

Durante la fase di sincronizzazione differenziale, sia in real time che all'apertura della connessione, entrano in gioco due eventi importanti: *onDocUpdate*, per aggiornare l'interfaccia utente in funzione delle variazioni ricevute, e *app.sync.DO.onVariationProcessing* per monitorare il progresso delle operazioni di sincronizzazione differenziale.

### Evento onDocUpdate

La sincronizzazione real time consente di avere il database locale sempre aggiornato rispetto alle variazioni che avvengono nell'intero sistema informativo. Tuttavia questo non basta: se viene aggiornato un documento mostrato a video, anche l'interfaccia utente si deve aggiornare.

Questo può avvenire se tutte le istanze caricate in memoria del medesimo documento vengono aggiornate con i dati ricevuti dal sistema di sincronizzazione. Il framework contiene un servizio di notifica di tali variazioni chiamato *docUpdate*, che, una volta attivato, genera le notifiche necessarie per avere un aggiornamento in tempo reale dell'interfaccia utente.

Per attivare il servizio *docUpdate* si deve impostare a *true* la proprietà *app.sync.DO.notifyDocUpdates* sia nella sessione locale che nelle sessioni online, comprese quelle proxy.

A questo punto, quando una sessione riceve una variazione di un documento, essa notifica l'evento *onDocUpdate* alle videate aperte e alle istanze del documento caricate in memoria, se la variazione è di modifica o cancellazione, oppure alle istanze di documenti che contengono una collection di quel tipo di documenti, se la variazione è di inserimento.

Il comportamento standard del framework è sufficiente per aggiornare i documenti visualizzati a video, applicando le variazioni di aggiornamento e cancellazione. Per le variazioni di inserimento, è invece necessario implementare l'evento *onDocUpdate*, perché il framework non può sapere se il nuovo documento deve appartenere ad una determinata collection o meno.

Vediamo come esempio l'evento *onDocUpdate* relativo all'applicazione di esempio *ToBuy* che viene usato per mostrare a video un nuovo item di una lista della spesa se esso è stato inserito da un'altra sessione che gestisce la stessa lista.

```
App.TBBE.ShoppingList.prototype.onDocUpdate = function (doc, options)
{
  if (doc instanceof App.TBBE.ListItem && doc.inserted) {
    var item = doc; // type:TBBE.ListItem
    if (item.ShoppingListID === this.ID) {
      // A new (my) list item has arrived
      this.Items.add(item);
    }
  }
};
```

L'evento viene gestito per il documento *ShoppingList* che contiene una collection di documenti *ListItem*. Se la sessione riceve una variazione di inserimento di un'istanza di *ListItem,* essa notifica l'evento a tutte le istanze in memoria di documenti *ShoppingList* che in questo modo hanno la possibilità di aggiungere il nuovo *ListItem* alla propria collection.

Il codice dell'evento deve verificare che il documento sia effettivamente di tipo *ListItem* e che la variazione sia di inserimento (*doc.inserted*); inoltre si deve controllare che il documento *Item* appartenga proprio alla lista a cui è stato notificato l'evento. Se tutte queste condizioni sono vere, il documento *Item* può essere aggiunto alla propria collection di articoli della lista. Se questa collection è la sorgente dati di una datamap che mostra a video la lista, anche l'interfaccia utente si aggiornerà automaticamente.

### Considerazioni di performance

La gestione dell'aggiornamento delle istanze in memoria dei documenti richiede l'analisi ed il confronto fra tutte le istanze caricate in memoria e ognuno dei documenti che viene aggiornato dalla sincronizzazione. Queste operazioni richiedono l'uso di una piccola, ma non insignificante, quantità di risorse; il servizio *docUpdate* rappresenta quindi un'ottima soluzione quando il numero degli aggiornamenti è relativamente basso, nell'ordine della decina di documenti al minuto, come ad esempio può avvenire in un'applicazione di chat.

Per ottimizzare le prestazioni di *docUpdate* sono presenti due proprietà:

- *app.sync.DO.docUpdateCacheSize*: è il numero minimo di documenti da attendere prima di iniziare il ciclo di aggiornamento. In questo modo l'interfaccia utente non si aggiorna continuamente, ma gestendo gli aggiornamenti a blocchi di documenti.
- *app.sync.DO.docUpdateCacheTimeout*: è il tempo massimo, in millisecondi, da attendere prima di iniziare il ciclo di aggiornamento se sono presenti documenti da aggiornare ma non si raggiunge ancora il numero minimo di documenti specificato con la proprietà precedente.

Regolando opportunamente queste proprietà, l'applicazione applicherà gli aggiornamenti in modo da non aggiornare troppo spesso l'interfaccia utente, eventualità che potrebbe rivelarsi fastidiosa per l'utente.

