Web API e file system

Integra le tue applicazioni con il RESTo del mondo.

Introduzione

L'architettura moderna dello sviluppo di applicazioni web include spesso l'utilizzo e l'esposizione di Web API.

Una Web API è un'interfaccia del back-end che permette di esporre dati e funzionalità accessibili attraverso il protocollo HTTP. Il principale vantaggio è quello di poter sfruttare questi servizi da diverse piattaforme e device, integrando le funzionalità del proprio sistema informativo con il resto del mondo digitale.

L'integrazione permessa dalle Web API è essenziale per mettere in comunicazione sistemi prodotti da parti diverse, siano essi di back-end o front-end. È possibile utilizzare questa architettura anche nel caso di back-end e front-end sviluppati all'interno dello stesso progetto, tuttavia in questo caso Instant Developer Cloud propone una modalità di integrazione più immediata, sicura ed efficace: il framework di sincronizzazione.

Il modello architetturale di Web API che si è affermato negli ultimi anni, fino a diventare praticamente lo standard, è il modello REST. Una Web API REST permette uno scambio di messaggi stateless, in cui ogni richiesta è svincolata dalle precedenti, senza l'ausilio di cookie o protocolli diversi da HTTP.

In questo capitolo vedremo come utilizzare Web API esistenti o esporre proprie interfacce. Un particolare approfondimento verrà fatto sullo standard ODATA che consente la generazione e l'importazione automatica delle Web API, permettendo così l'integrazione quasi immediata fra applicazioni che utilizzano questo standard.

Il file system

Prima di iniziare il trattamento delle Web API vere e proprie, è necessario approfondire la gestione del file system nei vari contesti di utilizzo. Il file system, infatti, comprende le funzioni necessarie all'accesso locale o remoto alle risorse, quindi per utilizzare Web API o implementarne di proprie, sono proprio i metodi del file system che entrano in gioco.

Strutture del file system

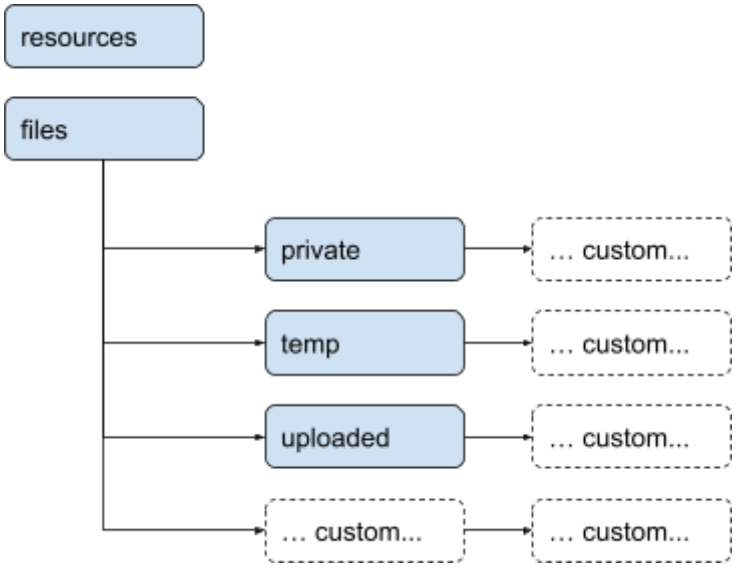
Il framework di Instant Developer Cloud contiene un'interfaccia file system unificata rispetto ai tre contesti operativi di utilizzo delle applicazioni:

- Il contesto server, in cui l'applicazione è in funzione nell'ambiente Node.js e può utilizzare il file system del sistema operativo del server.
- Il contesto browser offline (PWA), in cui l'applicazione è in funzione nel browser, e può utilizzare le funzioni di gestione file che il browser mette a disposizione.

- Il contesto device, in cui l'applicazione è in funzione in una shell Cordova e quindi può utilizzare il file system del device tramite i plugin nativi.

In tutti e tre questi contesti, l'interfaccia unificata di Instant Developer consente al medesimo codice applicativo di ottenere un funzionamento corrispondente.

Ogni applicazione installata ha una propria *home directory* nella quale memorizza i file che essa gestisce. La struttura della *home directory* è mostrata nel disegno seguente:



Nella cartella *resources* vengono memorizzati i file inseriti come risorse di progetto. Le risorse considerate per un'applicazione sono quelle inserite direttamente nell'applicazione stessa e quelle presenti nelle librerie condivise del progetto. Ogni risorsa viene memorizzata con un nome del file costituito dal suo *id* in formato *guid 36*, più l'estensione che corrisponde al nome del file. La cartella *resources* viene ricostruita ad ogni installazione, quindi deve essere considerata in sola lettura. I file contenuti nella cartella *resources* cambiano solo quando cambia anche il loro nome, quindi come impostazione predefinita vengono mantenuti nella cache dei browser per un anno intero.

La cartella *files* è la vera e propria *home directory* del file system dell'applicazione. Essa contiene le cartelle predefinite *private*, *temp* e *uploaded* che hanno un significato particolare:

- *private*: contiene file non visibili da internet, quindi senza URL pubblica.
- *temp*: contiene file temporanei visibili da internet. Questi file non vengono automaticamente cancellati dai server, ma possono essere rimossi dal browser o dai dispositivi in alcune circostanze.
- *uploaded*: contiene file che rappresentano risorse caricate dai dispositivi. I file in questa cartella vengono mantenuti nella cache dei browser per un anno intero, in quanto si suppone che essi non cambino mai il loro contenuto e, quando questo avviene, cambino anche il nome.

Ogni cartella può contenere a sua volta ulteriori sottocartelle personalizzate che mantengono le caratteristiche di quella base.

L’oggetto File System

L’oggetto che rappresenta il file system all’interno dell’applicazione viene creato dal framework all’avvio di una sessione e può essere utilizzato dal codice applicativo tramite *app.fs*. Sono disponibili tre semplici metodi per creare ed inizializzare oggetti di tipo *File*, *Directory* e *Url*.

Per istanziare un nuovo oggetto *File* è sufficiente utilizzare il seguente codice:

```
let f = app.fs.file(path, type);
```

Dove *path* è il percorso del file completo di nome ed estensione relativo alla cartella base definita in funzione del parametro *type*. I valori possibili di *type* sono contenuti nella lista valori *App.Fs.internalType* e sono i seguenti:

- *permanent*: è il valore di default; la cartella base sarà *files*.
- *temp*: rappresenta un file temporaneo; la cartella base sarà *files/temp.*
- *resource*: rappresenta un file di risorsa in sola lettura; la cartella base sarà *resources.*
- *private:* rappresenta un file privato; la cartella base sarà *files/private.*

Per istanziare un nuovo oggetto *Directory* è possibile utilizzare il seguente codice:

```
let d = app.fs.directory(path, type);
```

Dove *path* è il percorso della directory completa di nome relativa alla cartella base definita in funzione del parametro *type*.

