Delphi DataSnap
開發手册
embarcadero

# 目錄

第 1 章 開發 DataSnap/RESTful 應用系統 ........ 8

1-1 開發 DataSnap/RESTful 服務伺服器 ........ 8

1-2 開發用戶端 Win32 應用程式 ........ 18

第 2 章 JSON 程式設計 ........ 29

2-1 JSON 是什麼? ........ 29

2-2 VCL 框架中支援 JSON 的類別 ........ 34

2-2-1 TJSONAncestor 類別 ........ 37

2-2-2 TJSONValue 類別 ........ 37

2-2-3 TJSONPair 類別 ........ 38

2-2-4 TJSONString 類別 ........ 39

2-2-5 TJSONObject 類別 ........ 40

2-2-6 TJSONNumber 類別 ........ 41

2-2-7 TJSONArray 類別 ........ 42

2-3 JSON 程式設計 ........ 46

2-3-1 開發數值物件伺服器 ........ 46

AddEmployee 方法的實作 ........ 48

GetEmployeeJ 方法的實作 ........ 49

GetAllEmployeesJ 方法的實作 ........ 50

2-3-2 開發範例用戶端 ........ 51

2-4 使用 JSON 封裝和傳遞資料 ........ 57

2-4-1 開發 DataSnap REST 伺服器 ........ 57

2-4-2 開發範例用戶端 ........ 59

版權所有 請勿翻印

2-5 結論 .......... 62
第 3 章 DataSnap/REST 伺服器的授權和認證 .......... 64
3-1 開發 DataSnap/RESTful Web 用戶端應用程式 .......... 64
3-2 認證和授權 .......... 71
傳遞使用者登錄資訊 .......... 72
使用 Delphi 用戶端傳遞認證資訊 .......... 72
驗證使用者 .......... 73
授權使用者 .......... 74
認證和授權用戶端範例 .......... 77
使用 JavaScript 用戶端傳遞認證資訊 .......... 81
使用 TDSAuthenticationManager 的特性值編輯器授權 .......... 86
使用程式碼註解授權 .......... 88
版權所有 請勿翻印
第 4 章 DataSnap 回叫機制 .......... 91
4-1 DataSnap 基本的回叫機制 .......... 91
範例 DataSnap 伺服器 .......... 92
同步用戶端 .......... 93
回叫用戶端 .......... 94
TDBXCallback 抽象類別 .......... 96
TDSCallbackMethod 實體類別 .......... 96
4-2 進階 DataSnap 回叫功能 .......... 99
開發回叫 DataSnap 伺服器 .......... 100
開發回叫 DataSnap 用戶端 .......... 104
不同用戶端藉由回叫功能溝通 .......... 109

修改 DataSnap 伺服器 ................................................ 110

修改 DataSnap 用戶端應用程式 .................................... 111

開發輕薄型回叫 DataSnap 用戶端 ................................ 118

4-3 結論 ........................................................................ 124

第 5 章 使用 DataSnap 過濾器 .................................................. 125

5-1 使用內建的過濾器 ........................................................ 125

5-1-1 建立 DataSnap 過濾器伺服器 ...................................... 125

5-1-2 建立使用 DataSnap 過濾器的用戶端應用程式 .............. 127

5-2 開發客製化過濾器 ........................................................ 131

使用客製化過濾器 ........................................................ 136

5-3 使用 DataSnap 的請求過濾器 ........................................ 139

5-3-1 請求過濾器的種類 ........................................................ 140

5-3-2 使用請求過濾器 ............................................................ 142

藉由 TDSRestConnection 元使用請求過濾器 .................. 143

5-4 結論 ........................................................................ 149

第 6 章 DataSnap 生命週期和管理功能 .......................................... 150

6-1 DataSnap 伺服端服務的生命週期 .................................... 150

6-1-1 Server 生命週期 .......................................................... 151

6-1-2 Session 生命週期 ........................................................ 154

6-1-3 Invocation 生命週期 .................................................... 158

6-2 DataSnap 管理功能 ........................................................ 159

6-3 結論 ........................................................................ 169

第 7 章 開發移動式 DataSnap 用戶端 .......................................... 170

版權所有 請勿翻印

7-1 DataSnap Mobile Connector ........ 170
7-2 開發 Android 用戶端 ........ 175
7-2-1 建立 DataSnap 伺服器 ........ 176
7-2-2 建立 Android 用戶端 ........ 179
7-3 開發 iOS 用戶端 ........ 184
7-4 結論 ........ 190
第 8 章 DataSnap 監督功能 ........ 191
8-1 DataSnap 10.3 新增監督功能 ........ 191
8-1-1 DataSnap 伺服端監督功能 ........ 191
8-2 DataSnap Session 功能 ........ 195
8-3 TCP 連結監督功能 ........ 200
8-3-1 使用 TCP 連結監督功能 ........ 202
8-4 KeepAlive 功能 ........ 207
8-5 結論 ........ 209
開發高效率 DataSnap 篇 ........ 210
第 9 章 使用 FireDAC 開發 DataSnap 應用系統 ........ 212
9-1 開發可查詢的 FireDAC DataSnap 系統 ........ 214
9-1-1 開發 DataSnap 伺服器 ........ 214
9-1-2 開發 DataSnap 客戶端 ........ 218
9-1-3 開發 RESTful DataSnap 伺服器 ........ 223
9-1-4 開發 RESTful 手機客戶端 ........ 225
9-1-5 如何更新旅館資料 ........ 228
修改 FireDAC DataSnap 伺服器 ........ 228

版權所有 請勿翻印

修改 FireDAC DataSnap 手機客戶端 ............................ 231

9-2 開發可異動多資料表的 FireDAC DataSnap 系統 .................. 235

9-2-1 修改 FireDAC DataSnap 伺服器 .............................. 236

9-2-2 修改手機客戶端 ...................................................... 239

9-2-3 使用 JSON 更新資料 ................................................ 245

修改範例 DataSnap 伺服器 ........................................... 246

修改範例客戶端 .......................................................... 249

9-3 使用 TFDJSONDataSets 功能 ........................................... 251

9-3-1 開發 RESTful DataSnap 伺服器 ............................... 255

9-3-2 開發 RESTful 客戶端 ........................................ 257

9-3-3 開發 RESTful 多資料表查詢 .............................. 260

修改範例 DataSnap 伺服器 ........................................... 260

修改範例客戶端 .......................................................... 261

9-3-4 開發 CRUD 功能伺服端 ...................................... 262

9-3-4 開發 CRUD 功能客戶端 ...................................... 264

第 10 章 開發安全，高效率的 DataSnap 應用系統 ...................... 269

10-1 開發 DLL 型態的 DataSnap 伺服器 ................................... 270

10-1-1 開發 ISAPI 型態的 DataSnap 伺服器 ........................ 270

10-1-2 部署 ISAPI DataSnap 伺服器 ................................. 276

10-1-2 開發客戶端 ......................................................... 287

10-2 DataSnap 伺服器效能比較 ................................................ 291

ApacheBench ........................................................... 291

WeigHttp ................................................................. 295

版權所有 請勿翻印

Apache JMeter........................................................ 296

10-3 調校效能 ........................................................... 299

調校 EXE 型態的 DataSnap 伺服器 ..................................... 299

EXE 型態的 DataSnap 伺服器改良版 1 .......................... 302

EXE 型態的 DataSnap 伺服器改良版 2 .......................... 304

EXE 型態的 DataSnap 伺服器改良版 3 .......................... 308

調校 DLL 型態的 DataSnap 伺服器 ...................................... 312

DLL 型態的 DataSnap 伺服器改良版 4........................... 312

DLL 型態的 DataSnap 伺服器改良版 5........................... 313

10-4 改善程式碼調校效能 ........................................................ 316

10-4-1 開啟資料庫連結池............................................. 316

10-4-2 結合 ArrayDML...................................................... 318

DLL 型態的 DataSnap 伺服器改良版 5-結合 ArrayDML.... 318

10-5 結論 ................................................................. 322

版權所有 請勿翻印

本書的範例程式請至下列網頁下載:

http://embarcadero.qcomgroup.com.tw/download/dds.zip

# 第1章 開發 DataSnap/RESTful應用系統

Delphi/BCB 的 DataSnap 技術早期稱為 Midas，Midas 主要是使用來開發多層分散式應用系統的技術，它是以 COM 技術為核心發展出來的多層框架，隨著 PC 資訊技術不斷的演進，DataSnap 也從基於 COM 核心的框架發展到以 JSON 和 REST 技術的分散式架構。使用 JSON 和 REST 架構做為 DataSnap 核心技術的好處是 JSON 和 REST 都是可以跨平台的標準，JSON 提供了跨平台交換資料的標準格式，而 REST 更是基於 HTTP 通訊協定的技術。因此使用 Delphi 10.3 開發的 DataSnap/RESTful 伺服器雖然是執行於 Windows 平台，但卻可以服務所有平台的用戶端的呼叫和請求。

版權所有 請勿翻印

從本章開始我們將討論如何使用 Delphi 10.3 開發基於 JSON 和 REST 技術的分散式像應用系統，我們將從如何開發 DataSnap/RESTful 伺服器，DataSnap/RESTful 用戶端談起，然後一直討論到進階的 DataSnap 技術，現在讓我們從如何開發 DataSnap/RESTful 伺服器說起。

## 1-1 開發 DataSnap/RESTful 服務伺服器

在 Delphi 整合發展環境中點選 File|New|Other...啟動 New Items 對話盒並且在 DataSnap Server 選項中選擇建立 DataSnap Server 圖像，如下圖所示:

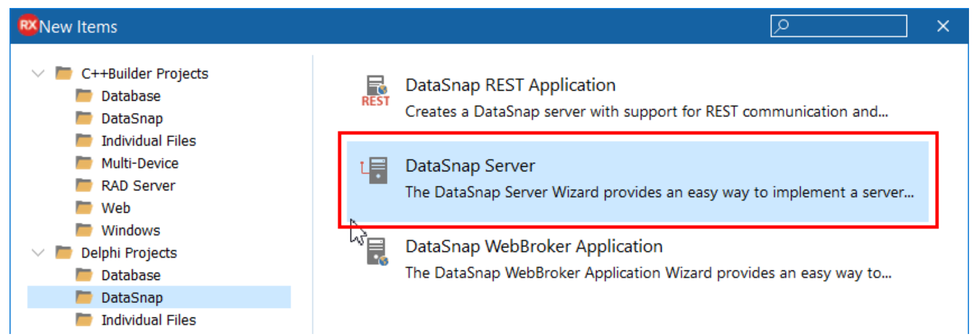
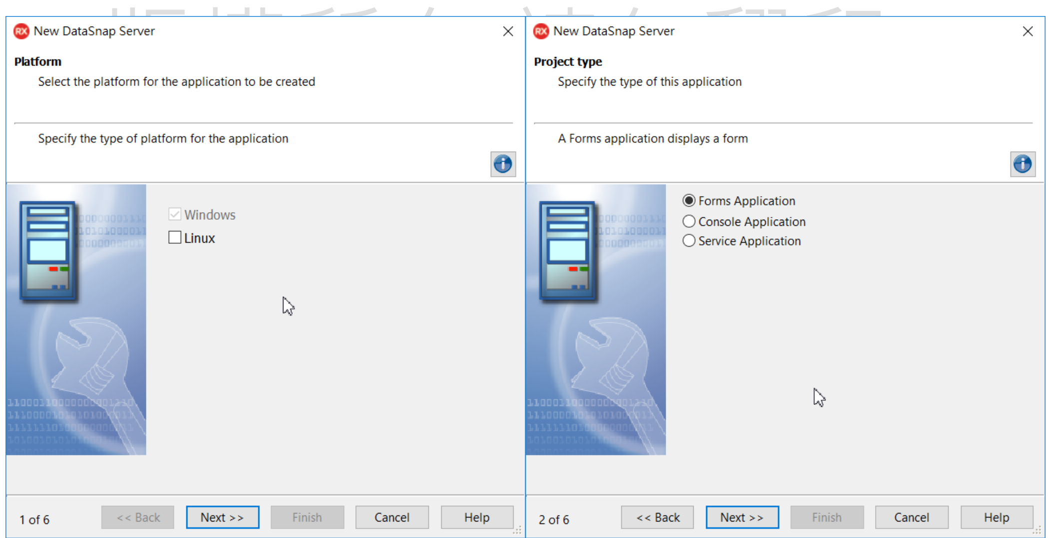



圖 1-1 建立 DataSnap 伺服器應用程式

點選了 DataSnap Server 圖像之後 Delphi 會開始詢問開發人員如何設定此 DataSnap 伺服器。首先 DataSnap 精靈會詢問要建立什麼型態的 DataSnap 伺服器，Delphi 10.3 提供三種不同的伺服器型態，分別是:

| 伺服器型態 | 說明 |
|---|---|
| VCL Forms Application | 使用VCL圖形使用者介面框架的應用程式 |
| Console Application | 主控程式型態的伺服器 |
| Service Application | Windows服務型態的伺服器 |

為了說明方便起見，讓我們在這選擇建立 VCL 表單型態的伺服器，如下所示:



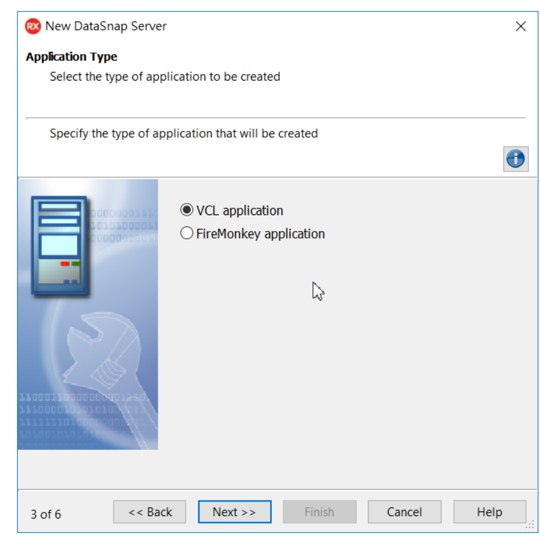



圖 1-2 選擇建立 DataSnap 伺服器的型態

接著 DataSnap 精靈會詢問自動建立的伺服器包含的基礎功能，例如使用什麼通訊協定?是否需要使用安全認證功能? 是否需要建立範例服務方法? 在這裡讓我們點選對話盒下方的 Select all 勾選盒要求建立所有的基礎功能，如下圖所示:

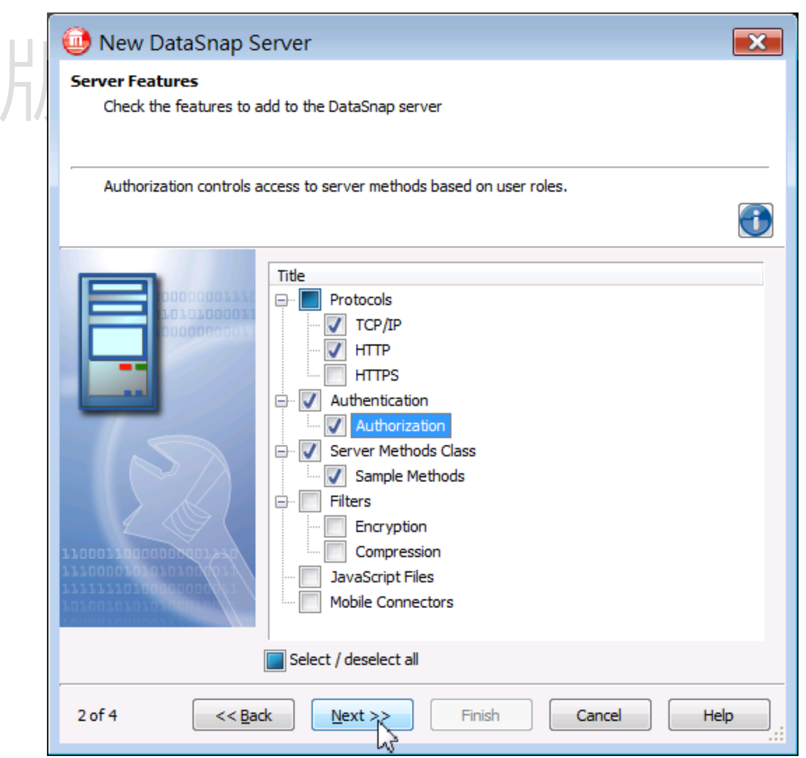



圖 1-3 選擇自動建立的伺服器功能

圖 1-3 中的 Filters，JavaScript Files 和 Mobile Connectors 等功能會在稍後的章節中說明，讀者現在可以暫時不用擔心這些選項。

接著 DataSnap 精靈會詢問在上一步驟選擇使用的通訊協定的通信埠，在內定上 TCP/IP 通訊協定是使用通信埠 211，HTTP 通訊協定是使用通信埠 8080，開發人員可以設定其他的通信埠，或是點選對話盒中 Find Open Port 按鈕讓 DataSnap 精靈自動搜尋其他可使用的通信埠，如下圖所示:

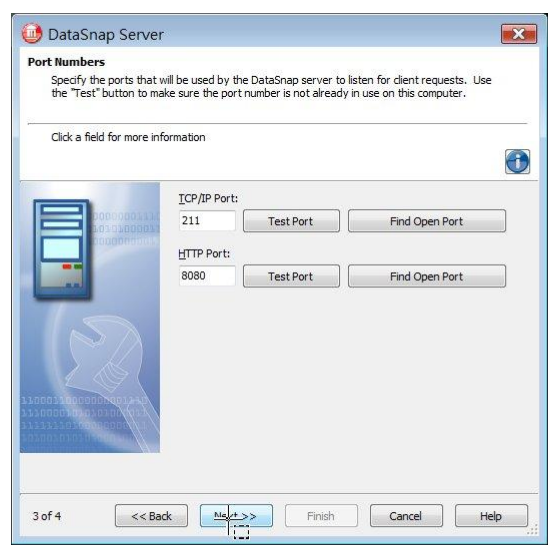



圖 1-4 選擇使用的通信埠

版權所有 請勿翻印

最後 DataSnap 精靈會詢問伺服器輸出服務的父代類別，可以做為父代類別的分別是:

| 父代類別 | 說明 |
| --- | --- |
| TComponent | 如果開發人員想使用最少的資源輸出服務方法，就選擇此類別 |
| TDataModule | 如果開發人員習慣使用資料模組或是需要把舊的資料模組昇級，就可以選擇這個類別 |
| TDSServerModule | 最典型的父代類別，如果開發人員沒有特別的昇級考量或是建立新的DataSnap伺服器，那麼請儘量選擇使用這個類別。 |

在這裡讓我們選擇使用 TDSServerModule 如下圖所示:

圖 1-5 選擇伺服器服務類別的父代類別

下表列出了選擇使用不同的父代類別所產生的程式碼差異，例如如果選擇使用 TComponent 做為父代類別，那麼就需要使用**{$METHODINFO ON}**和**{$METHODINFO OFF}**這兩個編譯器指令，因為使用這兩個編譯器指令的類別才會由 Delphi 編譯器自動產生 RTTI 資訊以自動輸出類別中的方法。

版權所有 請勿翻印

| 使用 TDSServerModule 父代類別 | 使用 TComponent 父代類別 |
|---|---|
| TServerMethods5 = class(TDSServerModule)<br>  private<br>    { Private declarations }<br>  public<br>    { Public declarations }<br>    function EchoString(Value: string):<br>string;<br>    function ReverseString(Value: string):<br>string;<br>  end; | **{$METHODINFO ON}**<br>  TServerMethods5 = class(TComponent)<br>  private<br>    { Private declarations }<br>  public<br>    { Public declarations }<br>    function EchoString(Value: string):<br>string;<br>    function ReverseString(Value: string):<br>string;<br>  end;<br>**{$METHODINFO OFF}** |

點選 Finish 按鈕之後，DataSnap 精靈便會自動幫助我們根據剛才在 DataSnap 精靈中進行的設定來產生專案，例如在下面的專案經理中我們把這個專案命名為 pDataSnapDemoServer，並且分別儲存其中的檔案名稱如下所示:

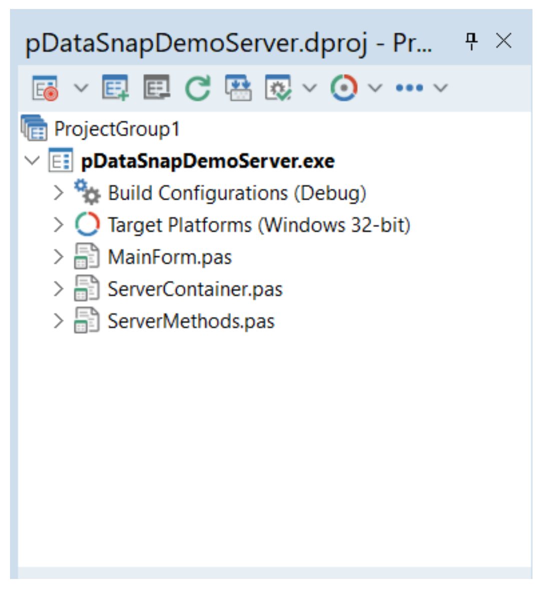



圖 1-6 DataSnap 精靈自動建立的專案

如果現在我們開啟專案中的 ServerContainer 程式單元就可以看到類似下圖的結果，在 ServerContainer 中包含了數個不同的元件，這些元件的用途如下所述:

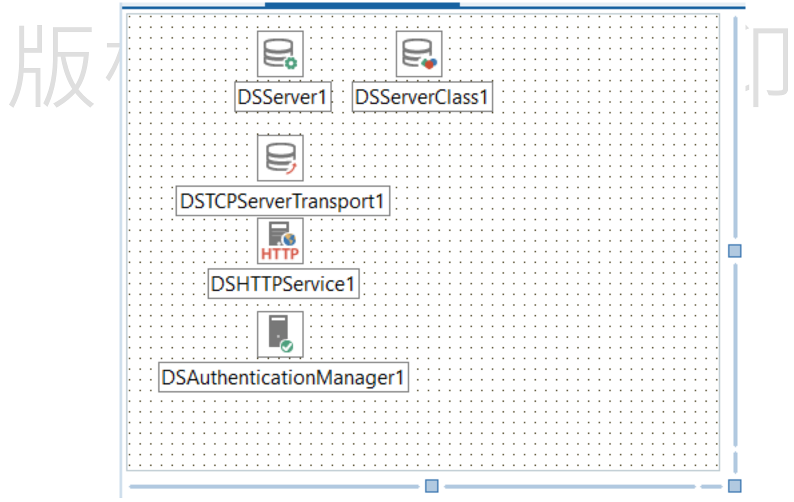



圖 1-7 DataSnap 精靈自動建立的 ServerContainer 模組中包含的元件

| 元件名稱 | 元件類別 | 意義 |
|---|---|---|
| DSServer1 | TDSServer | 提供DataSnap伺服器基本功能 |
| DSTCPServerTransport1 | TDSTCPServerTransport | 提供TCP/IP通訊協定的支援 |
| DSHTTPService1 | TDSHTTPService | 提供HTTP通訊協定和RESTful架構 |

| | | 的基本支援 |
|---|---|---|
| DSAuthenticationManager1 | TDSAuthenticationManager | 提供安全認證功能 |
| DSServerClass1 | TDSServerClass | 提供自動輸出伺服端類別讓用戶端呼叫的基本功能 |

好了，在 Delphi 10.3 自動產生了專案之後，基本上現在這個專案已經提供了 TCP/IP 以及 RESTful 的服務功能了，不信嗎?我們現在就可以測試一下這個自動產生的專案。

前面說過 TDSServerClass 類別中定義的方法都可以自動輸出並且被用戶端呼叫，由於現在專案中的 DSServerClass1 定義了 EchoString 和 ReverseString 這兩個範例方法，因此我們自然應該可以呼叫它們。因此 4 現在讓我們編譯並且執行此伺服器，然後啟動瀏覽器(筆者使用 Google 的 Chrome)，在瀏覽器中輸入如下的 URL:

```
http://localhost:8080/DataSnap/rest/TServerMethods5/ReverseString/嗨, 大家好!
```

因為這個 DataSnap 伺服器也是一個 REST 伺服器，因此我們可以使用標準的 REST 語法架構來呼叫它的服務。讓我們拆解一下上面 URL 的意義。

版權所有 請勿翻印

由於 REST 是使用 HTTP 通訊協定而且前面我們在建立此 DataSnap 伺服器時選擇使用了 HTTP 以及 8080 通信埠，因此 http://localhost:8080 就代表要使用 HTTP 通訊協定連結(請求)本機中位於 8080 通信埠的服務提供者。『DataSnap/』是 TDSHTTPService 元件使用的 DSContent 特性值，而『rest/』則是它使用的 RESTContext 特性值，這兩個數值都可以藉由設定 TDSHTTPService 元件的 DSContent 和 RESTContext 特性來改變。TServerMethods5 則是伺服器中提供輸出方法的類別，由於我們要呼叫 ReverseString 這個方法，因此使用了『/ReverseString』。最後的『/嗨, 大家好!』則是傳遞給 ReverseString 方法的參數，因為 ReverseString 方法定義接受一個字串型態的參數，例如下面就是 ReverseString 的定義，我們可以看到它接受一個命為 Value 的參數，因此『/嗨, 大家好!』就會設定成傳遞給 ReverseString 的 Value 參數值。

```
function ReverseString(Value: string): string;
```

例如下圖就是在 Chrome 中呼叫 DataSnap 伺服器的結果，證明了它的確支援 RESTful 架構並且使用 JSON 格式傳遞資料:

localhost:8080/Dat...

localhost:8080/DataSnap/Rest/TServerMethods5/ReverseString/嗨,%20大家好!