Per default la cache degli aggiornamenti è spenta, quindi ogni aggiornamento viene applicato subito. Un buon valore di partenza può essere *docUpdateCacheSize* = 30 e *docUpdateCacheTimeout* = 1000.

Se invece l'applicazione riceve un numero molto grande di aggiornamenti, esistono delle strategie di aggiornamento diverse che non prevedono l'uso di *docUpdate*, ma dell'evento *OnVariationsProcessing*.

### Evento onVariationsProcessing

L'evento *app.sync.DO.onVariationsProcessing* viene notificato alla sessione locale o alla sessione proxy mentre esse ricevono e gestiscono le variazioni della controparte. Fra i parametri vengono passati il numero di variazioni gestite, il numero di variazioni ancora da gestire ed infine l'elenco delle classi di documento che sono state aggiornate.

Nella sessione locale è possibile utilizzare questo evento per avvisare l'utente che stanno avvenendo delle variazioni e, se il numero è molto elevato, per bloccare l'interfaccia utente durante la gestione. Inoltre, il parametro che elenca le classi di documento aggiornate può essere usato per determinare quali sono le videate da aggiornate rileggendo i dati dal database.

Nella sessione proxy lo stesso evento può essere usato per inviare un messaggio alle sessioni online e alle applicazioni correlate (*app.sync.relatedApps*) in modo da consentire loro di rileggere i dati dal database.

Tramite l'evento *onVariationsProcessing* è quindi possibile gestire la UX dell'applicazione e l'aggiornamento dei dati senza utilizzare *docUpdate*, nei casi in cui esso potrebbe non essere la soluzione migliore.

## Gestione degli errori

Durante l'uso del sistema di sincronizzazione possono avvenire diversi tipi di errori:

1) Eccezioni legate alla disconnessione dalla rete.
2) Errori di collegamento con la sessione proxy.

Tali errori vengono gestiti tramite gli eventi *onConnect* e *onDisconnect* dell'oggetto *app.sync*.

La sincronizzazione dei documenti aggiunge uno strato di gestione automatica dei documenti e delle loro variazioni. Anche queste operazioni possono causare errori o eccezioni, sia nella sessione locale che in quella proxy. In questi casi verrà lanciato l'evento *app.sync.DO.onError* nella sessione in cui avviene l'errore e anche nella sessione locale se l'errore era avvenuto in quella proxy.

Le principali cause di errori nella sincronizzazione dei documenti sono le seguenti:

1) Eccezioni nel codice degli eventi *onGetTopics* e *onResyncClient*. Queste eccezioni dovrebbero essere rilevate nella fase di beta test e vengono tracciate sia nei log strutturati delle installazioni che tramite il sistema di analitiche per quanto riguarda le sessioni locali dei dispositivi.
2) Disallineamento della struttura del database locale rispetto al cloud: in questi casi la sincronizzazione dovrebbe essere rifiutata a livello di connessione. Tuttavia non è possibile escludere casi in cui una versione aggiornata dell'applicazione non riesca ad allineare la struttura del database locale. In questi casi si consiglia di registrare l'errore tramite la gestione dell'evento *onError* e programmare un reset globale del database locale tramite il metodo *App.<NomeDatabase>.resetSchema*.

Non è possibile fornire una linea guida generale per quanto riguarda le correzioni degli errori di sincronizzazione a livello di database perché esse dipendono dal tipo di applicazione, dalle possibili interazioni con gli utenti e dalle caratteristiche dei dati locali.

Una possibile strategia è quella di registrare le condizioni di errore avvenute durante l'evento *onError* per poi mostrare una videata di attenzione all'utente che lo invita ad attivare una funzione di riallineamento dei dati, che però potrebbe richiedere qualche minuto.

Il riallineamento dei dati potrebbe poi avvenire come segue:

1) Disattivazione della sincronizzazione tramite *app.sync.enabled = false*.
2) Reset della struttura del database tramite *App.<NomeDatabase>.resetSchema.*
3) Riattivazione della sincronizzazione tramite *app.sync.enabled = true*.
4) Attesa dell'evento di fine sincronizzazione tramite messaggio lanciato dall'evento *app.sync.DO.onSync* che registra il termine delle operazioni di riallineamento.
5) Chiusura della videata di attenzione all'utente.

Se dovesse avvenire un errore anche durante il riallineamento, si potrebbe invitare l'utente a contattare il servizio di assistenza o a provare a disinstallare e reinstallare l'applicazione.

### Test della sincronizzazione dei documenti

Il test della sincronizzazione dei documenti avviene avviene nello stesso modo delle altre funzioni della sincronizzazione, cominciando dalla modalità FEBE. Nella sessione locale si consiglia di attivare il log del sistema di sincronizzazione tramite la seguente riga di codice:

```
app.sync.DO.logLevel = App.SyncPlugin.logLevels.verbose;
```

In questo modo sarà possibile seguire l'andamento delle operazioni di sincronizzazione dei documenti nel log della sessione locale.