Infine, per istanziare un nuovo oggetto URL è possibile utilizzare il seguente codice:

```
let u = app.fs.url(URL);
```

Dove il parametro URL è in formato canonico:
protocollo:///<nomehost><:porta></percorso><?querystring>

Scrivere file

Per creare un nuovo file nel file system e scrivere una riga di testo, è possibile usare il seguente codice:

```
let f = app.fs.file("test.txt");
yield f.create();
yield f.write("hello world\n");
yield f.close();
```

Il file viene creato nella *home directory* per i file pubblici, ovvero nella cartella *files.* Se si desidera vedere il contenuto del file in un browser, si possono aggiungere le seguenti righe:

```
let url = yield f.getPublicUrl();
app.open(url);
```

Se si desidera aggiungere righe ad un file invece di crearlo vuoto, è possibile usare il metodo *append* al posto di *create*.

Per creare un file in una directory personalizzata, tale directory deve essere già presente nel file system, come mostrato nel seguente esempio:

```
let d = app.fs.directory("test-files");
yield d.create();
let f = app.fs.file("test-files/test.txt");
yield f.create();
yield f.write("hello world\n");
yield f.close();
```

Leggere file

Per leggere un file contenuto nel file system sono disponibili due modalità: lettura binaria o lettura di file di testo in forma completa o riga per riga. Vediamo un esempio di lettura completa di un file di testo in memoria.

```
let f = app.fs.file("test-files/test.txt");
let fcontent = yield f.readAll();
```

Il metodo *readAll* non richiede né l'apertura né la chiusura del file, visto che il file viene letto completamente in una sola operazione in memoria. Non è consigliabile quindi usarlo per file di grandi dimensioni, cioè dell'ordine del megabyte o superiori. In questi casi è preferibile utilizzare il metodo *readLine* che esegue la lettura del file riga per riga.

```
let f = app.fs.file("test-files/test.txt");
yield f.open();
f.encoding = "utf-8";
yield f.readLine(function (s) {
  console.log(s);
  f.break();
});
yield f.close();
```

Il metodo *readLine* richiede come primo parametro una funzione che riceve in input le righe del file, lette una alla volta. All'interno della funzione è possibile interrompere la lettura chiamando il metodo *file.break*. Nell'esempio, verrà letta solo la prima riga e poi la chiamata a *readLine* sarà completa e il codice proseguirà con la chiusura del file.

La lettura in formato binario può essere eseguita tramite il metodo *read*, che restituisce un oggetto JavaScript di tipo *ArrayBuffer*.

Leggere le informazioni di un file o una directory

Gli oggetti *File* e *Directory* possiedono proprietà e numerosi metodi nati per recuperare informazioni. Fra i quali:

- *file.encoding*: codifica del file, di default "utf-8". Deve essere impostata prima di utilizzare il metodo di lettura *readLine*.

- *file.publicUrl*: contiene la URL con cui accedere al file da remoto. Nota bene: per recuperare questa informazione in modo coerente in tutti i contesti di utilizzo, è necessario chiamare il metodo *file.getPublicUrl()*.
- *file.originalName*: se il file viene restituito da un'operazione di upload, contiene il nome originale del file. Il file fisico viene rinominato per ragioni di privacy e per evitare conflitti con i file esistenti.
- *file.exists(), directory.exists()*: restituiscono *true* se il file o la directory sono già esistenti nel file system.
- *file.name(), file.extension(), file.length()*: restituiscono il nome, l'estensione e la lunghezza del file in byte.
- *directory.list()*: restituisce un array di *File* o *Directory* contenuti nella directory attuale. È possibile richiedere la ricerca a più livelli.

Se si desidera conoscere la lunghezza di un file in byte è possibile, ad esempio, utilizzare il seguente codice:

```
let l = null;
let f = app.fs.file("test-files/test.txt");
if (yield f.exists()) {
  l = yield f.length();
}
console.log(l);
```

Operazioni su file e directory

Gli oggetti *File* e *Directory* possiedono diversi metodi per la propria gestione. I principali sono:

- *file.create*: crea il file; la directory di destinazione deve già esistere.
- *directory.create:* crea una directory, creando anche quelle intermedie.
- *file.remove, directory.remove*: cancella il file o la directory (e il suo contenuto).
- *file.rename, directory.rename:* cambia nome e sposta il file o la directory in un'altra directory.
- *file.zip, directory.zip*: comprime il file o la directory.
- *file.unzip*: decomprime l'archivio zip ricreando il contenuto compresso.

Funzioni crittografiche

Gli oggetti *File* possiedono i seguenti metodi per la gestione del contenuto criptato:

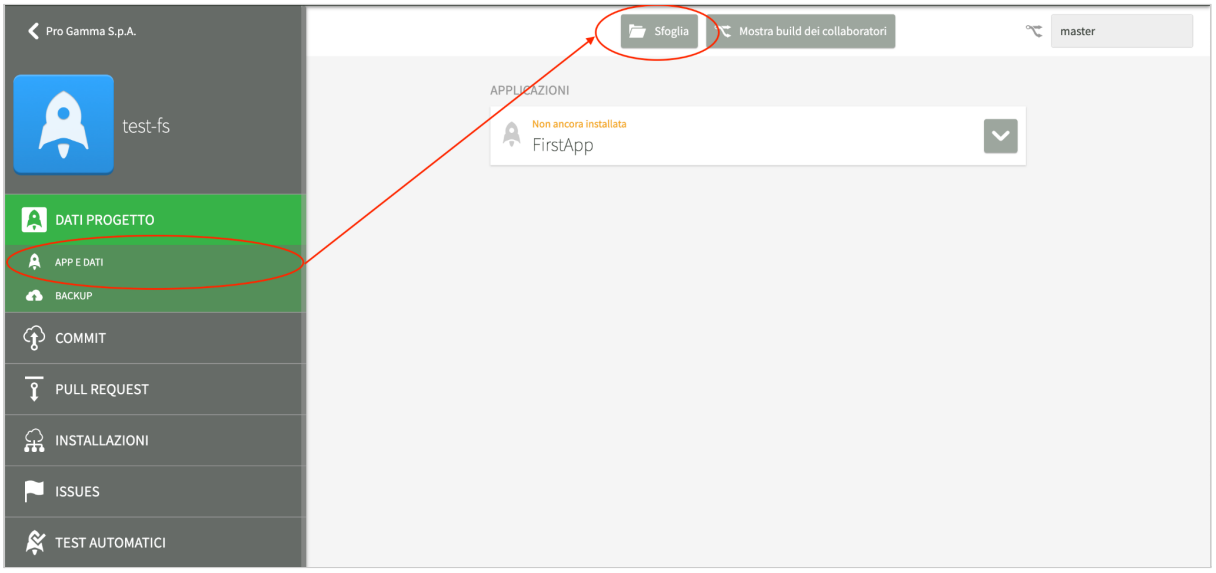
- *file.encrypt*: sostituisce il file con la versione criptata, utilizzando una chiave simmetrica.
- *file.decrypt*: sostituisce un file criptato con la versione in chiaro, utilizzando una chiave simmetrica.

Per la gestione dei contenuti criptati, si consiglia di utilizzare i metodi della libreria *App.Crypt*, che permette di gestire crittografia simmetrica, asimmetrica, hashing, e generazione di chiavi con un'interfaccia consistente nei vari contesti di utilizzo: server, browser e dispositivi.