{"result":["!\u597D\u5BB6\u5927 ,\u55E8"]}

圖 1-8 使用瀏覽器呼叫 DataSnap 伺服器中的範例方法

那麼，現在讀者是不是知道了如何在瀏覽器中呼叫 EchoString 呢? 沒錯，只要使用下面的 URL 在瀏覽器中就可以呼叫 EchoString 方法了。

```
http://localhost:8080/DataSnap/rest/TServerMethods5/EchoString/嗨, 大家好!
```

如何，是不是很簡單呢? 由於支援了 REST 架構，因此任何支援 HTTP 的用戶端都可以呼叫 DataSnap 伺服器中的服務了。

現在讓我們繼續為這個 DataSnap 伺服器加入一些有用的功能。讓我們看看如何讓 DataSnap 輸出資料到傳統的 Delphi Win32 用戶端以便開發分散式資料庫應用系統。

首先開啟專案中的 ServerMethods 程式單元，由於它是從 TDSServerClass 類別繼承下來的，因此任何在其中宣告，定義的方法和資料都可以輸出讓用戶端呼叫，現在讓我們使用 dbExpress 的 TSQLConnection，TSQLDataSet，TDataSetProvider 和 TClientDataSet 元件連結到資料，如下所示(有關 dbExpress 程式設計的主題，請參考 dbExpress 相關的書籍):

版權所有 請勿翻印

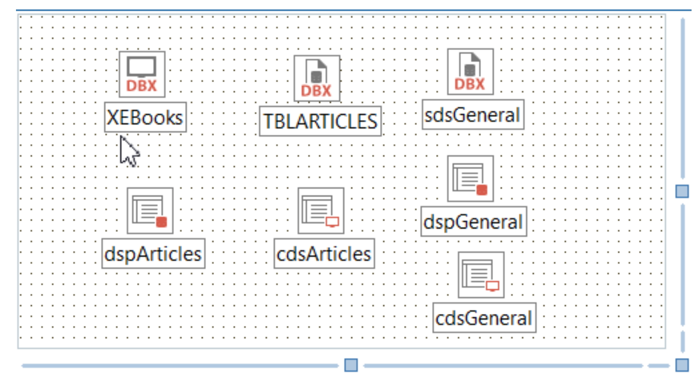



圖 1-9 在 ServerMethods 程式單元中使用 dbExpress 元件連結資料庫

在說明如何開發用戶端之前，再讓我們為伺服器定義兩個方法，這個兩個方法除了在稍後會被用戶端呼叫之外，稍後也可以再次使用瀏覽器來呼叫它們以證明這個 DataSnap 伺服器的確支援 ERSTful 架構並且使用 JSON 封裝和傳遞資料。

首先讓我們在輸出類別 TServerMethods5 中再定義 GetArticleID 方法，它使用 TSQLDataSet 元件執行 SQL 敘述以便取得 Articles 資料表中最大筆數加 1 的數值並且回傳到用戶端:

```
function TServerMethods5.GetArticleID: Integer;
begin
  sdsGeneral.Active := False;
  try
    sdsGeneral.CommandText := 'select count(*) from TBLARTICLES';
    sdsGeneral.Active := True;
    Result := sdsGeneral.Fields[0].AsInteger + 1;
  finally
    sdsGeneral.Active := False;
  end;
end;
```

現在我們就可以使用下面的 URL 在瀏覽器中呼叫 GetArticleID，

```
http://localhost:8080/DataSnap/Rest/TserverMethods5/GetArticleID
```

由於筆者在使用瀏覽器呼叫 GetArticleID 方法時，在 Articles 資料表中已經存在了 2 筆資料，因此在下面的瀏覽器截圖中可以清楚的看到 GetArticleID 方法果然以 JSON 格式封裝數值『3』回傳到用戶端:

圖 1-10 在瀏覽器中呼叫 GetArticleID 方法

第二個方法是則是使用 JSON 陣列回傳所有 Articles 資料表中文章名稱的 QueryAllArticles 方法，下面是 QueryAllArticles 的實作:

```
001   Sfunction TServerMethods5.QueryAllArticles: TJSONArray;
002   var
003     ja : TJSONArray;
004     aBK : TBookmark;
005   begin
006     if (not cdsArticles.Active)  then
007       cdsArticles.Active := True;
008
009     ja := TJSONArray.Create;
010     try
011       aBK := cdsArticles.GetBookmark;
012       cdsArticles.First;
```

```
013       while (not cdsArticles.Eof) do
014       begin
015
ja.AddElement(TJSONString.Create(cdsArticles.FieldByName('NAME').AsString));
016         cdsArticles.Next;
017       end;
018     finally
019       cdsArticles.GotoBookmark(aBK);
020       cdsArticles.FreeBookmark(aBK);
021     end;
022     Result := ja;
023   end;
```

在 009 行先建立一個 TJSONArray 物件，011 行先使用 TBookMark 物件保留 cdsArticles 目前的記錄位置，013 行進入 while 迴圈從第一筆記錄取得 Name 欄位的數值並且以 TJSONString 物件封裝並且儲存在 TJSONArray 物件中，一直到最後一筆記錄。最後 019 行使用 TBookMark 物件讓 cdsArticles 回到呼叫 QueryAllArticles 方法之前的位置，020 行釋放 TBookMark 物件，最後在 022 行回傳 TJSONArray 物件到用戶端。

版權所有 請勿翻印

現在同樣使用瀏覽器並且在 URL 位置使用:

```
http://localhost:8080/DataSnap/Rest/TserverMethods5/QueryAllArticles
```

呼叫 QueryAllArticles 方法，在下面的截圖中可以看到我們果然可以在瀏覽器中成功呼叫 QueryAllArticles 方法，QueryAllArticles 方法是以正確的 JSON 陣列封裝並且回傳所有的文章名稱，而且文章名稱都是以 Unicode 編碼的:

圖 1-11 在瀏覽器中呼叫 QueryAllArticles 方法

由上面兩個實作方法證明了 Delphi 10.3 的 DataSnap 能夠使用 JSON 格式回傳任何的資料，只要這些資料使用 JSON 規範來封裝即可。

現在我們已經完成了第一個範例 DataSnap/RESTful 伺服器，現在請編譯並且執行這個 DataSnap/RESTful 伺服器，因為接著我們要說明如何開發用戶端應用程式來呼叫這個 DataSnap/RESTful 伺服器提供的服務。

現在回到專案群組，在其中建立一個新的 VCL Form 應用程式專案，並且命名它為 pDataSnapDemoClient，如下圖所示:

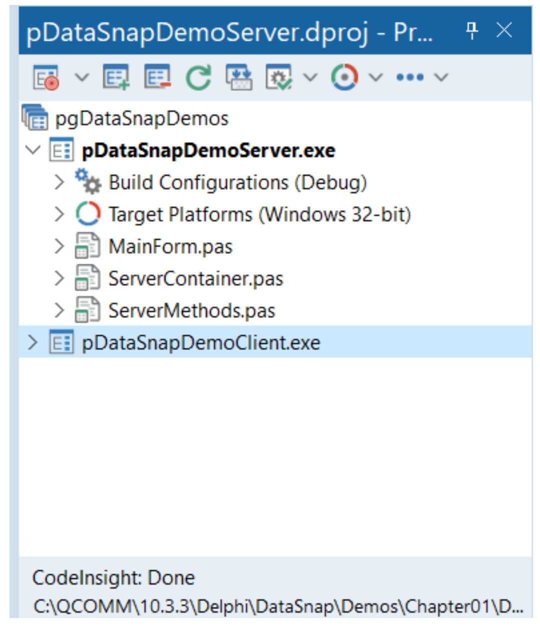



圖 1-12 在專案中建立一個 VCL Form 應用程式專案做為用戶端應用程式

版權所有 請勿翻印

Delphi 提供了數種方式讓用戶端應用程式連結和呼叫 DataSnap 伺服器，其中最簡單，方便的方式就是使用 DataSnap 用戶端精靈。現在就讓我們使用它來自動在用戶端專案中建立 DataSnap 用戶端模組以連結和呼叫 DataSnap 伺服器。

點選 File | New | Other...功能表啟動 New Items 對話盒，在 DataSnap Server 項目中點選 DataSnap Client Module 圖像以便在用戶端應用程式專案中建立 DataSnap 用戶端模組，如下圖所示:

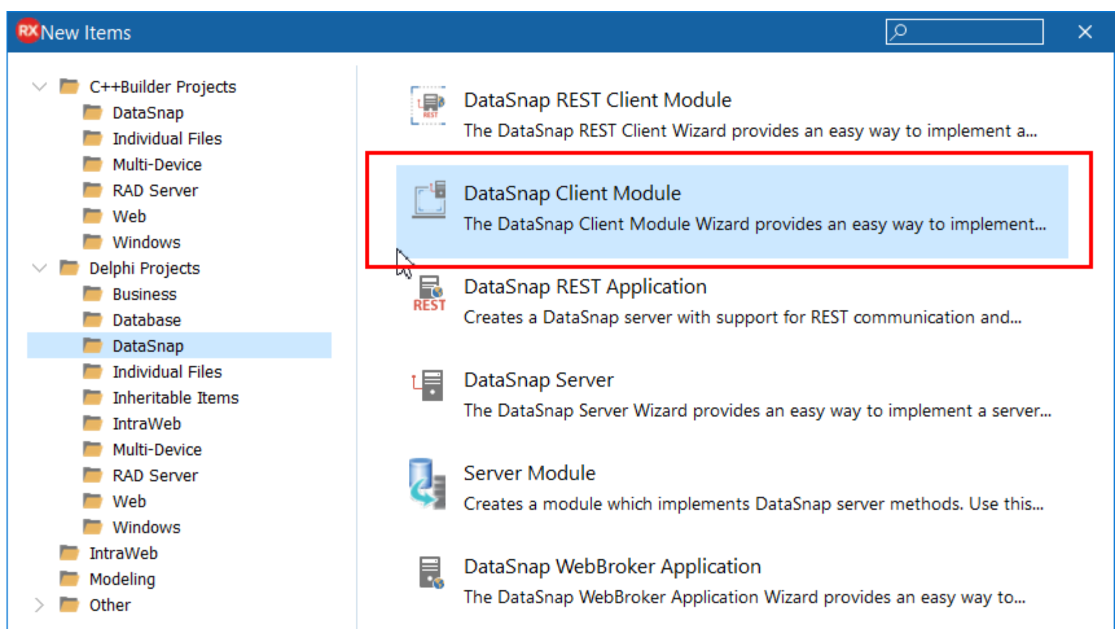



圖 1-13 在用戶端應用程式中建立 DataSnap Client Module

點選 DataSnap Client Module 圖像之後 DataSnap Client Module 精靈會詢問有關 DataSnap 伺服器的特性以便進行對應的設定，首先會詢問要連結的 DataSnap 伺服器的位置，由於我們的範例 DataSnap 伺服器是於本機中，因此我們選擇 Local Server:

版權所有 請勿翻印

圖 1-14 設定 DataSnap 伺服器的特性

接著 DataSnap Client Module 精靈會詢問要連結的 DataSnap 伺服器的型態，由於在前面我們是開發 VCL Form 應用程式型態的伺服器，因此在這裡我們選擇 DataSnap stand alone server 型態:

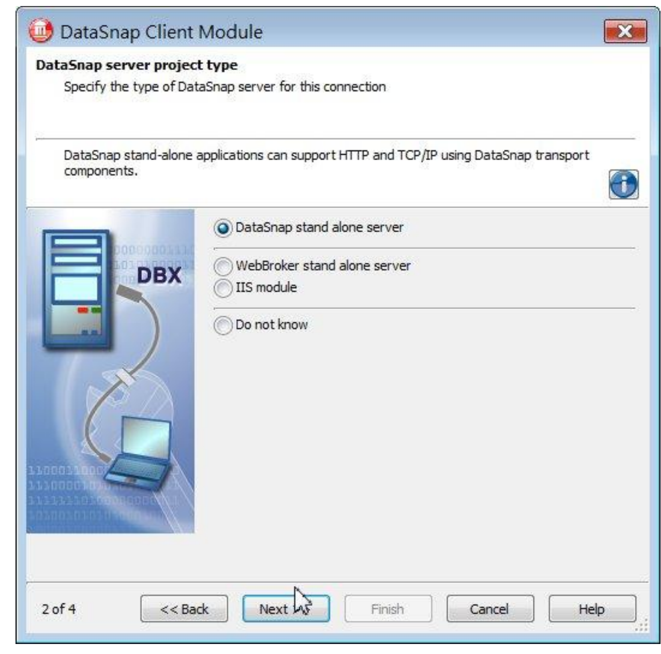



圖 1-15 設定 DataSnap 伺服器的型態

接著 DataSnap Client Module 精靈會詢問要使用什麼通訊協定連結 DataSnap 伺服器，由於現在我們是開發原生 Win32 用戶端應用程式，因此讓我們選擇使用 TCP/IP 通訊協定:

版權所有 請勿翻印

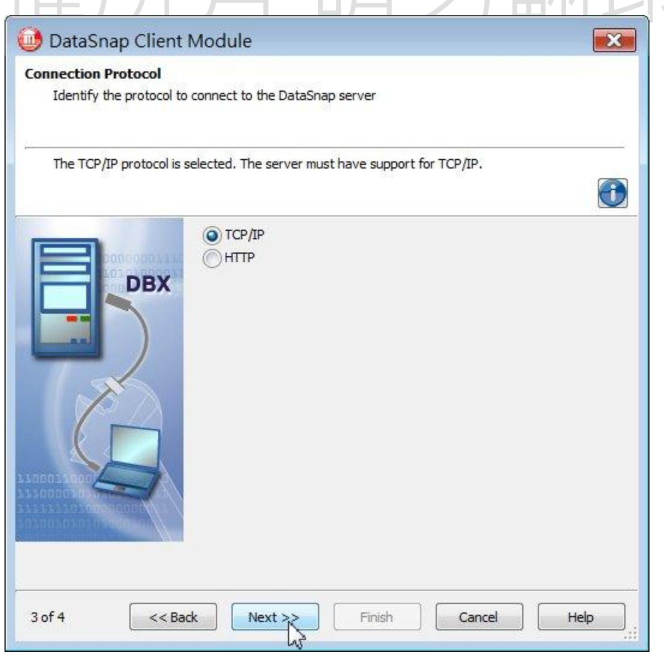



圖 1-16 選擇連結 DataSnap 伺服器的通訊協定

接著 DataSnap Client Module 精靈會詢問使用通信埠以及伺服器的登錄資訊，我們使用 TCP/IP 而且 DataSnap 伺服器是使用內定通信埠 211，因此

在下面的 Port 選項中設定 211，由於我們尚未說明如何使用 DataSnap 的安全認證機制，因此在這裡可以先不用管 User name 和 Password 的選項設定:

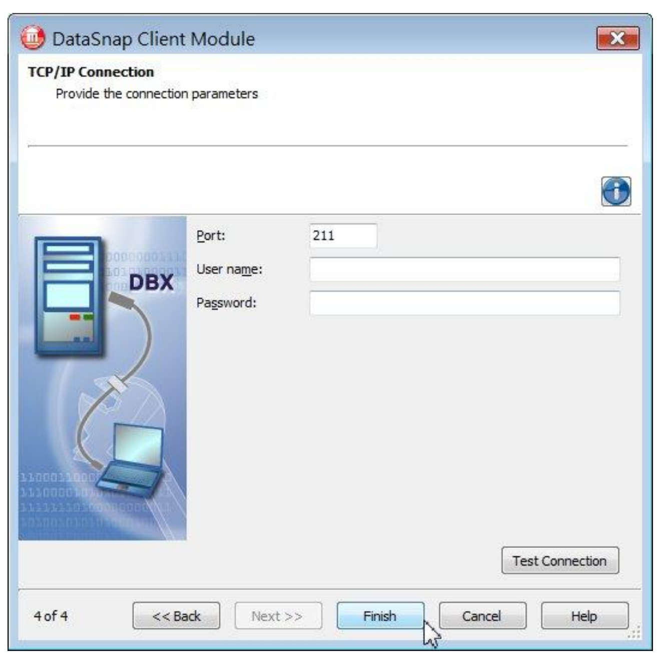



圖 1-17 設定通訊協定的通信埠以及登錄資訊

版權所有 請勿翻印

點選 Finish 按鈕之後讓我們儲存這個 DataSnap Client Module 為 ClientModule.pas，並且儲存 DataSnap Client Module 精靈產生的原始程式為 ServerProxy.pas，此時專案管理員應該類似如下圖所示:

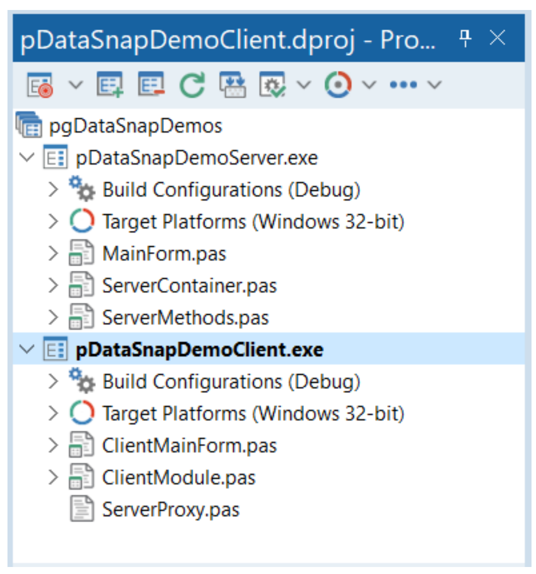



圖 1-18 專案管理員

如果我們開啟 ClientModule 就會看到 DataSnap Client Module 精靈在 DataSnap Client Module 中產生的 TSQLConnection 元件，如下圖所示，筆

者已經將這個 TSQLConnection 元件命名為 sqlcnDataSnapServer，請注意在物件檢視器中它的 Driver 已經被設定為 DataSnap 了:

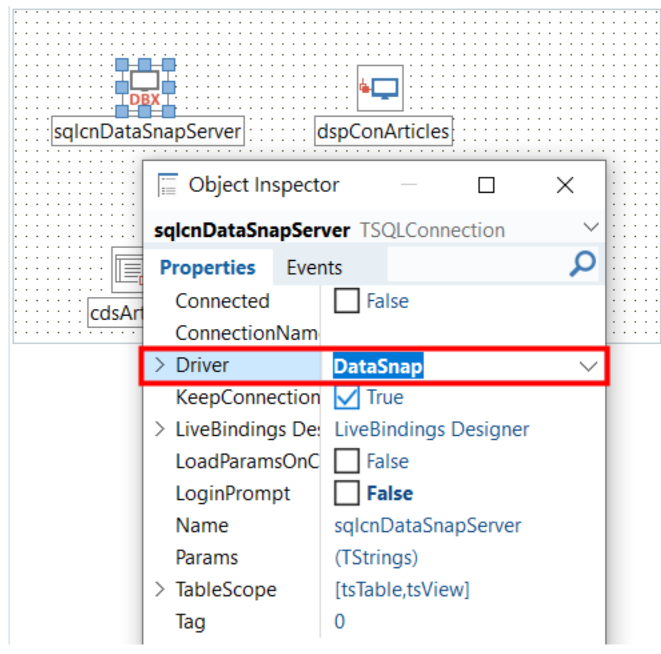



圖 1-19 DataSnap Client Module 精靈在 DataSnap Client Module 中產生的 TSQLConnection 元件

版權所有 請勿翻印

現在如果範例 DataSnap 伺服器已經在執行中，那麼我們可以到整合發展環境的 Data Explorer 中，連結此 DataSnap 伺服器，例如在下圖中我們於 Data Explorer 中對 DATASNAP 選項中的 LOCAL SEVER 進行設定之後就可以連結到範例 DataSnap 伺服器:

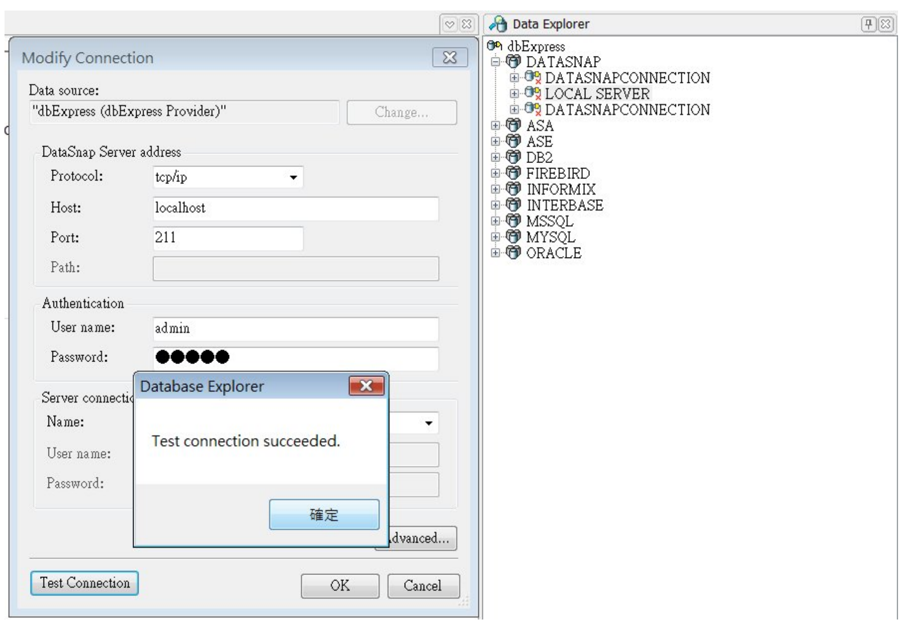



圖 1-120 使用 Data Explorer 連結範例 DataSnap 伺服器

如果我們再開啟 LOCAL SEVER，就可以看到類似下圖的結果，我們果然可以在 LOCAL SERVER 的 Procedures 選項中看到 TServerMethod5 輸出的方法，對於熟悉 Midas 技術的讀者來說，請注意 TServerMethod5 甚至輸出了遠端資料模組(TRemoteDataModule)的 IAppServer 介面之中的方法，這也代表舊的 Midas/DataSnap 應用系統也可以很容易的昇級到新的基於 REST/JSON 架構的 DataSnap 應用系統。

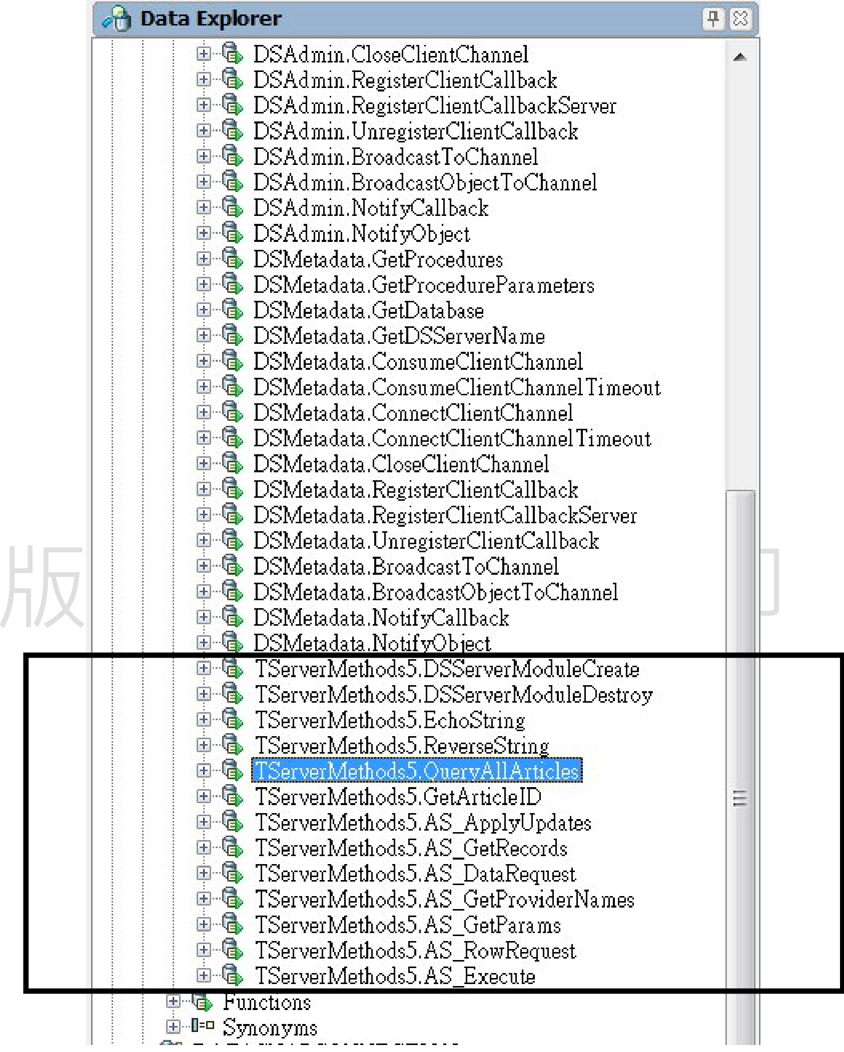



圖 1-19 在 Data Explorer 中檢視範例 DataSnap 伺服器輸出的服務方法

OK，現在我們就可以開始實作用戶端應用程式了，首先讓我們看看如何從 DataSnap 伺服器中取得文章資料表中的資料。在 dbExpress 程式中我們知道使用 TDataSetProvider 元件連結 TSQLDataSet，再使用 TClientDataSet 就可以取得資料。但當我們要連結基於 JSON/REST 的 DataSnap 伺服器時，我們需要使用 TDSProviderConnection 元件連結遠端 DataSnap 伺服器中輸出方法的服務，如果遠端 DataSnap 伺服器中提供了 TDataSetProvider 元件，那麼就可以使用 TClientDataSet 元件藉由 TDSProviderConnection 元件取得遠端的 TDataSetProvider 元件了。

因此現在讓我們在用戶端的 DataSnap ClientModule 中放入 TDSProviderConnection 和 TClientDataSet 元件，如下圖所示:

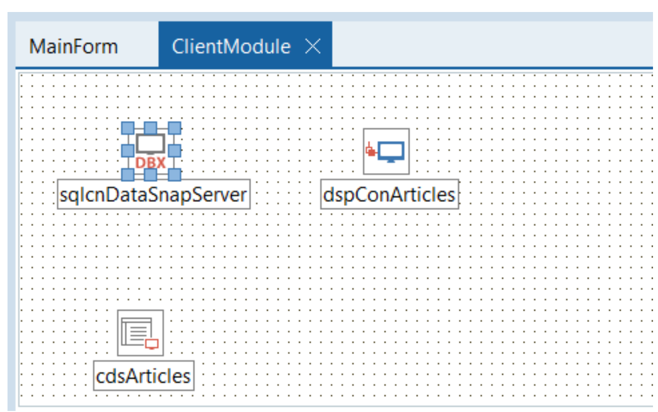



圖 1-20 在 Client Module 中加入 TDSProviderConnection 和 TClientDataSet 元件

接著設定 TDSProviderConnection 的特性如下:

| 特性 | 特性值 |
|---|---|
| ServerClassName | TServerMethods5 |
| SQLConnection | SqlcnDataSnapServer |
| Name | dspConArticles |

版權所有 請勿翻印

設定 TDSProviderConnection 元件的 ServerClassName 特性值為 TServerMethods5 是因為 DataSnap 伺服器中輸出服務方法的類別就是 TServerMethods5。

接著設定 TClientDataSet 的特性如下:

| 特性 | 特性值 |
|---|---|
| RemoteServer | dspConArticles |
| ProviderName | dspArticles |
| Name | cdsArticles |

在設定 cdsArticles 時，一旦設定了 RemoteServer 特性值，那麼當我們設定 ProviderName 特性值時會感覺到物件檢視器暫停了一下，之後可以設定的特性值會出現在下拉盒中，這是因為此時 Delphi 整合發展環境就藉由 TDSProviderConnection 連結到 DataSnap 伺服器並且搜尋其中輸出的 TDataSetProvider 元件，由於前面範例 DataSnap 伺服器輸出了名為 dspArticles 的 TDataSetProvider 元件，因此在 ProviderName 特性下拉盒中會出現 dspArticles。

現在於用戶端主表單中使用 TDataSource 元件連結 cdsArticles，就再設定 cdsArticles 的 Active 特性值為 True 之後，就可以看到遠端的 Article 資料表的資料果然出現在用戶端的應用程式中了，如下圖所示:

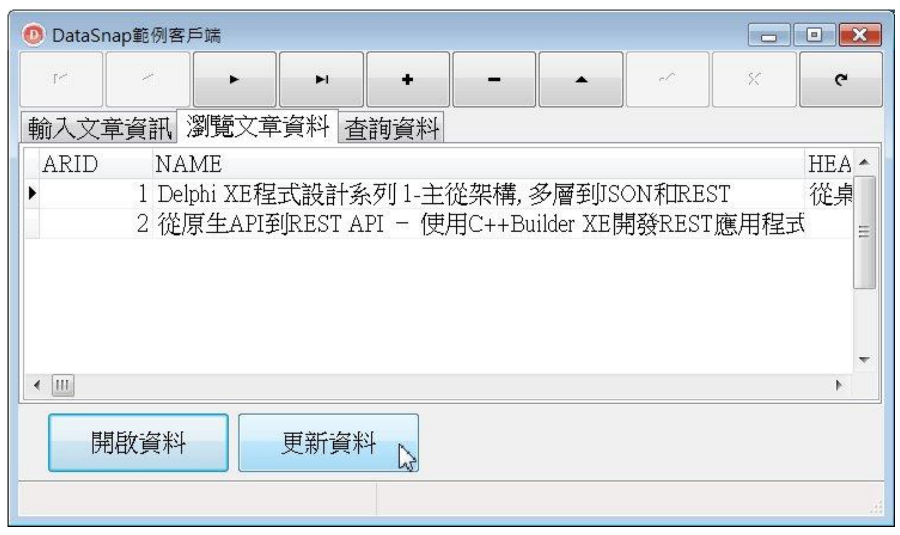



圖 1-25 在 Client Module 中加入 TDSProviderConnection 和 TClientDataSet 元件

那麼我們如何在用戶端應用程式對於 JSON/REST 的 DataSnap 伺服器進行資料的異動呢?事實上這就是使用 dbExpress 和 DataSnap 的好處了，我們仍然可以使用 dbExpress 的方式對於 JSON/REST 的 DataSnap 伺服器進行資料的異動。

版權所有 請勿翻印

例如現在如果我們要在用戶端新增資料到 JSON/REST 的 DataSnap 伺服器，那麼我們可以首先在 Client Module 中為 cdsArticles 定義 AfterInsert 事件處理函式，在 AfterInsert 事件處理函式中我們自動為新增資料的 ARID 和 ADate 欄位產生欄位值，ARID 是文章的序號而 ADate 則是文章新增到資料表的日期:

```
procedure TcmArticles.cdsArticlesAfterInsert(DataSet: TDataSet);
begin
  cdsArticles.FieldByName('ARID').Value := GetArticleID;
  cdsArticles.FieldByName('ADate').Value := Now;
end;
```

