Struttura del database

Indice generale

Struttura del database

Scopri il rapporto tra le applicazioni Instant Developer Cloud e i database, sia nel cloud che offline.

Introduzione

Instant Developer Cloud riserva un posto di primo piano alla gestione e all'integrazione dei database relazionali nei propri progetti software. L'attenzione data a questo tipo di sistemi di memorizzazione deriva da una serie di fattori:

1) Oggi i database relazionali sono i sistemi di memorizzazione più diffusi e conosciuti.
2) Nelle applicazioni di carattere gestionale, essi rappresentano il sistema di memorizzazione più semplice, flessibile ed efficace.
3) L'obiettivo di Instant Developer Cloud è quello di consentire lo sviluppo di sistemi omnichannel, anche offline, a partire dalla stessa base di codice. Per questo è necessario utilizzare un sistema di memorizzazione disponibile sia nel cloud che nei device. I database relazionali possono essere usati a questo scopo.
4) Le applicazioni devono essere integrate con i database relazionali esistenti.
5) Esistono sistemi di database open source, come ad esempio PostgreSQL, che garantiscono alta affidabilità, prestazioni, flessibilità e sono integrabili in uno stack applicativo a basso costo di gestione.
6) In casi specifici, le applicazioni sviluppate con Instant Developer Cloud possono interagire con qualunque altro sistema di memorizzazione tramite API cloud.

Nonostante questi fattori positivi, i database relazionali presentano difficoltà di gestione che il framework di Instant Developer Cloud affronta per semplificarne l'utilizzo.

1) *Aggiornamento dello schema*: utilizzando database sia nel cloud che nei dispositivi, occorre gestire l'aggiornamento sincronizzato degli schemi relazionali. A tal fine, il framework applicativo di Instant Developer Cloud si occupa del problema in totale autonomia: nel cloud lo schema dei dati viene aggiornato dalla console durante l'installazione dell'applicazione; nei dispositivi questa operazione avviene quando la versione aggiornata dell'applicazione viene eseguita per la prima volta.
2) *Diversi dialetti di SQL*: i vari tipi di database server hanno dialetti SQL diversi, quindi è impossibile scrivere il testo delle query una sola volta. Instant Developer Cloud risolve questo problema tramite un framework di gestione delle query in grado di tradurre a runtime le query fra i vari tipi di database supportati: PostgreSQL, SQLite, MySQL, SQL Server e Oracle.
3) *Disadattamento fra struttura relazionale e oggetti*: la struttura relazionale dei dati non si adatta bene al paradigma della programmazione ad oggetti, rendendo necessario un approccio misto alla gestione dei dati. Questo problema viene risolto tramite il framework ORM contenuto nelle applicazioni Instant Developer Cloud, che permette un approccio *object oriented* alla gestione dei dati tramite entità e collection.
4) *Problemi di sicurezza*: i database relazionali sono un target primario per gli attacchi hacker, sia a livello di sistema che applicativo. Instant Developer Cloud aiuta a risolvere questo problema tramite un sistema di controllo delle query che impedisce gli attacchi di tipo SQL Injection.

Quali database è possibile integrare?

Instant Developer Cloud permette di sviluppare applicazioni che interagiscono con la gran parte dei database e delle architetture oggi disponibili. Vediamo i casi d'uso principali:

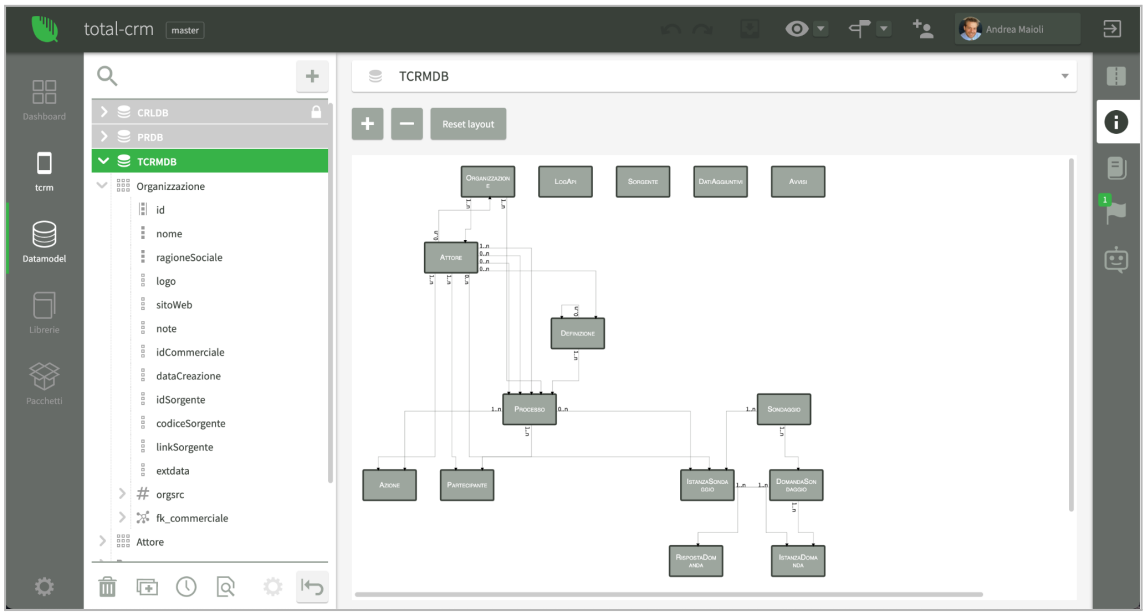
1) All'interno dei server IDE di sviluppo e dei server di produzione di Instant Developer Cloud è già presente un'istanza di PostgreSQL gestita tramite la console di controllo.
2) Quando l'applicazione è in esecuzione nei device, essa converte la connessione PostgreSQL in una connessione SQLite, gestendo automaticamente le differenze di comportamento fra questi diversi tipi di database server.
3) L'applicazione in esecuzione in un server può accedere a qualunque tipo di database modificando la stringa di connessione. Se il database server è di tipo Oracle, SQL Server, MySQL o PostgreSQL, esso viene nativamente supportato dal framework di Instant Developer Cloud, altrimenti sarà necessario fornire il driver Node.js come pacchetto personalizzato e una classe di interfaccia per l'adattamento degli schemi e delle query.
4) Tramite un componente di Instant Developer Cloud chiamato Cloud Connector, è possibile collegare ai server nel cloud i database presenti on-premise all'interno delle reti aziendali anche quando essi non accettano connessioni dall'esterno.

Definizione degli schemi relazionali

La definizione degli schemi entità-relazioni dei database viene effettuata in modo visuale tramite l'IDE di Instant Developer Cloud. Gli oggetti del database modellabili sono i seguenti:

1) Tabelle
2) Campi
3) Relazioni (primary e foreign key)
4) Indici

Non è quindi supportata la definizione visuale di viste, stored procedure o trigger, o altri oggetti specifici di un particolare tipo di database. Se si necessita di tali oggetti sarà possibile definirli tramite comandi SQL personalizzati.



Esempio di schema entità-relazioni

Tipi di dati

I tipi di dati gestibili per i campi delle tabelle sono quelli tradizionali; non sono stati previsti tipi speciali per evitare problemi di compatibilità con i dispositivi offline. A riguardo dei tipi di dato si segnala in particolare:

1) Per la definizione delle chiavi primarie si consiglia di utilizzare un unico campo di tipo *UUID* con lunghezza 24. Questo tipo di campo viene rilevato e automaticamente gestito da Instant Developer Cloud in modo portabile.
2) Per i campi di tipo carattere si consiglia di utilizzare il tipo *varchar*, senza specificare una lunghezza massima.
3) Per la memorizzazione di file, si consiglia di utilizzare il file system, quindi l'uso dei campi *BLOB* e *CLOB* è sconsigliato.
4) Il tipo *JSON* viene tradotto nel database come *varchar* per poterne gestire la compatibilità. L'indicazione del tipo viene utilizzata a livello di framework ORM per mappare l'informazione nel tipo di proprietà corretta a livello di entità.
5) I campi di tipo *datetime*, *date* e *time* vengono sempre memorizzati includendo il timezone. Per default viene usata la zona "0". Il framework converte questi dati nel timezone della sessione tramite i metodi della classe *app.locale*.
6) Se un campo può contenere solo un insieme limitato di valori, si consiglia di definire una lista valori a livello di database e poi indicarla come *dominio* del campo.

Note sulle foreign key

Le relazioni fra tabelle (foreign key) vengono definite in modo visuale nell'IDE.

Fra le proprietà di una relazione, è presente il flag *Crea Indice* che permette di associare un indice alla relazione. Per default viene attivato per tutte le nuove relazioni. Questo è particolarmente importante per le relazioni di tipo possessivo, quelle che vengono usate per collegare un record padre ai propri figli, come avviene, ad esempio, quando si caricano le righe di un ordine data la sua testata. Se, al contrario, una relazione viene usata prevalentemente a scopo identificativo, ad esempio quando dalla riga d'ordine si accede ai dati del prodotto ordinato, è possibile deselezionare questo flag.