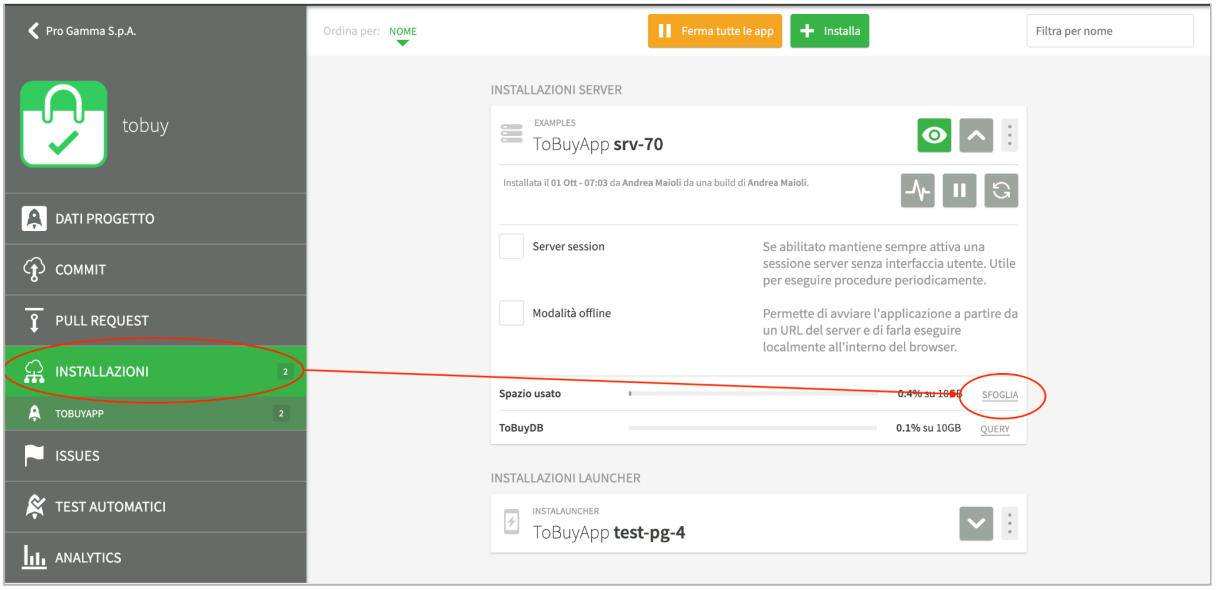
Controllare il contenuto del file system

Per controllare il contenuto del file system di un'applicazione su un server IDE o di produzione è possibile utilizzare la pagina *file browser* della console.

Per accedere al file system dell'applicazione sul server IDE, entrare nel sottomenu *APP E DATI* e poi cliccare il pulsante *Sfoglia*.



Per accedere al file system di un'installazione dell'applicazione su un server di produzione, entrare nella pagina delle installazioni e poi cliccare il pulsante *Sfoglia*.



Per controllare il contenuto del file system di un'applicazione installata in un browser o in un dispositivo, è necessario utilizzare gli inspector del browser o del dispositivo. Si consiglia di fare riferimento alla documentazione del browser o del dispositivo per maggiori informazioni.

Accedere alle risorse

I file di risorsa inseriti nel progetto vengono copiati in una parte del file system dell'applicazione in sola lettura. Per accedere a questi file è possibile utilizzare il metodo *app.getResourceFile* che restituisce un oggetto *File* dato il nome della risorsa o il relativo riferimento, come mostrato negli esempi seguenti:

```
let f = app.getResourceFile($panca);
console.log(f.readAll());

let f = app.getResourceFile("panca");
console.log(f.readAll());
```

Si ricorda che i file di risorsa sono pubblici, cioè visibili da internet. Tuttavia essendo i loro nomi sempre in formato *guid 36,* occorre conoscerne il nome per poter accedere.

File system remoti

Nel manuale Struttura del database è stato introdotto il componente Cloud Connector, in grado di far interagire i server nel cloud con servizi presenti on-premise come ad esempio database relazionali o parti di file system.

Operare sui file system on-premise condivisi tramite Cloud Connector è molto semplice: basta creare l'oggetto che rappresenta il file system remoto con questa riga di codice:

```
let rfs = new App.RemoteFS(app, "<nome-cc>://<nome-fs>", "api-key");
```

Dove *<nome-cc>* è il nome del Cloud Connector che contiene il file system su cui operare, *<nome-fs>* è il nome del file system come configurato nelle opzioni del Cloud Connector ed infine *<api-key>* è la chiave di connessione del Cloud Connector.

L'oggetto *RemoteFS* estende l'oggetto File System base, cioè quello ottenuto tramite *app.fs*, quindi tutti metodi e le proprietà visti finora sono disponibili anche nel file system remoto.

L'unica operazione da aggiungere è lo spostamento di un oggetto *File* tra un file system remoto e quello locale. Per ottenere questo risultato è disponibile il metodo *file.put* che sposta il file nell'oggetto corrispondente in un diverso file system. Esempio di codice:

```
let rfs = new App.RemoteFS(app, "<nome-cc>://<nome-fs>", "api-key");
let rf = rfs.file("panca.txt");
let lf = app.fs.file("panca.txt");
rf.put(lf); // sposta il file dal cloud connector al server
lf.put(rf); // sposta il file dal server al cloud connector
```

Si ricorda che l'uso dei file system remoti è possibile solo quando l'applicazione è in funzione nel server e non nei browser o nei dispositivi. Inoltre gli spostamenti tramite *put* sono consentiti solo tra un file system remoto e quello del server, non direttamente fra due istanze di file system remoto.

L’oggetto URL

Concludiamo la panoramica delle funzionalità del file system illustrando l’oggetto *url*, che rappresenta una URL su cui eseguire operazioni come *get*, *post*, eccetera. Per definire un oggetto *url* è sufficiente usare il metodo corrispondente del file system:

```
let u = app.fs.url("http://www.bing.com/HPImageArchive.aspx?format=js&n=8")
```

A questo punto è possibile chiamare i metodi: *get, post, put, delete, head, patch* che restituiscono la risposta completa del server, che nell’esempio è il servizio immagini di Bing:

```
let result = yield uu.get();
console.log(result);
let p = JSON.parse(result.body);
console.log(p);
```

Tutti i metodi di chiamata ammettono un parametro *options* che permette di specificare le seguenti opzioni:

- *params*: è un oggetto che contiene la query string della chiamata. I nomi dei parametri e i valori vengono automaticamente codificati.
- *headers*: è un oggetto che contiene gli header della chiamata.
- *timeOut*: timeout della richiesta, in millisecondi.
- *authentication*: è un oggetto che permette di specificare le proprietà *username* e *password*, per il livello di autenticazione *basic*.
- *responseType*: indica il tipo di risposta desiderata: “*arraybuffer*” o “*text*”.
- *body*: se la chiamata è di tipo *post* o *patch*, rappresenta il body della chiamata.
- *bodyType*: specifica il content-type della richiesta, in caso di *post*, *patch o put*.

La possibilità di utilizzare chiamate URL è disponibile in ogni contesto di utilizzo. Tuttavia, se l’applicazione è in esecuzione nel contesto browser, le chiamate devono soddisfare i criteri CORS. Si consiglia di verificare che il server consenta la condivisione delle risorse cross-origin, altrimenti la chiamata genererà un’eccezione.