Un esempio sincronizzazione dei documenti è presente nel progetto sync-design-patterns, avviando l'anteprima FEBE e scegliendo la voce *SyncingDocuments* nel menu della sessione locale.

## Ottimizzazione della sincronizzazione

Il sistema di sincronizzazione è piuttosto complesso e si occupa di gestire al meglio situazioni che possono essere molto diverse fra loro. In questo paragrafo analizziamo i parametri disponibili per personalizzare il sistema e gestire alcune situazioni particolari.

### Parametri della sincronizzazione

In questo paragrafo vengono elencate le proprietà disponibili per personalizzare il comportamento del sistema di sincronizzazione. Non verranno elencate tutte le proprietà di configurazione, ma solo quelle che ne influenzano le prestazioni e che possono essere oggetto di un'analisi di performance.

*app.sync.connectionTimeout*: è il tempo necessario per completare la connessione alla sessione proxy, in millisecondi. Per default vale 5000, ma potrebbe essere necessario aumentarlo se l'evento *onConnect* della sessione proxy richiede tempo per essere eseguito.

*app.sync.autoDisconnectTimeout*: è il periodo di inattività dopo il quale la connessione viene chiusa se *app.sync.autoConnect* è pari a *App.Sync.autoConnectTypes.automatic*. Per default è 30000 e non dovrebbe essere necessario modificarlo.

*app.sync.autoReconnectTimeout*: è il tempo fra due tentativi di connessione, nel caso in cui il precedente sia andato in timeout. È un valore che parte da 125 ms e può arrivare al massimo fino a 4000 ms. Non dovrebbe essere necessario modificarlo.

*app.sync.autoConnect*: determina la politica di gestione della connessione. Il valore di default è *App.Sync.autoConnectTypes.automatic*, che apre automaticamente la connessione nel momento in cui la sincronizzazione viene abilitata, esegue la resincronizzazione e poi mantiene aperta la connessione per un massimo di 30 secondi di inattività, in base al parametro *autoDisconnectTimeout* visto prima. Se durante questo intervallo la sessione locale o la proxy generano dei messaggi, il timeout viene resettato.

Il valore di default è conservativo rispetto al numero di sessioni proxy e di consumo di batteria, tuttavia, dato che la maggior parte delle sessioni utente ha una durata pari a qualche minuto, si consiglia di impostare questa proprietà a *App.Sync.autoConnectTypes.-alwaysConnected:* in questo modo la sincronizzazione in tempo reale sarà sempre attiva fino a che l'applicazione rimane in foreground.

*app.sync.maxMessages*: è il numero di messaggi che vengono gestiti contemporaneamente nello stesso blocco. Il valore di default è 100, che è conservativo rispetto alla memoria utilizzata. Si consiglia di provare un valore di 1000 per diminuire i tempi necessari alle operazioni di sincronizzazione massive di documenti.

*app.sync.DO.commandTimeout*: è il tempo massimo entro il quale un comando remoto dei documenti deve essere completato. Il default è 30 secondi, ma può essere aumentato se si hanno chiamate a metodi remoti che possono richiedere ancora più tempo.

*app.sync.DO.notifyDocUpdates*: abilita l'aggiornamento in tempo reale delle istanze di documenti caricati in memoria. Se impostato a *true*, i documenti in memoria verranno aggiornati rispetto alle variazioni della sincronizzazione, altrimenti no. Si consiglia di attivare questa funzione solo se il numero di variazioni è nell'ordine delle decine al minuto.

*app.sync.DO.docUpdateCacheSize*: numero minimo di variazioni che è necessario raggiungere perché ne venga notificato l'aggiornamento tramite *docUpdate*. Impostando un valore maggiore di zero, si riduce il numero di aggiornamenti che l'utente percepisce, tuttavia si perde il tempo reale effettivo visto che occorre attendere un numero minimo di variazioni, o che trascorra il timeout indicato sotto. Si consiglia di impostare un valore di 10.

*app.sync.DO.docUpdateCacheTimeout*: è il tempo massimo che si desidera attendere prima di eseguire un aggiornamento tramite *docUpdate* se non è ancora stato raggiunto il numero di variazioni indicato dal parametro precedente. Si consiglia un valore pari a 1000.

*app.sync.DO.maxAge:* è il tempo massimo, misurato in giorni, per il quale vengono mantenute memorizzate le variazioni nel cloud. I dispositivi che non si sono mai sincronizzati per il numero di giorni indicati da questo parametro dovranno essere completamente resincronizzati. Il valore di default è 30 giorni.

## Eventi di disconnessione critica

Normalmente la connessione viene mantenuta attiva dal sistema di sincronizzazione fino a che esso è attivo, cioè fino a che la proprietà *app.sync.enabled* è *true*.