接著我們就可以在主表單的『更新資料』按鈕的 OnClick 事件處理函式中呼叫 cdsArticles 的 ApplyUpdates 方法更新回 JSON/REST 的 DataSnap 伺服器即可:

```
procedure TMainForm.btnApplyUpuetsClick(Sender: TObject);
begin
  cmArticles.cdsArticles.ApplyUpdates(0);
```

```
  UpdateStatusBar(IntToStr(cmArticles.cdsArticles.ChangeCount) + '筆資料被異動了');
end;
```

例如下圖就是使用用戶端應用程式新增一筆資料回 JSON/REST 的 DataSnap 伺服器:

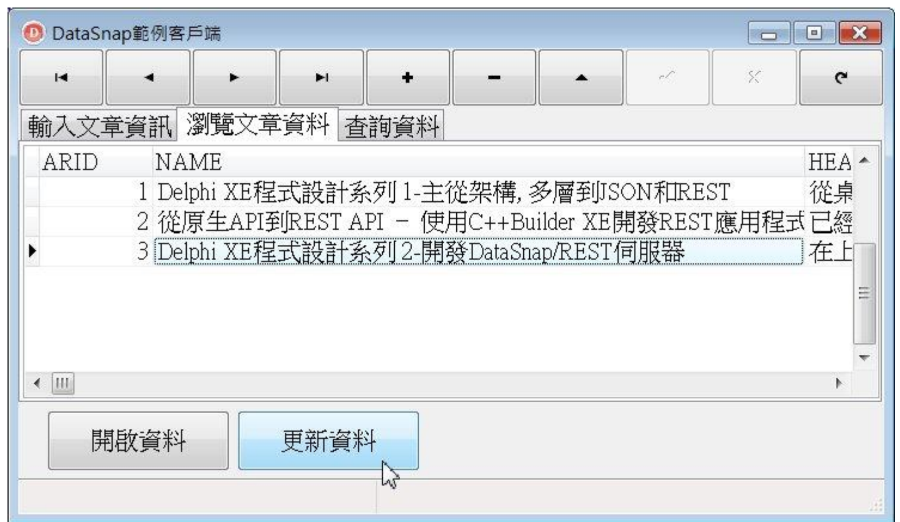



圖 1-26 呼叫 TClientDataSet 的 ApplyUpdates 方法更新資料回 JSON/REST 的 DataSnap 伺服器

一旦用戶端使用 TDSProviderConnection 和 TClientDataSet 元件向 JSON/REST 的 DataSnap 伺服器取得資料之後，在用戶端它就向是直接使用 dbExpress 取得資料一樣，在用戶端也可以使用 TClientDataSet 的查詢資料能力來搜尋資料，例如下面的程式碼即使用了 TClientDataSet 的 Lookup 方法查詢資料:

版權所有 請勿翻印

```
procedure TMainForm.btnQueryARIDClick(Sender: TObject);
begin
  edtARName.Text := cmArticles.cdsArticles.Lookup('ARID', StrToInt(edtARID.Text),
'Name');
end;
```

下圖就是查詢資料的結果，輸入文章 ID 即可查詢到相對應的文章。

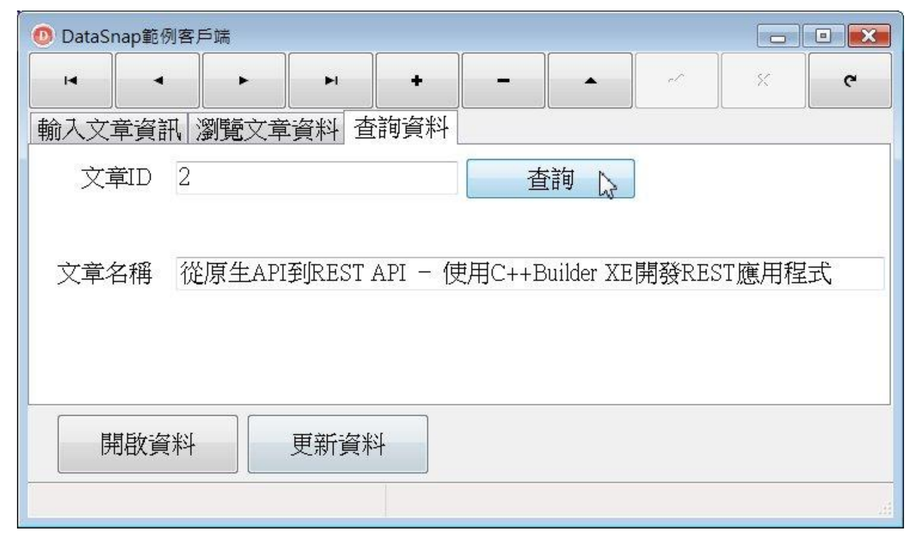



圖 1-27 使用 TClientDataSet 的 Lookup 方法查詢資料

在這個範例文章資料表中同時有一個 BLOB 型態的欄位，它的目的是儲存文章的 PDF 電子檔案，那麼我們又如何從用戶端把 PDF 檔案儲存到 JSON/REST 的 DataSnap 伺服器中呢?使用 dbExpress 和 DataSnap 可以免除我們把二進位資料轉換成 JSON 格式的麻煩，我們仍然可以要求 TClientDataSet 元件幫助我們完成這個工作，我們只需要呼叫用戶端 TClientDataSet 代表 BLOB 欄位的物件的 LoadFromFile 方法即可。例如在下面的程式碼中，cdsArticlesCONTENTS 是代表遠端 BLOB 欄位的 TBlobField 型態的物件，因此我們在決定需要上傳那一個 PDF 檔案到遠端文章資料表之中後，就可以直接呼叫 cdsArticlesCONTENTS 的 LoadFromFile 即可，非常的簡單，也是使用傳統 dbExpress 程式設計的方式。

版權所有 請勿翻印

```
procedure TMainForm.btnOpenFileClick(Sender: TObject);
begin
  if (odlgopenFile.Execute) then
  begin
    if (cmArticles.cdsArticles.State <> dsEdit) then
      cmArticles.cdsArticles.Edit;
      cmArticles.cdsArticlesCONTENTS.LoadFromFile(odlgopenFile.FileName);
      cmArticles.cdsArticles.Post;
      cmArticles.cdsArticles.ApplyUpdates(0);
  end;
end;
```

最後筆者要提醒的是，如果讀者在 JSON/REST 的 DataSnap 伺服器中加入了新的方法讓用戶端呼叫，那麼必須在用戶端重新產生連結伺服器的用戶端類別才能夠看到新增的方法，要重新產生連結伺服器的用戶端類別，讀者可以到

Client Module 中，右擊 TSQLConnection 元件並且從快顯功能表中選擇『Generate DataSnap client classes』，如下圖所示即可:

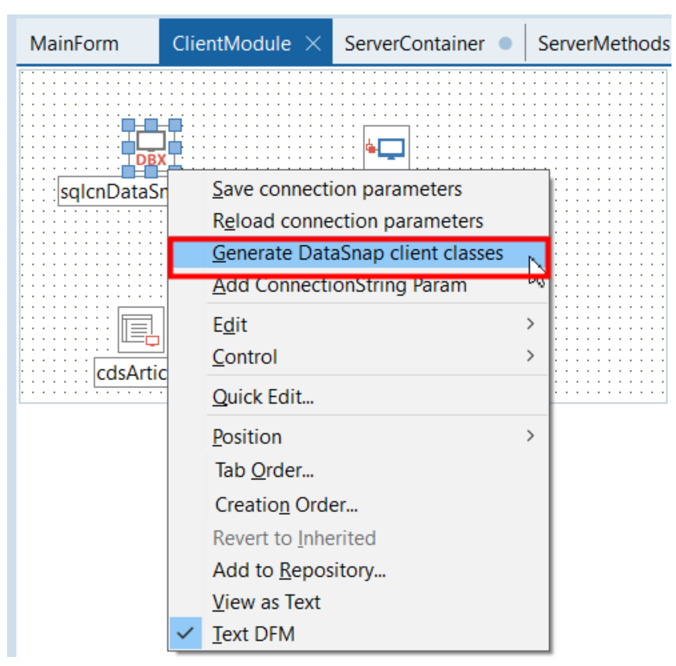



圖 1-28 用戶端的 TSQLConnection 元件重新產生用戶端連結伺服器的程式碼

版權所有 請勿翻印

在重新產生了用戶端類別之後，就可以在用戶端應用程式中使用程式碼來呼叫 JSON/REST 的 DataSnap 伺服器新加入的方法了。

其實 Delphi XE 的 DataSnap 框架也提供了自動產生用戶端類別的機制，但這屬於高階的 DataSnap 技術，我們會在以後的章節中討論這個技術。

好了，現在您已經瞭解了如何開發 DataSnap 伺服器和用戶端應用程式了，在以後的章節中我們將繼續討論進階的 DataSnap 技術，現在是您動手開發您自己的 DataSnap 應用系統的時候了。

# 第2章 JSON程式設計

從 Delphi 2009 開始 Delphi 便開始支援 JSON，DataSnap 2009 開始使用 JSON 做為開發分散式應用系統的基礎技術，開始逐漸捨棄使用 COM/DCOM/COM+。由於 JSON 具備跨平台，跨程式語言和跨工具的特性，因此 DataSnap 在結合 JSON 和 dbExpress 之後也逐漸具備了跨平台的基礎。但是 Delphi 2009 對於 JSON 的支援只限於使用在 DataSnap 之中，因此如果開發人員需要進行通用的 JSON 程式設計，Delphi 2009 在這方面仍然顯得不足。

版權所有 請勿翻印

到了 Delphi 2010 情形開始改變，因為 VCL 框架開始內建支援 JSON 的類別，因此 Delphi 的開發人員就可以利用這些和 JSON 相關的類別來進行 JSON 程式設計的工作。到了 Delphi XE 之後，JSON 已經成為 DataSnap 的核心技術了，因為 Delphi 即將進入跨平台的世代，JSON 具備跨平台的特性自然就成為最佳的資料傳遞和交換的標準，因此在 Delphi XE 之後的 RTL 以及 DataSnap 中又再次大幅強化了對於 JSON 支援的能力。由於 DataSnap 在分散式和 Web 架構中都使用 JSON 封裝資料，此外 dbExpress 也能夠處理 JSON 封裝的資料，因此 Delphi 的程式師必須瞭解和掌握 JSON 技術。

本章的重點就是討論 VCL 框架對於 JSON 的支援，本章將討論如何使用 VCL 框架中和 JSON 相關的類別來進行 JSON 開發的工作。不過在討論這些類別之前，我們需要先簡單的介紹什麼是 JSON。

## 2-1 JSON 是什麼?

簡單的說，JSON 是一種資料封裝格式以及資料交換格式，JSON 是 JavaScript Object Notation 的縮寫，它是一種輕量級的資料交換格式，易於一般人閱讀和編寫，同時也易於機器解析和生成。

JSON 是基於 JavaScript（Standard ECMA-262 3rd Edition - December 1999）的一個子集，JSON 採用完全獨立語言的文字格式，但是也

使用了類似於 C 語言家族的習慣（包括 C, C++, C#, Java, JavaScript, Perl, Python 等），這些特性使 JSON 成為理想的資料交換語言。

JSON 是由下面的兩個基本架構組成的:

➢ “名稱/值”對的集合（A collection of name/value pairs）。在不同的程式語言中，這個架構經常以物件（object），紀錄（record），結構（struct），字典（dictionary），雜湊表（hash table），有鍵列表（keyed list），或者關聯陣列 （associative array）來實作。

➢ 值的有序列表（An ordered list of values）。在大部分的程式語言中，這個架構經常使用陣列（array）來實作。

OK，看了上述的定義之後，讀者可能還是覺得模模糊糊，也許讓我們使用實際的範例來說明可以讓讀者更容易瞭解。

假設現在我們有一個 DataSnap 伺服器，它使用 JSON 和用戶端進行資料交換的工作，那麼現在用戶端向這個 DataSnap 伺服器查詢筆者所撰寫的書籍，例如本書『實戰 Delphi 2010』，那麼如何把這個資訊以 JSON 的格式傳遞給用戶端呢?

版權所有 請勿翻印

首先讓我們觀察下圖，在 JSON 中物件的代表方式是以“{”開始，以“}”結束，另外在前面我們也說明 JSON 是以『名稱/值』的格式來代表資料，因此從下圖中我們可以看到名稱和值之間是以“：”符號分隔，其中『名稱』是字串格式，而值則是以數值格式來代表:

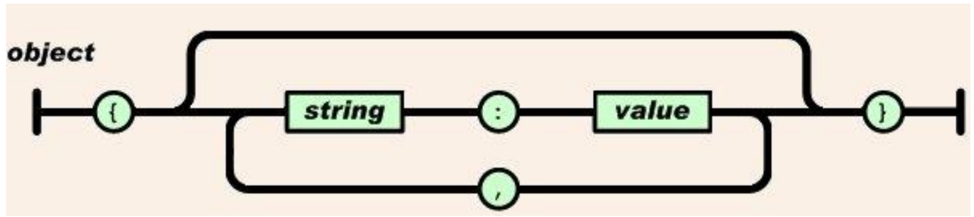



圖 1 JSON 封裝物件的格式

在 JSON 規範中上圖的字串規範如下圖所示:

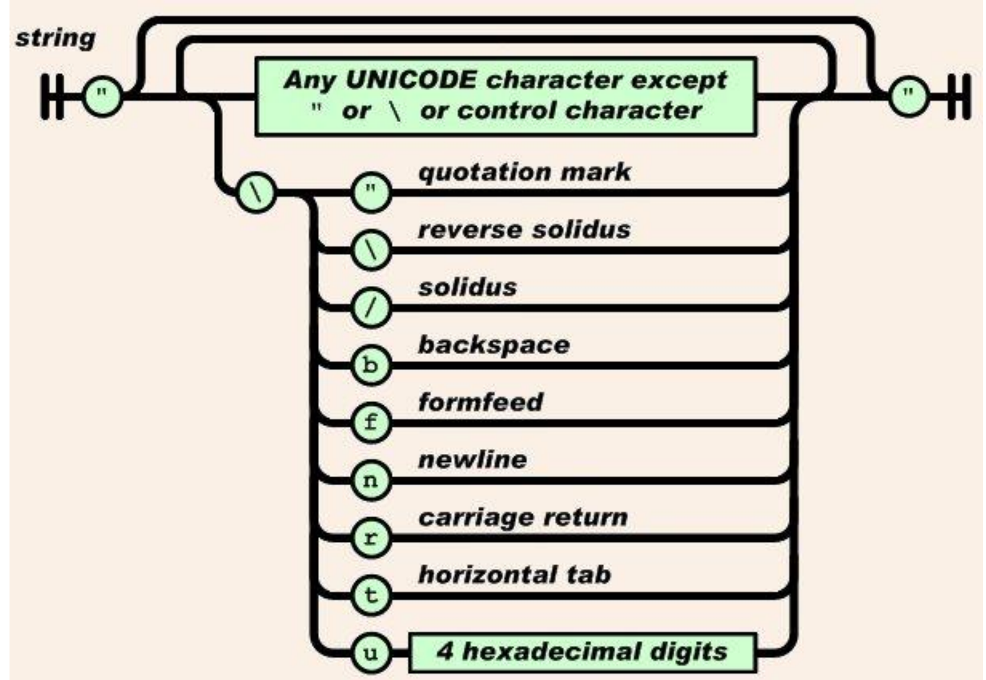



圖 2 JSON 封裝字串的格式

從上圖中我們可以知道在 JSON 中字串是以”開始，以”結束，可包含 Unicode 的字元組。而圖 1 中的數值格式則如下圖所示:

版權所有 請勿翻印

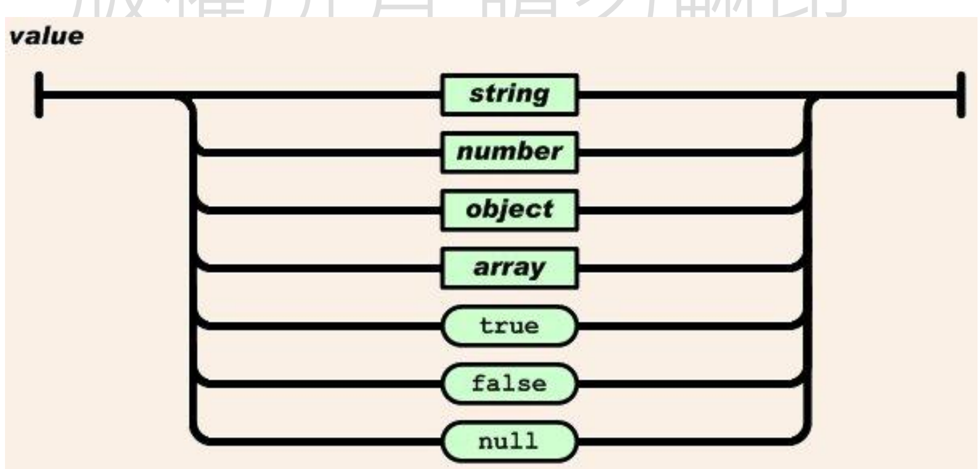



圖 3 JSON 封裝數值的格式

從圖 3 中可知，JSON 的數值可以是字串，數字，物件，陣列或是 true/false/null 值。由於 JSON 的數值可以是物件或是陣列，而物件又可以包含字串:數值配對，因此 JSON 規範可以封裝任何複雜的資料。

有了上述對於 JSON 基本的瞭解之後，我們就可以使用如下的格式從 DataSnap 伺服器傳遞資料到用戶端:

```
{"書名":"實戰 Delphi 2010"}
```

從這個格式中我們可以知道這是用 JSON 中封裝物件的規則，把『字串:數值』配對包含在{和}符號之中。

如果我們希望也傳遞作者的資訊，那麼可以使用逗號分離每一個『字串:數值』配對，如下所示，讀者也可以回頭再參考圖 1 就可以發現這是圖 1 的規則。

```
{“書名”:”實戰 Delphi 2010”,“作者”:”李維”}
```

如果現在再把書籍出版年份也傳遞，那麼就可以使用下面的格式:

```
{"書名":"實戰 Delphi 2010","作者":"李維","出版年份":2010}
```

請注意的是，出版年份的數值是 2010 這個數字，因為如圖 3 所示數值可以是數字(number)，而在 JSON 中數字的定義如下:

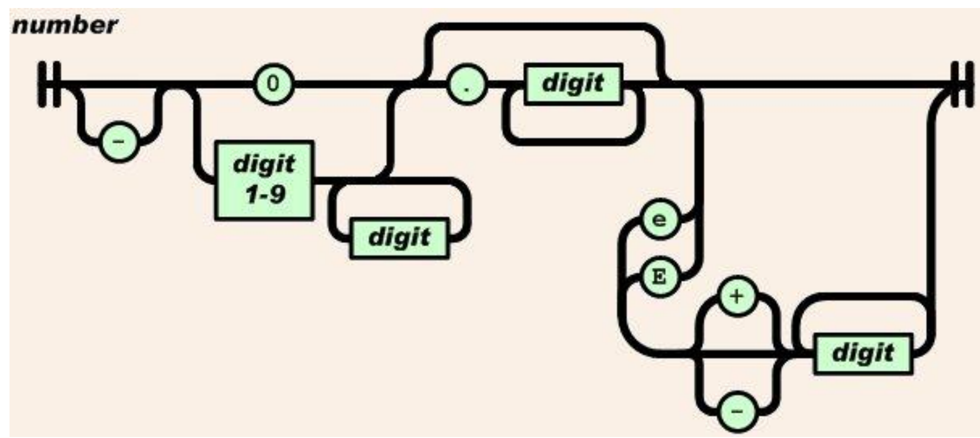



圖 4 JSON 封裝數值的格式

從圖 4 我們可以知道，在 JSON 中數字可以是整數或是浮點數。

```
{"書名":"實戰 Delphi 2010","作者":"李維","出版年份":2010}
```

因此如果我們再傳遞書籍價格，那麼可以使用如下的格式:

```
{"書名":"實戰 Delphi 2010","作者":"李維","出版年份":2010,"價格":45.95}
```

最後如果我們再加入書籍已出版否資訊，那麼可以使用如下的格式:

```
{"書名":"實戰 Delphi 2010","作者":"李維","出版年份":2010,"價格":45.95,"已出版否":false}
```

從上面的範例中我們可以看到 JSON 規範如何封裝各種不同型態的資料。那麼如果我們需要一次傳遞多筆資料到用戶端的話，又要如何封裝呢?這可以使用 JSON 中陣列的規範。

圖 5 是 JSON 陣列的封裝規則，陣列“[”開始，以“]”結束，而陣列中的每一個元素都是數值，每一個元素使用逗號分隔。回頭參考圖 3 JSON 封裝數值的格式，由於數值可以是物件，因此我們如果需要一次傳遞多筆資料到用戶端，那麼就可以使用 JSON 的陣列來封裝。

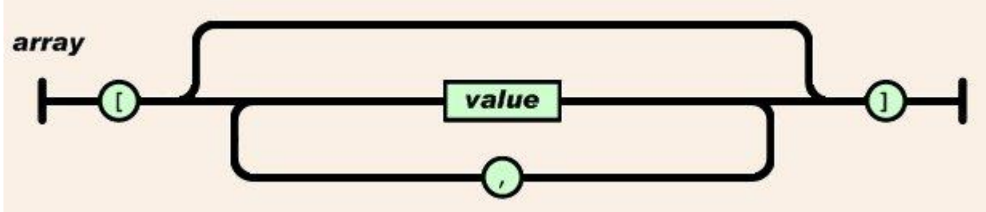



圖 5 JSON 封裝陣列的格式

例如下面就是使用 JSON 的陣列封裝性三個代表書籍的物件，陣列中的每一個元素數值是物件，每一個元素數值使用逗號分隔:

```
[{“書名”:”實戰 Delphi 2010”},{“書名”:”Inside VCL”},{“書名”:”Borland 傳奇”}]
```

如何? 一旦瞭解了 JSON 封裝資料的規則之後讀者是不是覺得使用 JSON 非常的簡潔呢? 正由於 JSON 在封裝資料的規則比較簡單，因此在解析 JSON 資料時速度也比較快，JSON 的簡潔規則讓 JSON 擁有較少的資料流量和較快的處理速度，因此讓 JSON 在 Web 應用方面佔有優勢，也愈來愈受歡迎，這也是 DataSnap 現在選擇使用 JSON 做為基礎技術的原因。

在離開本小節之前，讓我們比較一下使用 XML 和 JSON 在封裝和交換資料方面的差異。下面是使用 XML 封裝一個員工的資訊:

版權所有 請勿翻印

```
<?xml version="1.0" encoding="utf-8"?>
<user>
     <name>李大明</name>
     <password>123456</password>
     <department>R&D</department>
     <gender>男</gender>
     <age>26</age>
</user>
```

如果我們使用 JSON 來封裝的話，那麼就如下所示：

```
{
     "name":"李大明",
     "password":"123456",
     "department":"R&D",
     "gender":"男",
     "age":"26"
}
```

我們可以看到 JSON 使用了較 XML 少的資料量來封裝相同的資訊，此 JSON 在資料交換方面擁有比較多的優勢。

讀者可以參考 www.json.org 以瞭解更多有關 JSON 的資訊。

在對於 JSON 有了基本的瞭解之後，更重要的是我們需要知道如何在 Delphi 中進行 JSON 的程式設計。

## 2-2 VCL 框架中支援 JSON 的類別

Delphi 在 2009 開始使用 JSON 做為 DataSnap 2009 封裝和傳遞資料的格式，但是 Delphi 2009 對於 JSON 的支援和 DataSnap 的功能撰寫的非常緊密，開發人員除了能夠在 DataSnap 2009 中使用 JSON 之外，並不容易使用 Delphi 2009 來開發其他應用型態的 JSON 應用程式。到了 Delphi 2010 這個現象獲得了大幅的改善，Delphi 2010 提供了許多通用的 JSON 相關類別，開發人員可以使用這些通用的 JSON 相關類別開發任何型態的 JSON 應用程式而不只限於 DataSnap 應用程式。

Delphi 2010 在新的 DBXJSON.pas 程式單元中提供了這些 JSON 相關的類別，如果讀者仔細觀察前面 JSON 規則圖的話，就可以發覺如果我們需要使用類別來實作支援 JSON 的規範，那麼至少需要下面的類別:

| 類別 | 說明 |
| --- | --- |
| JSONValue | 用來代表和實作圖 3 的實體和關係 |
| JSONPair | 用來代表 JSON 規範中的『名稱/值』配對實體和關係 |
| JSONObject | 用來代表和實作圖 1 的實體和關係 |
| JSONString | 用來代表和實作圖 2 的實體和關係 |
| JSONArray | 用來代表和實作圖 5 的實體和關係 |
| JSONNumber | 用來代表和實作圖 4 的實體和關係 |

版權所有 請勿翻印

當然在上述的基礎類別中還存在繼承的關係，例如由於 JSON 規範中數值可以代表字串，物件，陣列等實體，因此上面的 JSONObject，JSONString 等類別可以從 JSONValue 類別繼承下來。

Delphi 2010 中支援 JSON 等相關類別就是使用這個觀念實作出來的，因此只要讀者瞭解了前面解釋 JSON 規範的觀念，就可以非常直覺的瞭解這些實作類別。在下面的表格中整理了這些類別，並且提供了簡單的敘述:

| 類別 | 說明 |
| --- | --- |
| TJSONAncestor | 抽象類別，是所有 JSON 相關類別的根類別 |
| TJSONPair | 實作 JSON 規範中『名稱/值』配對的類別 |
| TJSONValue | 實作 JSON 規範中數值的類別，TJSONValue |

| | 是 TJSONAncestor 的繼承類別 |
|---|---|
| TJSONTrue | 實作 JSON 規範中代表 True 的類別，從 TJSONValue 繼承下來 |
| TJSONString | 實作 JSON 規範中代表字串的類別，從 TJSONValue 繼承下來 |
| TJSONNumber | 實作 JSON 規範中代表數值的類別，它從 TJSONString 繼承下來，因為 JSON 使用文字格式傳遞封裝和傳遞資料，因此所有數值都將轉換為文字字串型態傳遞，到達目的地之後再反轉回數值 |
| TJSONObject | 實作 JSON 規範中代表物件的類別，從 TJSONValue 繼承下來 |
| TJSONNull | 實作 JSON 規範中代表 Null 的類別，從 TJSONValue 繼承下來 |
| TJSONFalse | 實作 JSON 規範中代表 False 的類別，從 TJSONValue 繼承下來 |

下圖是這些類別之間的繼承和關連關係圖，請讀者對照下圖和以前圖 1 到圖 5 的 JSON 規範，就可以瞭解這些類別的實作是完全從 JSON 規範中設計出來的，簡單又符合直覺:

版權所有 請勿翻印

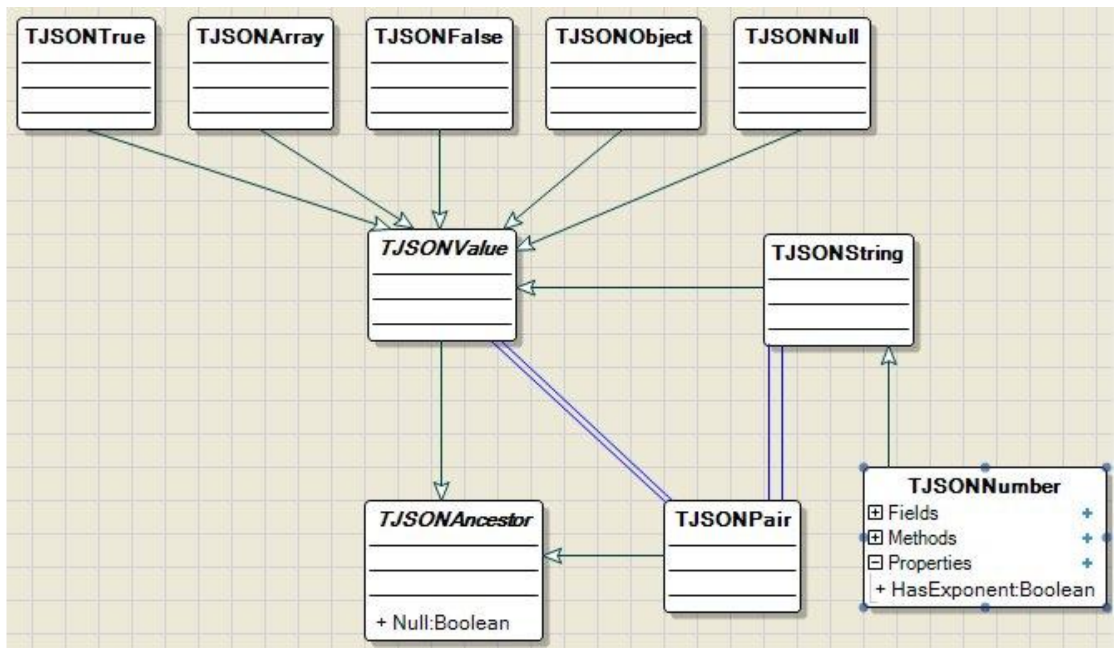



圖 6 Delphi 中支援 JSON 的類別架構

Delphi XE 又加入了許多新的對於 JSON 支援的服務，讓 Delphi 的開發人員能夠更簡單的在程式碼中處理 JSON。例如 Delphi XE 加入 TDBXJSONTools 類別

| 方法 | 說明 |
|---|---|
| ValueTypeToJSON | 封裝 dbExpress 型態的資訊為 JSON 的格式，主要的目的是使用 JSON 傳遞 DBX 數值的 Metadata |
| JSONToValueType | ValueTypeToJSON 的反相函式，把 JSON 封裝的 metadata 轉換回 DBX 的型態物件 |
| TableToJSON | 把 dbExpress 封裝的 TDBXReader 中的資料轉換為以 JSON 型態封裝的資料，這個函式允許開發人員在伺服端把 TDataSet 等物件封裝的資料以 JSON 的形式回傳到用戶端 |
| DBXToJSON | 封裝 DBX 的數值為 JSON 的型態，把 DBX 能夠代表的數值都轉換為以 JSON 封裝的形式，以便在伺服端和用戶端傳遞資料 |
| JSONToDBX | DBXToJSON 的反相函式，把 JSON 封裝的資料轉換回適當的 DBX 的數值型態 |
| StreamToJSON | 把 TStream 型態的資料轉換為 TJSONArray 封裝的資料，通常是使用在傳遞 2 進位型態的資料於伺服端和用戶端之間 |
| JSONToStream | StreamToJSON 的反相函式，把 TJSONArray 封裝的資料轉換為 TStream 型態的資料 |

版權所有 請勿翻印

TDBXJSONTools 類別的目的是幫助開發人員更容易的在伺服器和用戶端之間以 JSON 的形式傳遞 dbExpress 封裝的資料。

在 10.3 中為了增加處理 TDBXArrayList 和 TJSONArray 物件的便利性，因此增加了 2 個 Enumerator 類別:

| 類別 | 說明 |
|---|---|
| TJSONPairEnumerator | 封裝 TDBXArrayList 物件的 Enumerator 類別 |
| TJSONArrayEnumerator | 封裝 TJSONArray 物件的 Enumerator 類別 |

這 2 個 Enumerator 類別的目的是為了讓開發人員可以使用 for..in...迴圈來處理 TJSONPair 和 TJSONArray 物件中的物件。

接下來讓我們簡單的說明這些類別提供的服務，如此一來讀者就可以瞭解如何在程式碼中使用它們。

## 2-2-1 TJSONAncestor 類別

TJSONAncestor 類別是所有 JSON 相關類別的根類別，它主要定義了三個虛擬方法讓它的衍生類別來複載實作，下面的表格敘述了這些虛擬方法:

| 虛擬方法 | 說明 |
| --- | --- |
| function ToString: UnicodeString; virtual; | ToString 虛擬方法會把 JSON 的內容轉換為文字字串形式回傳，TJSONAncestor 的衍生類別需要實作這個虛擬方法，例如 TJSONObject 類別會實作這個虛擬方法並且把物件內容轉換為字串的形態回傳 |
| function EstimatedByteSize: Integer; virtual; abstract; | EstimatedByteSize 虛擬方法回傳 JSON 物件預估的位元組大小，TJSONAncestor 的衍生類別需要實作這個虛擬方法，每一個衍生類別都會回傳它本身需要佔據位元組的大小。EstimatedByteSize 虛擬方法的目的是在封裝和傳遞 JSON 物件時程式碼能夠配置足夠的記憶體大小之用。 |
| function ToBytes(const Data: TBytes; const Offset: Integer): Integer; virtual; abstract; | ToBytes 虛擬方法能夠把 JSON 物件以位元組的形式回傳，TJSONAncestor 的衍生類別需要實作這個虛擬方法，每一個衍生類別都會在這個複載的虛擬方法中把自己轉換為位元組形態。 |

版權所有 請勿翻印

TJSONAncestor 本身是一個抽象類別，因此開發人員並不應該直接在程式碼中使用它，而是應該使用下面介紹的衍生類別。

## 2-2-2 TJSONValue 類別

TJSONValue 類別本身也是一個抽象類別，它是直接從 TJSONAncestor 繼承下來的，TJSONValue 類別只是做為一個 Placeholder，它主要的目的是代表圖 3 中 JSON 規範 value 的概念，因此它的定義非常簡單，如下所示:

```
TJSONValue = class abstract(TJSONAncestor)
end;
```

稍後介紹的許多類別都是從 TJSONValue 繼承下來的，幾乎和圖 3 顯示的規範架構一樣。

## 2-2-3 TJSONPair 類別

TJSONPair 類別是實作 JSON 規範中『名稱/值』配對的類別，它從 TJSONAncestor 直接繼承下來，由於 TJSONPair 是實作『名稱/值』配對，因此它提供了兩個最重要的特性，JsonString 和 JsonValue:

```
property JsonString: TJSONString read GetJsonString write SetJsonString;
property JsonValue: TJSONValue read GetJsonValue write SetJsonValue;
```

JsonString 是代表『名稱/值』配對中名稱的特性，而 JsonValue 則代表其中值的特性。請注意的是 JsonValue 的型態是 TJSONValue，而根據前面表格所敘述，TJSONValue 可以代表任何從 TJSONValue 繼承下來的類別，因此 JsonValue 可以代表 TJSONString，TJSONNumber 或是 TJSONObject 等。

那麼如何建立 TJSONPair 呢? TJSONPair 定義了四個複載的建構函式，其中最重要的三個如下所示:

```
constructor Create(const Str: TJSONString; const Value: TJSONValue); overload;
constructor Create(const Str: UnicodeString; const Value: TJSONValue); overload;
constructor Create(const Str: UnicodeString; const Value: UnicodeString); overload;
```

版權所有 請勿翻印

上面第一個建構函式可以藉由 TJSONString 和 TJSONValue 物件來建立 TJSONPair 物件，第二個建構函式可以使用 Unicode 字串和 TJSONValue 物件來建立 TJSONPair 物件，至於第 3 個建構函式則是使用兩個 Unicode 字串來建立 TJSONPair 物件。

例如，如果我們要建立如下的 JSON『名稱/值』配對:

```
"書名":"實戰 Delphi 2010"
```

那麼我們可以使用如下的程式碼:

```
var
  jp : TJSONPair;
begin
  jp := TJSONPair.Create(TJSONString.Create('書名'), TJSONString.Create('實戰 Delphi
2010'));
  try
    Memo1.Lines.Add(jp.ToString);
  finally
    jp.Free;
  end;
```

上面的程式碼使用了第一個建構函式建立 TJSONPair 物件，TJSONPair 類別的 ToString 方法可以把 JSON 物件之中的內容以文字字串形式回傳，最後釋放 TJSONPair 物件時，TJSONPair 物件會自動釋放傳入建構函式之中的兩個 TJSONString 物件。

當然您也可以使用第二個建構函式來建立 TJSONPair 物件，如下所示:

```
jp := TJSONPair.Create('書名', TJSONString.Create('實戰 Delphi 2010'));
```

或是使用第 3 個建構函式，因為在這個範例中『名稱/值』配對中的名稱和值都是字串:

```
jp := TJSONPair.Create('書名', '實戰 Delphi 2010');
```

當然，如果您需要建立如下的 JSON『名稱/值』配對:

```
"價格":45.95
```

由於值是數字，因此我們只能使用第一個或是第二個建構函式來建立:

```
jp := TJSONPair.Create(TJSONString.Create('價格'), TJSONNumber.Create(45.95));
```

或是:

```
jp := TJSONPair.Create('價格', TJSONNumber.Create(45.95));
```

一旦建立了 TJSONPair 物件，我們也可以藉由存取它的 JsonString 和 JsonValue 特性來存取『名稱/值』配對中的名稱或是值了，例如:

```
Memo1.Lines.Add('JSONString : ' + jp.JsonString.ToString);
Memo1.Lines.Add('JSONValue : ' + jp.JsonValue.ToString);
```

## 2-2-4 TJSONString 類別

TJSONString是從TJSONValue類別繼承下來，它是使用來代表Unicode字串的資料，它可以是『名稱/值』配對中的名稱或是值或是兩者。

TJSONString 最重要的方法應該是它的建構函式了，它接受一個 Unicode 字串做為參數:

```
constructor Create(const Value: UnicodeString); overload;
```

在 Unicode 字串使用建立 TJSONString 物件之後，如果開發人員需要加入額外的字元，那麼可以呼叫 AddChar 虛擬程序，AddChar 會在原本的 Unicode 字串之後加入參數 Ch 的字元內容。

```
procedure AddChar(const Ch: WideChar); virtual;
```

如果開發人員需要 TJSONString 物件中字串的內容值，可以呼叫 ToString:

```
  function ToString: UnicodeString; override;
```

如果開發人員只是需要 TJSONString 物件中字串的內容值，而不需要額外的開始”符號以及結束的”符號，那麼可以呼叫 TJSONString 的 Value 函式

```
  function Value: UnicodeString; override;
```

例如下面的程式碼:

```
var
 js : TJSONString;
begin
 js := TJSONString.Create('實戰 Delphi 2010');
 try
  Memo1.Lines.Add('TJSONString.ToString : '+ js.ToString);
  Memo1.Lines.Add('TJSONString.Value : '+ js.Value);
 finally
  js.Free;
 end;
```

版權所有 請勿翻印

下面是呼叫 ToString 和 Value 的差異:

```
TJSONString.ToString : "實戰 Delphi 2010"
TJSONString.Value : 實戰 Delphi 2010
```

## 2-2-5 TJSONObject 類別

TJSONObject 類別從 TJSONValue 直接繼承下來，它代表 JSON 規範中封裝物件的類別。要建立 TJSONObject 物件非常簡單，只需要呼叫它的建構函式即可:

```
constructor Create;
```

那麼如果我們需要建立如下的 JSON 物件內容，那麼應該如何做呢?

```
{“書名”:”實戰 Delphi 2010”}
```

請注意上面的結構，其實是在 JSON 物件之中包含一個 TJSONPair 物件，因此我們只需要執行下面的步驟即可:

- 建立 TJSONObject 物件
- 建立 TJSONPair 物件

➢ 把 TJSONPair 物件加入到 TJSONObject 物件

要把 TJSONPair 物件或是 JSON『名稱/值』配對加入到 TJSONObject 物件中，我們可以使用 TJSONObject 類別中下面兩個複載的 AddPair 方法:

```
  procedure AddPair(const Pair: TJSONPair); overload; virtual;
  procedure AddPair(const Str: TJSONString; const Val: TJSONValue); overload; virtual;
```

因此如果我們需要建立如下內容的 TJSONObject 物件:

```
{"書名":"實戰 Delphi 2010","出版年份":2010}
```

那麼可以使用如下的程式碼:

```
var
  jo : TJSONObject;
  jp : TJSONPair;
begin
  jo := TJSONObject.Create;
  try
    jp := TJSONPair.Create('書名', '實戰 Delphi 2010');
    jo.AddPair(jp);
    jo.AddPair(TJSONString.Create('出版年份'), TJSONNumber.Create(2010));
    Memo1.Lines.Add(jo.ToString);
  finally
    jo.Free;
  end;
```

同樣的 TJSONObject 類別的 ToString 方法可以把其中的內容以文字字串型態回傳:

```
  function ToString: UnicodeString; override;
```

## 2-2-6 TJSONNumber 類別

TJSONNumber 類別是從 TJSONString 類別繼承下來的，這是因為在 JSON 規範中整數和浮點數都必須使用字串形式來代表，由於 TJSONString 已經提供了使用字串形式來代表 JSON 內容的功能，因此 TJSONNumber 只需要從它繼承下來並且把整數和浮點數轉換為文字字串型態即可。

在前面我們看過多次建立 TJSONNumber 物件的範例了，它的建構函式接受一個 double 值的參數:

```
constructor Create(const Value: Double); overload;
```

一旦建立了 TJSONNumber 物件，呼叫它的 ToString 方法即可取得其內容。

在離開本小節之前再讓我們看看三個剩下簡單的類別，它們是 TJSONTrue，TJSONFalse 和 TJSONNull。這三個類別分別實作圖 3 中的 true，false 和 null 三個 JSON 數值。例如我們如果需要下面的 JSON 內容:

```
{"書名":"實戰 Delphi 2010","出版否":false, "撰寫中":true, "出版商":null}
```

那麼我們使用下面的程式碼即可:

```
var
  jo : TJSONObject;
  jp: TJSONPair;
begin
  jo := TJSONObject.Create;
  try
    jp := TJSONPair.Create('書名', '實戰 Delphi 2010');
    jo.AddPair(jp);
    jp := TJSONPair.Create('書名', TJSONFalse.Create);
    jo.AddPair(jp);
    jp := TJSONPair.Create('撰寫中', TJSONTrue.Create);
    jo.AddPair(jp);
    jp := TJSONPair.Create('出版商', TJSONNull.Create);
    jo.AddPair(jp);
    Memo1.Lines.Add(jo.ToString);
  finally
    jo.Free;
  end;
```

## 2-2-7 TJSONArray 類別

TJSONArray 類別從 TJSONValue 繼承下來，它的目的是實作 JSON 定義中 JSON 陣列的元素，在 JSON 規範中 JSON 陣列的定義如圖 7 所示:

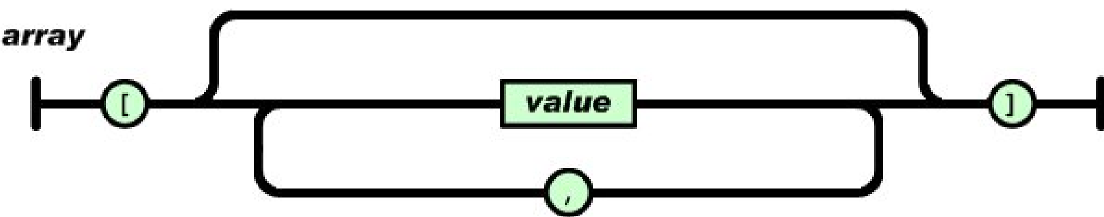



圖 7 JSON 陣列的定義

JSON 陣列可封裝 JSON 數值為陣列的元素，每一個封裝的 JSON 數值元素以分號分隔。由於在 VCL 中封裝 JSON 數值的類別是 TJSONValue，如前所述 TJSONValue 是抽象類別，也是每一個封裝 JSON 元素的祖先類別，因此這代表 TJSONArray 類別幾乎可以封裝任何的複雜的資料，例如當 dbExpress 以 JSON 形式傳遞資料時，以會使用 TJSONArray 封裝 TDataSet 包含的資料。

TJSONArray 類別提供了數個方法可讓開發人員在其中加入元素，例如下面即是 TJSONArray 類別提供相關加入元素的方法:

```
  procedure AddElement(const Element: TJSONValue);
  function Add(const Element: UnicodeString): TJSONArray; overload;
  function Add(const Element: Integer): TJSONArray; overload;
  function Add(const Element: Double): TJSONArray; overload;
  function Add(const Element: Boolean): TJSONArray; overload;
  function Add(const Element: TJSONObject): TJSONArray; overload;
  function Add(const Element: TJSONArray): TJSONArray; overload;
```

開發人員可以呼叫 AddElement 在 TJSONArray 物件中加入 TJSONValue 型態的物件元素，或是呼叫 Add 方法直接在其中加入字串，整數，浮點數，甚至是 TJSONObject 或是 TJSONArray 型態的物件。請注意的是，由於 Add 方法會回傳 TJSONArray 物件本身，因此我們可以類似下面的程式碼在 TJSONArray 物件中加入元素:

版權所有 請勿翻印

```
procedure TForm18.Button1Click(Sender: TObject);
var
  aJA : TJSONArray;
begin
  aJA := TJSONArray.Create;
  try
    aJA.Add('JSON 陣列元素').Add(Now).Add(True).Add(False).Add(TJSONObject.Create);
    Memo1.Lines.Text := aJA.ToString;
  finally
    aJA.Free;
  end;
end;
```

下面是執行的結果畫面:

圖 8 使用 TJSONArray 物件封裝元素

要取得 TJSONArray 中的元素，開發人員可以呼叫 Get 方法並且傳入元素的索引位置:

```
function Get(const Index: Integer): TJSONValue;
```

例如我們可以使用下面的程式碼取得前面範例中 TJSONArray 物件中的每一個元素:

```
procedure TForm18.Button2Click(Sender: TObject);
var
  aJA : TJSONArray;
  iIndex : Integer;
begin
  aJA := TJSONArray.Create;
  try
    aJA.Add('JSON 陣列元素').Add(Now).Add(True).Add(False).Add(TJSONObject.Create);

    for iIndex := 0 to aJA.Size - 1 do
    begin
      ListBox1.Items.Add(aJA.Get(iIndex).ToString);
    end;
  finally
    aJA.Free;
  end;
end;
```

下面是執行的結果畫面:

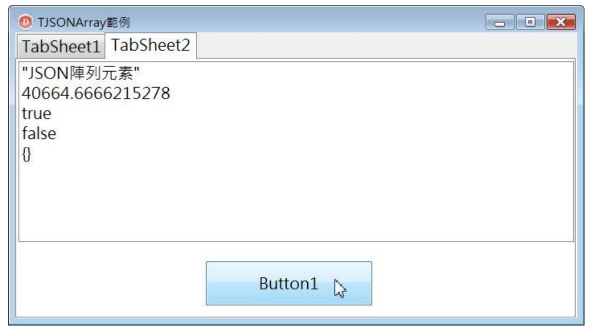



圖 9 使用 Get 方法擷取 TJSONArray 中的元素

當然在上面的程式碼中使用了 for 迴圈從 TJSONArray 中把每一個元素取出，但我們也可以使用 10.3 中新的 TJSONArrayEnumerator 類別來進行一樣的工作，例如下面的程式碼在 010 行建立 TJSONArrayEnumerator 物件並且傳入 TJSONArray 物件做為參數，接著就可以在 011 行藉由呼叫 TJSONArrayEnumerator 類別的 MoveNext 方法來判斷是否還有元素可存取，如果 MoveNext 回傳 True 的話，就存取 TJSONArrayEnumerator 物件的 Current 特性以取得目前 TJSONArray 中對應的元素。

```
001   procedure TForm18.Button3Click(Sender: TObject);
002   var
003     aJA : TJSONArray;
004     jae : TJSONArrayEnumerator;
005   begin
006     aJA := TJSONArray.Create;
007     try
008       aJA.Add('JSON 陣列元素
').Add(Now).Add(True).Add(False).Add(TJSONObject.Create);
009
010       jae := TJSONArrayEnumerator.Create(aJA);
011       while jae.MoveNext do
012         ListBox2.Items.Add(jae.Current.ToString);
013     finally
014       jae.Free;
015       aJA.Free;
016     end;
017   end;
```

在讀者瞭解了如何使用這些 JSON 相關的類別之後，讓我們看看如何使用它們在分散式應用系統之中。

前面的章節簡單的了如何使用 DataSnap 開發基於 JSON 分散式應用系統，Delphi 藉由原本的 Midas/DataSnap 和 dbExpress 的元件提供這個新的分散式計算能力，但使用這些元件都是資料的方式來存取遠端服務，然而我們也可以使用前面介紹的 JSON 類別再結合 DataSnap 來實作出存取遠端數值物件(Value Object)或是所謂的 DTO(Data Transfer Object)的應用，讓用戶端可以存取遠端的物件。

在下面的範例中我們將展示如何開發使用 VO/DTO 的 DataSnap 應用程式伺服器，並且在用戶端擷取其中封裝的資料和物件。這個範例將開發一個可以讓用戶端查詢資料和物件的 DataSnap 應用程式伺服器，當用戶端查詢物件時，我們將使用 TJSONObject 封裝物件並且傳遞到用戶端。

## 2-3-1 開發數值物件伺服器

首先建立一個 VCL Form 應用程式專案，在它的主表單中加入如下的元件:

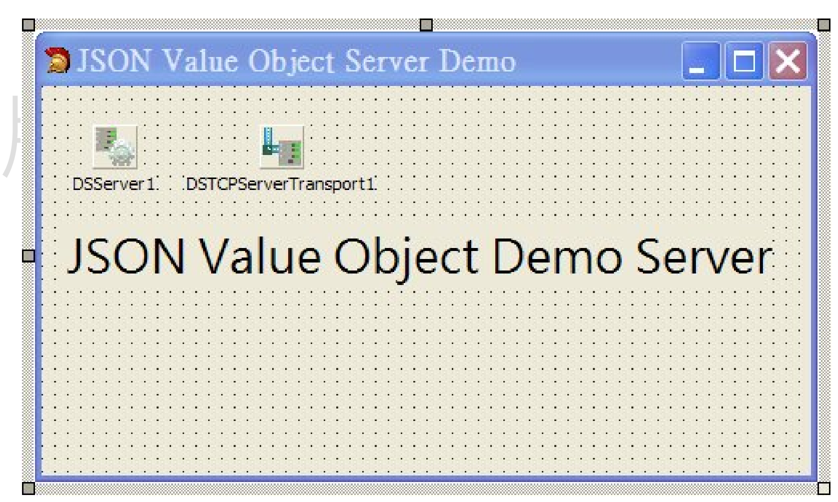



圖 10 VO DataSnap 應用程式伺服器

接著在專案中建立一個新的程式單元，於其中實作一個簡單的 TEmployee 類別如下:

```
unit uEmployee;

interface

uses SysUtils, classes;

type
```

```
  TEmployee = class
  private
    { Private declarations }
    FName : string;
    FEMail : string;
    FPhone : string;
  public
    { Public declarations }
    constructor Create(sName  : string = ''; sEMail : string = ''; sPhone : string = '');

    property Name : string read FName write FName;
    property EMail : string read FEmail write FEMail;
    property Phone : string read FPhone write FPhone;
  end;

implementation

{ TEmployee }

constructor TEmployee.Create(sName, sEMail: string; sPhone: string);
begin
  FName := sName;
  FEmail := sEMail;
  FPhone := sPhone;
end;

end.
```

再於專案中建立一個新的程式單元稱為 uEmployeeValueObject，其中定義 TEmployeeVO 類別如下，請注意 TEmployeeVO 類別使用了{$MethodInfo ON}和{$MethodInfo Off}編譯器指令要求編譯器把 TEmployeeVO 類別的 Reflection 資訊編譯到執行檔中。

```
  {$MethodInfo ON}
  TEmployeeVO = class(TComponent)
  private
    { Private declarations }
    FEmployees : TObjectList<TEmployee>;

```

```
  function CreateEmployeeJSONObject(employee : TEmployee) : string;
  function CreateEmployeeJSONArray : string;
  function CreateEmployeeJSONObjectJ(employee : TEmployee) : TJSONObject;
public
  { Public declarations }
  destructor Destroy; override;
  function GetEmployeeJ(const sName : string) : TJSONObject;
  function GetAllEmployeesJ : TJSONArray;
  procedure AddEmployee(const sName : string; const sEMail : string; const sPhone : string);
end;
{$MethodInfo Off}
```

TEmployeeVO 宣告了三個方法說明如下:

| 方法名稱 | 說明 | 備註 |
|---|---|---|
| AddEmployee | 讓用戶端呼叫增加員工資訊 | |
| GetEmployeeJ | 讓用戶端以員工姓名查詢員工物件 | 請注意 GetEmployeeJ 回傳 TJSONObject 型態的結果值，這代表範例應用程式將把員工物件封裝在 TJSONObject 物件之中並且回傳給用戶端 |
| GetAllEmployeesJ | 讓用戶端查詢所有的員工資訊 | 查詢後端所有員工資訊，請注意 GetAllEmployeesJ 回傳型態為 TJSONArray 的結果值，這代表範例應用程式將把所有查詢結果封裝在 TJSONArray 物件中回傳 |

版權所有 請勿翻印

下面小節將簡單的說明這些方法的實作。

## AddEmployee 方法的實作

AddEmployee 方法非常的簡單，它從用戶端接受三個字串型態的參數，並且根據它們來建立 TEmployee 物件，最後再把建立的 TEmployee 物件加入在宣告為 TObjectList<TEmployee>泛型型態的變數 FEmployees 之中。

```
procedure TEmployeeVO.AddEmployee(const sName, sEMail, sPhone: string);
var
  employee : TEmployee;
begin
```

```
 if (FEmployees = nil) then
   FEmployees := TObjectList<TEmployee>.create(true);
 employee := TEmployee.Create(sName, sEmail, sPhone);
 FEmployees.Add(employee);
end;
```

## GetEmployeeJ 方法的實作

GetEmployeeJ 方法就非常的有趣了，它展示了如何於 DataSnap 應用程式伺服器中使用 TJSONValue 的相關類別，GetEmployeeJ 回傳型態為 TJSONObject 的物件回用戶端。

GetEmployeeJ 根據用戶端傳遞來查詢的員工姓名，在 Fem ployees 中一一的搜尋具有相同名稱的 TEmployee 物件，找到之涕在 013 行呼叫 CreateEmployeeJSONObjectJ 來建立回傳的 TJSONObject 物件。

```
001   function TEmployeeVO.GetEmployeeJ(const sName: string): TJSONObject;
002   var
003     ie : TList<uEmployee.TEmployee>.TEnumerator;
004     employee : TEmployee;
005   begin
006     Result := nil;
007     ie := FEmployees.GetEnumerator;
008     while (ie.MoveNext) do
009     begin
010       employee := ie.Current;
011       if (employee.Name = sName) then
012       begin
013         Result := CreateEmployeeJSONObjectJ(employee);
014         break;
015       end;
016     end;
017   end;
```

CreateEmployeeJSONObjectJ 方法根據找到的 TEmployee 物件來建立 TJSONObject 物件，它呼叫我們前面學習過的 AddPair 方法，把每一個 TEmployee 物件的特性名稱和特性值做為一個 JSON 的中『名稱/值』配對加入到 TJSONObject 物件。

```
function TEmployeeVO.CreateEmployeeJSONObjectJ(employee: TEmployee): TJSONObject;
var
```

```
  aJO : TJSONObject;
begin
  aJO := TJSONObject.Create;
  aJO.AddPair(TJSONString.Create('姓名'), TJSONString.Create(employee.Name));
  aJO.AddPair(TJSONString.Create('EMail'), TJSONString.Create(employee.EMail));
  aJO.AddPair(TJSONString.Create('電話'), TJSONString.Create(employee.Phone));
  Result := aJO;
end;
```

## GetAllEmployeesJ 方法的實作

GetAllEmployeesJ 方法會把所有存在泛型型態變數 FEmployees 之中的 TEmployee 物件封裝在 TJSONArray 物件中回傳給用戶端。

在 013 行仍然是呼叫 CreateEmployeeJSONObjectJ 為每一個 CreateEmployeeJSONObjectJ 建立一個 TJSONObject 物件，並且加入到回傳的 TJSONArray 物件中。

```
001   function TEmployeeVO.GetAllEmployeesJ: TJSONArray;
002   var
003     ie : TList<uEmployee.TEmployee>.TEnumerator;
004     employee : TEmployee;
005     jo : TJSONObject;
006     ja : TJSONArray;
007   begin
008     ie := FEmployees.GetEnumerator;
009     ja := TJSONArray.Create;
010     while (ie.MoveNext) do
011     begin
012       employee := ie.Current;
013       jo := CreateEmployeeJSONObjectJ(employee);
014       ja.AddElement(jo);
015     end;
016     Result := ja;
017   end;
```