Si noti la possibilità di definire le regole di aggiornamento e cancellazione delle foreign key. Impostando il valore *no-action* sia per l'aggiornamento che per la cancellazione, la relazione diventerà di tipo logico, cioè l'IDE ne terrà conto, ma non verrà fisicamente creata nel database.

Nota importante: per i database di tipo SQLite tutte le relazioni sono di tipo logico, a prescindere dalle regole impostate. Questo deriva dalla particolare implementazione presente nei vari dispositivi che non consente di creare foreign key di tipo fisico.

Importazione degli schemi di database

Se un database esistente viene connesso al server IDE tramite Cloud Connector è possibile importare la struttura esistente tramite l'apposita funzione presente nell'IDE. In questo caso verrà impostato anche il flag *Schema sola lettura* che previene l'aggiornamento dello schema da parte di Instant Developer Cloud.

Aggiornamento dello schema

L'aggiornamento degli schemi avviene in modalità diverse a seconda del caso d'uso.

1) Se il database è presente nel server IDE, esso viene aggiornato quando l'applicazione viene lanciata in anteprima, oppure tramite il comando *Aggiorna schema*.
2) Se il database è presente in un server Instant Developer Cloud, esso viene aggiornato quando l'applicazione viene installata tramite la console.
3) Se il database è di tipo SQLite, esso viene aggiornato quando l'applicazione viene lanciata per la prima volta.

Gli schemi di database non vengono quindi aggiornati se:

1) Il database ha il flag *Schema sola lettura* attivato.
2) L'applicazione non viene installata tramite la console di Instant Developer Cloud e non è di tipo offline. **Nota:** in caso di installazione in modalità manuale, l'aggiornamento dei database non viene gestito da Instant Developer Cloud.

Non è consigliabile attivare l'aggiornamento dello schema di un database per il quale lo schema è stato creato tramite importazione.

La procedura di aggiornamento dello schema viene eseguita tramite il confronto fra la struttura precedente del database, memorizzata in un record di una tabella speciale chiamata *z_schema* che viene creata da Instant Developer Cloud, e quella attuale. Il confronto non tiene quindi conto dello schema fisico attuale, ma dello schema del database corrispondente al contenuto della tabella *z_schema*. In questo senso, modificando manualmente lo schema del database si potrebbe arrivare ad una situazione non più gestibile dal framework.

Si noti infine che non tutte le modifiche possono essere gestite tramite aggiornamento, a causa di limitazioni imposte dai database server. Prima di installare l'applicazione, si consiglia di verificare che l'aggiornamento possa essere eseguito testando l'operazione sul server IDE con una versione del database corrispondente a quella di produzione.

Riferimenti ai database per nome

I database vengono creati e riferiti *per nome* sia nei server cloud che nelle applicazioni offline. Nei server IDE, al nome del database viene preposto il nome dell'utente che sta usando l'IDE.

È quindi possibile avere più applicazioni che utilizzano lo stesso database, anche in progetti diversi. In questi casi è necessario che l'aggiornamento dello schema avvenga sempre dall'applicazione più aggiornata.

Se si utilizza lo stesso database in progetti diversi, si consiglia di condividere lo schema tramite componenti; in questo modo lo schema potrà essere facilmente allineato fra i progetti. Si consiglia inoltre di marcare come *Schema sola lettura* tutti i database tranne quello da cui si desidera pubblicare le modifiche.

Durante l'installazione delle applicazioni tramite la console, essa verifica la coerenza dell'aggiornamento richiesto dall'installazione e, nel caso, chiede conferma dell'operazione.

Scrittura di query ed esecuzione di comandi

Passiamo ora all'illustrazione dei metodi di accesso e modifica dei dati dei database. Per estrarre dati dal database si utilizza il metodo *query* dell'oggetto database su cui essa deve essere eseguita. Se invece si vuole modificare i dati, occorre utilizzare il metodo *execute*.

Questi metodi possono essere chiamati su una specifica istanza della classe database e, in tal caso, useranno la connessione collegata a tale istanza, oppure possono essere chiamati in modo statico sulla classe database stessa. In questo caso le chiamate agiranno tramite un'istanza di default dedicata alla sessione e quindi useranno tutte la medesima connessione. Di solito si preferisce questa seconda modalità di esecuzione dei comandi SQL.

Entrambi i metodi richiedono l'inserimento del testo del comando da eseguire. In base a come viene fornita questa stringa, si ottengono due tipi diversi di query o comandi: le *query strutturate* e le *query libere*. Ora vedremo i dettagli dei due tipi di query, tenendo conto che le medesime considerazioni valgono per i comandi (*execute*).

Query strutturate

Una query strutturata viene inserita nel codice tramite un particolare editor SQL. Per iniziare l'inserimento di una query strutturata, è necessario scrivere la parte iniziale del comando, o anche solo "" (stringa vuota) come parametro del metodo query. A questo punto, cambiando riga nell'editor di codice, la chiamata al metodo viene trasformata in un testo multirighe che verrà editato tramite un editor di codice SQL che si sovrappone a quello JavaScript.

Nell'esempio di codice seguente vediamo un esempio di una query strutturata che recupera il nome di un prodotto dato l'id passato come parametro.

```
App.Session.prototype.queryExample = function (id)
{
  var rs = yield App.NDB.query(app," \
    select                  \
      P.ProductName         \
    from                    \
      Products P            \
    where                   \
      P.ProductID = id      \
  ");
  return rs.rows[0].ProductName;
};
```

Cliccando all'esterno dell'editor SQL esso viene chiuso e il testo della query o del comando viene formattato nuovamente come testo multirighe.

La modalità più semplice per inserire una query strutturata è quella di comporre per prima la *from list*, cioè prima di tutto indicare le tabelle ed i join tra di esse. A questo punto è possibile comporre la *select list*, le clausole *where* e infine la *order by.* Non è necessario invece comporre un eventuale *group by* perché verrà proposta in autonomia dall'editor.

L'inserimento del codice SQL viene facilitato tramite un sistema di autocompletamento che si basa su un parser in tempo reale della query in fase di inserimento. Per massimizzare le prestazioni di questo sistema, si consiglia di mantenere la query sintatticamente corretta.

Si può notare che all'interno della query è possibile referenziare tutte le variabili JavaScript in contesto. Esse verranno sostituite al momento di esecuzione della query utilizzando il formato più appropriato in base al tipo di dati in esse contenuto.

Nelle query strutturate è infine possibile inserire funzioni di database. Se si utilizzano le funzioni proposte dal sistema di autocompletamento, si otterranno query portabili in tutti i tipi di database supportati. Se invece si utilizzano funzioni specifiche, esse potrebbero non essere disponibili in tutti i contesti.

Nell'esempio seguente vediamo un esempio di concatenamento fra stringhe dove viene usato l'operatore || (doppio pipe). Tale espressione verrà tradotta a runtime nell'operatore utilizzato dallo specifico tipo di database su cui la query viene eseguita.

```
App.Session.prototype.employeeExample = function (id)
{
  var rs = yield App.NDB.query(app," \
    select                                              \
      E.FirstName || ' ' || E.LastName as FullName      \
    from                                                \
      Employees E                                       \
    where                                               \
      E.EmployeeID = id                                 \
  ");
  return rs.rows[0].FullName;
};
```

Riassumendo, i vantaggi delle query strutturate sono i seguenti:

1) Formattazione automatica del testo multirighe.
2) Autocompletamento del codice SQL.
3) Traduzione automatica per i vari tipi di database supportati.
4) Gestione delle variabili in contesto
5) Gestione della sicurezza delle query (per evitare SQL Injection).

L'elenco delle funzioni di database utilizzabili in modo portabile tra i vari tipi di database è disponibile nella libreria Database presente nella sezione corrispondente del progetto.

Se un determinato comando o query deve essere eseguito sia in modalità cloud che offline, si consiglia di utilizzare solo query strutturate e solo funzioni portabili in modo che possano essere eseguite senza problemi in tutti i contesti di utilizzo.

Query libere

Una query libera consiste in un testo SQL che viene passato al metodo query tramite una variabile di tipo stringa. Un esempio di query libera è il seguente:

```
App.Session.prototype.employeeExample2 = function (id)
{
  var sql = "select E.FirstName + ' ' + E.LastName as \"FullName\", \
    from Employees E where E.EmployeeID = " + App.Utils.sqlEscape(app, id);
  //
  var rs = yield App.NDB.query(app, sql);
  return rs.rows[0].FullName;
};
```

Come è possibile vedere, la costruzione della stringa SQL è completamente a carico del programmatore che ne garantisce la correttezza e la sicurezza in tutti i contesti d'uso. Il framework si limiterà a mandare in esecuzione il comando così com'è, senza eseguire alcuna verifica in tal senso.