Trasferimento di file tramite URL

L’oggetto *url* contiene anche i metodi *download* e *upload* che permettono di scaricare un file da un server o di caricare un file verso un server remoto. Questi metodi richiedono come primo parametro un oggetto *File* che rappresenta il contenuto da caricare o il posto dove scaricare la risorsa.

È possibile monitorare il progresso dell’operazione di upload o download tramite gli eventi *onUploadProgress* e *onDownloadProgress* che possono essere intercettati definendo la corrispondente funzione sull’istanza di *url*, come mostrato nell’esempio seguente:

```
var u = app.fs.url("https://server.com/my-file.jpg");
yield u.download();
u.onDownloadProgress = function (byteTransferred, total) {
  if (view.cancelOperation)
    return false; // to cancel download, return false
};
```

Consumare Web API

In questo paragrafo vediamo come utilizzare una Web API REST generica, così definita in quanto non utilizza uno specifico standard semantico per le chiamate. In questi casi, il metodo di utilizzo è semplice: si crea un'istanza di *url* che rappresenta l'endpoint della Web API e si utilizzano i metodi *get* e *post* per recuperare o inviare dati.

Esecuzione di una chiamata get

Vediamo un esempio di come recuperare dati da una Web API REST. Verrà utilizzato un endpoint di test che espone i dati della tabella Employees del database NorthWind.

```
let endpoint = "https://examples.instantdevelopercloud.com/
                datamapdesignpatterns/Employee";
let u = app.fs.url(endpoint);
//
let options = {
  authentication : {
      username:"alladin",
      password:"opensesame"
  }
};
//
let resp = yield u.get(options);
let ok = resp.status === 200;
if (ok) {
  let data = JSON.parse(resp.body);
  let emplist = data.value;
  //
  for (let i = 0; i < emplist.length; i++) {
    let emp = emplist[i];
    console.log(emp);
  }
}
//
return ok;
```

Dopo aver effettuato la chiamata *get* all'endpoint, passando i parametri per l'autenticazione *basic* richiesti dall'endpoint, è opportuno controllare il codice di ritorno che solitamente è 200 quando l'operazione ha avuto successo.

A questo punto la risposta, che è in formato JSON come comunemente accade, viene trasformata in oggetto e viene estratto l'array dei dati degli impiegati. In questo esempio i dati vengono semplicemente stampati nella console, ma possono anche essere caricati in una datamap o in una collection.

Esecuzione di una chiamata post

Per modificare i dati vengono normalmente utilizzati i metodi *post*, *patch* e *delete*. Siccome stiamo trattando di Web API generiche, per sapere quale metodo utilizzare in base all'operazione richiesta occorre fare riferimento alla documentazione della Web API.

Nell'esempio seguente utilizziamo il metodo *post* per creare un nuovo impiegato, utilizzando sempre l'endpoint precedente.

```
let endpoint = "https://examples.instantdevelopercloud.com/
                datamapdesignpatterns/Employee";
let u = app.fs.url(endpoint);
//
let authdata = {
  username:"alladin",
  password:"opensesame"
};
let body = {
  EmployeeID : 444,
  FirstName : "Giuseppe",
  LastName : "Ferrari",
  Title : "Sig",
  Country : "Italia",
  Notes : "Corre veloce"
}
//
let resp = yield u.post({
  authentication: authdata,
  body: body
});
if (resp.status !== 201) {
  let data = JSON.parse(resp.body);
  console.log("errore", data.error);
}
```

Dopo aver inviati i dati del nuovo impiegato all'endpoint occorre controllare la risposta per verificare se l'operazione ha avuto successo o sono avvenuti degli errori.

Web API e documenti

Il modo migliore per utilizzare chiamate a Web API esterne è quello di inserire il codice di recupero dati o di invio delle modifiche all'interno di una classe documento.

In questo modo l'applicazione potrà interagire con il documento, senza avere conoscenza di come il documento viene *permanentizzato*, cioè letto o scritto in un data store. Si aumenta così il livello di incapsulamento del sistema.

I documenti favoriscono questa modalità di utilizzo delle Web API attraverso gli eventi *beforeLoad* e *onSave*.

L'evento *beforeLoad* permette di intercettare le chiamate ai metodi di caricamento dati (*loadByKey*, *loadCollection, collection.load)* e sostituirli con chiamate a Web API, restituendo poi i risultati trovati nel parametro *collection*. Il tipo di caricamento richiesto viene passato nel parametro *options.loadType*, mentre i filtri di caricamento sono presenti nelle proprietà del documento a cui viene notificato l'evento.

L'evento *onSave* permette di intercettare le fasi di salvataggio del documento e sostituirle con chiamate Web API. A seconda della modalità prevista dalla Web API, è possibile intercettare la fase di *beforeSave*, oppure quella di *inserting*, *updating* e *deleting.* Si ricorda di impostare a *true* la proprietà *options.skip* che permette di disabilitare il salvataggio nel database eseguito dal framework.

Esporre Web API

Vediamo adesso come implementare un servizio di Web API generico, utilizzando l'applicazione in esecuzione in un server nel cloud.

Sessioni REST ed evento onCommand

Il cuore dell'implementazione di un servizio di Web API in un'applicazione Instant Developer Cloud consiste nell'evento *app.onCommand*, che viene notificato quando l'applicazione viene richiamata specificando *mode=rest* nella query string.

Ad esempio, l'applicazione di esempio ToBuy può venire aperta in un browser tramite l'URL https://prod3-pro-gamma.instantdevelopercloud.com/ToBuyApp. Quando il browser effettua la chiamata, viene creata nel server una sessione browser che inizia notificando l'evento *app.onStart*.

L'applicazione ToBuy permette di caricare le immagini dei prodotti, quindi espone una Web API che può essere invocata dall'applicazione in esecuzione nei dispositivi quando viene scattata una foto. L'endpoint della WebAPI sarà il medesimo dell'applicazione e la query string deve contenere *mode=rest,* come mostrato nell'esempio seguente: https://prod3-pro-gamma.instantdevelopercloud.com/ToBuyApp?mode=rest&fn=xyz. Effettuando la chiamata, si vedrà il messaggio *no files* che indica il fatto che non sono stati rilevati file da caricare.

L'invocazione dell'applicazione indicando *mode=rest* sulla query string crea una nuova sessione nel server, una sessione di tipo REST che, appunto, serve per identificare il comando richiesto dalla Web API, per eseguirlo e per restituire i risultati al chiamante; a questo punto la sessione viene terminata. Il ciclo di vita di una sessione REST deve essere il più breve possibile in quanto ogni chiamata deve gestire una singola richiesta, nel minor tempo possibile, in modalità *stateless*.

L'evento *onCommand* specifica il parametro *request* che contiene tutti i dati passati dal chiamante, fra i quali:

- *request.query*: oggetto JavaScript che contiene i parametri query string della chiamata.

- *request.headers*: oggetto JavaScript che contiene gli header della chiamata.
- *request.cookies*: oggetto JavaScript che contiene i cookie della chiamata.
- *request.params*: oggetto JavaScript che contiene i campi della form se la chiamata è di tipo *post*.
- *request.files*: array di oggetti *File* caricati nel server dalla richiesta post di tipo *multipart*. I file caricati vengono salvati nella cartella *uploaded* del file system.
- *request.protocol, request.host, request.url, request.method*: proprietà della chiamata.
- *request.isBot*: vale *true* se la sessione REST è stata attivata a seguito di una chiamata da parte di un crawler di un motore di ricerca.