最後不要忘記在主表單中註冊 TEmployeeVO 類別，如此一來用戶端才能看見並且呼叫 TEmployeeVO 輸出的方法:

```
procedure TForm10.FormCreate(Sender: TObject);
begin
```

```
  if DSServer1.Started then
    DSServer1.Stop;
  RegisterServers;
  DSServer1.Start;
end;

procedure TForm10.FormDestroy(Sender: TObject);
begin
  DSServer1.Stop;
end;

procedure TForm10.RegisterServers;
begin
  uEmployeeValueObject.RegisterServerClasses(Self, DSServer1);
end;
```

現在編譯並且執行 DataSnap 應用程式伺服器，並且準備開發用戶端。

## 2-3-2 開發範例用戶端

版權所有 請勿翻印

其實這個範例的重點就在於用戶端如何呼叫遠端支援 TJSONValue 相關類別的方法，這是因為在 DataSnap 2009 中無法支援 TJSONValue 相關類別，到了 DataSnap 2010 才支援。但是在筆者撰寫本章時 Delphi 2010 的 Beta 版仍然無法自動產生代表遠端類別的用戶端 Proxy 類別，因此想要呼叫前面實作的 GetEmployeeJ 和 GetAllEmployeesJ 方法，我們必須瞭解如何修改由 Delphi 產生的 DataSnap 用戶端類別。

> 筆者認為當 Delphi 2010 正式版釋出時，CodeGear 應該會把這個問題修正，如果您發現您使用的 Delphi 已經能夠產生正確的用戶端 Proxy 類別，那麼就不需要如下所敘述的修改了。

首先建立一個 VCL Form 應用程式，放入 TSQLConnection 元件，設定 Driver 為 DataSnap 再把 connected 設定為 True(請確定 DataSnap 應用程式伺服器已經在執行)，接著用點選滑鼠右鍵，選擇 Generate DataSnap client classes 選項，儲存產生的程式單元為 uServerProxy.pas，然後搜尋 GetEmployeeJ 方法，把 012 行修改為如下:

```
001    function TEmployeeVOClient.GetEmployeeJ(sName: string): TJSONObject;
002    begin
```

```
003     if FGetEmployeeJCommand = nil then
004     begin
005       FGetEmployeeJCommand := FDBXConnection.CreateCommand;
006       FGetEmployeeJCommand.CommandType := TDBXCommandTypes.DSServerMethod;
007       FGetEmployeeJCommand.Text := 'TEmployeeVO.GetEmployeeJ';
008       FGetEmployeeJCommand.Prepare;
009     end;
010     FGetEmployeeJCommand.Parameters[0].Value.SetWideString(sName);
011     FGetEmployeeJCommand.ExecuteUpdate;
012     Result := FGetEmployeeJCommand.Parameters[1].Value.GetJSONValue as TJSONObject;
013   end;
```

這是因為遠端 GetEmployeeJ 方法回傳 TJSONObject 型態的物件，因此我們需要把 012 行呼叫遠端方法執行的結果轉變型態為 TJSONObject 型態。

同樣的，我們也需要修改 GetAllEmployeesJ 方法，轉變型態為回傳 TJSONArray 物件，如下所示:

```
function TEmployeeVOClient.GetAllEmployeesJ: TJSONArray;
begin
  if FGetAllEmployeesJCommand = nil then
  begin
    FGetAllEmployeesJCommand := FDBXConnection.CreateCommand;
    FgetAllEmployeesJCommand.CommandType := TDBXCommandTypes.DSServerMethod;
    FGetAllEmployeesJCommand.Text := 'TEmployeeVO.GetAllEmployeesJ';
    FGetAllEmployeesJCommand.Prepare;
  end;
  FGetAllEmployeesJCommand.ExecuteUpdate;
  Result := FGetAllEmployeesJCommand.Parameters[0].Value.GetJSONValue as TJSONArray;
end;
```

瞭解了如何以及為什麼需要修改 DataSnap 用戶端類別之後，我們就可以開始實作呼叫遠端方法的程式碼了。

首先下面的程式碼藉由自動產生的 TEmployeeVOClient 類別呼叫 AddEmployee 方法，把用戶端輸入的員工資料加入在遠端的應用程式伺服器中:

```
procedure TForm11.btnAddEmployeeClick(Sender: TObject);
var
  evo : TEmployeeVOClient;
begin
```

```
  evo := TEmployeeVOClient.Create(Self.SQLConnection1.DBXConnection);
  try
    evo.AddEmployee(edtAddName.Text, edtAddEMail.Text, edtAddPhone.Text);
    edtAddName.Text := '';
    edtAddEMail.Text := '';
    edtAddPhone.Text := '';
  finally
    evo.Free;
  end;
end;
```

下圖是執行範例用戶端應用程式並且點選『增加員工』按鈕以執行上述程式碼的畫面:

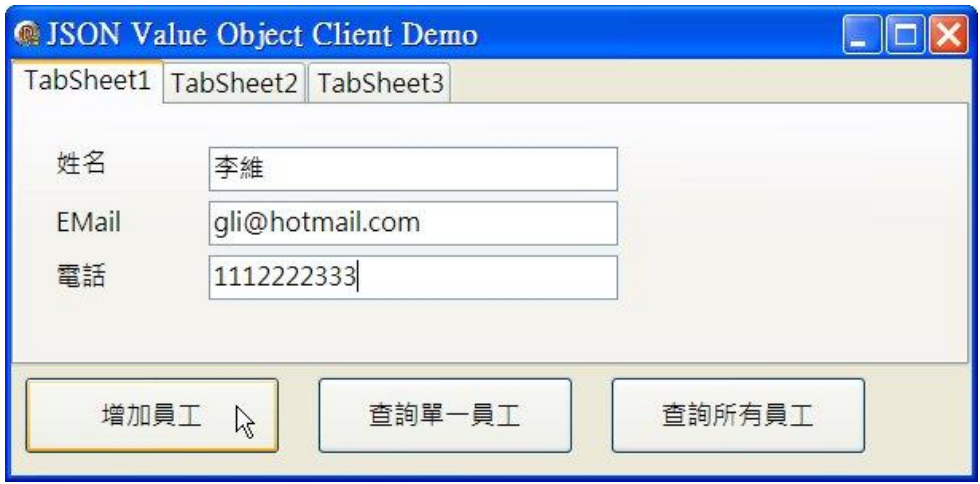



圖 11 用戶端應用程式呼叫遠端 AddEmployee 方式加入員工資訊

接著是藉由 TEmployeeVOClient 呼叫 GetEmployeeJ 查詢員工資料的實作程式碼:

```
procedure TForm11.btnQueryEmployeeClick(Sender: TObject);
var
  evo : TEmployeeVOClient;
  jo : TJSONObject;
  employee : TEmployee;
begin
  evo := TEmployeeVOClient.Create(Self.SQLConnection1.DBXConnection);
  try
    jo := evo.GetEmployeeJ(edtQname.Text);
    employee := TEmployee.Create(jo.Get(0).JsonValue.ToString,
jo.Get(1).JsonValue.ToString, jo.Get(2).JsonValue.ToString);
```

```
    Self.edtQEMail.Text := employee.EMail;
    Self.edtQPhone.Text := employee.Phone;
  finally
    FreeAndNil(employee);
    FreeAndNil(jo);
    evo.Free;
  end;
end;
```

在上面的程式碼中呼叫 GetEmployeeJ 取得代表員工的 TJSONObject 物件，再根據 TJSONObject 物件中的資訊於用戶端建立 TEmployee 物件，再顯示員工資訊於表單中，最後不要忘記釋放 TEmployee 物件，TJSONObject 物件和 TEmployeeVOClient 物件。

下圖是先增加李維這筆員工資料之後，再輸入李維來查詢的畫面:

圖 12 輸入員工姓名查詢員工資料

點選上圖中的『查詢單一員工』按鈕之後我們的確可以取得遠端員工物件的資訊，如下圖所示:

圖 13 查詢的結果畫面

但是為什麼上圖中的員工資訊，例如員工的 Email 有引號包圍呢? 這當然是因為這是 JSON 封裝字串的規範，而在上面的程式碼中當我們建立 TEmployee 物件時是直接呼叫 ToString 方法，如果我們不希望 TEmployee 物件的特性值有引號包圍，那麼我們可以修改程式碼如下，改呼叫 Value 方法:

```
   employee := TEmployee.Create(jo.Get(0).JsonValue.Value, jo.Get(1).JsonValue.Value,
jo.Get(2).JsonValue.Value);
```

那麼就會有如下正確的結果:

版權所有 請勿翻印

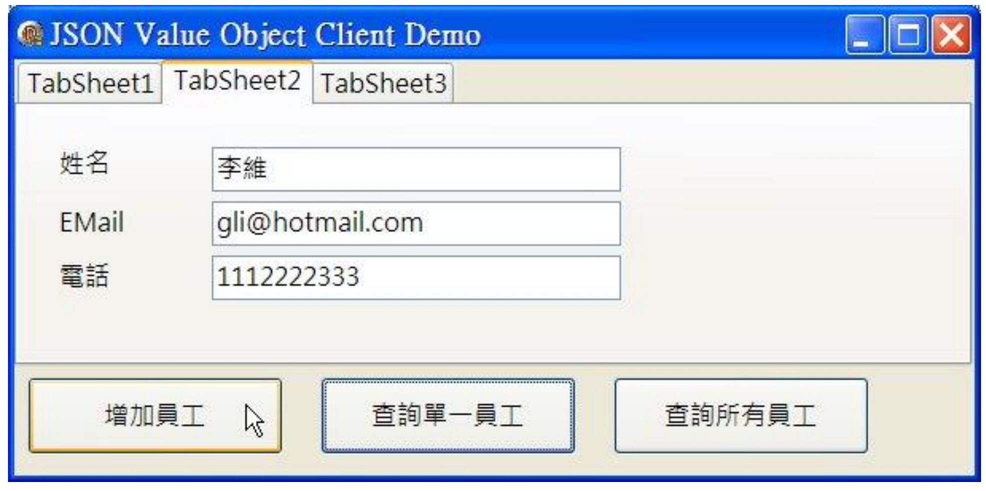



圖 14 查詢的結果畫面

最後讓我們看看如何查詢所有的員工資料。下面的程式碼藉由 TEmployeeVOClient 呼叫 GetAllEmployeesJ 取得 TJSONArray 物件，然後進入 for 迴圈把其中的每一個元素取出再轉變型態為 TJSONObject 物件，最後再根據 TJSONObject 物件一一的建立用戶端的員工物件。

```
procedure TForm11.btnQueryAllEmployeesClick(Sender: TObject);
var
```

```
  evo : TEmployeeVOClient;
  jo : TJSONObject;
  ja : TJSONArray;
  employee : TEmployee;
  iIndex: Integer;
begin
  evo := TEmployeeVOClient.Create(Self.SQLConnection1.DBXConnection);
  try
    ja := evo.GetAllEmployeesJ;
    for iIndex := 0 to ja.Size - 1 do
    begin
      jo := ja.Get(iIndex) as TJSONObject;
      employee := TEmployee.Create(jo.Get(0).JsonValue.ToString,
jo.Get(1).JsonValue.ToString, jo.Get(2).JsonValue.ToString);
      Memo1.Lines.Add(employee.Name);
      FreeAndNil(employee);
    end;
  finally
    FreeAndNil(ja);
    FreeAndNil(employee);
    evo.Free;
  end;
end;
```

下圖是執行此查詢的結果，我們可以看到用戶端的確可以查詢到遠端的所有員工的資訊。

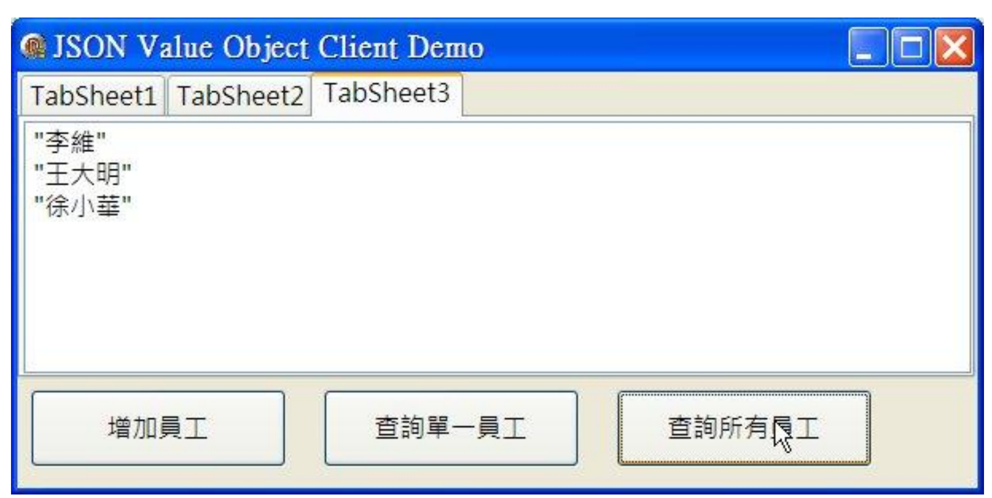



圖 15 所有員工資料查詢的結果畫面

當然，我們一樣可以修改上面的程式碼如下:

```
    employee := TEmployee.Create(jo.Get(0).JsonValue.Value, jo.Get(1).JsonValue. Value,
jo.Get(2).JsonValue. Value);
```

那麼我們會看到如下的結果:

圖 16 所有員工資料查詢的結果畫面

## 2-4 使用 JSON 封裝和傳遞資料

版權所有 請勿翻印

現在再讓我們看看如何使用 TDBXJSONTools 類別幫助開發人員封裝和傳遞複雜的資料。

### 2-4-1 開發 DataSnap REST 伺服器

首先在 Delphi 整合發展環境中建立一個 DataSnap REST 應用程式專案，接著在 ServerMethodsUnit1 程式單元中宣告下面的兩個方法:

```
function GetData : String;
function GetImage : TJSONArray;
```

GetData 方法將把 TDataSet 中包含的資料以 JSON 的形式傳遞到用戶端，讓讀者瞭解如何把資料集轉換為 JSON 資料。

GetImage 方法把是把資料集中 Blob 欄位封裝的圖形資料轉換為 JSON 形式的資料，讓讀者瞭解如何把 2 進位的資料換為 JSON 資料。

下面的 GetData 方法的實作程式碼，首先它使用 TDBXCommand 物件執行 SQL 命令，TDBXCommand 物件的 ExecuteQuery 方法執行 SQL 敘述之後會回傳 TDBXReader 物件，其中就封裝了 SQL 敘述執行的結果資料集，接著我們就可以呼叫 TDBXJSONTools 的類別方法 TableToJSON 把 TDBXReader 物件封裝的資料轉換為以 JSON 封裝的字串資料，再回傳給用戶端。

```
function TServerMethods1.GetData: String;
var
  aCommand : TDBXCommand;
begin
  Result := '';
  aCommand := CHINESEDEMO.DBXConnection.CreateCommand;
  try
    aCommand.Text := 'select CATEGORY, COMMON_NAME, TOPOTYPE from BIOLIFE';
    Result := TDBXJSONTools.TableToJSON(aCommand.ExecuteQuery, 10, True).ToString;
  finally
    aCommand.Free;
  end;
end;
```

GetImage 方法則是把 BIOLIFE 資料表中 Graphic 欄位包含的圖形資料先儲存到 TMemoryStream 物件中，再呼叫 TDBXJSONTools 的類別方法 StreamToJSON 把 2 進位形態的資料轉換為 JSON 封裝的字串形態資料再傳遞回用戶端。

```
function TServerMethods1.GetImage: TJSONArray;
var
  aMS : TMemoryStream;
begin
  BIOLIFE.Active := True;
  aMS := TMemoryStream.Create;
  try
    BIOLIFEGRAPHIC.SaveToStream(aMS);
    aMS.Position := 0;
    Result := TDBXJSONTools.StreamToJSON(aMS, 0, aMS.Size);
  finally
    BIOLIFE.Active := False;
    aMS.Free;
  end;
end;
```

現在編譯並且執行此範例 DataSnap REST 應用程式伺服器。

## 2-4-2 開發範例用戶端

接著在專案群組中建立一個 VCL Form 應用程式專案，再於專案中建立 DataSnap Client Module，然後在主表單中撰寫如下的程式碼呼叫伺服器的 GetData 方法:

```
procedure TForm18.Button1Click(Sender: TObject);
begin
  Memo1.Lines.Text := ClientModule1.ServerMethods1Client.GetData;
end;
```

下圖是執行上述程式碼的結果，我們可以清楚的看到資料集的資料被封裝成 TJSONObject 傳遞到用戶端，每一筆資料又藉由 TJSONArray 封裝。

圖 17 傳遞 TDataSet 包含的資料

接著再於主表單中撰寫如下的程式碼以呼叫伺服器的 GetImage 方法:

```
procedure TForm18.btnJSONImageClick(Sender: TObject);
var
  aJA : TJSONArray;
  aStream : TStream;
begin
  aJA := ClientModule1.ServerMethods1Client.GetImage;
  Memo1.Lines.Text := aJA.ToString;
  try
    aStream :=  TDBXJSONTools.JSONToStream(aJA);
    Image1.Picture.Bitmap.LoadFromStream(aStream);
  finally
```

```
    aStream.Free;
  end;
end;
```

在上面的程式碼中首先把呼叫 GetImage 方法取得的 JSON 封裝的資料顯示在主表單的 TMemo 控制項中，接著再藉由呼叫 TDBXJSONTools 的類別方法 JSONToStream 把伺服端傳遞來的 TJSONArray 物件轉換回 TStream 物件(其實是 TBytesStream 物件)，最後藉由主表單中的 TImage 元件把圖形資料顯示在用戶端的主表單中。

下面的畫面是伺服端回傳的圖形資料，我們可以清楚的看到圖形資料被封裝成 TJSONArray 物件:

圖 18 以 JSON 型態封裝的圖形資料

而下面的畫面則是使用 TDBXJSONTools. JSONToStream 方法把上圖中的字串型態的資料轉換回圖形資料並且顯示在 TImage 元件中的結果，我們可以看到藉由 TDBXJSONTools 類別，開發人員可以輕易的封裝和傳遞複雜型態的資料。

圖 19 藉由 TDBXJSONTools 類別把資料轉換回圖形

由於此範例伺服器是 REST 伺服器，因此我們也可以使用瀏覽器來呼叫 GetData 和 GetImage 方法。例如下圖就是在瀏覽器中呼叫 GetData 方法的結果:

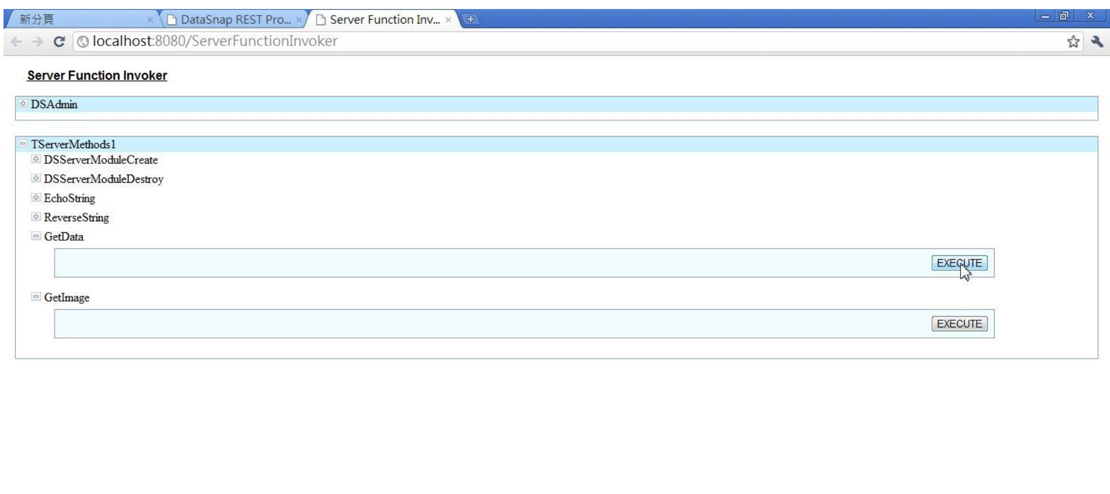
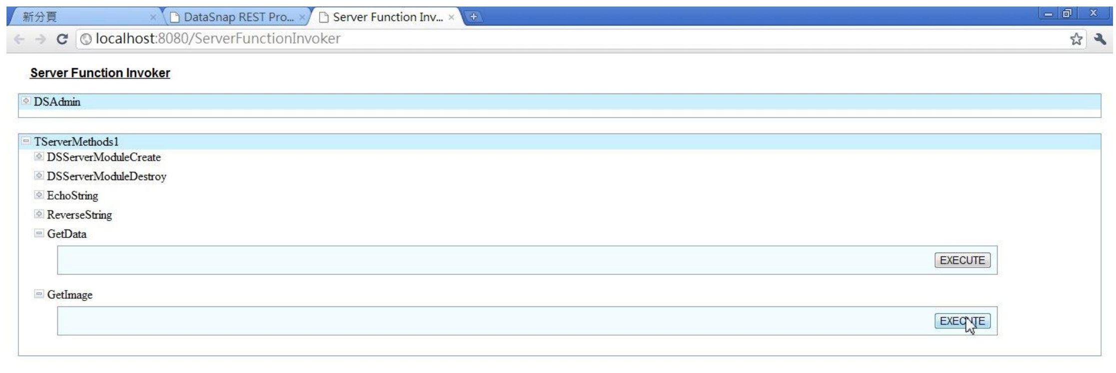



圖 20 使用瀏覽器呼叫伺服器的 GetData 方法

而下圖則是在瀏覽器中呼叫 GetImage 方法的結果:



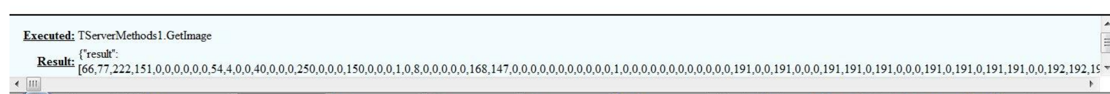




圖 21 使用瀏覽器呼叫伺服器的 GetImage 方法

最後讀者想提醒讀者的是，雖然結合 DataSnap 相關的 JSON 類別和 TDBXJSONTools 類別可以封裝和傳遞任何複雜的資料，但使用 JSON 傳遞大量的資料仍然是不智的，因為以 JSON 傳遞大量的資料不僅緩慢而且會消耗網路頻寬，JSON 只適合傳遞少量的資料，其目的是跨平台，和整合跨平台的應用。如果開發人員真的要使用 JSON 傳遞傳統資料，也應該儘量減少資料量。

版權所有 請勿翻印

對於主從架構，多層應用系統而言 Delphi 和 C++Builder 的開發人員可以在傳遞大量資料時仍然使用 dbExpress，需要在 Web，跨平台傳遞資料時再選用 JSON，Delphi 和 C++Builder 提供同時提供了效率(dbExpress)和彈性(JSON)的方式讓開發人員自行選擇傳遞/存取資料的技術。

## 2-5 結論

本章敘述了 Delphi VCL 框架中支援 JSON 開發的相關類別，讀者從本章中可以瞭解這些不但是依照 JSON 規範設計的，而且非常的直覺，好用，只要我們對於 JSON 規範有基本的掌握就可以順利的使用它們。

此外新版的 DataSnap 不但使用 JSON 做為封裝和傳遞資料的基礎，現在也加入支援使用 TJSONValue 和衍生類別做為遠端方法的參數和回傳值，如此一來可以讓開發人員撰寫客製化的 JSON 應用，例如使用 VO/DTO 設計樣例來傳遞資料和物件，使用這樣的技術，開發人員也可以使用.NET，Java 或是任何支援 JSON 的程式語言來開發用戶端應用程式。

DataSnap 10.3 提供了 TDBXJSONTools 類別可幫助開發人員以 JSON 封裝資料集或是任何 2 進位形式的資料，再搭配 TJSONArray 和 TJSONObject 類別，開發人員就可以使用 JSON 封裝和傳遞任何複雜型態的資料了。

版權所有 請勿翻印

# 第3章 DataSnap/REST伺服器的授權和認證

在前一章中我們討論了如何開發 DataSnap/REST 伺服器和 Win32 的用戶端應用程式，然而由於 REST 和 JSON 適用於任何平台和用戶端的應用，因此即使 Web 或是移動的用戶端也能夠連結到 DataSnap/REST 伺服器並且呼叫其中的服務，本章將從如何開發 DataSnap/REST 的 Web 應用程式說起，接著會討論如何對於 DataSnap/REST 伺服器中公開的服務方法進行授權和認證的安全機制功能。

版權所有 請勿翻印

## 3-1 開發 DataSnap/RESTful Web 用戶端應用程式

> 注意:本範例需要使用 VCL For Web(IntraWeb)元件組

現在讓我們說明如何開發 VCL For Web 的應用程式來呼叫使用 DataSnap/REST 伺服器。在 Delphi 整合發展環境中開啟上一章的專案群組，在專案群組中建立 VCL For Web 應用程式， 如下圖所示:

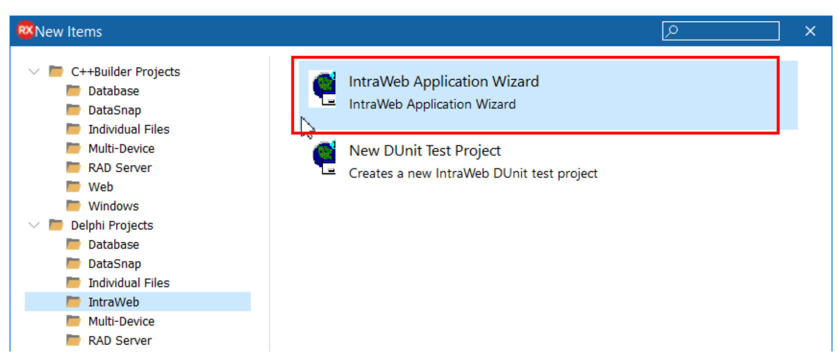



圖 3-1 建立 VCL For Web 應用程式

點選了 OK 按鈕之後，再讓我們選擇建立 StandAlone 型態的應用程式，如下圖所示:

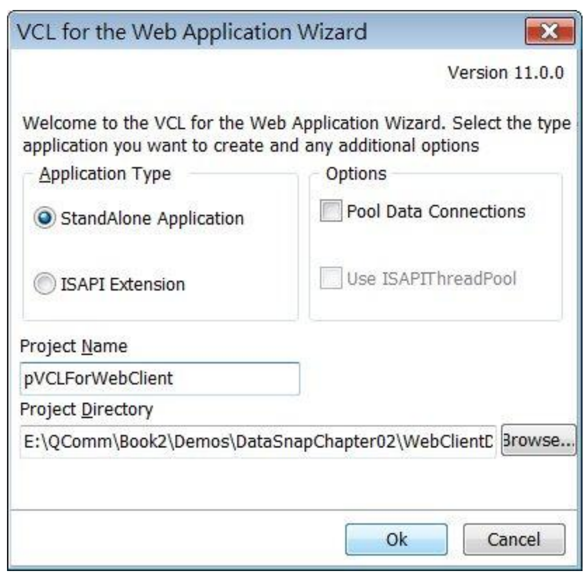



圖 3-2 建立 StandAlone 型態的應用程式

開啟 VCL For Web 專案中的 UserSession 模組，並且在其中放入 TSQLConnection 元件連結前一章的範例 DataSnap/REST 伺服器，再放入 TDSProviderConnection 和 TClientDataSet 元件連結範例 DataSnap/REST 伺服器中的 TDataSetProvider 元件 dspArticles，如同前一章討論如何開發 DataSnap 用戶端程式一樣，此時 VCL For Web 專案中的 UserSession 看起來如下所示:

版權所有 請勿翻印

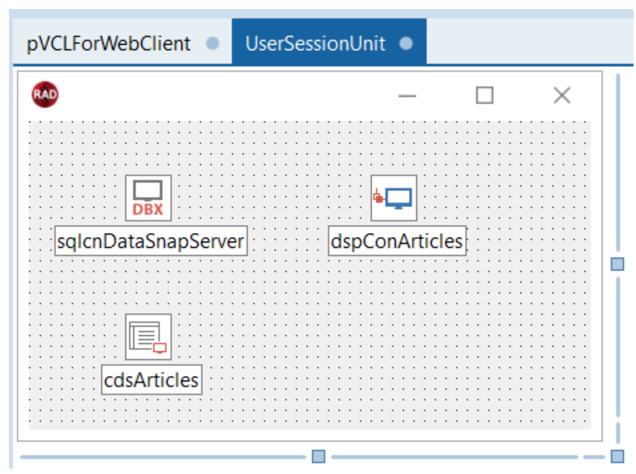
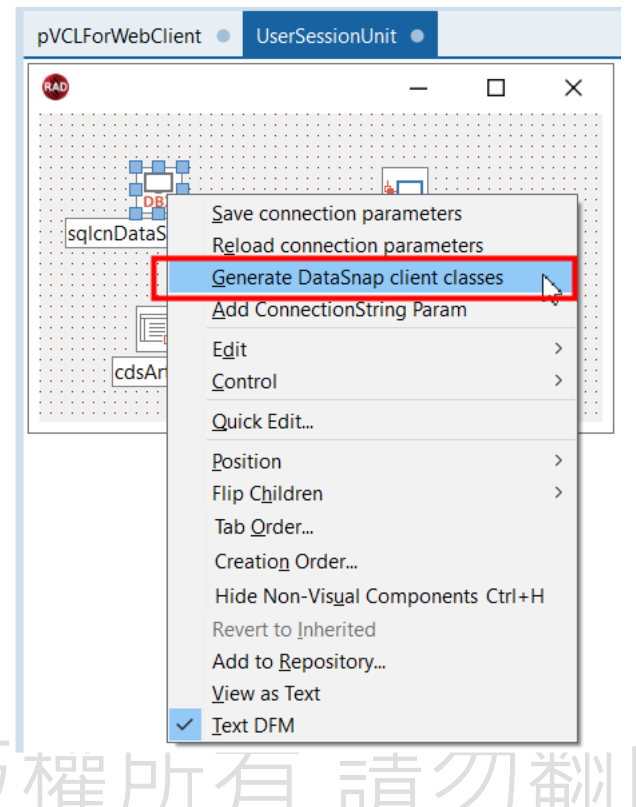



圖 3-3 在 UserSession 程式單元模組中加入 dbExpress 元件連結 DataSnap/REST 伺服器

現在我們就可以使用 UserSession 中的 TSQLConnection 元件自動產生 DataSnap 用戶端類別以便呼叫範例 DataSnap/REST 伺服器輸出的服務。請

使用滑鼠右擊 sqlcnDataSnapServer 元件，從快顯功能表中選擇『Generate DataSnap client classes』，如下圖所示:




版權所有 請勿翻印

圖 3-4 點選 TSQLConnection 元件，選擇產生 DataSnap 用戶端類別

然後儲存自動產生的 DataSnap 用戶端類別為 ServerProxy.pas，並且重新命名主表單為 WebMainForm.pas，此時 VCL For Web 專案應該如下圖所示:

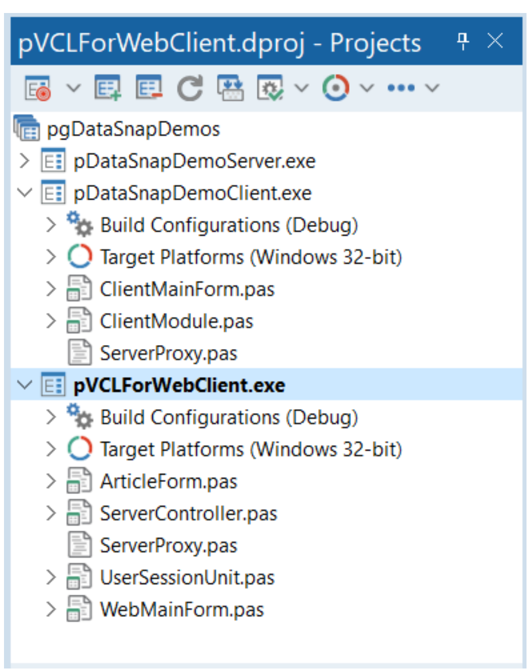


圖 3-5 儲存 DataSnap 用戶端類別之後 VCL For Web 專案的內容

開啟專案中的 WebMainForm，放入 TIWDBNavigator，TIWDBGrid，TIWMemo，TIWButton 和 TDataSource 元件如下:

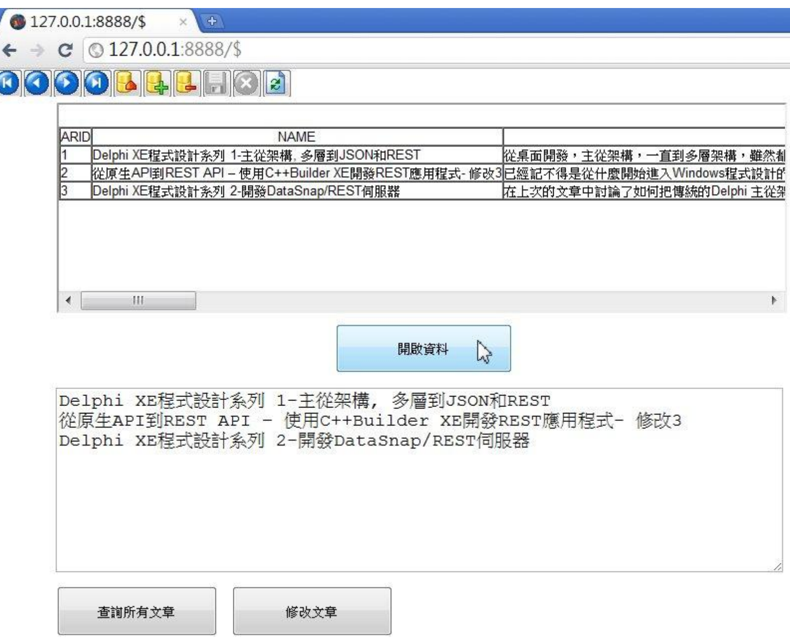



圖 3-6 VCL For Web 主表單連結 DataSnap/REST 伺服器

版權所有 請勿翻印

接著在『開啟資料』按鈕的事件處理函式中開啟 UserSession 中的 TClientDataSet:

```
procedure TiwMainForm.iwbtnOpenTableClick(Sender: TObject);
begin
  UserSession.cdsArticles.Active := True;
end;
```

VCL For Web 應用程式便輕鬆的從範例 DataSnap/REST 伺服器中取得資料顯示在 Web 主表單中，『查詢所有文章』按鈕則呼叫了 DataSnap/REST 伺服器的 QueryAllArticles 方法取得所有文章的 TJSONArray，再一一的從其中取出文章名稱並且顯示在主表單的 TIWMemo 元件中:

```
procedure TiwMainForm.iwbtnQueryAllArticlesClick(Sender: TObject);
var
  aServer : TServerMethods5Client;
  ja : TJSONArray;
  js : TJSONString;
  iIndex: Integer;
begin
```

```
  aServer :=
TServerMethods5Client.Create(UserSession.sqlcnDataSnapServer.DBXConnection);
  try
    ja := aServer.QueryAllArticles;
    for iIndex := 0 to ja.Size - 1 do
    begin
      js := ja.Get(iIndex) as TJSONString;
      IWMemo1.Lines.Add(js.Value);
    end;
  finally
    aServer.Free;
  end;
end;
```

接著在 VCL For Web 專案中再建立另外一個 VCL For Web 新的表單，並且設計其表單如下:

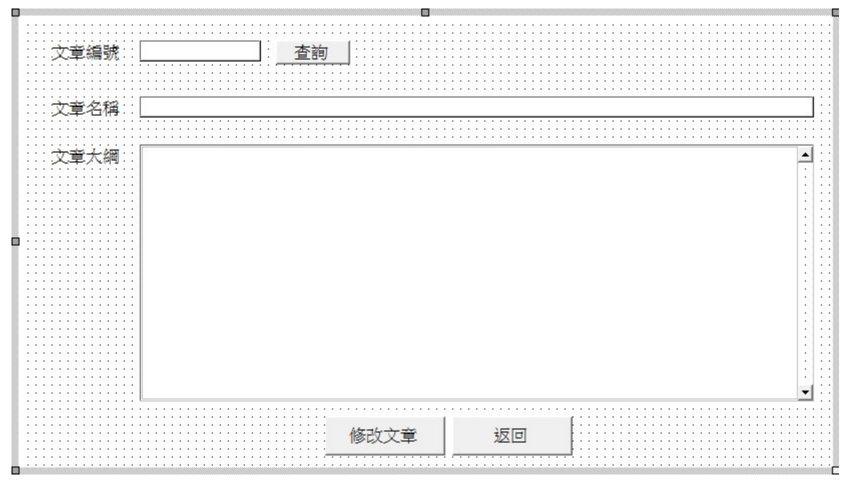



圖 3-7 呼叫 DataSnap/REST 伺服器查詢和修改資料的表單

要查詢文章非常的簡單，直接使用 dbExpress 的 Lookup 方法即可:

```
procedure TiwfmArticle.iwbtnQueryArticleClick(Sender: TObject);
begin
  if (UserSession.cdsArticles.Active = False) then
    UserSession.cdsArticles.Active := True;
  iwedtArticleName.Text := UserSession.cdsArticles.Lookup('ARID',
StrToInt(iwedtArticleID.Text), 'Name');
  iwmmHeadline.Lines.Text := UserSession.cdsArticles.Lookup('ARID',
StrToInt(iwedtArticleID.Text), 'HEADLINE');
end;
```

要修改文章也很簡單，除了也可以直接使用 dbExpress 之外，當然也可以使用 JSON 格式直接呼叫 DataSnap/REST 伺服器提供的服務即可。例如如果用戶端是 JavaScript，Ruby，PHP 或是其他不使用 dbExpress 的用戶端，那麼就可以藉由呼叫 DataSnap/REST 伺服器提供的服務取得 JSON 封裝的資料再加以處理即可。

例如在圖 3-6 中的『修改文章』按鈕使用了下面的程式碼讓使用者在 Web 表單中對於文章資料的異動更新回 DataSnap/REST 伺服器中:

```
procedure TiwfmArticle.iwbtnUpdateArticleClick(Sender: TObject);
var
  aServer : TServerMethods5Client;
  jaArticle : TJSONArray;
begin
  aServer :=
TServerMethods5Client.Create(UserSession.sqlcnDataSnapServer.DBXConnection);
  try
    jaArticle := CreateUpdateArticleObject;
    aServer.ModifyArticle(jaArticle);
    UserSession.cdsArticles.Refresh;
  finally
    aServer.Free;
  end;
end;
```

『修改文章』按鈕的 OnClick 事件處理函式先建立 TServerMethods5Client 物件，再呼叫 CreateUpdateArticleObject 方法建立一個以 JSON 格式封裝的異動過的文章資料，最後再藉由 TServerMethods5Client 物件呼叫 DataSnap/REST 伺服器輸出的 ModifyArticle 方法把異動過的資料更新回伺服器/資料庫之中。

而 CreateUpdateArticleObject 方法在 032 行建立 TJSONArray 物件，接著呼叫 CreateArticleIDObject，CreateArticleNameObject 和 CreateArticleHeadlineObject 方法分別建立封裝文章 ID，名稱和標題的 TJSONObject 物件並且加入到 TJSONArray 物件中，最後再回傳 TJSONArray 物件給前面的 iwbtnUpdateArticleClick 事件處理函式並且呼叫伺服器的 ModifyArticle 方法。

```
001    function TiwfmArticle.CreateUpdateArticleObject: TJSONArray;
002    var
```

下圖就是在 VCL For Web 應用程式的 Web 表單中先查詢文章資料，再於 Web 表單中修改文章資料，最後點選『修改文章』按鈕把文章資料更新回伺服器中。

```
003    jpArticle : TJSONPair;
004
005    function CreateArticleIDObject : TJSONObject;
006    begin
007      jpArticle := TJSONPair.Create;
008      Result := TJSONObject.Create;
009      jpArticle.JsonString := TJSONString.Create('ARID');
010      jpArticle.JsonValue :=  TJSONString.Create(iwedtArticleID.Text);
011      Result.AddPair(jpArticle);
012    end;
013
014    function CreateArticleNameObject : TJSONObject;
015    begin
016      jpArticle := TJSONPair.Create;
017      Result := TJSONObject.Create;
018      jpArticle.JsonString := TJSONString.Create('NAME');
019      jpArticle.JsonValue :=  TJSONString.Create(iwedtArticleName.Text);
020      Result.AddPair(jpArticle);
021    end;
022
023    function CreateArticleHeadlineObject : TJSONObject;
024    begin
025      jpArticle := TJSONPair.Create;
026      Result := TJSONObject.Create;
027      jpArticle.JsonString := TJSONString.Create('HEADLINE');
028      jpArticle.JsonValue :=  TJSONString.Create(iwmmHeadline.Lines.Text);
029      Result.AddPair(jpArticle);
030    end;
031  begin
032    Result := TJSONArray.Create;
033    Result.AddElement(CreateArticleIDObject);
034    Result.AddElement(CreateArticleNameObject);
035    Result.AddElement(CreateArticleHeadlineObject);
036  end;
```

下圖就是在 VCL For Web 應用程式的 Web 表單中先查詢文章資料，再於 Web 表單中修改文章資料，最後點選『修改文章』按鈕把文章資料更新回伺服器中。

圖 3-8 VCL For Web 應用程式呼叫 DataSnap/REST 伺服器修改資料

在使用 VCL For Web 開發 Web 應用程式時，開發人員仍然可以使用 DataSnap 技術以及 JSON 封裝的資料，和開發一般的 Wi32 DataSnap 用戶端視窗應用程式是一樣的。

## 3-2 認證和授權

版權所有 請勿翻印

在開發 DataSnap/REST 伺服器時很重要的一個設計要素就是認證和授權，所謂認證是指可連結 DataSnap/REST 伺服器的使用者，而授權則是指每一個登錄和連結的使用者可呼叫的服務。例如我們在上一章中設計的 DataSnap/REST 伺服器提供了許多的方法可讓用戶端呼叫，使用。但是如果我們希望只有系統合法的使用者才能夠連結使用，而且有一些服務方法是只有特定的使用者才可以呼叫使用，那麼我們應該如何完成這個控制存取的功能?

例如假設我們把上一章伺服器中的服務方法重新設計如下:

```
TServerMethods5 = class(TDSServerModule)
 …
 public
   {訪客可呼叫的方法}
   function QueryAllArticles : TJSONArray;
   function QueryHeadline(Name : string) : string;

   {Admin, Gordon 可呼叫的方法}
   function GetArticleID : Integer;
   function ModifyArticle(jaArticle : TJSONArray) : boolean;
   function AddArticle(Name : string; Headline : string) : boolean;
```

```
  {只有 Gordon 可呼叫的方法}
  function GetArticle(Name : String) : TDataSet;
end;
```

在 TServerMethods5 類別中我們把服務方法分成不同的群組，而且有的群組的方法只能讓特定的使用者呼叫，例如 AddArticle 方法只有 Admin 群組的使用者和 Gordon 這個特定的使用者可以呼叫，而 GetArticle 方法則只限 Gordon 可以呼叫。那麼我們如何能夠實作這些認證和授權的功能呢?

其實我們只要能夠瞭解 DataSnap 如何執行下面的工作就可以立刻掌握如何使用 DataSnap 的認證和授權的功能:

- 如何傳遞使用者登錄資訊
- 如何驗證使用者
- 如何授權使用者可呼叫的方法

在下面的小節將詳細的說明如何在 DataSnap 中完成上面的工作。

## 傳遞使用者登錄資訊

版權所有 請勿翻印

對於 DataSnap/REST 伺服器進行安全控管的第一步當然就是認證系統合法的使用者，當用戶端連結 DataSnap/REST 伺服器欲進行服務呼叫的一開始，DataSnap/REST 伺服器必須先認證此用戶端是否是合法的使用者。在 DataSnap 中用戶端可以使用兩種不同的技術讓 DataSnap/REST 伺服器對於連結的用戶端進行認證的工作。第一種是使用 dbExpress 技術，當然目前這僅限於 Delphi 和 C++Builder 的用戶端，第 2 種方法是使用 JavaScript，那麼任何支援 JavaScript 的用戶端都可以使用這種方式讓 DataSnap/REST 伺服器認證。

讓我們先說明如何使用 Delphi 來傳遞認證資訊給 DataSnap/REST 伺服器。

### 使用 Delphi 用戶端傳遞認證資訊

在說明如何使用 Delphi 用戶端傳遞認證資訊之前，我們需要先解釋 DataSnap 伺服器如何進行認證的工作。當開發人員建立 DataSnap/REST 伺服器時，如果勾選了使用安全機制，那麼 DataSnap 精靈會在 ServerContainer 模組中加入 TDSAuthenticationManager，DataSnap/REST 伺服器中認證和授權的工作就是由 TDSAuthenticationManager 負責。

TDSAuthenticationManager 元件提供了兩個事件處理函式進行認證和授權，它的 OnUserAuthorize 會對連結的用戶端進行認證的工作，它的 OnUserAuthenticate 事件處理函式則會在用戶端呼叫服務方法時檢查是否授權呼叫，下面的表格說明了這兩個事件處理函式的功能:

| 伺服器型態 | 說明 |
| --- | --- |
| OnUserAuthenticate | 用戶端連結時DataSnap呼叫此事件處理函式認證是否為合法的系統使用者 |
| OnUserAuthorize | 用戶端呼叫伺服器的服務時，DataSnap呼叫此事件處理函式檢查此用戶端是否能夠呼叫此特定的服務方法 |

因此要對用戶端進行認證是否為合法的系統使用者，第一個方法就是在 TDSAuthenticationManager 元件的 OnUserAuthenticate 事件處理函式中進行認證。OnUserAuthenticate 的定義原型如下:

```
TDSAuthenticationEvent = procedure(Sender: TObject; const Protocol: UnicodeString; const
Context: UnicodeString; const User: UnicodeString; const Password: UnicodeString; var valid:
boolean; UserRoles: TStrings) of object;
```

版權所有 請勿翻印

TDSAuthenticationEvent 接受數個參數，這些參數由 DataSnap 框架傳遞進入，下面的表格說明其中 User，Password，valid 和 UserRoles 參數的意義:

| 參數名稱 | 說明 |
| --- | --- |
| User | 用戶端使用者的登錄ID |
| Password | 用戶端使用者的登錄密碼 |
| valid | 如果是合法的使用者就需要設定valid為True，否則就設定valid為False |
| UserRoles | 開發人員可根據連結的用戶端指定使用者到不同的角色角色，例如一般的使用者可指定為'users'角色群組，系統管理員可指定為'admin' 角色群組。稍後在授權使用者可呼叫的服務方法時可以根據角色群組來授權 |

而 TDSAuthenticationEvent 的 Protocol 和 Context 參數則會根據連結的用戶端的種類而包含不同的參數值，下面的表格說明了這兩個參數根據不同的用戶端被指定的參數值:

| 參數名稱 | 說明 |
| --- | --- |
| Delphi/C++Builder使用dbExpress連結的用戶端 | Protocol參數值='tcp/ip' |

| | Context = '' |
|---|---|
| Web應用程式用戶端 | Protocol 參數值 ='http'或'https'<br><br>Context = '' |
| RESTful用戶端，例如在瀏覽器中使用如下的REST呼叫語法呼叫DataSnap/REST伺服器時:<br><br>http://localhost:8080/**datasnap**/**rest**/TServerMethods5/EchoString/test | Protocol參數值='datasnap'<br><br>Context = '/rest' |

## 授權使用者

至於當用戶端呼叫伺服器時是否授權用戶端呼叫則由 OnUserAuthorize 事件處理函式決定，它的定義原型如下:

```
TDSAuthorizationEvent = procedure(Sender: TObject; AuthorizeEventObject:
TDSAuthorizeEventObject; var valid: boolean) of object;
```

TDSAuthorizationEvent 接受兩個參數，其中 AuthorizeEventObject 是型態為 TDSAuthorizeEventObject 的物件，而 TDSAuthorizeEventObject 又是從 TDSServerMethodUserEventObject 繼承下來的，TDSServerMethodUserEventObject 類別提供了四個重要的特性，這些特性在下面的表格中說明:

版權所有 請勿翻印

| 特性 | 說明 |
|---|---|
| UserName | 用戶端使用者的登錄名稱 |
| UserRoles | 此用戶端使用者屬於的角色群組，也就是在前面TDSAuthenticationEvent事件處理函式中設定的角色群組 |
| AuthorizedRoles | 其中包含了可呼叫此服務方法的角色群組 |
| DeniedRoles | 其中包含了不可呼叫此服務方法的角色群組 |

由於 TDSServerMethodUserEventObject 又是從 TDSServerMethodEventObject 類別繼承下來的，而 TDSServerMethodEventObject 類別又提供了三個重要的特性，其說明如下:

| 特性 | 說明 |
|---|---|
| ServerClass | 被呼叫的服務方法所屬的類別 |
| MethodAlias | 被呼叫的服務方法的名稱，這是以字串型態代表的名稱，其格式為'類別名稱.方法名稱' |
| MethodInstance | 被呼叫的服務方法的樣例(指標) |

由上面的說明我們可以瞭解，開發人員可以從 AuthorizeEventObject 參數中取得是那一個使用者，那一個角色群組要呼叫什麼類別的什麼方法。因此開發人員就可以在事件處理函式中使用程式碼來決定是否授權使用者呼叫服務方法。

而 TDSAuthorizationEvent 的第二個參數 valid 決定了用戶端是否能夠呼叫服務方法，如果設定為 True 的話，那麼用戶端就被允許呼叫服務方法，設定為 False 的話用戶端就會被拒絕呼叫。

瞭解了這 DataSnap 框架如何控制認證和授權後，就可以開始說明用戶端要如何傳遞認證資訊給 DataSnap/REST 伺服器。

➢ 使用 TSQLConnection 元件

最簡單的方法就是使用 TSQLConnection 元件來設定用戶端連結的使用者名稱和密碼來讓 DataSnap/REST 伺服器進行認證的工作，在用戶端的 TSQLConnection 的 Driver 特性中的 DSAuthUser 和 DSAuthPassword 兩個子特性就可以讓開發人員設定使用者名稱和密碼，例如下圖顯示了這兩個子特性:

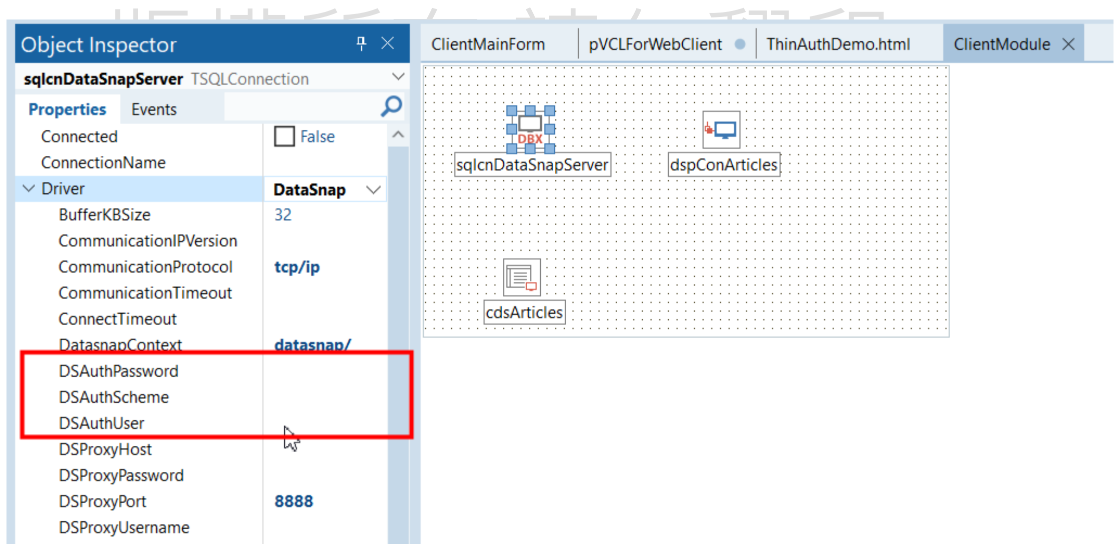



圖 3-9 使用 TSQLConnection 元件設定登錄伺服器的使用者名稱和密碼

開發人員可以在物件檢視器中設定這兩個認證資訊。

➢ 使用 Delphi 程式碼

當然使用 TSQLConnection 元件設定認證資訊固然方便，但產生的問題是沒有彈性，在實際的應用中程式應該讓使用者在執行用戶端應用程式時輸入登錄資訊，再據此連結使用 DataSnap/REST 伺服器。要使用程式碼傳遞用戶端認

證資訊到 DataSnap/REST 伺服器，開發人員必須使用 TDBXDatasnapProperties 物件。

TSQLConnection 元件的 ConnectionData 特性是 TConnectionData 物件，而 TConnectionData 中的 Properties 特性是 TDBXProperties 物件，當 TSQLConnection 是連結 DataSnap/REST 伺服器時，TSQLConnection. ConnectionData. Properties 則是 TDBXDatasnapProperties 物件。TDBXDatasnapProperties 類別中就包含了 DSAuthUser 和 DSAuthPassword 這兩個可傳遞到 DataSnap/REST 伺服器認證資訊的特性。

因此在程式碼中，開發人員可以使用下面的程式碼把使用者登錄的資訊傳遞到 DataSnap/REST 伺服器進行認證:

```
001   procedure TForm17.Button4Click(Sender: TObject);
002   var
003     prop : TDBXDatasnapProperties;
004     aServer : TServerMethods5Client;
005     cm : TClientModule1;
006   begin
007     cm := TClientModule1.Create(nil);
008     try
009
cm.sqlcnDataSnapServer.ConnectionData.Properties.Values[TDBXPropertyNames.DSAuthentica
tionUser] := edtUserName.Text;
010
cm.sqlcnDataSnapServer.ConnectionData.Properties.Values[TDBXPropertyNames.DSAuthentica
tionPassword] := edtPassword.Text;
011       aServer := cm.ServerMethods5Client;
012       aServer.AddArticle(edtAddedArticleName.Text, mmHeadline.Text);
013     finally
014       cm.Free;
015     end;
016   end;
```

在上面的程式碼中，007 行先建立 DataSnap 用戶端模組物件，並且在 011 行使用用戶端模組物件取得代表遠端 DataSnap/REST 伺服器物件 aServer 之前，先在 009 行設定使用者名稱以及在 010 行設定使用者密碼，才於 011 行取得 aServer，接著才在 012 行呼叫 DataSnap/REST 伺服器的 AddArticle 服務方法加入文章資訊。

為什麼要在 011 行之前先藉由 TDBXDatasnapProperties 設定使用者名稱和密碼? 這是因為 011 行呼叫的 GetServerMethods5Client 方法中會開啟用戶端模組中的 TSQLConnection，下面的程式碼中 005 行顯示了這個動作，因此在呼叫 GetServerMethods5Client 之前必須先設定好使用者名稱和密碼。

```
001   function TClientModule1.GetServerMethods5Client: TServerMethods5Client;
002   begin
003     if FServerMethods5Client = nil then
004     begin
005       sqlcnDataSnapServer.Open;
006       FServerMethods5Client:=
TServerMethods5Client.Create(sqlcnDataSnapServer.DBXConnection, FInstanceOwner);
007     end;
008     Result := FServerMethods5Client;
009   end;
```

## 認證和授權用戶端範例

現在讓我們看看一個使用程式碼來進行用戶端認證的範例，首先讓我們開啟上一章的範例 DataSnap/REST 伺服器，開啟它的 ServerContainer 模組，在模組中的 TDSAuthenticationManager 元件的 OnUserAuthenticate 事件處理函式中撰寫如下的程式碼:

版權所有 請勿翻印

```
procedure TServerContainer5.DSAuthenticationManager1UserAuthenticate(
  Sender: TObject; const Protocol, Context, User, Password: string;
  var valid: Boolean; UserRoles: TStrings);
begin
  valid := CheckUser(User, Password, UserRoles);
end;
```

在 DSAuthenticationManager1UserAuthenticate 事件中我們把 valid 指定為呼叫 CheckUser 的執行結果來判斷目前連結的用戶端是否為合法的使用者。而 CheckUser 的實作如下:

```
function TServerContainer5.CheckUser(const User, Password: string;
  UserRoles: TStrings): boolean;
begin
  Result := True;
  if ((User = '') and (Password = '')) then
    UserRoles.Add('guest')
  else
```

```
  if ((User = 'admin') and (Password = 'admin')) then
    UserRoles.Add('Admin')
  else
  if ((User = 'Gordon') and (Password = '123456')) then
    UserRoles.Add('Master')
  else
    Result := False;
end;
```