Esistono quattro casi particolari in cui nella sessione locale la connessione viene rifiutata e contemporaneamente viene disabilitato l'intero sistema di sincronizzazione:

1) Timeout di connessione dovuto ad un tempo troppo lungo di gestione dell'evento *onConnect* nella sessione proxy.
2) Schema dei documenti non allineato tra dispositivo e cloud.
3) Rifiuto della connessione impostando *cancel=true* nell'evento *onConnect* della sessione proxy.
4) Generazione di un'eccezione non gestita nell'evento *onConnect* della sessione proxy.

In questi quattro casi, alla sessione locale viene notificato l'evento *onConnect* con parametro *success=false*. Se il codice di gestione dell'evento osserva che la proprietà *app.sync.enabled* è pari a *false*, può rilevare l'evento di disconnessione critica, cioè uno stato in cui la sincronizzazione si autodisabiliterà e non funzionerà più a meno che il codice dell'applicazione non attivi nuovamente la proprietà *app.sync.enabled*.

Come nel caso degli errori, la gestione della disconnessione critica dipende dal tipo di applicazione e dal tipo di utenti a cui è rivolta. Una possibile soluzione è quella di aprire una videata che mostra l'errore all'utente e chiedere se vuole rimanere disconnesso oppure se tentare una riconnessione.

## Modalità *fastDiff*

Abbiamo visto che durante le operazioni di sincronizzazione differenziale, vengono gestite le variazioni inviate dalla controparte tramite messaggi di sincronizzazione.

La gestione di una variazione, in particolare, comporta le seguenti operazioni:

1) Caricamento in memoria del documento da variare in base alla sua chiave primaria, che a sua volta notifica gli eventi *beforeLoad* e *afterLoad*.
2) Applicazione delle variazioni presenti nel messaggio.
3) Validazione e salvataggio del documento, che a sua volta notifica gli eventi *onValidate* e *onSave*.
4) Se richiesto, viene gestito *docUpdate*.

Questo insieme di operazioni comporta almeno l'esecuzione di una query di caricamento per ogni variazione, ma se la gestione del ciclo di vita del documento è complessa, potrebbero entrare in gioco diverse query e magari la gestione di altri documenti.

Per ottimizzare questa situazione è possibile usare il metodo di documento *isSynchronizing*, che restituisce *true* se il documento è in fase di gestione della variazione dovuta alla sincronizzazione. Tuttavia, questa operazione richiede modifiche al codice dei documenti e l'effetto di ottimizzazione rimane limitato.

Il sistema di sincronizzazione dei documenti mette a disposizione *fastDiff*, una diversa politica di gestione delle differenze che è **da 2 a 10 volte più veloce** di quella standard. *fastDiff* può essere attivata per una singola classe di documenti, oppure per tutti i documenti dell'applicazione con la seguente riga di codice inserita nell'evento *onStart* dell'applicazione:

```
App.Document.fastDiff = true;
```

Il principio di funzionamento di *fastDiff* consiste in una diversa politica di gestione delle variazioni che consiste in:

1) Creazione di una nuova istanza del documento da variare invece che operare il caricamento dal database, senza notificare alcun evento.
2) Applicazione della variazione.
3) Salvataggio sul database della variazione senza notificare alcun evento.

Per ogni variazione viene quindi creata un'unica query di modifica dati che il dispositivo riesce a gestire in modo ottimizzato perché le operazioni di scrittura possono essere bufferizzate, non essendo mai intercalate da query di lettura dati.

*fastDiff* può essere usato solo nella sessione locale perché si considera che i dati provenienti dal cloud sono affidabili, e quindi non è necessaria la validazione completa dei dati. Inoltre il documento deve poter essere salvato senza contare sul codice dell'evento *onSave*, che in questo caso non viene notificato.

Si consideri infine che *fastDiff* non gestisce *docUpdate*, quindi per l'aggiornamento visuale dei dati in funzione delle variazioni occorre utilizzare *onVariationProcessing* come visto nei paragrafi precedenti.

Tipicamente un dispositivo può gestire 50 variazioni al secondo; tramite *fastDiff* questo numero è nell'ordine di 500 al secondo. *fastDiff* è quindi la soluzione adeguata quando la propria applicazione prevede la variazione di un numero di documenti importante.

## Compressione delle differenze

Nell'ottica di ottimizzare la gestione delle variazioni, occorre considerare un secondo caso notevole: lo stesso documento può subire modifiche successive alle medesime proprietà o a proprietà diverse, e alla fine potrebbe essere direttamente cancellato dal database.

Ogni singola modifica diventa una variazione, quindi un messaggio di sincronizzazione da inviare e gestire separatamente. Per ottimizzare la situazione è possibile consolidare tutte le modifiche allo stesso documento all'interno del medesimo messaggio, in modo da velocizzare tutte le operazioni relative.