Nota: oltre a *mode=rest* sulla query string, esistono altre circostanze per cui viene chiamato l'evento *onCommand* di un'applicazione; potrebbe quindi essere necessario tenerne conto nel codice di gestione dell'evento. In particolare:

- Nel caso di un'applicazione installata in un dispositivo, se viene attivata tramite un link che contiene il protocollo personalizzato dell'app. Questo caso si può distinguere testando *app.runsLocally()*.
- Nel caso di una chiamata da parte di un crawler di un motore di ricerca. Questo caso si distingue testando *request.isBot*.
- Nel caso di una chiamata di tipo Web API ODATA, illustrata nei paragrafi seguenti. Per distinguere questo caso è possibile testare *app.isWebApiRequest().*

Rispondere ad una richiesta

La risposta ad una chiamata REST deve essere inviata entro 60 secondi (intervallo personalizzabile modificando *app.sessionTimeout*). Per rispondere occorre utilizzare il metodo *app.sendResponse* che consente di inviare un codice di risposta http e la risposta vera e propria, restituendo una stringa, un array buffer o un oggetto *File*. Specificando altri tipi di dati, essi verranno convertiti in stringa. Tramite il parametro *options* è infine possibile indicare il content-type della risposta e inviare un elenco di header di risposta.

Prima di rispondere può essere opportuno verificare i token di autenticazione ed eventualmente gestire o cancellare eventuali file caricati. Si noti che è possibile utilizzare *app.sendResponse* una sola volta e che dopo averlo utilizzato la sessione viene terminata. Deve pertanto essere l'ultima operazione eseguita dal codice dell'evento *onCommand*.

Vediamo adesso un semplice esempio di gestione di una risposta per una Web API che permette di sapere se il server è attivo e funzionante.

```
App.Session.prototype.onCommand = function (request)
{
  if (app.isWebApiRequest()) {
    // Gestione Web API ODATA
  }
  else if (app.runsLocally()) {
    // Gestione url protocol offline
  }
  else if (request.isBot) {
    // Gestione crawler motore di ricerca
  }
```

```
  else if (request.query.cmd === "hello") {
    // Gestione comando hello
    app.sendResponse(200, "Hello. Local time is: " +
                          app.locale.now().format("LLL"));
  }
  else {
    app.sendResponse(401, "Unknown command");
  }
};
```

Gestire i file caricati

Un importante caso d'uso delle sessioni REST è quello di accettare file caricati da dispositivi, come ad esempio fotografie, documenti, eccetera. A tal fine, le sessioni REST sono predisposte per salvare sul disco del server, nella cartella *uploaded* del file system, i file che vengono inviati nella richiesta REST se essa è in formato multipart.

Per salvare i file su disco, non vengono usati i loro nomi originali, che vengono invece sostituiti da nomi di file generati in formato *guid 36*. In questo modo non è possibile per chi carica i file sapere con quale nome verranno salvati.

I file caricati vengono passati all'evento *onCommand* tramite l'array *request.files* ed è quindi possibile manipolarli, cancellarli, spostarli in sottocartelle o in ogni altro modo. Si ricorda che la directory *uploaded* e le sue sotto-directory mantengono i file nella cache dei browser per un anno. Sono quindi la destinazione ideale per documenti caricati che devono permanere nelle cache dei browser.

La dimensione massima accettabile per un file caricato è di 50 MB.

Oltre a gestire i file caricati, una Web API può anche servire file al richiedente, specificando l'oggetto *File* da servire come secondo parametro del metodo *app.sendResponse*. In questi casi si consiglia di aggiungere l'header *Cache-Control* alla risposta in modo da poter fissare per un determinato tempo il file nella cache del dispositivo chiamante.

Questa modalità di servire i file può rappresentare un metodo di sicurezza estremo per garantire la massima privacy ai file del proprio file system. Normalmente non è possibile accedere ai file pubblici del server perché i nomi sono sconosciuti, ma una volta conosciuto il link, questo può essere comunicato ad altri.

Se invece il file viene servito tramite una Web API, inviandolo a partire da un file privato, esso non può essere acceduto se non tramite la Web API stessa, che può quindi effettuare tutti i controlli del caso sui parametri della richiesta, ad esempio su un token di autenticazione a scadenza ravvicinata.

Gestire i crawler dei motori di ricerca

Se l'URL dell'applicazione viene sottoposta ad un motore di ricerca, oppure viene linkata in un sito sottoposto ad un motore di ricerca, il crawler del motore inizierà a contattarla per recuperarne i contenuti da indicizzare.

In questi casi il framework di Instant Developer Cloud riconosce che la chiamata all'applicazione non proviene da un browser vero e proprio, quindi, invece che servire i file JavaScript che servono per iniziare la sessione browser, attiva una sessione REST per dare la possibilità all'applicazione di rispondere alla richiesta del crawler con un contenuto testuale.

Solitamente la richiesta viene gestita caricando una lista di dati o il dettaglio di un documento. Questi dati vengono usati per costruire una pagina HTML fittizia che viene infine data come risposta alla richiesta del crawler e, in questo modo, il motore di ricerca potrà indicizzare le informazioni in essa contenute.

Quando l'utente cerca le informazioni tramite il motore di ricerca, verrà utilizzato come link di attivazione della pagina proprio quello dell'applicazione stessa, che, questa volta, attiverà una sessione browser vera e propria.

Per maggiori informazioni su queste funzionalità, vedi il paragrafo Esempio di indicizzazione sui motori di ricerca.

Testare la Web API esposta

Per testare le Web API si consiglia di utilizzare prima l'applicazione in anteprima nell'IDE e poi l'applicazione installata in un server di produzione.

Per testare l'applicazione in anteprima nell'IDE è necessario generare un link che rappresenta l'endpoint di connessione della Web API esposto dall'applicazione in anteprima. A tal fine si consiglia di inserire nell'evento *app.onStart* il seguente codice:

```
if (App.ide) {
  let urlParts = app.requestConnectionUrl().split("/");
  let url = urlParts[0] + "//" + urlParts[2] + "/" + urlParts[6] + "/" +
            urlParts[7] + "/" + urlParts[8] + "?mode=rest";
  console.log("WebAPI endpoint", url);
}
```

Lanciando l'applicazione in anteprima, nella console dell'IDE apparirà il link che rappresenta l'endpoint specifico della Web API per questa sessione di anteprima. A questo punto è possibile utilizzare questa informazione per effettuare le chiamate da un altro browser oppure tramite PostMan o altri software di test per le Web API.

Durante il test della Web API nell'IDE si deve considerare che non viene generata una sessione REST aggiuntiva, ma la sessione utilizzata è la stessa sessione browser che mostra l'interfaccia utente dell'anteprima. È quindi possibile interagire con le videate aperte e con la console di debug.

La seconda fase di test consiste nell'installazione dell'applicazione in un server di produzione per poter testare l'endpoint effettivo. In questa fase si consiglia di attivare il log strutturato delle sessioni e di inserire nell'evento *onCommand* la riga di codice che dà il nome alla sessione, come la seguente:

```
app.sessionName = "(A) API request " + parseInt(Math.random() * 100000);
```

In questo modo ogni chiamata al server verrà aggiunta al log strutturato in una sessione separata.