In questo modo è possibile eseguire un codice SQL completamente personalizzato, prendendosi tuttavia in carico una responsabilità importante. Per aiutare la gestione dei parametri esterni, si noti l'utilizzo del metodo di utilità *sqlEscape* che, in base al tipo di dati, prepara il parametro ad essere incluso in modo sicuro nella stringa.

Recupero dei risultati

I metodi *query* ed *execute* sono entrambi metodi asincroni che vengono automaticamente sequenzializzati tramite *yield*. In questo caso, il valore restituito dal metodo *query* è un oggetto di classe *DataMap* che contiene i risultati. Il valore restituito dal metodo *execute* è invece un oggetto che restituisce il numero di righe modificate e il prossimo valore della *sequence* o del campo *auto_increment*, se presente in un comando di inserimento.

L'istanza di *DataMap* è un oggetto molto complesso che può essere utilizzato per collegare efficacemente i risultati estratti con l'interfaccia utente. Per maggiori informazioni si consiglia la lettura del capitolo relativo: Datamap.

Per gli scopi del presente capitolo, è sufficiente notare che i risultati estratti dal database vengono restituiti nell'array *rows* di questo oggetto. Ogni item dell'array è a sua volta un oggetto che contiene le proprietà e i valori estratti. Per questa ragione, l'esempio precedente restituisce il nome e cognome dell'impiegato tramite l'espressione `rs.rows[0].FullName`.

Gestione della connessione

Sia le query che i comandi richiedono una connessione con il database per poter essere eseguite. La gestione della connessione è a carico del framework ed è legata alla specifica istanza di database su cui si sta operando. Il framework si comporta come segue:

1) La prima volta che viene usata un'istanza di database (o l'istanza di default) per eseguire una query o un comando, essa richiede al framework una connessione al database. Tale connessione viene estratta da un pool di connessioni interno.
2) L'istanza di database rimane in possesso della connessione finchè ci sono transazioni aperte o finché vengono inviate ulteriori richieste al database. Dopo un determinato tempo di inattività, la connessione viene rilasciata al pool di connessioni.
3) Il pool di connessioni non chiude immediatamente la connessione ma si riserva di riutilizzarla anche per altre sessioni prima di chiuderla definitivamente in caso di inutilizzo prolungato.

Transazioni

Normalmente sia le query che i comandi vengono eseguiti in modalità autocommit, cioè in una singola transazione separata. In alcuni casi è invece importante utilizzare le transazioni in modo esplicito tramite i metodi della classe database: *beginTransaction*, *commitTransaction* e *rollbackTransaction*.

Il primo caso riguarda il coordinamento di più comandi o query che devono essere eseguiti in modo atomico, cioè o tutti o nessuno, per garantire l'integrità dei dati del database.

Il secondo caso riguarda l'esecuzione di un grande numero di comandi di modifica ai dati o di query. In questo caso l'uso di una singola transazione aumenta notevolmente la velocità dello script. Questo incremento di performance deriva dal fatto che al momento del commit di una transazione, il log del database deve essere scritto in modo definitivo sul dispositivo di memorizzazione, e questo può richiedere anche diversi millisecondi per ogni transazione, anche se composta da un unico comando SQL.

Nell'esempio seguente vediamo come usare *beginTransaction* e *commitTransaction* per isolare e rendere più veloce l'aggiornamento del prezzo di una lista di prodotti.

```
App.Session.prototype.productExample2 = function (productsToIncrease,
increasePerc)
{
  yield App.NDB.beginTransaction(app);
  for (let i = 0; i < productsToIncrease.length; i++) {
    yield App.NDB.execute(app," \
      update Products set                          \
        UnitPrice = UnitPrice * increasePerc       \
      where                                        \
        ProductID = productsToIncrease[i]          \
    ");
  }
  yield App.NDB.commitTransaction(app);
};
```

Gestione delle eccezioni

Se la query o il comando producono un errore, viene lanciata un'eccezione che può essere catturata con un normale blocco *try / catch* JavaScript. Il testo dell'errore viene passato al blocco *catch*.

Se si verifica un'eccezione in un comando SQL mentre una transazione è aperta, essa viene marcata come *da abortire* e al momento del commit viene comunque inviato al database un comando di rollback. Questo comportamento viene utilizzato per default su alcuni tipi di database e viene generalizzato da Instant Developer Cloud per rendere il codice portabile in tutti i tipi di contesto di esecuzione.

A tal fine si consiglia di utilizzare sempre comandi SQL che non possono causare errori a livello di database. Ad esempio, prima di effettuare un inserimento su una tabella che si riferisce ad altre tabelle, si deve verificare che tali riferimenti siano validi.

Trattamento delle date

Come abbiamo indicato in precedenza, i valori di tipo *date*, *time* e *datetime* vengono memorizzati nel database nel timezone di default +0. Quando vengono estratti i dati dal database, essi diventano degli oggetti *Date* JavaScript che possono essere utilizzati all'interno dei metodi della classe *app.locale* per convertire o formattare tali valori nel timezone della sessione.

Il seguente esempio scrive nella console la data di creazione di un ordine nel formato previsto dalla sessione in corso.

```
App.Session.prototype.orderExample = function (id)
{
  var rs = yield App.NDB.query(app," \
    select                  \
      O.OrderDate           \
    from                    \
      Orders O              \
    where                   \
      O.OrderID = id        \
  ");
  //
  var m = app.locale.moment(rs.rows[0].OrderDate);
  console.log(m.format("lll"));
};
```

Per maggiori informazioni sull'utilizzo della classe *moment*, fare riferimento alla relativa documentazione online.

Gestione dei database nel cloud

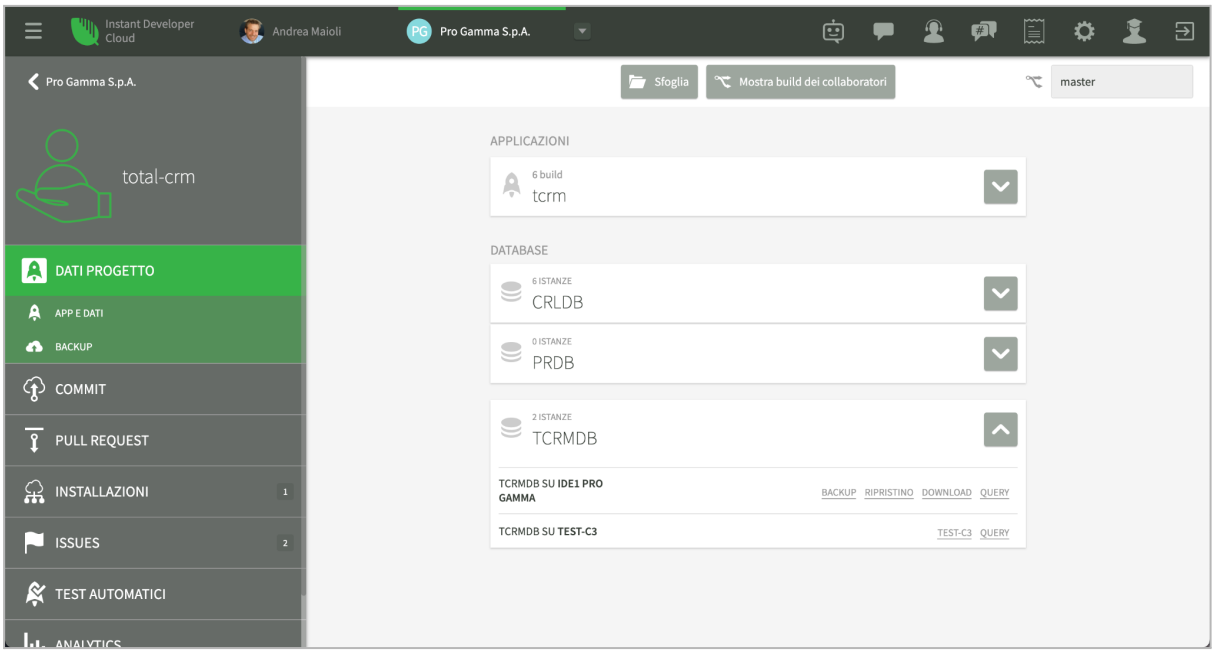
In questo paragrafo viene illustrato come gestire le varie istanze di database PostgreSQL nei server della piattaforma Instant Developer Cloud, ovvero nei Server IDE e nei Server App IDC (server di produzione). Per ulteriori informazioni su questi tipi di server, vedere “Composizione della piattaforma” nel capitolo “Introduzione”.

Istanze IDE e di produzione

Ogni database contenuto in un progetto può avere diverse istanze, a seconda dei server su cui l’applicazione corrispondente viene installata. In particolare:

1) Sui server IDE è presente un’istanza del database per ogni programmatore che partecipa al gruppo di lavoro. Avere istanze separate permette ai vari collaboratori di continuare il proprio lavoro senza essere influenzati dalle modifiche apportate da altri e non ancora rese pubbliche.
2) Sui server di produzione in cui è installata l’applicazione sarà presente un’istanza del database relativa all’ambiente di produzione dell’applicazione.