A tal fine, il sistema di sincronizzazione documenti contiene il metodo *app.sync.DO.-minimizeVariations* che esegue la scansione dell'intera tabella *z_syncDO* delle variazioni ed esegue le compressioni possibili.
Al termine delle operazioni, che possono richiedere anche diversi secondi, il metodo restituisce un oggetto con i dati relativi alle compressioni effettuate.

Il modo migliore per utilizzare *minimizeVariations* è all'interno di una sessione server che esegue l'operazione di notte, oppure quando il numero di sessioni è minimo: si deve tener conto, infatti, che l'operazione esegue una scansione dell'intera tabella delle variazioni e poi aggiorna un numero importante di esse. È quindi una operazione intensiva a livello di utilizzo del database e può influenzare il comportamento delle altre sessioni di sincronizzazione.

## Sincronizzazione multi-tenant

La sincronizzazione multi-tenant permette di ottenere un ulteriore livello di ottimizzazione della sincronizzazione differenziale, pensato per il caso in cui è possibile partizionare l'insieme dei dispositivi che si devono sincronizzare in modo che ogni insieme possa modificare solo documenti diversi dagli altri insiemi.

È questo il caso di un'applicazione multi-tenant, in cui i dati di più organizzazioni diverse vengono memorizzati nel medesimo database centralizzato. In questo caso i dispositivi di ogni organizzazione vedono e modificano solo i propri documenti e non ci sono sovrapposizioni fra gli insiemi di dispositivi e gli insiemi di documenti che essi modificano.

In questo caso, attivando la sincronizzazione multi-tenant si ottiene un'ottimizzazione che consiste nel poter gestire una tabella delle variazioni diversa per ogni tenant. Il numero totale di variazioni che devono essere memorizzate viene quindi suddiviso in più tabelle e questo rende più efficiente la ricerca delle variazioni nella fase di resincronizzazione. Inoltre se i tenant sono di grandezza molto diversa fra loro, i tenant più piccoli avranno sincronizzazioni veloci perché non devono subire il carico combinato dei tenant maggiori.

Per attivare la sincronizzazione multi-tenant è necessario impostare la proprietà *app.sync.tenant* ad un codice che identifica univocamente il tenant collegato alla sessione e che non cambia nel tempo. L'impostazione deve essere fatta in tutti i tipi di sessioni online che modificano i documenti e quindi può avvenire negli eventi *onStart*, *onConnect* e *onCommand* a seconda del tipo di sessione.

Il codice impostato in *app.sync.tenant* viene usato come suffisso per determinare il nome della tabella *z_syncDO* che contiene le variazioni per una determinata sessione, e quindi per un determinato tenant. Il codice deve essere costituito solo da lettere o numeri. Se, ad esempio, si imposta:

```
app.sync.tenant = "C123";
```

Per quella sessione la tabella delle variazioni sarà *z_syncDOC123*.

## Sincronizzazione di immagini e file

In alcune applicazioni, oltre ai dati dei documenti devono essere sincronizzati anche immagini, video, audio o più in generale file di qualunque natura.

In primo luogo è importante notare che i file non devono essere memorizzati all'interno del database tramite campi di tipo *blob* o *clob*. Questa modalità rende il sistema poco performante e può generare problemi in fase di sincronizzazione dei documenti a causa della dimensione dei messaggi che, in questo caso, devono contenere anche i blob.

La politica consigliata per la gestione di file collegati ai documenti è a due livelli. Nel primo, i file vengono caricati nel file system del server e nel documento si memorizza la URL pubblica del file stesso. Quando il documento viene sincronizzato, il dispositivo riceve la URL e a questo punto può accedere al file e visualizzarlo direttamente nell'interfaccia utente.

C'è un secondo livello di gestione che permette di poter accedere ai file anche quando il dispositivo è offline, cioè completamente disconnesso. In questo secondo caso è possibile scaricare localmente il file una volta ricevuta la URL tramite sincronizzazione.

L'operazione di scaricamento avviene tramite i relativi metodi del file system, mentre il momento migliore per farlo dipende dal tipo di sincronizzazione (completa, differenziale o fast). In generale si consiglia di implementare una funzione che allinea il contenuto dei documenti con il file system, scaricando tutti i file necessari, e di lanciare questa procedura al termine della fase di resincronizzazione.

È possibile vedere un esempio del primo livello di gestione qui descritto nel progetto di esempio To Buy; in particolare il documento *App.TBBE.Item* contiene la proprietà *FileName* per identificare la URL della foto scattata ad un determinato articolo della lista della spesa.

## Sincronizzazione e Cloud Connector

Alcune applicazioni accedono ai dati tramite Cloud Connector e in questi casi il database si trova su un server diverso, di solito on-premise, o in un diverso cloud.