Web API in formato OData

Nei paragrafi precedenti abbiamo visto come consumare o esporre Web API REST scrivendo manualmente il codice di gestione delle chiamate. Ora vedremo come automatizzare questo processo tramite lo standard OData.

OData è uno standard che definisce un insieme di best practice per la progettazione ed il consumo di Web API REST. Delineando header, status, URL e metodi HTTP in base all'operazione che si vuole completare, lo standard permette di focalizzarsi sulle funzionalità degli applicativi piuttosto che sulla loro interazione con la Web API. Definendo un linguaggio comune, le Web API OData possono essere consumate da qualsiasi client che adotti lo standard.

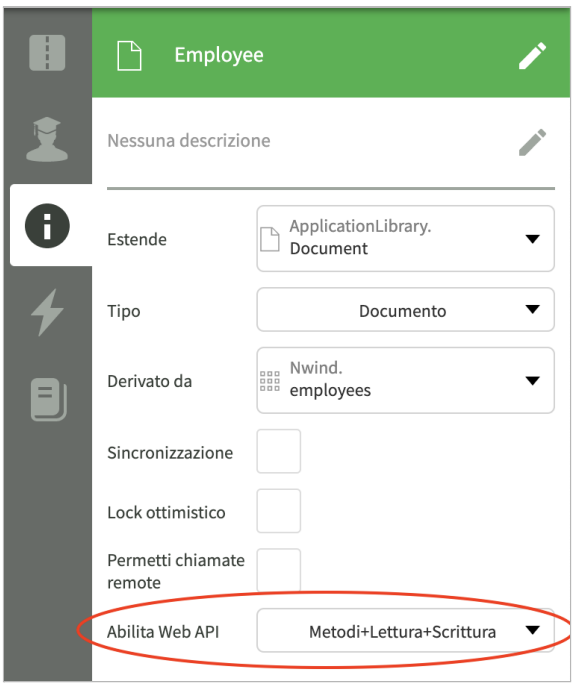
Instant Developer Cloud implementa lo standard OData in maniera completa, infatti:

1) Contiene un framework di generazione automatica di Web API OData a partire dai documenti contenuti nell'applicazione.
2) Permette di importare una Web API OData esistente generando automaticamente le classi di documento corrispondenti per consumarla.

Esposizione di documenti via Web API OData

Esporre un documento del progetto via OData è molto semplice: è sufficiente impostare la proprietà *Abilità Web API* a design time nella classe documento. I possibili valori sono:

- *Metodi*: sarà possibile solo chiamare i metodi del documento specificatamente abilitati per l'utilizzo tramite Web API.
- *Metodi+Lettura*: sarà possibile effettuare delle query di lettura dati estraendo documenti e collection, oltre che chiamare i metodi specificatamente abilitati per l'utilizzo tramite Web API.
- *Metodi+Lettura+Scrittura*: sarà possibile effettuare delle query di lettura dati, inserire, modificare e cancellare i documenti, oltre che chiamare i metodi specificatamente abilitati per l'utilizzo tramite Web API.



Autenticazione e autorizzazioni

Abilitando le Web API per una classe di documento, esse saranno subito attive al momento dell'installazione dell'applicazione su un server. È quindi necessario aggiungere un livello di autenticazione per controllare che il chiamante abbia il permesso di accedere ai documenti esposti. A tal fine è possibile implementare l'evento *onCommand*, che viene notificato prima di procedere con l'elaborazione della richiesta, come mostrato nell'esempio seguente:

```
App.Session.prototype.onCommand = function (request)
{
  if (app.isWebApiRequest()) {
    let token = "Basic " + App.Utils.toBase64(app, "alladin:opensesame");
    if (request.headers.authorization !== token) {
      app.sendResponse(401, { error: {
            message: "Authorization token is not valid"}
      }, {contentType: "application/json"});
    }
  }
};
```

Per disabilitare la gestione automatica della richiesta Web API è sufficiente chiamare il metodo *app.sendResponse*. Nei casi reali si consiglia di verificare il token di autenticazione in base ad una lista di token validi, che possono essere memorizzati anche su una tabella del database.

Ai token di autorizzazione possono essere associati dei parametri da utilizzare per condizionare le chiamate di lettura o scrittura dei dati. Se, ad esempio, volessimo limitare la possibilità di caricare gli ordini dal database solo a quelli dell'utente associato al token, potremmo operare come segue:

1) Nell'evento *onCommand*, viene impostata una proprietà di sessione (*app.employeeID)* che identifica l'ID dell'utente per cui limitare l'estrazione degli ordini.
2) Nell'evento *beforeLoad* del documento *Order*, nel caso di chiamata Web API viene utilizzata tale proprietà per impostare un filtro, come mostrato nell'esempio seguente:

```
App.BE.Order.prototype.beforeLoad = function (collection, options)
{
  if (app.isWebApiRequest())
    this.employeeID = app.employeeID;
};
```

Nell'evento *beforeLoad*, infatti, le proprietà del documento *this* rappresentano i filtri QBE che verranno applicati nella fase di caricamento.

Si noti che, in base al tipo di applicazione, può essere più appropriato definire una classe documento apposita che contiene tutti i metodi di accesso e di modifica dei dati esposti via Web API, invece che esporre i documenti applicativi veri e propri.

In questo modo è più facile avere un quadro completo dei possibili accessi dall'esterno, anche se si dovrà implementare un metodo per ogni operazione consentita.

Il compromesso fra velocità di implementazione e controllo degli accessi può essere liberamente gestito miscelando opportunamente le tecniche illustrate in questo paragrafo: esposizione diretta dei documenti applicativi o esposizione indiretta tramite un documento di interfaccia.

Si ricorda che, in ogni caso, è necessario gestire un livello di autenticazione generale tramite l'evento *onCommand*.

Testare la Web API OData esposta

Nel caso di Web API OData non è possibile eseguire il test tramite l'anteprima nell'IDE, ma è invece necessario installare l'applicazione in un server di test o di produzione.

In questo modo sarà possibile importare la Web API in un client OData specificando l'endpoint, cioè la URL dell'installazione o la URL dei metadati, che è uguale all'endpoint seguito da $metadata. Ad esempio:

https://examples.instantdevelopercloud.com/datamapdesignpatterns/$metadata

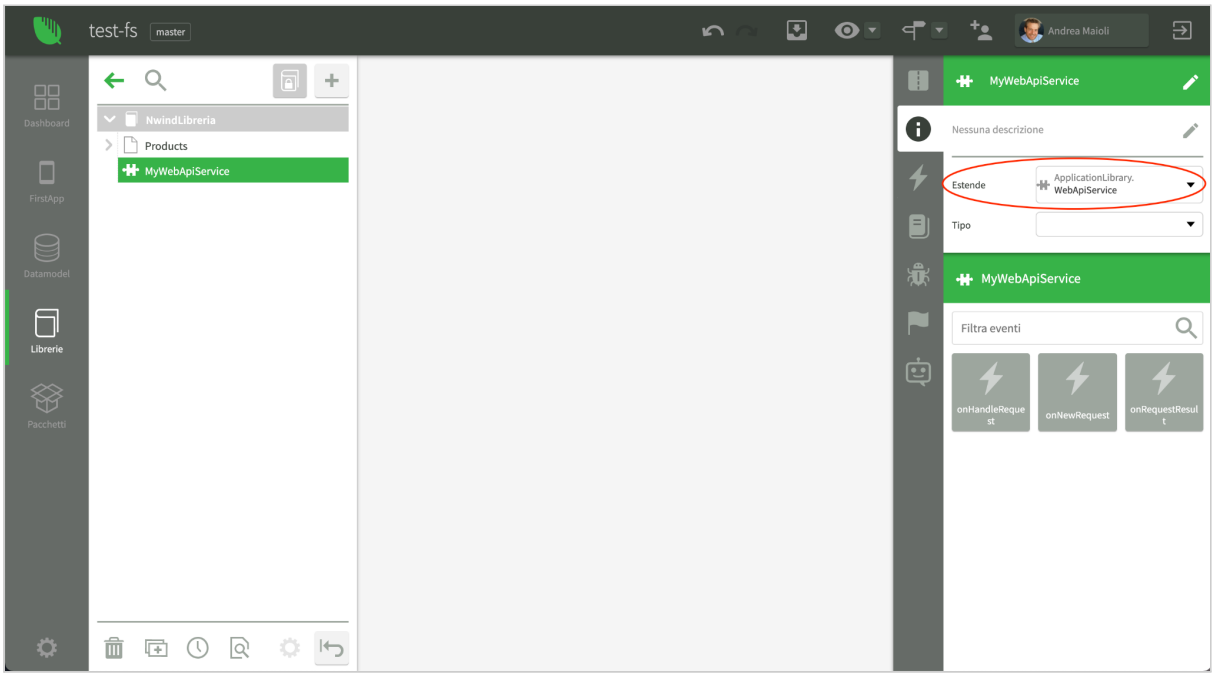
Le eccezioni di funzionamento e i warning potranno essere seguiti nel log strutturato dell'installazione, una volta attivato e configurato specificando *app.sessioneName* nell'evento *app.onCommand*.

Personalizzare il framework di gestione delle Web API OData

Abbiamo visto che la generazione delle Web API OData richiede solo l'attivazione del servizio per i documenti che la richiedono. Tutta la gestione delle richieste e delle relative risposte è quindi a carico del framework di Instant Developer Cloud.

Abbiamo visto anche che è possibile interagire con la richiesta in arrivo tramite l'evento *app.onCommand*, prima che il framework la gestisca e questo ci dà la possibilità, ad esempio, di gestire l'autenticazione.

Può essere necessario personalizzare più profondamente il funzionamento del framework, ad esempio loggando ogni richiesta di un certo tipo, oppure intercettando la risposta prima che venga inviata al chiamante. A tal fine è possibile inserire nel proprio progetto una classe di codice che estende *App.WebApiService*.



Per questa classe sono disponibili tre eventi:

- *onNewRequest*: notificato prima di processare la richiesta. È analogo ad *app.onCommand* e permette di gestire la richiesta prima di passarla al framework OData.
- *onHandleRequest*: notificato prima di gestire una richiesta OData già validata. All'evento vengono passati il tipo di azione e i parametri. È quindi possibile modificare i parametri dell'azione, gestire l'azione in autonomia o anche più semplicemente loggare la richiesta.
- *onRequestResult*: notificato prima di inviare la risposta al chiamante. Può essere usato per analizzare la risposta, inserire header o gestire l'invio in autonomia.

Vediamo un esempio dei primi due eventi precedenti. Nel primo esempio utilizziamo *onNewRequest* per controllare i parametri di autenticazione della richiesta.

```
App.NwindLibreria.MyWebApiService.prototype.onNewRequest =
function (request)
{
  if (!request.headers.Authorization) {
    this.sendError(401, "non autorizzato");
  }
};
```

Il metodo *WebApiService.sendError* viene usato per restituire un errore di gestione della richiesta, mentre *WebApiService.sendResponse* serve per inviare la risposta al chiamante. In entrambi i casi si disabilita la gestione automatica da parte del framework, in quanto, una volta fornita una risposta o segnalato un errore, la sessione REST viene terminata.

Nel secondo esempio utilizziamo *onHandleRequest* per attivare la paginazione dei risultati anche se non è stata richiesta dal chiamante.

```
App.NwindLibreria.MyWebApiService.prototype.onHandleRequest =
function (request, action)
{
  if (action.type === App.WebApiService.actionTypes.loadCollection) {
    action.options.pagingMode = action.options.pagingMode ||
      App.DataMap.dataPagingModes.offset;
    action.options.pageSize = action.options.pageSize || 30;
    action.options.preCount = true;
  }
};
```

Vediamo infine come fare in modo che il framework OData utilizzi la nostra classe per notificare gli eventi. Ciò avviene nell'evento *app.onCommand*, impostando la proprietà *webApiService* della classe *App.Document*, tramite la riga di codice seguente:

```
App.Session.prototype.onCommand = function (request)
{
  App.Document.webApiService = App.NwindLibreria.MyWebApiService;
  ...
}
```

È possibile impostare direttamente la classe oppure crearne un'istanza e usare questa. Il secondo caso è utile se l'istanza deve essere inizializzata in modo personalizzato.

Comandi OData non implementati

Il protocollo OData definisce una serie estesa di funzionalità, adatte a gestire un insieme di casistiche complesse. Il framework OData di Instant Developer Cloud implementa una parte delle specifiche, restituendo l'errore *501 - request type not implemented* in caso diverso.

In generale, la parte implementata riflette le funzionalità di manipolazione dei documenti descritte nel manuale Document Orientation e le funzionalità non implementate possono essere ottenute tramite una combinazione di quelle disponibili.
Ad esempio, OData permette di caricare un documento e una specifica collection figlia con una sola chiamata. Se volessimo caricare l'entità Order con ID pari a 10540 e contemporaneamente avere il contenuto della collection figlia OrderDetails, OData prevede una chiamata con questo formato:

```
https://<endpoint>/Order(10540)?$expand=OrderDetails
```

Questo comando, richiedendo l'uso di una funzionalità non implementata da Instant Developer Cloud, restituisce l'errore *501 - request type not implemented*.