Per gestire le varie istanze, nella console di Instant Developer Cloud è presente la pagina *App e Dati* relativa ad un progetto, come si vede nell’immagine seguente:



Gestire le istanze di database dell’applicazione

Tramite questa videata è possibile verificare quali istanze esistono per ogni database e gestire ognuna di esse.

Per le istanze contenute nei server IDE è possibile effettuare il backup di un database, ripristinare un database da un backup precedente o effettuare il download del file di backup al fine di gestire i dati in un ambiente diverso. Infine è possibile aprire la pagina di gestione query per il database desiderato.

Le operazioni di manipolazione che coinvolgono i server di produzione potranno essere effettuate tramite le pagine di gestione server a cui è possibile accedere dalla pagina *App e Dati.* È necessario possedere i permessi di gestione del server per accedere a tali operazioni.

Tramite la pagina *App e Dati* si possono affrontare facilmente alcuni passaggi del lavoro quotidiano di un gruppo di sviluppo software.

Allineamento dei database fra i membri del gruppo di lavoro: mentre l'aggiornamento della struttura del database fra i membri del gruppo avviene tramite il sistema di team working, quest'ultimo non gestisce allo stesso modo il contenuto dei database. Per gestire gli aggiornamenti del contenuto è possibile effettuare un backup dell'istanza IDE che contiene i dati aggiornati e poi effettuare il ripristino sulle altre istanze dei membri del gruppo di lavoro usando il backup aggiornato.

Caricamento dei dati in produzione: dopo aver installato l'applicazione in produzione per la prima volta, l'istanza del database di produzione sarà vuota. Anche in questo caso le operazioni di backup dall'istanza IDE e restore su quella di produzione permetteranno di spostare i dati nel server di produzione.

Ispezione dei dati di produzione tramite server IDE: in alcuni casi può essere interessante lavorare sui dati del server di produzione dal server IDE. È possibile ottenere questo risultato semplicemente effettuando il backup dell'istanza del server di produzione per poi ripristinarla nel server IDE.

Disaster recovery dei database di produzione

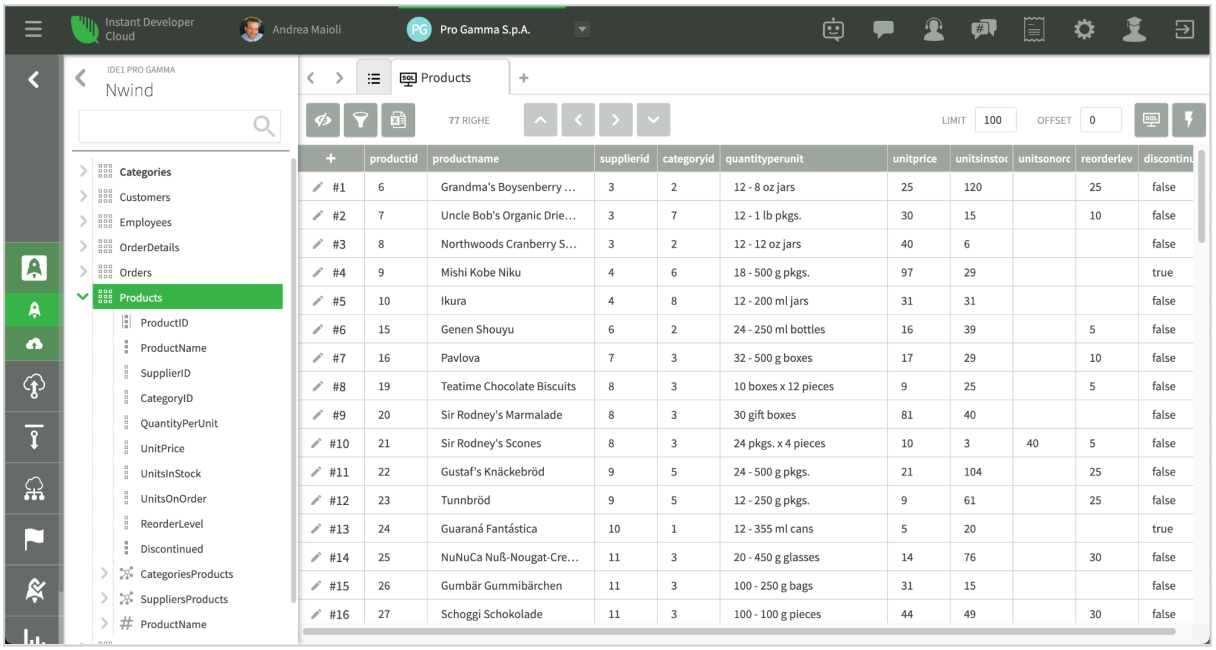
Le operazioni di backup, restore e download relative alle istanze di database non sono pensate per garantire un servizio di disaster recovery dei dati dei server, ma semplicemente per automatizzare le operazioni di copia e spostamento dati nei vari ambienti.

Per gestire la problematica di disaster recovery è invece disponibile il servizio di backup automatico dei server di produzione che è in grado di effettuare uno snapshot coerente dell'intero server su base giornaliera o oraria, mantenendo in linea un mese di dati. È possibile attivare il servizio di backup automatico dalla pagina di gestione del server o durante il processo di configurazione iniziale.

Nota importante: lo snapshot coerente dei server di produzione avviene automaticamente ogni volta che si installa una nuova versione dell'applicazione, indipendentemente dai servizi acquistati o dalla configurazione del server. In questo modo, se l'installazione fosse in qualche modo causa di perdite di dati, sarà possibile ritornare allo stato precedente ripristinando lo snapshot automatico creato prima dell'installazione tramite la pagina di gestione dei server della console di Instant Developer Cloud.

Esecuzione di query tramite la console

Tramite il link *Query* presente nella pagina *App e Dati* è possibile aprire la pagina di gestione query per l'istanza di database desiderata, sia essa presente in un server IDE che in uno di produzione.



Database browser della console

Questa pagina permette di preparare ed eseguire query su qualunque istanza di database. Le query possono essere salvate ed eseguite in un secondo tempo.

Oltre alle query è possibile effettuare operazioni di modifica e cancellazione, sia dalla griglia dati che tramite comandi SQL specifici. Le funzioni di esportazione e importazione dati completano le funzioni di manipolazione dati a livello di tabella.

Nota importante: modificando manualmente i dati di un'istanza contenuta in un server di produzione, viene bypassato il framework di sincronizzazione differenziale che serve per distribuire tali dati ai dispositivi mobile. Se sul server di produzione sono presenti applicazioni che utilizzano questo framework, si deve tenere conto che le modifiche dei dati tramite console non verranno propagate ai dispositivi.

Monitoraggio delle performance

Un accesso ai dati non ottimizzato è la causa più comune di applicazioni che presentano performance insufficienti. I motivi di questa criticità sono molteplici: possono risiedere anche in errori di progetto nello schema entità-relazioni, ma più spesso l'errore che ne rappresenta la causa è quello di misurare le performance dell'applicazione in un ambiente di test monoutente. In questo caso, infatti, anche se l'applicazione esegue un numero elevato di query o se esse sono poco performanti, le performance complessive possono sembrare adeguate.

Nei casi reali potremo avere decine, centinaia, o persino migliaia di utenti collegati all'applicazione e operanti sul database. In questi casi i tempi ottenuti nel caso monoutente si moltiplicano: se, ad esempio, una query richiede 100ms per essere eseguita nel caso monoutente (tempo largamente accettabile), la medesima query può richiedere 10 secondi se ci sono 100 utenti collegati (tempo non più accettabile).

Quando un server è sotto carico, oltre ai problemi di velocità di accesso ai dati si potrebbero incontrare problemi legati all'occupazione di memoria o anche ad altri colli di bottiglia non facilmente evidenziabili nel caso monoutente.