Anche se è teoricamente possibile utilizzare un database con Cloud Connector come datastore della sincronizzazione, questa opzione è altamente sconsigliata, in quanto l'accesso via Cloud Connector è più lento e meno scalabile del caso in cui il database è nello stesso server o nello stesso cloud, soprattutto quando si tratta di effettuare letture massime come quelle della tabella *z_syncDO*. Le query su Cloud Connector, inoltre, potrebbero fallire per problemi di connessione al server o di disponibilità del server di database, causando errori di sincronizzazione difficili da recuperare e da notificare all'utente.

Per queste ragioni, il datastore della sincronizzazione dovrà essere indicato fra quelli locali, eventualmente creandone uno anche solo con lo scopo di contenere la tabella *z_syncDO*.

I dati relativi ai documenti, invece, potranno risiedere anche su database remotizzati, ed in questo caso entreranno in gioco solo durante la resincronizzazione completa. Se l'accesso

avviene via Cloud Connector, gli eventi *onResyncClient* che estraggono i dati in autonomia dovranno farlo usando la parzializzazione per non sovraccaricare il Cloud Connector stesso.

# Analisi della sincronizzazione a runtime

In quest'ultimo paragrafo vediamo come tenere sotto controllo il sistema di sincronizzazione nella sua interezza, mentre si trova in funzione nell'ambiente di produzione.

Gli strumenti da utilizzare sono i seguenti:

1) Log strutturato delle sessioni, che comprende anche le sessioni proxy.
2) Query nella tabella *z_syncDO* del server.
3) Uso del sistema di analitiche per analizzare le performance complessive del sistema.

Si consiglia di eseguire le analisi indicate giornalmente nelle prime quattro settimane di funzionamento in produzione, settimanalmente nelle otto settimane successive, proseguendo poi una volta al mese.

## Log strutturato delle sessioni

L'analisi del log strutturato delle sessioni è utile per capire se si verificano problemi nelle sessioni proxy. È importante considerare sia errori che warning ed eliminarli tutti.

Per attivare il log strutturato di un'applicazione, è necessario valorizzare la proprietà *app.sessionName* negli eventi *onStart*, *onConnect* e *onCommand* in base al tipo di sessione. Si consiglia di usare un codice che identifica il tipo di sessione e il nome dell'utente, o un numero casuale. Ad esempio per le sessioni proxy, se si carica in memoria il documento *Utente* relativo al login della sessione locale, si potrebbe utilizzare:

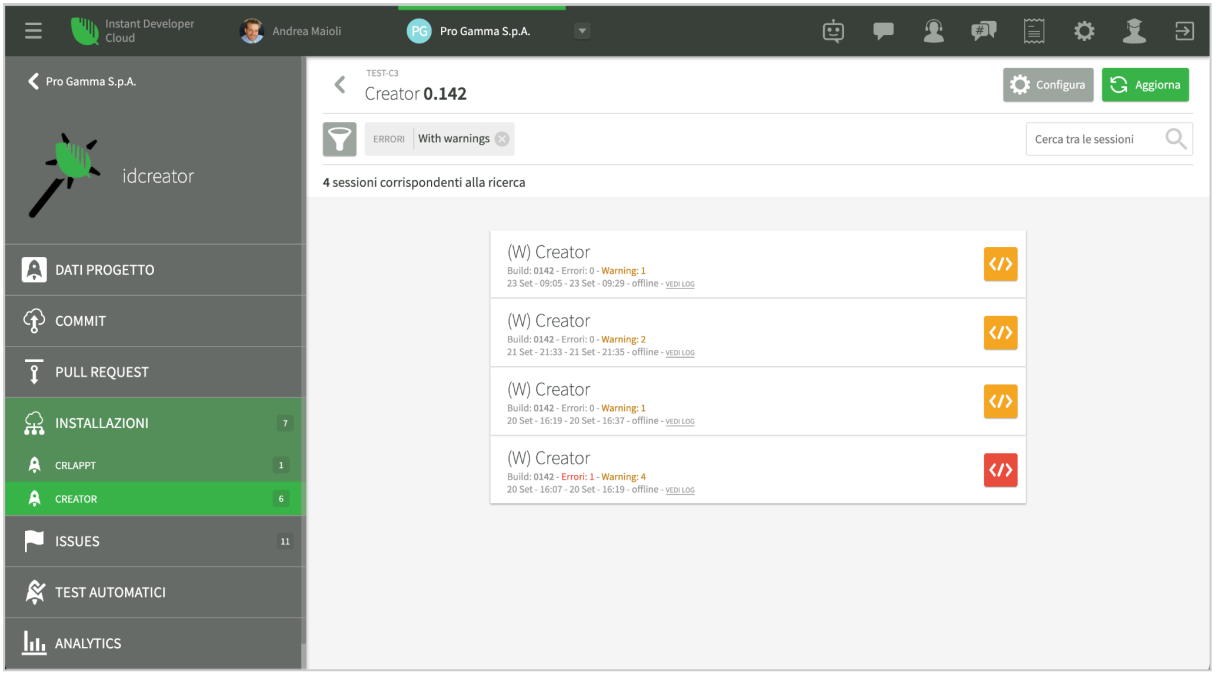
```
app.sessionName = "(S) " + app.loggedUser.userName;
```

Infine, nella videata della console relativa al log strutturato dell'installazione, occorre premere il pulsante *Configura* per attivare la raccolta dati di log.