Esso può essere sostituito in due modi. Il primo consiste nel caricare la testata specificando il parametro *childLevel* per caricare anche tutte le collection di primo livello:

```
https://<endpoint>/Order(10540)?childLevel=1
```

In alternativa è possibile effettuare due chiamate, una per caricare la testata:

```
https://<endpoint>/Order(10540)
```

e l'altra per caricare i dettagli dell'ordine indicando come filtro il campo *OrderID*:

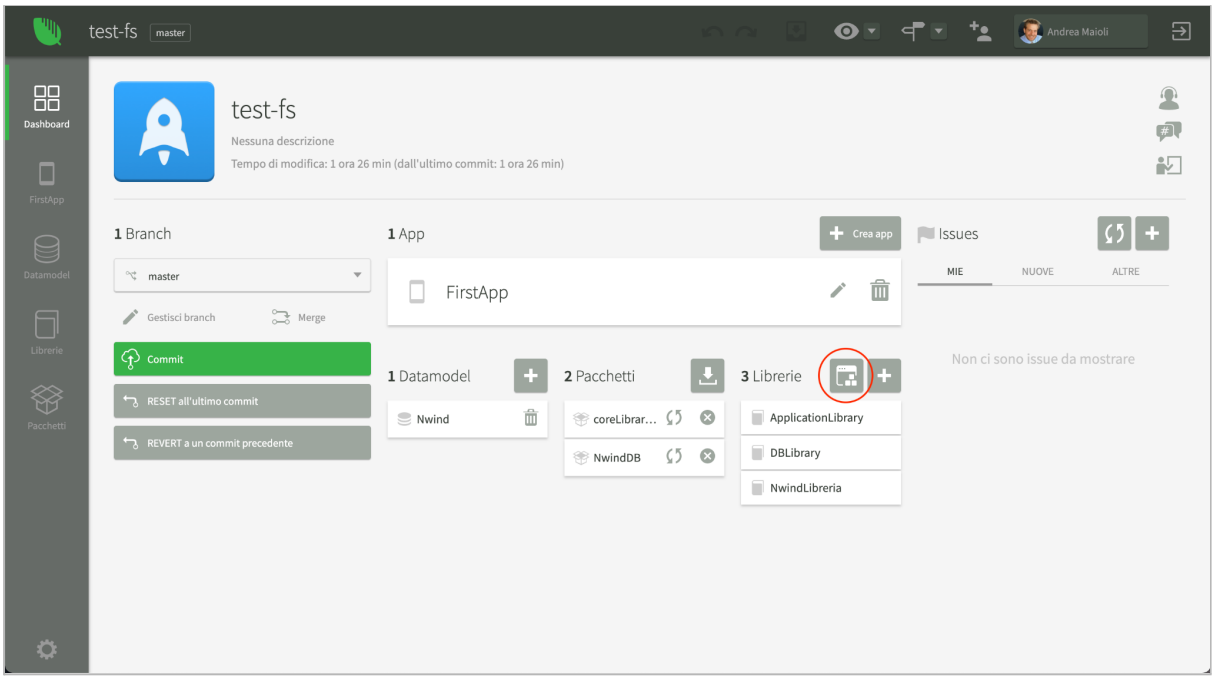
```
https://<endpoint>/OrderDetail?$count=true&$filter=OrderID%20eq%2010540
```

Questo tipo di problematiche può accadere se si accede alla Web API esposta direttamente tramite comandi https; se invece essa viene importata in un altro progetto questo non accadrà in quanto l'interfaccia dei documenti utilizzerà solo i metodi implementati.

Importazione e configurazione di Web API OData

In questo paragrafo vediamo come utilizzare una Web API esistente in formato OData in un progetto.

Anche in questo caso l'operazione è molto semplice. Basta inserire la URL dell'endpoint (che deve usare il protocollo https) nella funzione di importazione API che si trova nella pagina principale dell'IDE.



La URL dell'endpoint serve alla funzione di importazione per recuperare i metadati della Web API OData; dopo averla inserita verrà mostrato un elenco di entità che è possibile importare.

La procedura di importazione termina con la creazione nel progetto delle classi di documento relative alle entità importate. A questo punto è possibile utilizzarle come se fossero normali documenti: è quindi possibile aggiungere proprietà unbound e metodi, oltre a gestire eventi.

Ma poiché i documenti che gestiscono le entità OData non sono basati direttamente su tabelle del database, le funzionalità specifiche dell'accesso al database non sono disponibili. Ad esempio, non è possibile creare una query di caricamento del documento, oppure gestire proprietà derivate. I cicli di caricamento e salvataggio, tuttavia, notificano gli stessi eventi, a parte l'evento *onSave* che non serve in quanto il salvataggio vero e proprio avviene all'interno del server che espone la Web API.

Autenticazione e personalizzazione degli endpoint

In alcuni casi i documenti importati non saranno subito utilizzabili perché la Web API può richiedere un header di autenticazione, oppure perché l'endpoint di produzione può essere diverso da quello da cui è stata importata l'entità.

Come nel caso precedente, anche il framework che gestisce le chiamate a Web API OData può essere personalizzato, inserendo una apposita classe nel proprio progetto che estende *WebApiOData*.

A questo punto è necessario comunicare ai documenti che gestiscono le entità importate la classe da utilizzare per la gestione delle chiamate. A tal fine si consiglia di inserire in questa classe un metodo statico *init*, in cui inserire l'impostazione delle proprietà *webApiEndpoint* dei vari documenti, come mostrato nell'esempio di codice seguente:

```
App.NwindLibreria.MyODataAPI.init = function (app)
{
  App.NwindLibreria.Products.webApiEndpoint = {
    type : "NwindLibreria.MyODataAPI",
    url : "https://prod3-pro-gamma.instantdevelopercloud.com/academy"
  };
  ...
};
```

La proprietà *webApiEndpoint* di una classe documento è un oggetto JavaScript che deve contenere le proprietà:

- *type*: il nome della classe che gestisce la Web API importata.
- *url*: endpoint da utilizzare per questa entità.

È necessario specificare questa proprietà per ogni classe documento che gestisce un'entità importata.

A questo punto, nell'evento *app.onStart* è necessario richiamare il metodo statico *init* per attivare le impostazioni.

```
App.Session.prototype.onStart = function (request)
{
  App.NwindLibreria.MyODataAPI.init(app);
  ...
}
```

Per gestire l'autenticazione è ora possibile inserire l'evento *beforeHttpRequest* nella classe di gestione della Web API per inserire gli opportuni header, come mostrato nel seguente esempio:

```
App.NwindLibreria.MyODataAPI.prototype.beforeHttpRequest =
function (request, action)
{
  request.options.headers.Authorization = "Basic " +
          App.Utils.toBase64(app, "alladin:opensesame");
};
```

Utilizzare Web API Instant Developer Foundation

Nei paragrafi precedenti abbiamo visto come consumare o esporre Web API REST OData. Ora vedremo come importare Web API implementate in applicazioni Instant Developer Foundation.

Instant Developer Foundation è la piattaforma di sviluppo per creare applicazioni di business al passo con gli ultimi trend e con le best practice del momento, potenziando al massimo l'efficienza e la produttività.

In questo paragrafo parleremo solo dei passi da seguire all'interno delle applicazioni Instant Developer Cloud tralasciando quelli per la costruzione della Web API nell'ambiente Foundation.

Il primo passo è l'importazione della Web API, che avviene come già descritto nel caso delle Web API OData. Anche in questo caso occorre inserire la URL dell'endpoint, che deve sempre avere protocollo https. L'importatore utilizza poi un file di metadati, che si deve trovare sempre alla URL <endpoint>/$metadata. Come esempio è possibile vedere il seguente endpoint: https://progamma.com/NwindWebAPI/$metadata.

Dopo aver importato le classi documento, si dovrà personalizzare il framework di gestione aggiungendo una propria classe, come nel caso precedente. Questa volta però la classe deve estendere *WebApiFoundation* invece di *WebApiOData.* La procedura di personalizzazione continua poi come nel caso precedente.

A questo punto è possibile utilizzare i documenti esposti dall'applicazione Foundation come ogni altro documento importato nell'ambiente Instant Developer Cloud.