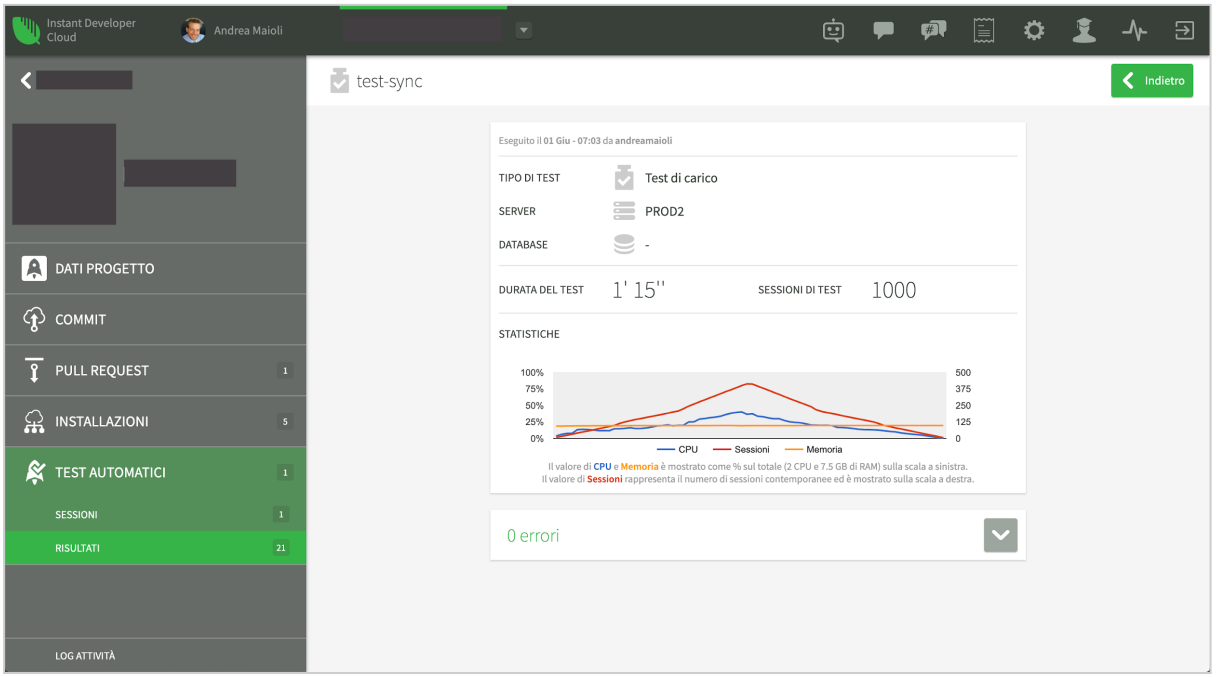
Test di carico

Per tutte queste ragioni, prima di rilasciare un'applicazione in ambiente di produzione reale, è consigliabile utilizzare il sistema di test di carico incluso in Instant Developer Cloud che consente di misurare le performance del sistema nei casi più vicini alla realtà.

Per realizzare i test di carico è necessario installare l'applicazione su un server di produzione di Instant Developer Cloud e poi utilizzare il sistema dei test automatici per registrare una o più sessioni di utilizzo che verranno usate come sessioni campione durante il periodo di carico.

A questo punto è possibile passare al test di carico vero e proprio, in cui il server viene caricato in modo da eseguire le varie sessioni campione fino al numero di sessioni totali richieste nel tempo previsto.

Al termine del test si potranno visualizzare i risultati in termini di grafici di CPU e di memoria utilizzata in funzioni del numero di sessioni attive e una lista di eventuali errori incontrati.



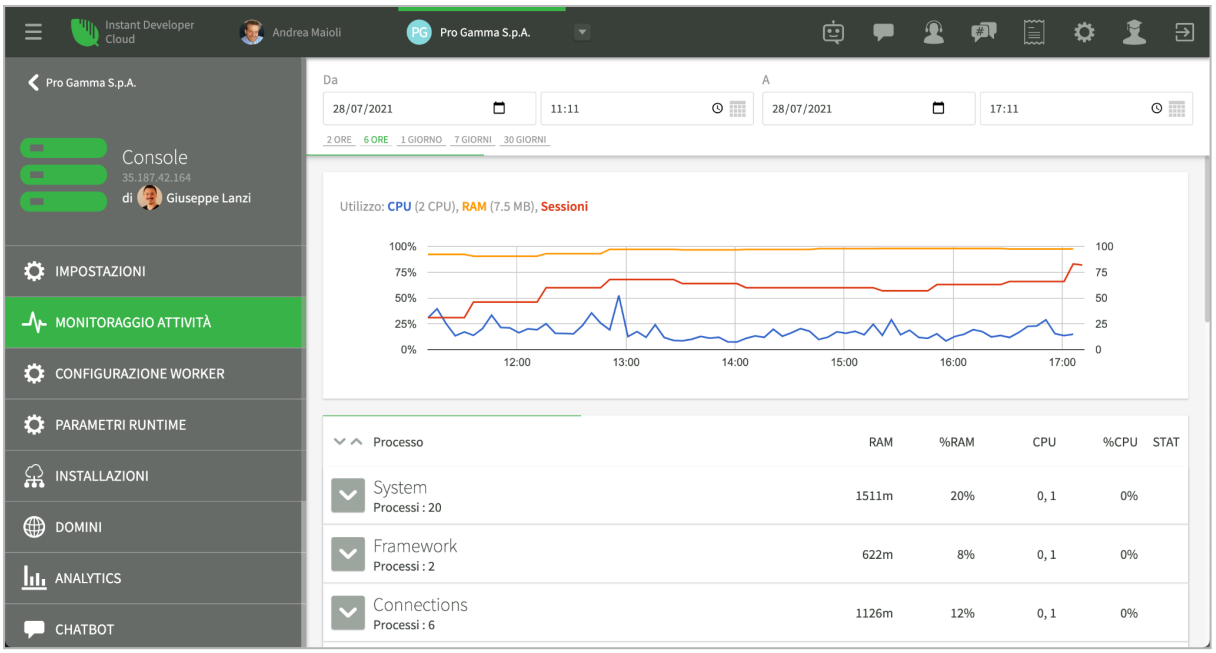
Esempio di risultati di un test di carico con 400 utenti contemporanei

Tipicamente le prime sessioni di test di carico rilevano molte criticità. Per capire quali processi sono più critici è possibile registrare le sessioni campione in modo che ognuna esegua un singolo processo; a questo punto sarà possibile effettuare test di carico puri, cioè con una sola sessione campione alla volta. Solo al termine dell'ottimizzazione dei singoli processi si potrà procedere ad un test di carico misto, in cui tutti i processi vengono eseguiti contemporaneamente.

Monitoraggio in tempo reale

Dopo aver ottimizzato l'applicazione tramite i test di carico si può essere confidenti che il sistema possa reggere il carico desiderato. A questo punto è consigliabile continuare l'analisi delle reali prestazioni del sistema tramite gli strumenti di monitoraggio dell'attività del server.

L'immagine seguente mostra la pagina dedicata a questa attività all'interno della console di Instant Developer Cloud.



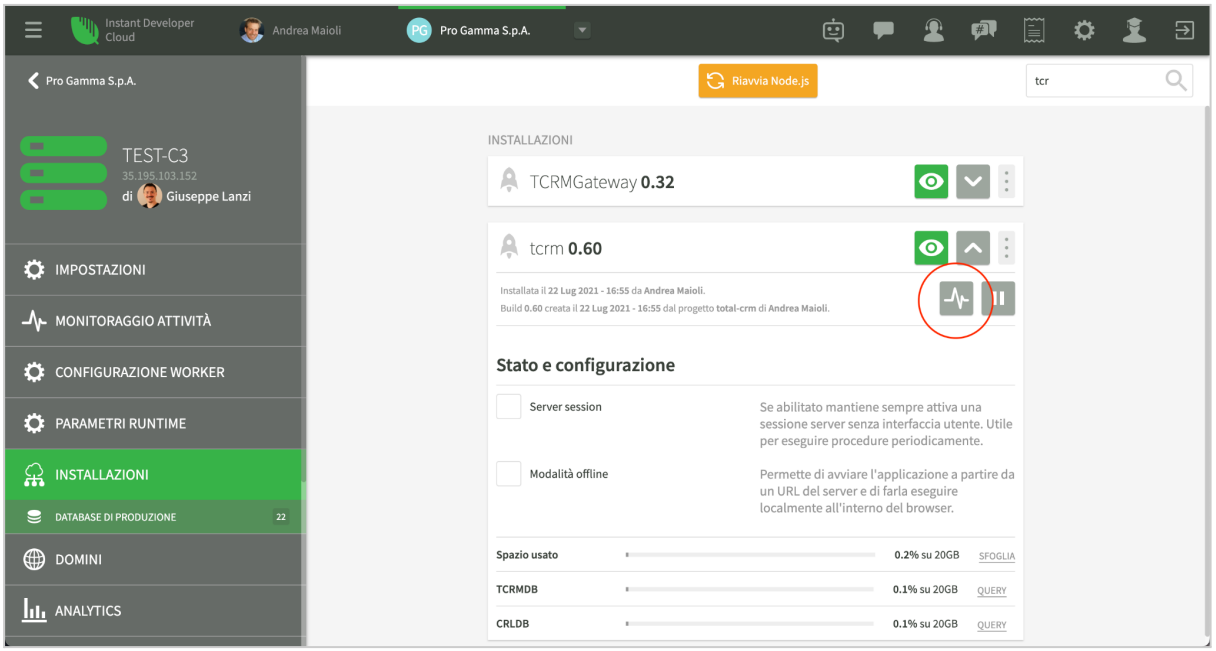
Esempio di monitoraggio in tempo reale delle attività di un server

La schermata è suddivisa in due zone principali. La zona dei grafici mostra i dati relativi al numero di sessioni e all'utilizzo di CPU e RAM nel periodo specificato.