L'immagine alla pagina successiva mostra un esempio di lista delle sessioni registrate tramite log strutturato già filtrata per includere le sessioni che hanno generato warning. Si consiglia di utilizzare il menù di filtro per ampliare il periodo temporale della selezione ed attivare il filtro per errori o warning.

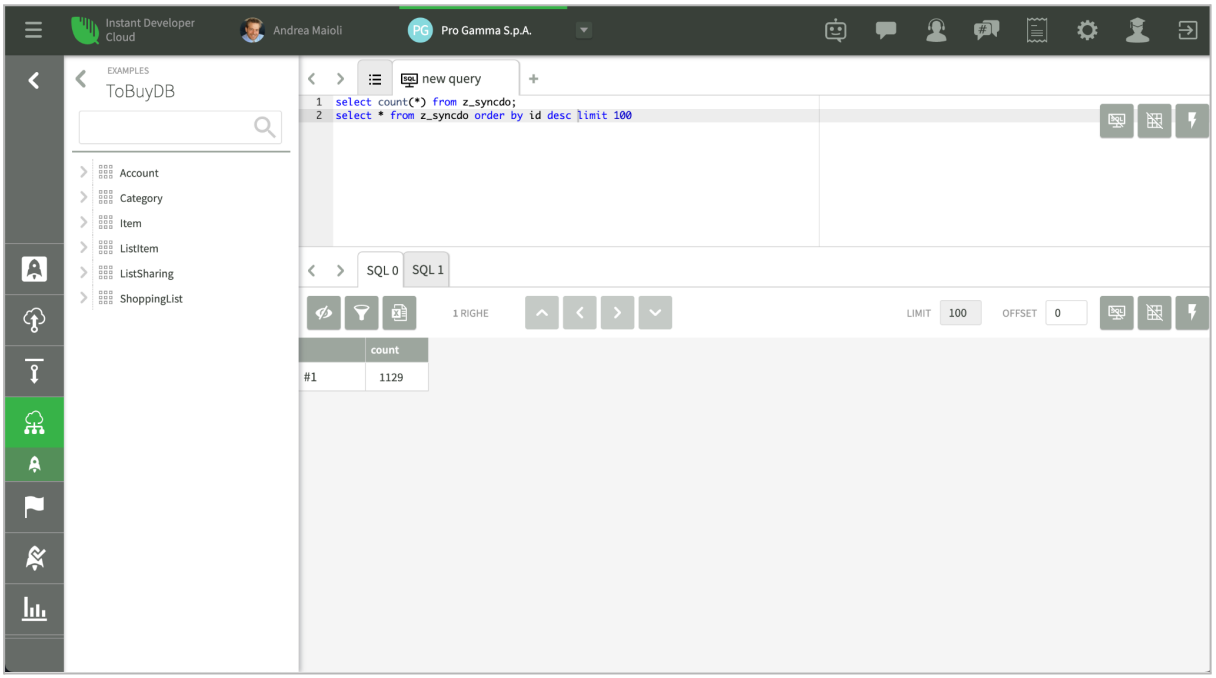
L'analisi del log strutturato consiste infatti  nella selezione delle sessioni con warning e con errori e di conseguenza nell'identificazione delle cause di errore, per procedere poi alla correzione.

Per avere un sistema di sincronizzazione stabile, è necessario che le sessioni online non causino errori o eccezioni, almeno a livello del codice dei documenti.




## Query nella tabella *z_syncDO*

Un secondo livello di analisi periodica riguarda il contenuto della tabella *z_syncDO*, che contiene tutte le variazioni ai documenti che vengono registrate per la distribuzione ai database locali dei dispositivi.