La zona dei processi mostra in tempo reale il risultato del comando *top* linux eseguito sul server. Questa zona è particolarmente importante perché permette di conoscere la reale allocazione delle risorse del server per le varie applicazioni e istanze di database in esso presenti. Sarà quindi più semplice comprendere a quale applicazione o a quale database possono essere imputati i cali di performance del sistema.

Log strutturato e delle query

Uno strumento molto utile per la comprensione del comportamento delle applicazioni in produzione è il cosiddetto log strutturato. Esso si attiva tramite la pagina delle installazioni nella sezione della console dedicata alla gestione dei server e richiede che il codice dell'applicazione valorizzi la proprietà *app.sessionName* per attivare la funzione di log strutturato per una determinata sessione.



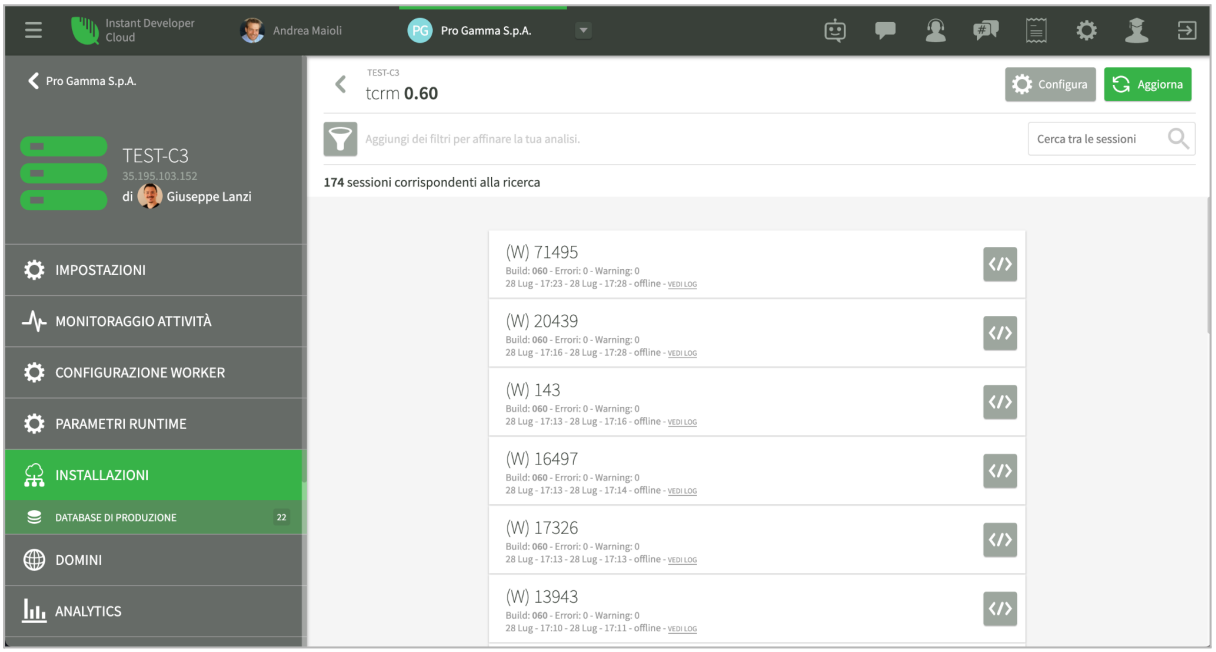
Accedere al log strutturato per un'installazione

La configurazione del log strutturato permette di attivare la registrazione dei messaggi di log, dei warning e degli errori delle sessioni applicative che sono state nominate tramite la proprietà *app.sessionName*, nonché la registrazione di tutte le query eseguite sul database.

Configurazione del log strutturato e attivazione della registrazione delle query

Dopo aver eseguito le analisi richieste, si consiglia di disattivare la registrazione delle query in quanto può richiedere l'uso di uno spazio significativo nel disco del server.

Tramite i filtri nella visualizzazione della lista delle sessioni è facile verificare quali sessioni sono state eseguite e in quali di esse sono avvenuti errori o warning. Cliccando sugli appositi comandi sarà mostrata anche la console della sessione.



Lista filtrabile delle sessioni presenti nel log strutturato

Si segnala infine che se l'applicazione è stata compilata attivando l'opzione *debug a runtime*, dalla lista delle sessioni del log strutturato sarà possibile collegare una sessione IDE ad una sessione in esecuzione nel server per controllare in real time il suo funzionamento.

Log applicativo dell'interfaccia rispetto al database

Un ultimo strumento dedicato all'analisi del funzionamento del database riguarda il tracciamento a basso livello dei comandi inviati al database. Questo tipo di log può essere attivato sia sui server di sviluppo che su quelli di produzione impostando a *true* la proprietà *logEnabled* della classe di database interessata. Se, ad esempio, si desidera attivare il log per il database *MyDB*, nell'evento *onStart* dell'applicazione sarà possibile scrivere:

```
App.MyDB.logEnabled = true;
```

Questa istruzione attiva la scrittura di tutti i comandi inviati al database da parte di tutte le sessioni in un unico file di testo memorizzato nel file system dell'applicazione di nome *temp/[nomeDB].log*. Per ogni comando vengono riportati la connessione su cui viene inviato, il testo, il numero di righe su cui agisce, quanti altri comandi erano già in coda sulla connessione e quanto tempo complessivo viene utilizzato per completarlo. Questo file permette di verificare il modo con cui le varie sessioni si interfacciano con il database momento per momento ed è quindi uno strumento utile per controllarne il funzionamento.

Nota importante: il file può diventare anche molto grande e non vengono imposti vincoli alla sua crescita. È quindi necessario disabilitare la funzione di log dopo un determinato tempo per evitare che il disco del server si riempia.

Il Cloud Connector

Un requisito presente con grande frequenza nelle applicazioni cloud è quello di interagire con risorse *on-premise*. Il caso principale consiste nel poter utilizzare dall'applicazione cloud dati provenienti da uno o più database presenti all'interno di reti aziendali.

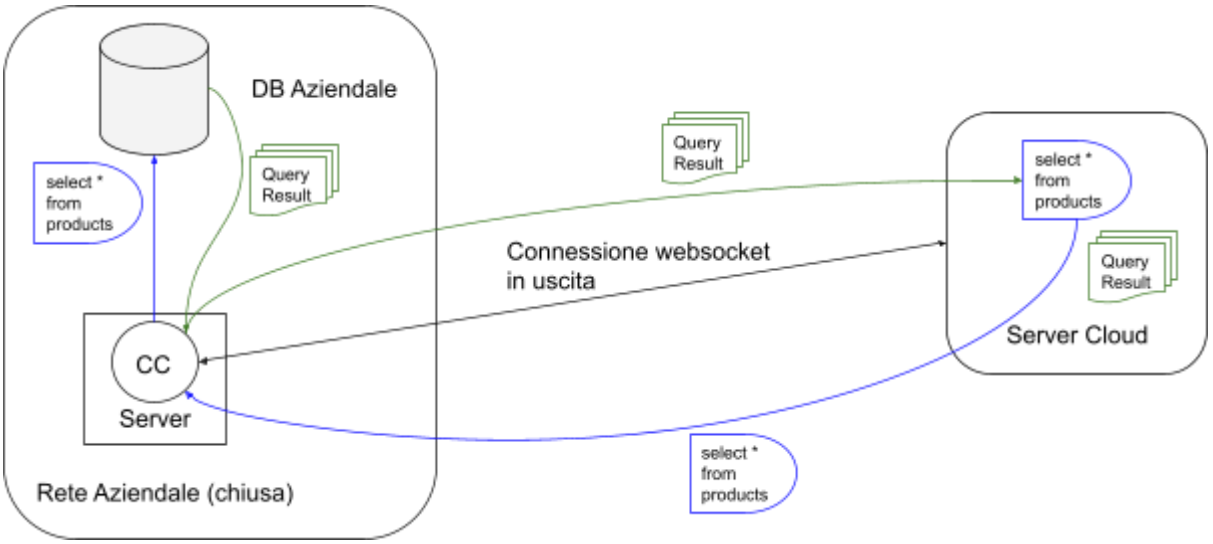
Questo tipo di integrazioni risulta spesso problematico perché:

1) Le reti aziendali sono reti chiuse e non è pertanto possibile accedere dall'esterno alle risorse necessarie.
2) La soluzione di creare API di accesso dal cloud verso l'azienda è costosa sia in termini di implementazione che di gestione della sicurezza della rete aziendale.
3) Una soluzione alternativa è quella di pubblicare i dati aziendali tramite API cloud, ma è costosa in termini di implementazione e di duplicazione dei dati.
4) La possibilità di creare VPN tra cloud e la rete interna pone problemi di gestione della sicurezza e di performance, oltre che di costi di infrastruttura e rigidità dell'architettura.

Tutti questi problemi vengono risolti utilizzando un componente presente nella piattaforma Instant Developer Cloud: il Cloud Connector.

Architettura di un Cloud Connector

Il Cloud Connector è un agente software configurabile basato su Node.js che, installato all'interno della rete aziendale, apre una connessione websocket in uscita verso i server cloud per cui è stato configurato e consente alla controparte cloud di inviare comandi verso le risorse visibili della rete interna.



Integrazione nel cloud di un database aziendale

Nell'immagine precedente le righe viola rappresentano l'esecuzione di una query dal cloud verso il database interno alla rete aziendale, quelle verdi la restituzione al server nel cloud dei risultati del comando.

Questa soluzione ha diversi vantaggi rispetto alle varie alternative. Infatti:

1) Non presenta costi di programmazione in quanto il Cloud Connector è un componente Node.js che richiede solo un file di configurazione per funzionare.
2) Non crea problemi di sicurezza perché il Cloud Connector viene configurato solo localmente tramite un file che specifica a quali server nel cloud esso si può collegare; non richiede quindi porte aperte in entrata nella rete aziendale. Inoltre il Cloud Connector si può collegare solo alle risorse aziendali specificamente elencate nella configurazione.
3) Fornisce performance elevate in quanto i protocolli di comunicazione websocket utilizzano tecniche di streaming di dati compressi e sono specializzate per la trasmissione dei comandi in modo asincrono.
4) Non richiede alcuna forma di gestione dell'infrastruttura come invece avviene nel caso di VPN. È sufficiente installare il componente su un server interno per attivare l'integrazione con il cloud.
5) Rappresenta un'architettura flessibile in quanto consente di collegare al medesimo server cloud un qualunque numero di reti aziendali, anche collegate alla stessa istanza di database cloud.

Risorse integrabili tramite il Cloud Connector

Tramite un Cloud Connector è possibile integrare con il cloud i seguenti componenti software presenti nelle reti aziendali:

1) Database di tipo SQL Server, Oracle, MySQL o PostgreSQL.
2) Porzioni del file system del server in cui è installato il Cloud Connector.
3) Un servizio di Active Directory presente all'interno della rete aziendale.
4) Qualunque risorsa web raggiungibile all'interno della rete aziendale.

Installazione e configurazione del Cloud Connector

Il Cloud Connector è un componente software disponibile per gli utenti di Instant Developer Cloud in codice sorgente aperto, scaricabile dal repository github. Le istruzioni di installazione sono contenute nel repository e i requisiti di installazione sono i seguenti:

1) Installazione di Node.js sul server in cui sarà in funzione il Cloud Connector.
2) Configurazione come servizio tramite il pacchetto Node *pm2*.
3) Installazione dei sorgenti del Cloud Connector e aggiornamento delle dipendenze.
4) Configurazione del Cloud Connector.

Il file di configurazione, di nome *config.json* e presente nella directory base dei sorgenti, contiene le seguenti sezioni:

1) Elenco dei server cloud (sia di produzione che IDE) a cui il Cloud Connector si deve collegare.
2) Elenco dei database utilizzabili dal cloud.
3) Elenco dei file system utilizzabili dal cloud.

Nel pacchetto github è presente un esempio di file di configurazione che è possibile usare come template. Esso è visibile a questo link.
Analizziamo ora le varie sezioni del file di configurazione *config.json* del Cloud Connector.

Nella proprietà *name* va impostato il nome del Cloud Connector così come sarà visto dai server IDE e di produzione.
La proprietà *remoteServers* indica quali server di produzione verranno contattati dal Cloud Connector; questa sezione non va cancellata anche se non si utilizzano server di produzione al limite lasciarla vuota con le sole parentesi quadre.
La proprietà *remoteUserNames* indica a quali utenti è concesso l'accesso al Cloud Connector; è possibile indicare il nome utente e in questo caso lo specifico utente sarà abilitato su tutti i server IDE sui quali può operare. Oppure si può indicare il server (con il suo indirizzo completo) e il nome utente a specificare che solo su quel server IDE sarà utilizzabile il Cloud Connector per quell'utente. Un'ultima possibilità è indicare solo l'indirizzo del server IDE per abilitare tutti i suoi utenti all'utilizzo di un Cloud Connector. Nell'esempio di configurazione del Cloud Connector qui di seguito abbiamo che l'utente *paolo-giannelli* è abilitato a questo Cloud Connector su tutti i server IDE sui quali opera; l'utente *giovanni* solo sul server *ide2-developer* e per il server *ide1-pro-gamma* tutti i suoi utenti sono abilitati.
La proprietà *remoteConfigurationKey* è utilizzata per abilitare la configurazione remota del Cloud Connector. Qui sotto un esempio delle proprietà descritte fino ad ora.

```
{
    "name": "pg-connector",
    "remoteServers": [
        "prod1-pro-gamma.instantdevelopercloud.com",
        "prod3-pro-gamma.instantdevelopercloud.com"
    ],
    "remoteUserNames": [
        "paolo-giannelli",
        "https://ide2-developer.instantdevelopercloud.com/@giobvanni",
        "https://ide1-pro-gamma.instantdevelopercloud.com"

    ],
    "remoteConfigurationKey": "e6159f20-2395-4405-8cfe-7ada5525a0f7",
    "datamodels": [...]
}
```

Nella sezione *datamodels* è possibile indicare i database che si vogliono condividere le cui proprietà sono diverse a seconda del tipo di database. Negli esempi seguenti sono visualizzate le proprietà standard di ogni tipo di database; per specifiche proprietà di un particolare database consultare la documentazione di Node.js specifica.

Esempio di connessione ad un database di tipo SQL Server.

```
{
    "name": "myDB1",
    "class": "SQLServer",
    "APIKey": "b475019c-1f9f-4bc9-b12c-4d36af07f664",
    "connectionOptions": {
        "user": "paolo-giannelli",
        "password": "password",
        "server": "127.0.0.1\\SQLEXPRESS",
        "database": "mydb3",
        "options": {
            "useUTC": false
        }
    }
}
```

Esempio di connessione ad un database di tipo Postgres.

```
{
    "name": "myDB2",
    "class": "Postgres",
    "APIKey": "7a556cbb-cfe6-4c5e-bc20-8a6bd90c5eff",
    "connectionOptions": {
        "user": "paolo-giannelli",
        "password": "password",
        "host": "localhost",
        "database": "mydb2"
    }
}
```

Esempio di connessione ad un database di tipo MySQL.

```
{
    "name": "myDB3",
    "class": "MySQL",
    "APIKey": "e334852c-e353-49b9-8855-ebfd87dba6a4",
    "connectionOptions": {
        "user": "paolo-giannelli",
        "password": "password",
        "connectString": "localhost/mydb4"
    },
    "maxRows": 10000
}
```

Esempio di connessione ad un database di tipo Oracle.

```
{
    "name": "myDB4",
    "class": "Oracle",
    "APIKey": "3be755df-1b48-4cc5-8c09-236716d37a46",
    "connectionOptions": {
        "user": "paolo-giannelli",
        "password": "password",
        "connectString": "localhost/mydb1"
    },
    "maxRows": 10000
}
```

Nella sezione *fileSystems* è possibile impostare le informazioni delle directory che si desidera condividere in sola lettura o anche in scrittura.
Un esempio di condivisione in lettura e scrittura.

```
"fileSystems": [
    {
        "name": "myFS",
        "path": "C:\\users\\paolo-giannelli\\temp",
        "permissions": "rw",
        "APIKey": "5f927c1b-5e7b-42cd-9203-37fcf2f01b98"
    }
]
```

Infine abbiamo la sezione *plugins* dove è possibile configurare l'accesso ad Active Directory.