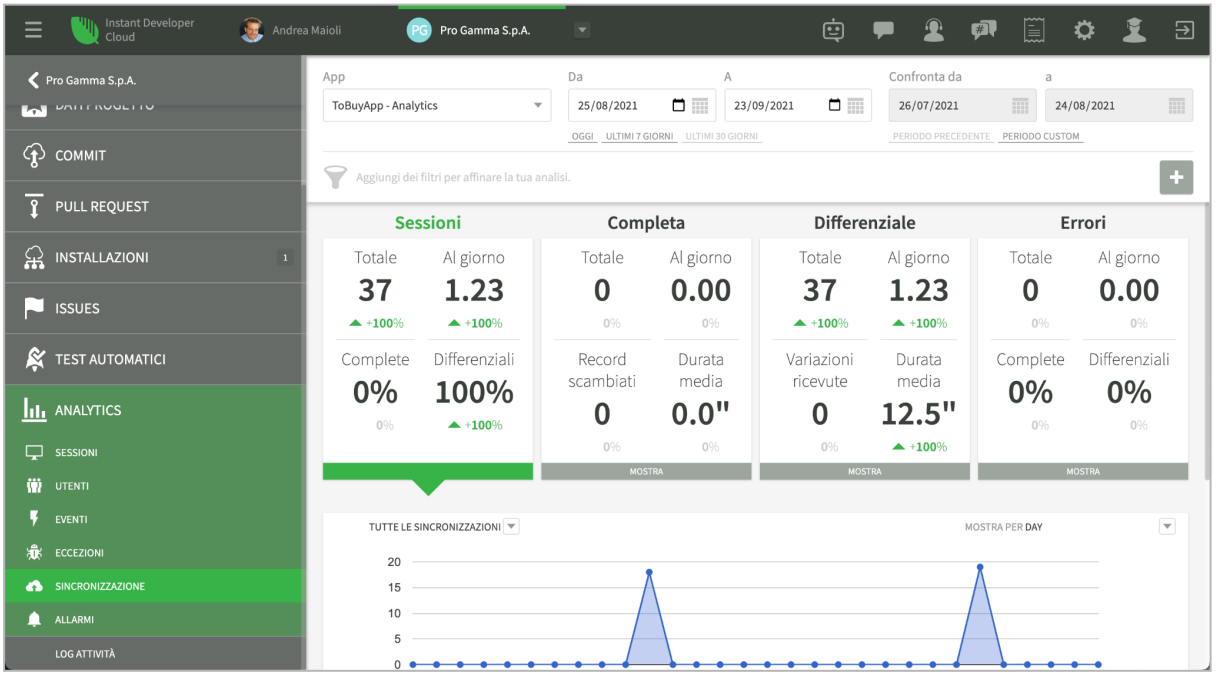

L'esecuzione delle query avviene tramite la pagina database browser della console. Si consiglia di controllare il numero complessivo di variazioni, che deve corrispondere a quello progettato in fase di analisi del sistema. Inoltre è importante scorrere almeno le ultime 1000 variazioni per verificare se esse corrispondono all'uso atteso del sistema.

Un errore comune è quello di modificare e salvare più volte lo stesso documento: anche se i risultati finali sono corretti, il numero di variazioni può diventare eccessivo. Un'altra causa del proliferare di variazioni è la modifica della medesima proprietà dello stesso documento da parte di più dispositivi. Anche in questo caso i dati del database sono corretti, ma la tabella delle variazioni può crescere a dismisura. In tutti questi casi si consiglia di implementare il meccanismo di consolidamento delle variazioni (*minimizeVariations*) e contemporaneamente di eliminare le cause di salvataggio multiplo.

Un ulteriore possibile uso della tabella delle variazioni è quello di comprendere quali dati vengono modificati da ogni sessione. Si possono osservare pattern inattesi o comunque modifiche non necessarie o non dovute. In questo modo si possono eliminare errori anche non legati al sistema di sincronizzazione.

## Uso del sistema di analitiche

Il sistema integrato di raccolta dati analitici comprende due moduli importanti: l'analisi delle eccezioni a livello di dispositivo e l'analisi delle performance del sistema di sincronizzazione.




Le pagine di analitica mostrano i pattern di utilizzo del sistema e danno le indicazioni migliori per capire se il sistema è in equilibrio o si possono prevedere problemi di carico.

In particolare:

1) Il numero di sessioni di sincronizzazione al giorno ci rende consapevoli del grado di utilizzo del sistema.
2) La percentuale di sincronizzazioni complete rispetto a quelle differenziali indica il carico a livello di accesso ai dati dell'applicazione rispetto a quello della sola tabella *z_syncDO*. Dopo il periodo iniziale, la percentuale di sincronizzazioni complete dovrebbe essere molto bassa, attorno al 10%.

3) Per quanto riguarda la sincronizzazione completa, il numero di record scambiati e la durata media possono dare indicazioni sulla dimensione del database da trasferire e sull'impatto dell'operazione sull'esperienza utente. Selezionando il box della sincronizzazione completa saranno visibili anche il numero di record ricevuti per ogni classe, mostrando così eventuali errori di scaricamento dati eccessivi o mancanti.
4) Per quanto riguarda la sincronizzazione differenziale, il numero di variazioni ricevute e la durata media possono dare indicazioni sulla frequenza di modifica dei documenti e sull'impatto dell'operazione sull'esperienza utente. Selezionando il box della sincronizzazione differenziale sarà visibile l'analisi della distribuzione delle durate delle operazioni.
5) Infine selezionando il box degli errori ci si può rendere conto di quali errori vengono rilevati dal sistema. Si può trattare di semplici cadute di connessione, ma si possono scoprire anche errori relativi alle logiche programmate nel codice dei documenti.