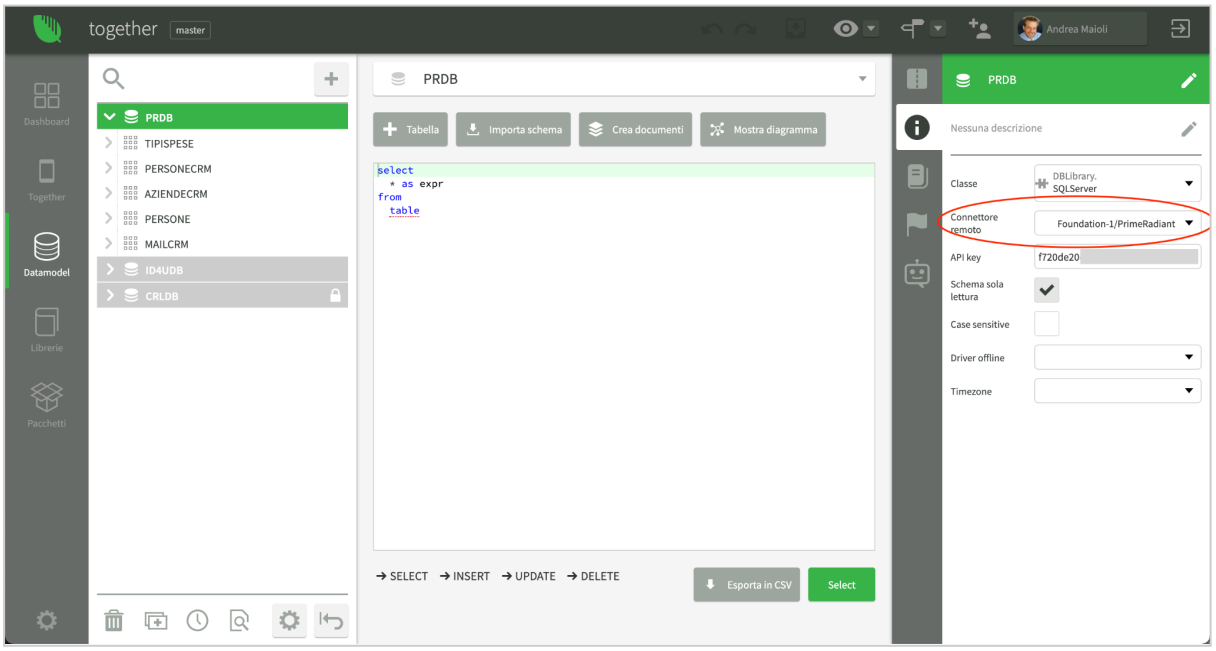
```
"plugins": [{
    "name" : "myAD" ,
    "class" : "ActiveDirectory" ,
    "APIKey" : "770c2c25-bad5-4d67-816d-eaa64753ddbf" ,
    "config" : {
        "url" : "ldapServerUrl" ,
        "baseDN" : "dc = esempio, dc = com" ,
        "nome utente" : "utente" ,
        "password" : "password"
    }
}
```

Utilizzo del Cloud Connector

Per utilizzare le risorse disponibili tramite il Cloud Connector occorre seguire questi passi:

1) Installare e configurare il Cloud Connector.
2) Aggiungere come server remoti i propri server di produzione e il proprio server IDE. **Nota**: per utilizzare il Cloud Connector dai server di produzione è necessario attivare il corrispondente servizio aggiuntivo su tali server (nei server IDE è sempre abilitato).
3) Impostare nel file di configurazione almeno una connessione ad un database e scegliere una API KEY.
4) Nell'IDE di Instant Developer Cloud creare un nuovo Data Model ed inserire la stessa API KEY inserita nel file di configurazione.

A questo punto la connessione al Data Model mostra il nome del Cloud Connector a cui quel Data Model è collegato, come mostrato nell'immagine seguente.



Si noti che il flag *Schema sola lettura* viene impostato in modo permanente, in quanto non è possibile modificare la struttura del database tramite il Cloud Connector.

A questo punto è possibile importare la struttura delle tabelle contenute nel database e cominciare ad usarle come se questo fosse connesso localmente al server nel cloud.

Per i database connessi mediante il Cloud Connector è possibile indicare anche il fuso orario al quale appartiene mediante la proprietà *timezone* in modo da identificare correttamente tutti i campi di tipo datetime.

A livello di codice dell'applicazione, i database connessi tramite Cloud Connector sono utilizzabili con le stesse modalità descritte nei paragrafi precedenti. Occorre però tenere presente la differente architettura di accesso ai dati. Quindi:

1) Le performance sono minori, se vengono comparate con il database PostgreSQL presente nei server IDE e di produzione.
2) Se la connessione websocket si disconnette, una query può generare eccezione di *Connection closed*. Riprovando dopo qualche istante la stessa query funzionerà perchè il Cloud Connector gestisce la connessione in modo automatico.
3) In caso di query con set di risultati molto ampio, prendere in considerazione la lettura dei dati in modalità paginata.

Cambio connettore a runtime

Se occorre collegare istanze diverse di un database alla stessa applicazione in un server di produzione in funzione dell'utente che esegue la login è possibile cambiare a runtime il Cloud Connector da utilizzare mediante il metodo *setRemoteConnector* come da esempio seguente.

```
let cc = "nome-cloud-connector//nome-datamodel";
let apiKey = "e334852c-e353-49b9-8855-ebfd87dba6a4";
App.nome-database.setRemoteConnector(app, cc, apiKey);
```

Accesso al file system

Per accedere al file system di un server messo a disposizione tramite il Cloud Connector è possibile utilizzare la classe *RemoteFS*.
Le istruzioni seguenti permettono di ottenere la lista dei file della directory *images*.

```
let cc = "nome-cloud-connector//nome-datamodel";
let apiKey = "e334852c-e353-49b9-8855-ebfd87dba6a4";
let rfs = new App.RemoteFS(app, cc, apiKey);
let dir = rfs.directory("images");
let lista = yield dir.list();
```

Le istruzioni seguenti permettono di creare un file sul file system remoto partendo da un file esistente nel server dell'applicazione.

```
let cc = "nome-cloud-connector//nome-datamodel";
let apiKey = "e334852c-e353-49b9-8855-ebfd87dba6a4";
let rfs = new App.RemoteFS(app, cc, apiKey);
let fr = rfs.file(nomeFile);
yield picture.put(fr);
```

Utilizzo Web API locali

Se nella rete locale di un server oppure su di un computer locale, non esposto con un indirizzo pubblico su internet, abbiamo delle Web API che vorremmo testare da un server IDE di Instant Developer Cloud possiamo utilizzare il Cloud Connector per accedervi.
Per accedere ad una Web API locale occorre avere un Cloud Connector che condivida una directory del file system locale e qui di seguito vediamo un esempio di codice.

```
let cc = "nome-cloud-connector//nome-datamodel";
let apiKey = "e334852c-e353-49b9-8855-ebfd87dba6a4";
let rfs = new App.RemoteFS(app, cc, apiKey);
let url = "http://192.168.0.32/MyApp/Supplier/" + key;
let u = rfs.url(url);
let retInfo = {};
//
try {
  let res = yield u.get();
  //
  // Tutto ok - nel body ho l'entità Supplier
  if (res.status === 200) {
    let sup = JSON.parse(res.body);
    retInfo = {supplier : sup, error : ""};
  }
  //
  // Richiesta errata - nel body ho una stringa di errore
  if (res.status === 400) {
    retInfo = {supplier : null, error : res.body};
  }
  //
  // Entità non trovata - nel body ho una stringa di errore
  if (res.status === 404) {
    retInfo = {supplier : null, error : res.body};
  }
  //
  if (res.error) {
    retInfo = {supplier : null, error : res.error};
  }
}
catch (err) {
  // Il Cloud Connector non è stato trovato
  retInfo = {supplier : null, error : err.stack};
}
```

Controllo remoto

Il Cloud Connector ha una serie di metodi che ne permettono il controllo da remoto attraverso l'applicazione a cui è collegato. È possibile eseguire il riavvio, la modifica di config.json, l'aggiornamento del software. Per consentire la configurazione da remoto occorre impostare il parametro *remoteConfigurationKey* da utilizzare nell'applicazione di controllo per autorizzare l'accesso. In particolare è possibile utilizzarei metodi:

- *app.listCloudConnectors()* per ottenere la lista dei Cloud Connector connessi al server su cui è in esecuzione l'applicazione;
- *app.pingCloudConnector()* per eseguire il ping al Cloud Connector per verificare se connesso e funzionante;
- *app.changeCloudConnectorConfig()* per cambiare la configurazione del Cloud Connector;
- *app.changeCloudConnectorSouceCode()* per aggiornare i sorgenti del Cloud Connector;
- *app.restartCloudConnector()* per riavviare il Cloud Connector.

Consultare la relativa guida nella reference dei singoli metodi nella documentazione in linea dell'IDE.
Di seguito un esempio di utilizzo del ping al Cloud Connector.

```
try {
  let t1 = new Date();
  yield app.pingCloudConnector("nome-connector");
  let t2 = new Date();
  console.log("Tempo di risposta: in millisecondi", t2 - t1);
}
catch (err) {
  console.log("Nessuna risposta", err);
}
```

Note

Attualmente Cloud Connector non supporta il *caching_sha2_password* come metodo di autenticazione su MySQL 8. Si consiglia invece di utilizzare il metodo di autenticazione *legacy*.
Nella connessione a SQL Server se viene utilizzato un certificato autofirmato alla lettura dei dati di struttura delle tabelle può verificarsi l'eccezione: *"Failed to connect to: undefined - self signed certificate when connecting to MSSQL Server"*.
In questi casi occorre impostare nelle opzioni del database i valori evidenziati qui sotto.

```
"connectionOptions": {
        "user": "paolo-giannelli",
        "password": "password",
        "server": "127.0.0.1\\SQLEXPRESS",
        "database": "mydb3",
        "options": {
            "encrypt": true,
            "trustServerCertificate": false
```