在 CheckUser 中我們檢查從用戶端傳遞來的使用者認證資訊，並且把不同的使用者指定到不同的角色群組中，例如如果用戶端使用者沒有輸入任何的使用者名稱和密碼，就被指定為 guest 角色群組，如果使用者名稱是 admin，密碼也是 admin 的話，就指定為 Admin 角色群組。

接著讓我們在 TDSAuthenticationManager 元件的 On User Authorize 事件處理函式中撰寫如下的程式碼:

```
procedure TServerContainer5.DSAuthenticationManager1UserAuthorize(
  Sender: TObject; EventObject: TDSAuthorizeEventObject;
  var valid: Boolean);
begin
  valid := CanAuthorize(EventObject);
end;
```

在上面的程式碼中也是把呼叫 CanAuthorize 函式的結果指定給 valid 參數。而 CanAuthorize 實作如下:

```
001   function TServerContainer5.CanAuthorize(
002     EventObject: TDSAuthorizeEventObject): Boolean;
003   begin
004     Result := True;
005
006     if ((EventObject.MethodAlias = 'TServerMethods5.QueryAllArticles') or
(EventObject.MethodAlias = 'TServerMethods5.QueryAllArticles')) then
007     begin
008       if ((EventObject.UserRoles.IndexOf('guest') <> -1) or
009           (EventObject.UserRoles.IndexOf('Admin') <> -1) or
010           (EventObject.UserRoles.IndexOf('Master') <> -1)) then
011         Result := True
012       else
013         Result := False;
```

```
014       exit;
015     end;
016
017     if ((EventObject.MethodAlias = 'TServerMethods5.ModifyArticle') or
(EventObject.MethodAlias = 'TServerMethods5.AddArticle')
018          or (EventObject.MethodAlias = 'TServerMethods5.GetArticleID')) then
019     begin
020       if ((EventObject.UserRoles.IndexOf('Admin') <> -1) or
021           (EventObject.UserRoles.IndexOf('Master') <> -1)) then
022         Result := True
023       else
024         Result := False;
025       exit;
026     end;
027   end;
```

CanAuthorize 函式先於 004 行設定回傳結果為 True 以便讓用戶端能夠呼叫所有伺服器輸出的服務方法(在上一章討論開發伺服器的內文中我們在 DataExplorer 中展示了 DataSnap 伺服器輸出了許多其他的系統方法)，接著在 006 行我們藉由 EventObject 物件的 MethodAlias 特性值，以

版權所有 請勿翻印

```
'類別名稱.方法名稱'
```

的格式來檢查目前用戶端呼叫的服務方法是什麼？如果是 TServerMethods5 類別輸出的方法，那麼就根據 EventObject 物件中 UserRoles 特性來判斷目前呼叫服務方法的使用者群組是那一個，再根據使用者群組來決定是否允許呼叫。例如如果是呼叫查詢函式，那麼 guest，Admin 和 Master 角色群組都可以呼叫，但如果是呼叫 ModifyArticle'等方法，那麼只有 Admin 和 Master 角色群組都可以呼叫。

當然，前面的程式碼只是為了展示如何使用 EventObject 的 UserRoles 特性，由於 004 行已經設定回傳結果為 True，因此我們可以把上面的程式碼改成如下的實作:

```
001   function TServerContainer5.CanAuthorize(
002     EventObject: TDSAuthorizeEventObject): Boolean;
003   begin
004     Result := True;
005
006     if ((EventObject.MethodAlias = 'TServerMethods5.QueryAllArticles') or
(EventObject.MethodAlias = 'TServerMethods5.QueryAllArticles')) then
```

```
007     begin
008       if (not ((EventObject.UserRoles.IndexOf('guest') <> -1) or
009             (EventObject.UserRoles.IndexOf('Admin') <> -1) or
010             (EventObject.UserRoles.IndexOf('Master') <> -1)) ) then
011         Result := False;
012       exit;
013     end;
…
```

完成了伺服器認證和授權的實作之後，現在請編譯並且執行範例 DataSnap/REST 伺服器。

接著建立一個用戶端應用程式，並且使用下面的程式碼呼叫 DataSnap/REST 伺服器中的 GetArticleID 服務方法:

```
procedure TForm17.Button6Click(Sender: TObject);
var
  aServer : TServerMethods5Client;
  cm : TClientModule1;
begin
  cm := TClientModule1.Create(nil);
  try

cm.sqlcnDataSnapServer.ConnectionData.Properties.Values[TDBXPropertyNames.DSAuthentica
tionUser] := edtUserName.Text;

cm.sqlcnDataSnapServer.ConnectionData.Properties.Values[TDBXPropertyNames.DSAuthentica
tionPassword] := edtPassword.Text;
    aServer := cm.ServerMethods5Client;
    Edit5.Text := IntToStr(aServer.GetArticleID);
  finally
    cm.Free;
  end;
end;
```

由於 GetArticleID 服務方法在 DataSnap/REST 伺服器中已經被定義為只有 Admin 群組和 Gordon 可以呼叫，因此如下圖所示，當我們呼叫 GetArticleID 而沒有輸入使用者名稱和密碼時，用戶端會被指定為 guest 角色群組，而 guest 角色群組是無法呼叫 GetArticleID 的，因此用戶端會得到一個呼叫例外錯誤:

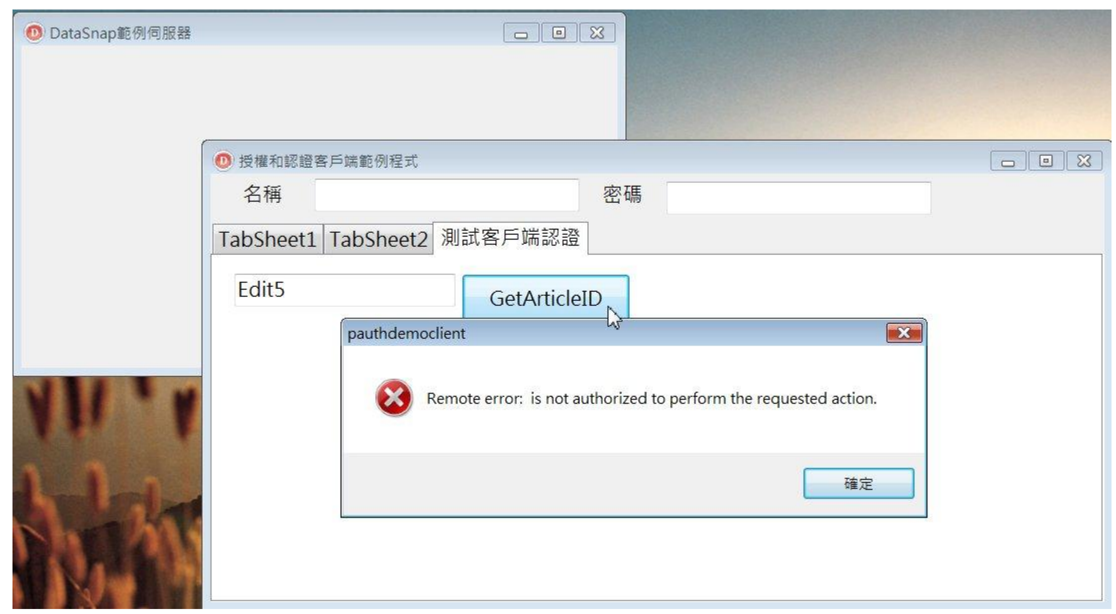



圖 3-10 呼叫 GetArticleID 失敗，因為沒有輸入使用者認證資訊

但是當用戶端使用 admin 或是 Gordon 這兩個使用者登錄時，如下面兩個圖形所示，就可以成功呼叫 GetArticleID，因為 admin 和 Gordon 會分別被指定 Admin 和 Master 這兩個角色群組，而這 Admin 和 Master 這兩個角色群組是被授權可呼叫 GetArticleID 服務方法的。

版權所有 請勿翻印

圖 3-11 呼叫 GetArticleID 成功，因為 admin 和 Gordon 這兩個使用者都允許呼叫 GetArticleID

## 使用 JavaScript 用戶端傳遞認證資訊

除了 Window 的用戶端應用程式之外，DataSnap XE 也可以產生 JavaScript 的程式碼來使用 DataSnap 的認證和授權的機制，因此允許任何支援 JavaScript 的開發工具，框架或是程式語言使用 DataSnap 的認證和授權的機制。DataSnap XE 提供了數個內建的 JavaScript 檔案允許用戶端使用 JavaScript 來連結和使用 DataSnap XE。這些 JavaScript 檔案不但允許用戶

端使用 JavaScript 來連結和使用 DataSnap 或是 DataSnap REST 伺服器，也可以使用 DataSnap XE 提供的回叫機制和認證和授權的機制等進階的功能，開發人員只需要瞭解如何使用這些內建的 JavaScript 檔案，再加入 DataSnap XE 能夠自動產生 DataSnap 或是 DataSnap REST 伺服器以 JavaScript 封裝的遠端服務方法，如此一來用戶端幾乎就可以使用這些內建的 JavaScript 檔案以及自動產生的 JavaScript 檔案來使用任何的 DataSnap 功能。由於本章是的重點是討論認證和授權的機制，因此本小節討論的重點就是如何藉由 JavaScript 讓非 Delphi/C++Builder 的用戶端使用 DataSnap XE 的論認證和授權的機制。

首先讓我們在 Delphi IDE 中建立一個 HTML 檔案，如下所示:

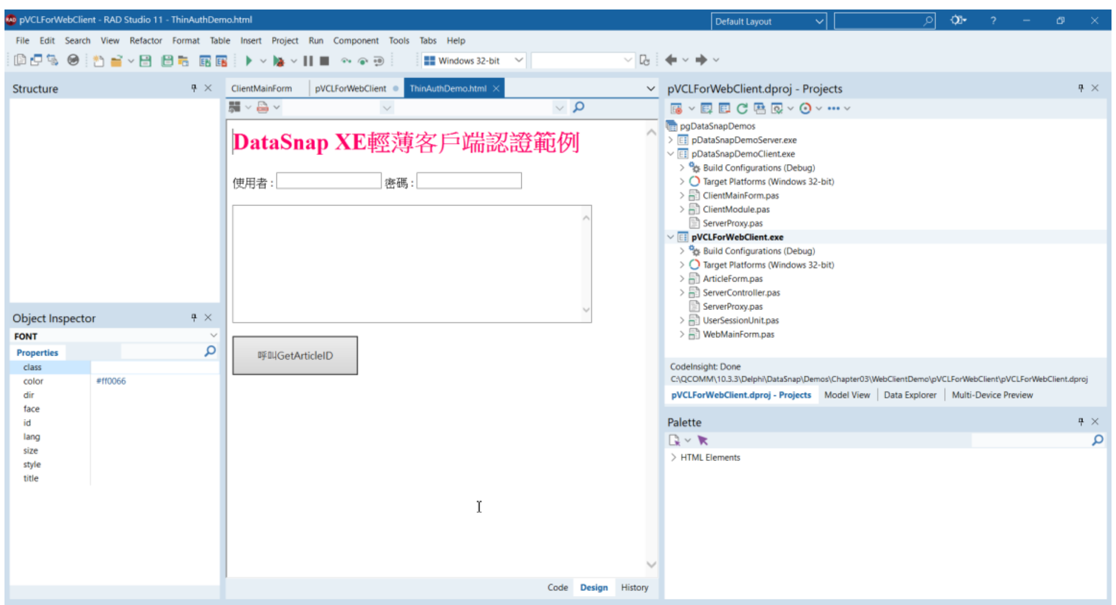



圖 3-12 在 Delphi IDE 中建立一個 HTML 檔案並且設計圖形使用者介面

要讓用戶端使用 JavaScript 使用 DataSnap XE 的論認證和授權的機制，那麼必須在 JavaScript 中執行下面的步驟:

1. 擷取用戶端輸入的使用者名稱和密碼

2. 使用 DataSnap XE 內建的 base64.js 檔案中的 convertStringToBase64 函式以下面的格式把用戶端輸入的使用者名稱和密碼轉換為 Base64 編碼的格式:

```
使用者名稱:密碼
```

3. 使用 DataSnap XE 內建的 connection.js 檔案中的 connectionInfo 格式建立一個 connectionInfo 的變數。connectionInfo 格式如下:

```
{"authentication":步驟 2 產生的結果}
```

4. 使用 JavaScript 程式碼建立 DataSnap/REST 伺服器中的服務類別物件並且把步驟 3 產生的結果傳入做為參數

5. 使用步驟 4 建立的服務類別物件呼叫遠端 DataSnap/REST 伺服器提供的服務

從上面的步驟來看藉由 JavaScript 使用 DataSnap XE 的功能並不困難，因為關鍵的 JavaScript 程式碼都已經由 Delphi/C++Builder 幫開發人員完成了，開發人員只需要瞭解如何使用這些內建和自動產生的 JavaScript 程式碼即可。下面就是一個 JavaScript 檔案藉由 DataSnap XE 的論認證和授權的機制呼叫前面範例 DataSnap 伺服器的實作程式碼。

在 007~012 行中加入使用 Delphi 內建的 JavaScript 程式碼，010 行的 ServerFunctions.js 是由 DataSnap XE 框架自動根據 DataSnap/REST 伺服器產生的封裝伺服器服務的 JavaScript 檔案。

```
001    <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
002    <html>
003      <head>
004        <title>DataSnap XE 輕薄客戶端認證範例
005    </title>
006    <meta HTTP-EQUIV="Content-Type" CONTENT="text/html; CHARSET=UTF-8"><meta
http-equiv="cache-control" content="no-cache">
007        <script type="text/javascript" src="base64.js"></script>
008        <script type="text/javascript" src="json-min.js"></script>
009        <script type="text/javascript" src="serverfunctionexecutor.js"></script>
010        <script type="text/javascript" src="ServerFunctions.js"></script>
011        <script type="text/javascript" src="connection.js"></script>
012        <script type="text/javascript">
013
014        function callServer()
015        {
016          var user = document.getElementById("edtName").value;
017          var password = document.getElementById("edtPassword").value;
018          var auth = convertStringToBase64(user + ":" + password);
019          var connectionInfo = {"authentication":auth};
020
021          var server = new TServerMethods5(connectionInfo);
```

```
022        var result = server.GetArticleID();
023
024        if(result != null)
025        {
026          var resultId = result.result;
027        }
028        else
029        {
030          var resultId = result.toJSONString();
031        }
032
033        document.getElementById("mmCallback").value = resultId;
034      }
035
036      </script>
037    </head>
038    <body>
039    <div>
040       <h1><font color="#ff0066">DataSnap XE 輕薄客戶端認證範例
041   </font></h1>
042       <form onsubmit="callServer(); return false;">
043           <p>使用者 : <input id="edtName"> 密碼 : <input id="edtPassword"></p>
044         <textarea rows="10" cols="60" id="mmCallback"></textarea><br><br />
045         <input id="btncallServer" type="submit" value="呼叫 GetArticleID" style="WIDTH:
217px; HEIGHT: 67px" size="34" />
046       </form>
047    </div>
048    </body>
049   </html>
```

016~017 行使用 JavaScript 擷取使用者在瀏覽器中輸入的使用者名稱和密碼，018 行執行前面討論的步驟 2，019 行執行前面討論的步驟 3，021 行執行前面討論的步驟 4，之後就可以使用 021 行建立的遠端服務物件來呼叫 DataSnap/REST 伺服器中提供的服務了。

現在讓我們試著在瀏覽器中執行這個範例 HTML 檔案，下圖是在瀏覽器中載入和執行此範例 HTML 檔案的畫面:

圖 3-13 在瀏覽器中執行範例 HTML 檔案

接著以 guest 登錄並且試著呼叫遠端 DataSnap/REST 伺服器中的 GetArticleID 方法，由於 guest 群組沒有權限呼叫 GetArticleID，因此瀏覽器無法呼叫 GetArticleID:

版 印

圖 3-14 使用 guest 登錄無法在瀏覽器中呼叫 GetArticleID

接著如果我們以 Gordon 登錄，再呼叫 GetArticleID 方法，由於 Gordon 的群組擁有呼叫 GetArticleID 的授權，因此瀏覽器成功的取得了執行 GetArticleID 方法的結果:

DataSnap XE輕薄客...
localhost:8080/ThinAuthDemo.html?
DataSnap XE輕薄客戶端認證範例
使用者: Gordon 密碼: 123456
5
呼叫GetArticleID

圖 3-15 使用 Gordon 登錄則可在瀏覽器中呼叫 GetArticleID

當開發人員在 Delphi 整合發展環境中建立 DataSnap REST 應用程式專案時，Delphi 會自動在專案中產生相關的 JavaScript 檔案並且儲存於專案的 js 子目錄之下，同時在專案中也可以看到這些 JavaScript 檔案，當開發人員開發了伺服器服務方法並且執行 DataSnap REST 伺服器時，DataSnap XE 也會自動在 js 子目錄中產生封裝伺服器服務方法的 ServerFunctions.js 檔案，之後開發人員就可以使用它來讓非 Delphi/C++Builder 的用戶端藉由這些 JavaScript 檔案來呼叫 DataSnap/REST 伺服器了。

版權所有 請勿翻印

## 使用 TDSAuthenticationManager 的特性值編輯器授權

使用程式碼來進行授權的控制是最具彈性的方法，但 Delphi 仍然提供了其他的方法讓開發人員進行授權的控制，第 2 種方法就是直接在整合發展環境中使用物件檢視器來進行。請開啟範例 DataSnap/REST 伺服器專案中的 ServerContainer 模組，點選 TDSAuthenticationManager 元件，於物件檢視器中 TDSAuthenticationManager 元件定義了 Roles 特性，如下所示:

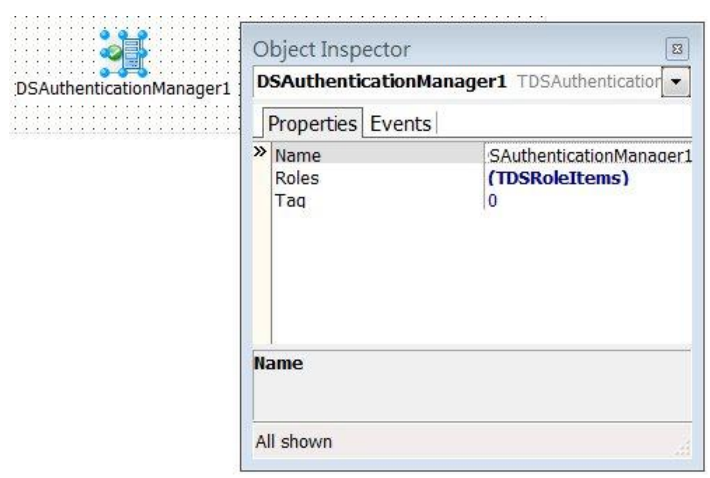


圖 3-16 點選 TDSAuthenticationManager 元件，使用物件檢視器進行授權的控制

請雙擊 Roles 特性，物件檢視器便會啟動 Roles 特性的特性值編輯器，開發人員就可以開始為每一個服務方法定義什麼角色群組可以呼叫，什麼角色群組無法呼叫。雙擊 Roles 特性之後 Roles 特性值編輯器後，請點選左上方的新增按鈕，此時在物件檢視器中就可以定義一個授權控制項目，其中開發人員可以定義三個子項目，它的們的說明如下:

| 特性名稱 | 說明 |
|---|---|
| ApplyTo | 這個授權適用的服務方法 |
| AuthorizedRoles | 可呼叫此服務方法的角色群組 |
| DeniedRoles | 不可呼叫此服務方法的角色群組 |

在定義 ApplyTo 特性值時，開發人員可以使用 3 種格式:

| ApplyTo格式 | 說明 | 範例 |
|---|---|---|
| 類別名稱 | 此授權定義適用此類別所有的服務方法 | 如果是 TServerMethods5 類別，那麼所有此類別的方法都適用此授權定義 |
| 類別名稱.服務方法名稱 | 此授權定義只適用此類別的此服務方法 | 如果是 TServerMethods5.GetArticle，那麼此授權只適用此服務方法 |
| 服務方法名稱 | 任何類別只要擁有相同的服務方法名稱就適用此授權定義 | 如果是 GetArticle，那麼任何類別的 GetArticle 方法都適用此授權定義 |

版權所有 請勿翻印

例如我們在 Roles 特性啟動特性值編輯器，就可以如下圖定義 ApplyTo，AuthorizedRoles 和 DeniedRoles 3 個子特性值:

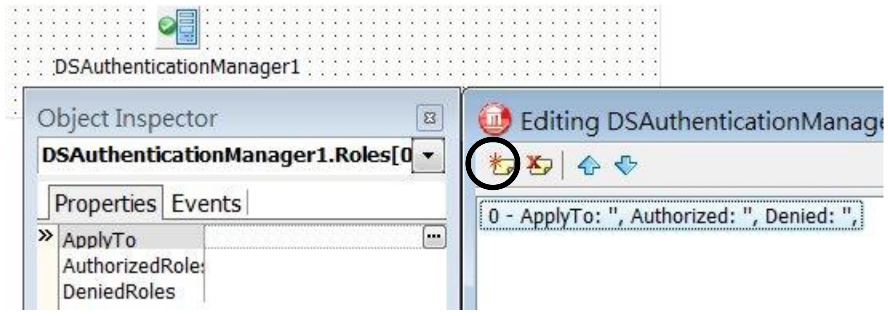



圖 3-17 使用 Roles 特性的特性值編輯器定義授權控制

而下圖顯示了我們定義 GetArticle 服務方法只能由 Master 角色群組呼叫，guest 和 Admin 角色群組都無法呼叫:

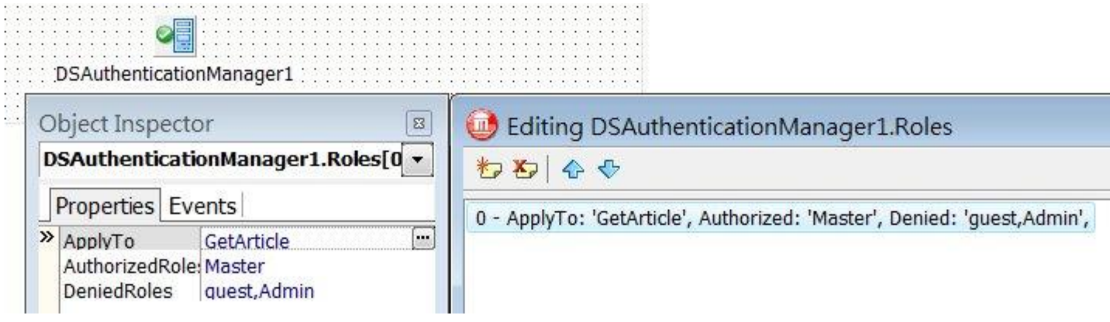



圖 3-18 定義 GetArticle 服務方法只能由 Master 角色群組呼叫，guest 和 Admin 角色群組無法呼叫

下圖則是使用 Roles 特性值編輯器完整的定義和前面使用程式碼進行授權控制相同的效果:

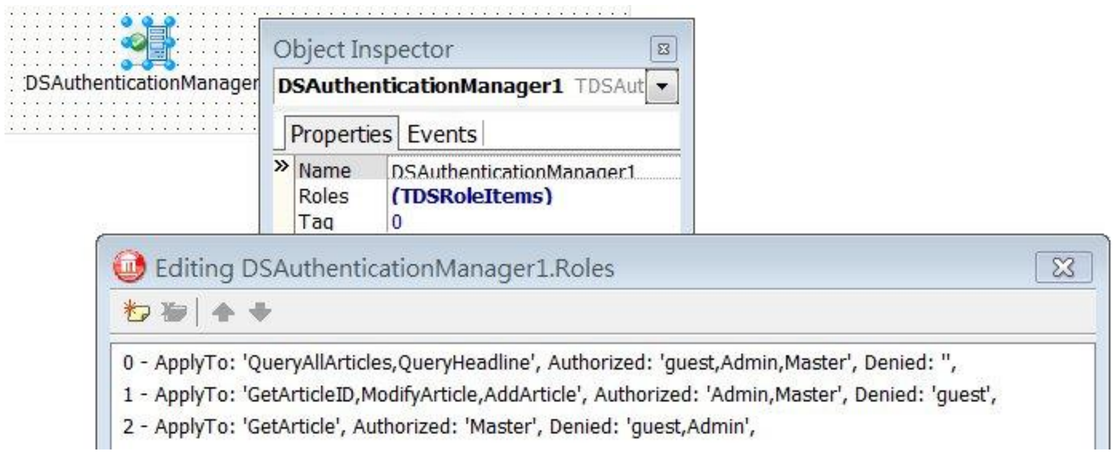



圖 3-19 藉由 TDSAuthenticationManager 的 Roles 特性值編輯器來定義授權的控制

## 使用程式碼註解授權

最後一種定義授權控制的方法是使用程式碼註解，開發人員只需要在類別或是類別的方法程式碼中使用 Delphi 程式語言的註解功能來定義什麼角色群組可以呼叫什麼類別或是什麼類別方法即可，而無需在 OnUserAuthorize 事件處理函式使用程式碼來控制。

DataSnap XE 定義了 TRoleAuth 類別來定義註解授權，它的宣告如下:

```
TRoleAuth = class(TCustomAttribute)
…
public
  …
```

```
   constructor Create(AuthorizedRoles: String; DeniedRoles: String = ''); overload;
virtual;
   constructor Create(AllowRoles: TStrings; DenyRoles: TStrings;
                   DesignTime: Boolean = False); overload; virtual;
…
```

TRoleAuth 有兩個建構函式，都可以接受可呼叫服務方法的角色群組和不可呼叫服務方法的角色群組。因此現在讓我們開啟範例 DataSnap/REST 伺服器專案中的 ServerMethods 程式單元，然後在類別定義中使用下面的程式碼註解來定義每一個服務方法的授權定義。由於 TRoleAuth 的第一個參數是可呼叫服務方法的角色群組，第二個參數是不可呼叫服務方法的角色群組，因此我們可以使用 CSV 的字串格式來定義這兩個角色群組。

```
   {訪客可呼叫的方法}
   [TRoleAuth('guest, Admin, Master', '')]
   function QueryAllArticles : TJSONArray;
   [TRoleAuth('guest, Admin, Master', '')]
   function QueryHeadline(Name : string) : string;

   {Admin 可呼叫的方法}
   [TRoleAuth('Admin, Master', 'guest')]
   function GetArticleID : Integer;
   [TRoleAuth('Admin, Master', 'guest')]
   function ModifyArticle(jaArticle : TJSONArray) : boolean;
   [TRoleAuth('Admin, Master', 'guest')]
   function AddArticle(Name : string; Headline : string) : boolean;

   {只有 Gordon 可呼叫的方法}
   [TRoleAuth('Master', 'guest, Admin')]
   function GetArticle(Name : String) : TDataSet;
```

現在我們就可以重新編譯範例 DataSnap/REST 伺服器並且執行它，再啟動範例用戶端來呼叫它的服務方法，下圖顯示了 Admin 群組果然無法呼叫 GetArticle 方法:

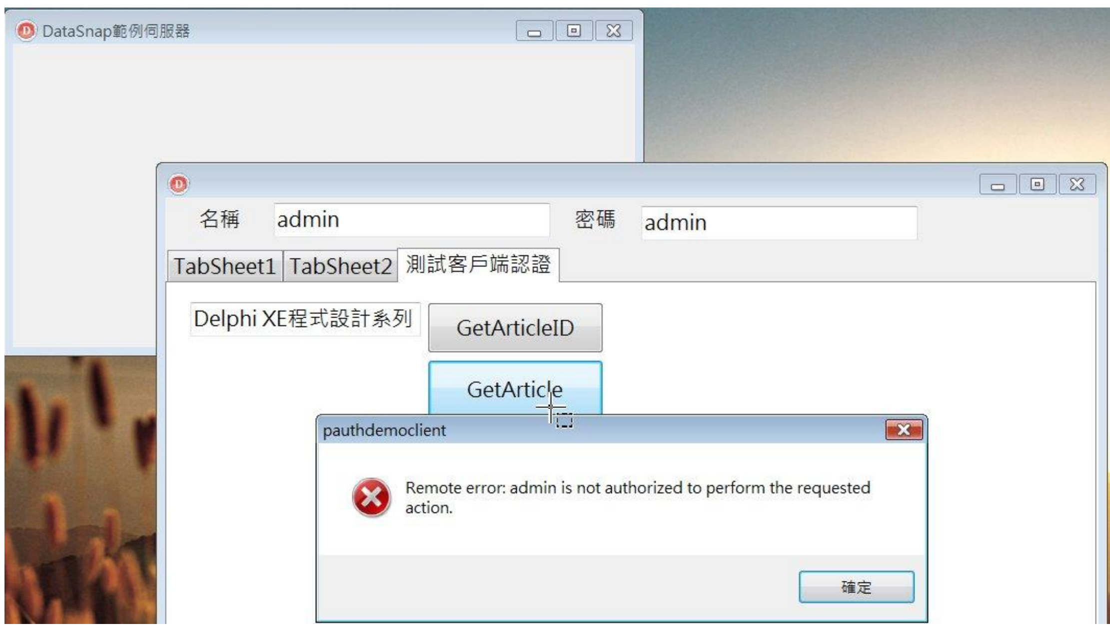



圖 3-20 藉由註解來定義授權的控制，Admin 角色群組不能呼叫 GetArticle 服務方法

而下圖也顯示了 Master 角色群組果然可以藉由程式碼註解方式來呼叫 GetArticle 方法:

圖 3-21 藉由註解來定義授權的控制，只有 Master 角色群組能呼叫 GetArticle 服務方法

您已經掌握了 DataSnap 的認證和授權的機制了，現在該您自己動手試試了。

# 第4章 DataSnap回叫機制

DataSnap 從 2010 版便開始加入回叫(Call Back)機制，當伺服端方法在執行的過程中可以回叫用戶端提供的方法以通知用戶端有關伺服端方法執行的狀態。DataSnap 10.3 之後又大幅強化了回叫機制的功能，加入了回叫通道(Callback Channel)以及通道廣播 (Channel Broadcast)等新功能，本章討論的重點便在說明如何使用 DataSnap 的各種回叫功能。

DataSnap 的回叫機制非常適合使用在需要較長時間執行的伺服端方法，例如如果伺服端方法需要執行長時間的查詢時就很適合使用，或是當程式啟動或是執行時需要進行許多查詢的工作，那麼也都可以使用回叫機制。

版權所有 請勿翻印

使用 DataSnap 最基本的回叫機制非常的簡單，開發人員只需要實作一個從 TDBXCallback 繼承下來的實體類別，並且在呼叫伺服端方法時把此實體類別的樣例當做參數傳遞給伺服端方法即可。至於回叫通道和通道廣播則需要更多的手續，但也提供了更強大的功能。在一開始我們將先說明如何使用 DataSnap 最基本的回叫機制，接著再說明回叫通道和通道廣播等功能。

## 4-1 DataSnap 基本的回叫機制

由於使用 DataSnap 2010 基本的回叫機制並不困難，開發人員只需要遵照下面的步驟即可完成:

- 在用戶端建立一個從 TDBXCallback 繼承下來的實體類別物件並且複載實作虛擬方法 Execute
- 呼叫伺服器方法時傳遞上述的物件給伺服端方法
- 實作伺服端方法時在每一個執行步驟之後呼叫用戶端傳遞來的物件中的虛擬方法 Execute 以代表執行進度

讓我們使用一個範例來說明讀者就可以很快的瞭解如何完成上述的步驟。

在下列的範例中本文將使用 Delphi 中新的 System.IOUtils 程式單元中的類別進行檔案搜尋和計數的工作，由於這將花上一些時間，因此我們正好使用它來展示使用同步和回叫機制的差異。

## 範例 DataSnap 伺服器

首先讓我們建立一個 DataSnap 伺服端，下面是這個伺服器輸出的伺服端方法，請注意的是，TServerMethods1 輸出了兩個方法 GetServerDirectoryInfo 和 GetServerDirectoryInfoAsync。這兩個方法都執行相同的工作，它們使用 TDirectory 類別搜尋和計數特定伺服端目錄下的檔案總數，它們的差異是 GetServerDirectoryInfo 使用同步的方式執行，因此當用戶端呼叫它時，用戶端會同時暫停反應直到 GetServerDirectoryInfo 執行完畢。

而 GetServerDirectoryInfoAsync 則是使用回叫機制的方式執行，因此當用戶端呼叫它之後，GetServerDirectoryInfoAsync 在執行的過程中則可以藉由用戶端傳遞來的參數 **ACallback: TDBXCallback**，來回叫回用戶端，告訴用戶端執行的狀態，用戶端因此也根據目前伺服端執行的情形來更新用戶端的資訊。

```
{$METHODINFO ON}
  TServerMethods1 = class(TPersistent)
  private
    { Private declarations }
    fTotalFiles : Integer;
    FResult : TJSONArray;
    FCallback: TDBXCallback;

    procedure ProcessPath(const sPath : string);
    procedure ProcessPathAsync(const sPath : string);
    procedure ShowMessage(sMessage : string);
    procedure ProcessThisDirectory(const sPath : string);
  public
    { Public declarations }
    function EchoString(Value: string): string;
    function GetServerDirectoryInfo(const sPath : string) : TJSONArray;
    function GetServerDirectoryInfoAsync(ACallback: TDBXCallback; const sPath : string):
TJSONArray;
  end;
{$METHODINFO OFF}
```

GetServerDirectoryInfoAsync 方法是如何回叫回用戶端呢?其實非常的簡單，因為用戶端在呼叫它時已經把用戶端的回叫方法當成參數傳遞過來了，因此 GetServerDirectoryInfoAsync 方法只需要藉由這個參數即可回叫回用戶端。

因此我們可以從下面 18 行的程式碼看到，伺服端直接使用這個回叫參數呼叫用戶端，並且建立一個 TJSONString 型態的物件做為參數，在這個 TJSONString 物件中告訴了用戶端目前伺服端正在處理那一個目錄。

```
001   function TServerMethods1.GetServerDirectoryInfoAsync(ACallback: TDBXCallback;
002     const sPath: string): TJSONArray;
003   begin
004     FCallback := ACallBack;
005     FTotalFiles := 0;
006     FResult := TJSONArray.Create;
007     ProcessPathAsync(sPath);
008     FResult.AddElement(TJSONString.Create('總檔案數' + ' : ' + IntToStr(FTotalFiles)));
009     Result := FResult;
010   end;
011
012   procedure TServerMethods1.ProcessPathAsync(const sPath: string);
013   var
014     rootDirectories : TStringDynArray;
015     i: Integer;
016   begin
017     ProcessThisDirectory(sPath);
018     FCallback.Execute(TJSONString.Create('處理目錄' + sPath + '中...'));
019     rootDirectories := TDirectory.GetDirectories(sPath);
020     for i := 0 to Length(rootDirectories) - 1 do
021       ProcessPathAsync(rootDirectories[i]);
022   end;
```

## 同步用戶端

範例的同步用戶端非常的簡單，它只是直接呼叫伺服端的 GetServerDirectoryInfo 方法。

```
procedure TForm3.btnGetServerInfoClick(Sender: TObject);
var
  aServer : TServerMethods1Client;
```

```
  ja : TJSONArray;
  jv : TJSONValue;
  I: Integer;
begin
  lStart := GetTickCount;
  aServer := TServerMethods1Client.Create(Self.SQLConnection1.DBXConnection);
  try
    ja := aServer.GetServerDirectoryInfo(Edit1.Text);
    lEnd := GetTickCount;
    for I := 0 to ja.Size - 1 do
    begin
      jv := ja.Get(I);
      lbResult.Items.Add(jv.ToString);
    end;
  finally
    aServer.Free;
    ShowRunTime(lEnd - lStart);
  end;
end;
```

在用戶端呼叫 GetServerDirectoryInfo 方法的過程中，用戶端暫停反應，使用者也無從瞭解伺服端執行的狀態。

## 回叫用戶端

再看看回叫用戶端，這個用戶端的關鍵從下面的 012 行開始，012 行建立了 TDSCallbackWithMethod 物件，並且建立一個匿名方法做為用戶端的回叫方法傳遞給伺服端。從 013 行開始的匿名方法在被用戶端回叫的時候首先在 013 行把伺服端傳遞來的參數型態轉換為 TJSONString 的型態，接著更新用戶端的 UI 以通知使用者伺服端目前正在處理那一個目錄。最後在 023 行用戶端回叫方法如果執行成功就需要回傳 TJSonTrue 物件，如果失敗的話就需要回傳 TJSonFalse 物件。

```
001   procedure TForm3.btnGetServerInfoAsyncClick(Sender: TObject);
002   var
003     aServer : TServerMethods1Client;
004     LCallback : TDSCallbackWithMethod;
005     ja : TJSONArray;
006     jv : TJSONValue;
007     I: Integer;
```

```
008   begin
009     lStart := GetTickCount;
010     aServer := TServerMethods1Client.Create(Self.SQLConnection1.DBXConnection);
011     try
012       LCallback := TDSCallbackWithMethod.Create(
013         function(const Args: TJSONValue): TJSONValue
014         var
015           asyncResult: TJSONString;
016           I: Integer;
017           LMessage: string;
018         begin
019           asyncResult := TJSONString(Args);
020           lbAsync.Items.Add(asyncResult.ToString);
021           lbAsync.Update;
022           Application.ProcessMessages;
023           Result := TJSonTrue.Create;
024         end
025         );
026       ja := aServer.GetServerDirectoryInfoAsync(LCallback, Edit1.Text);
027
028
029       lEnd := GetTickCount;
030       for I := 0 to ja.Size - 1 do
031       begin
032         jv := ja.Get(I);
033         lbResult.Items.Add(jv.ToString);
034       end;
035     finally
036       aServer.Free;
037       ShowRunTime(lEnd - lStart);
038     end;
039   end;
```

那麼什麼是 TDSCallbackWithMethod 類別呢? 這要從 TDBXCallback 抽象類別開始談起。

## TDBXCallback 抽象類別

在討論 TDSCallbackMethod 之前我們必須先說明 TDBXCallback 抽象類別，因為 TDSCallbackMethod 是從 TDBXCallback 繼承下來的實體類別。事實上 TDBXCallback 類別即是使用 DataSnap 回叫機制的關鍵，要使用回叫機制，開發人員必須實作一個從 TDBXCallback 繼承下來的實體類別，並且傳遞此實體類別的樣例給伺服端方法做為參數，如此一來伺服端方法就可以藉由這個樣例參數呼叫回用戶端，以通知用戶端伺服端方法執行的狀態。

TDBXCallback 的虛擬方法 Execute 是衍生類別需要複載實作的，Execute 接受一個型態為 TJSONValue 的參數並且回傳一個型態為 TJSONValue 的結果值，伺服端方法在回叫用戶端的方法時，可以把需要傳遞給用戶端的數值或是物件轉換為TJSONValue型態並且當成Execute方法的參數傳遞回用戶端，而用戶端的方法在被回叫執行完畢之後，也可以把執行結果轉換為 TJSONValue 型態並且回傳給伺服端。下面即是 TDBXCallback 抽象類別的宣告:

```
001     TDBXCallback = class abstract
002     public
003       function Execute(const Arg: TJSONValue): TJSONValue; virtual; abstract;
004     protected
005       procedure SetConnectionHandler(const ConnectionHandler: TObject); virtual;
006       procedure SetOrdinal(const Ordinal: Integer); virtual;
007     public
008       property ConnectionHandler: TObject write SetConnectionHandler;
009       property Ordinal: Integer write SetOrdinal;
010     end;
```

## TDSCallbackMethod 實體類別

瞭解了 TDBXCallback 扮演的角色之後解釋 TDSCallbackMethod 實體類別就簡單了，由於此範例應用程式要使用非同步呼叫機制，因此宣告 TDSCallbackMethod 從 TDBXCallback 繼承下來並且實作虛擬方法 Execute。在 TDSCallbackMethod 的 Execute 方法中它只是簡單的呼叫在建構函式中儲存下來的用戶端方法的方法指標。

```
unit AsyncUtils;

interface
```

```
uses
  Classes,
  DbxDatasnap,
  DBXJson;

type
  TDSCallbackMethod = reference to function(const Args: TJSONValue): TJSONValue;
  TDSCallbackWithMethod = class(TDBXCallback)
  private
    FCallbackMethod: TDSCallbackMethod;
  public
    constructor Create(ACallbackMethod: TDSCallbackMethod);
    function Execute(const Args: TJSONValue): TJSONValue; override;
  end;

implementation

constructor TDSCallbackWithMethod.Create(ACallbackMethod: TDSCallbackMethod);
begin
  FCallbackMethod := ACallbackMethod;
end;

function TDSCallbackWithMethod.Execute(const Args: TJSONValue): TJSONValue;
var
  aString: string;
begin
  Assert(Assigned(FCallbackMethod));
  Result := FCallbackMethod(Args);
end;

end.
```

下面的圖形分別是 DataSnap 伺服端執行的畫面以及同步用戶端，回叫用戶端的執行畫面，從圖 2 同步用戶端畫面可以看到它雖然比回叫用戶端執行的快(在筆者的機器中執行了 7.078 秒)，但是在同步用戶端呼叫 DataSnap 伺服器時它的整個 UI 是暫停的，因此它無法在表單下方的 ListBox 中顯示任何伺服端執行的資訊，使用者必須等待它完全執行完畢才能取回用戶端應用程式的控制權。

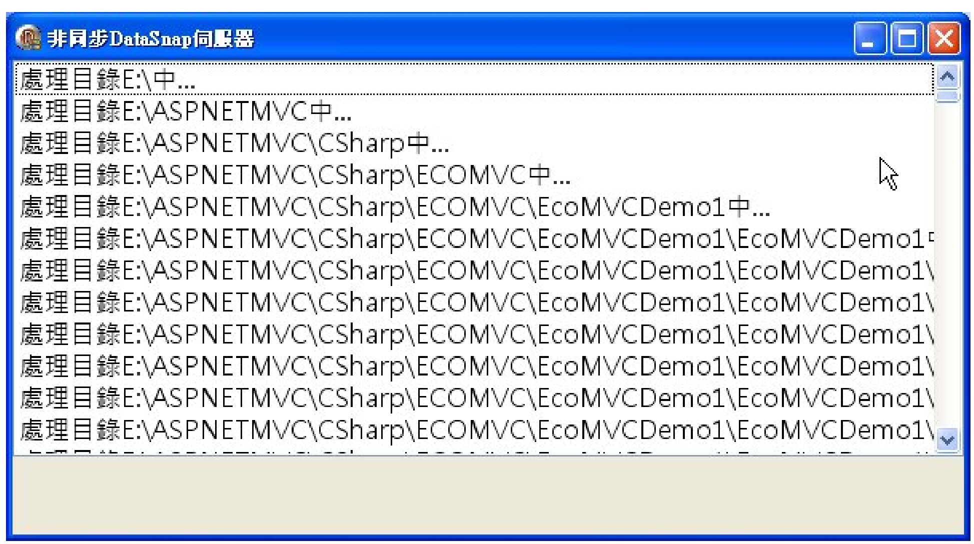



圖 1 DataSnap 伺服器執行畫面

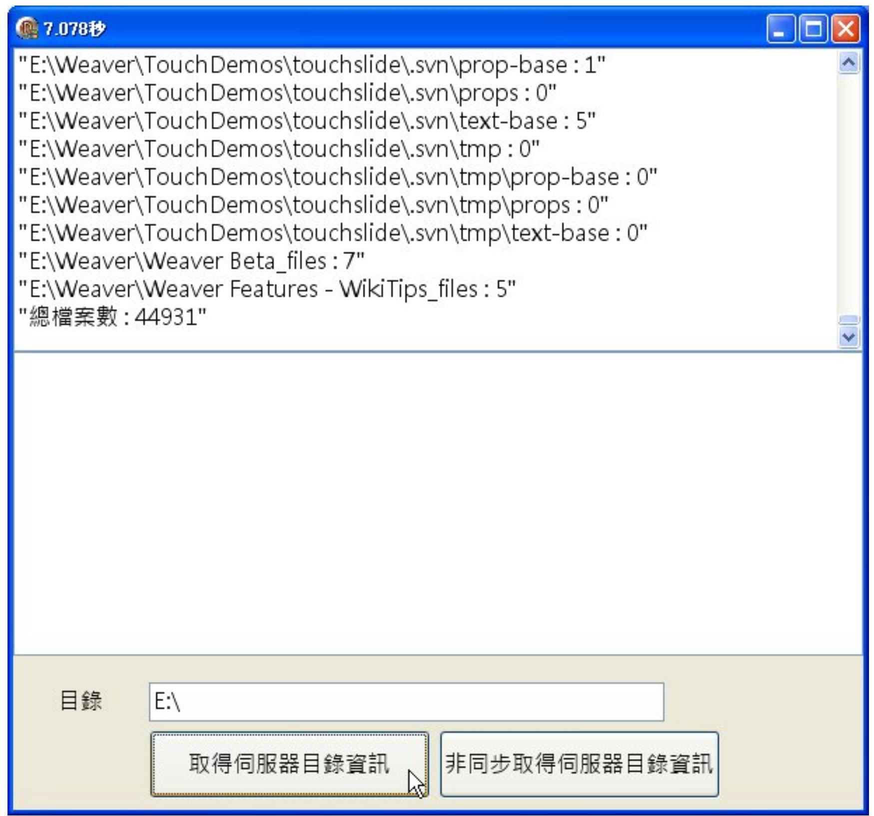



圖 2 同步客戶端呼叫 DataSnap 伺服器執行結果

相反的畫面 3 則是回叫用戶端的執行結果，我們看到它是比同步用戶端慢(在筆者的機器中執行了 12.984 秒)，但是在整個執行過程中使用者仍然可以控制用戶端應用程式，而且回叫用戶端能夠不停的在表單下方的 ListBox 中顯示目前伺服端正在處理的目錄資訊。因此就使用者經驗來說，回叫用戶端是比同步用戶端好多了。

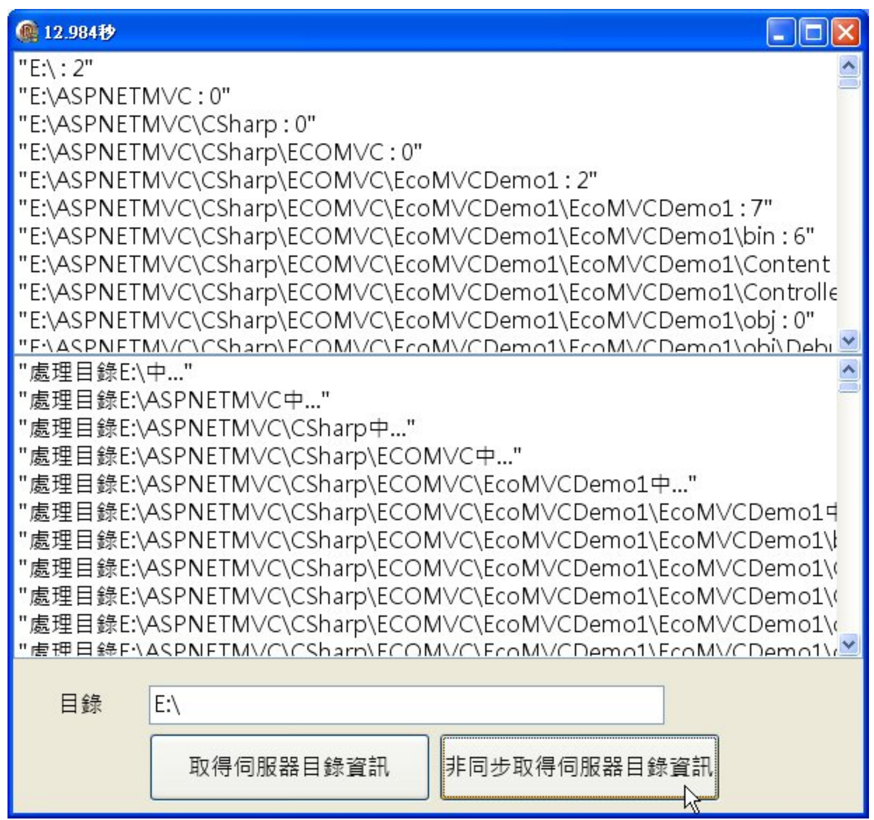



圖 3 DataSnap 伺服器藉由回叫功能呼叫非同步用戶端執行畫面

版權所有 請勿翻印

## 4-2 進階 DataSnap 回叫功能

前一小節討論了 DataSnap 基本的回叫功能，接著讓我們討論從 DataSnap 10.3 版加入的回叫通道等新的回叫機制。

DataSnap 10.3 在原有的基礎回叫機制之上加入了許多強大的新功能，從 DataSnap 10.3 開始開發人員可以使用下面的回叫功能:

- 用戶端可向伺服端註冊回叫通道，如此一來伺服器可以一次回叫所有在同回叫通道中所有註冊的用戶端回叫函式
- 用戶端可以同時註冊多個不同的回叫通道
- 用戶端可以藉由回叫通道呼叫不同的用戶端
- 新增回叫元件以幫助開發人員簡化開發回叫機制

DataSnap 10.3 新的回叫功能雖然很多，但使用起來仍然相當的容易，下面說明了如何使用這些 DataSnap 10.3 新的回叫功能的基本步驟:

➢ 用戶端使用 TDSClientCallbackChannelManager 向伺服器註冊一個回叫通道

➢ 伺服器使用 TDSServer 元件的 BroadcastMessage 方法回叫所有註冊的用戶端

當然，開發人員可以更進一步的使用 DataSnap 10.3 進階的回叫功能，不過在那之前也許我們應該先說明數個範例讓讀者瞭解如何使用這些基本的步驟。

## 開發回叫 DataSnap 伺服器

在 Delphi 整合發展環境中建立一個 DataSnap Server 專案:

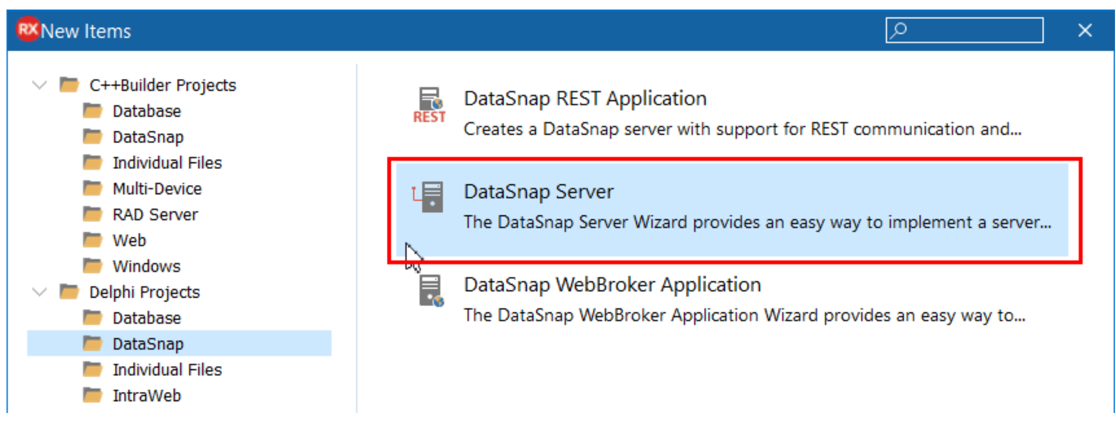



圖 4 建立 DataSnap 回叫伺服器

在設定此伺服器的特性時，讓我們目前只選擇使用 TCP/IP 通訊協定，如下圖所示:

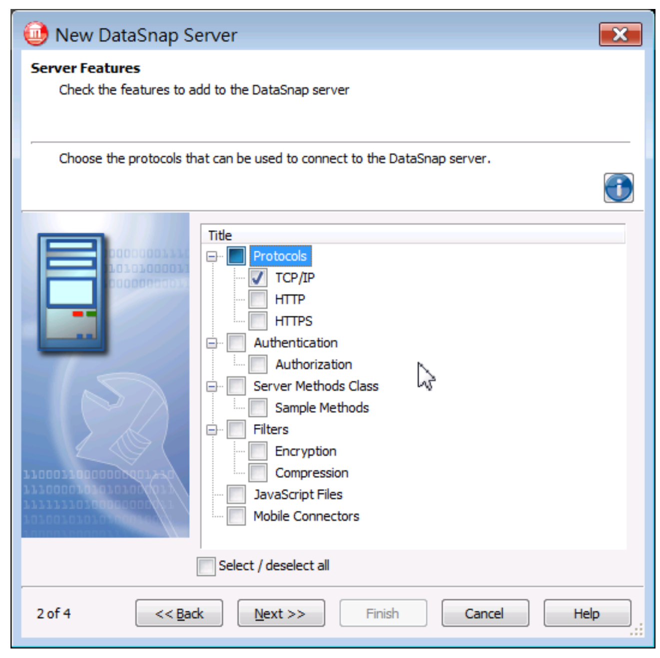



圖 5 建立 DataSnap 回叫伺服器，目前只選擇使用 TCP/IP 通訊協定

開啟專案中的 ServerContainer 程式單元，此時在 ServerContainer 中產生了兩個元件，TDSServer 以及 TDSTCPServerTransport，由於接下來我們將先展示 Windows 用戶端的回叫功能，因此現在使用 TCP/IP 通訊協定就足夠了。

版權所有 請勿翻印

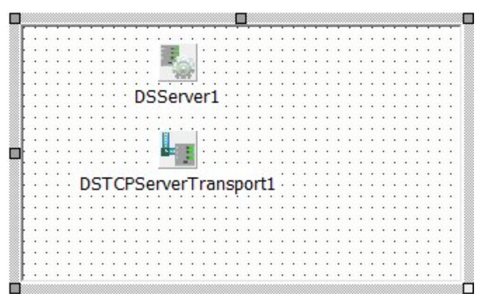



圖 6 ServerContainer 包含的元件

現在開啟專案主表單，並且在其中置入如下的元件:

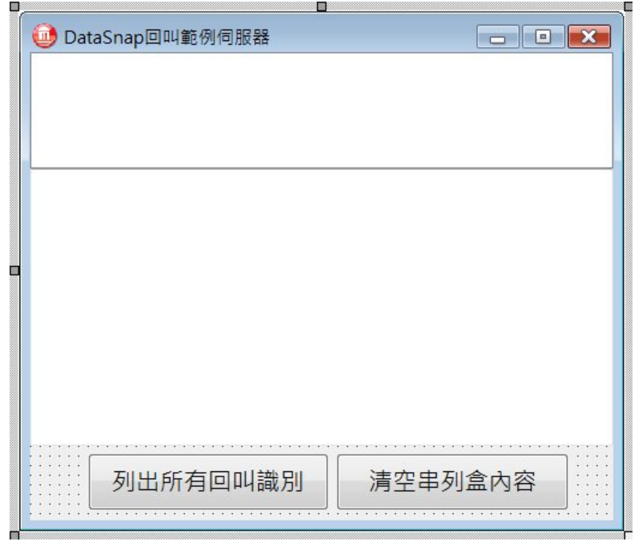



圖 7 DataSnap 回叫伺服器主表單

主表格上方使用了 TListBox 元件，它可以顯示所有用戶端註冊的回叫識別 ID，而下方的 TMemo 元件則是使用來回叫註冊用戶端，在稍後我們將在此 TMemo 元件中輸入資訊，這些輸入的資訊就會藉由回叫通道自動傳遞給用戶端。

版權所有 請勿翻印

DataSnap 伺服器要回叫所有註冊的用戶端是非常容易的，只需要藉由 TDSServer 類別定義的 BroadcastMessage 方法即可，TDSServer 類別中定義了兩個 BroadcastMessage 方法原型如下:

```
function BroadcastMessage(const ChannelName: String; const Msg: TJSONValue; const ArgType:
Integer = TDBXCallback.ArgJson): boolean; overload;

function BroadcastMessage(const ChannelName: String; const CallbackId: String; const Msg:
TJSONValue; const ArgType: Integer = TDBXCallback.ArgJson): boolean; overload;
```

這兩個 BroadcastMessage 方法的差異在於上述第一個 BroadcastMessage 可以傳遞訊息給它第一個參數 ChannelName 指定的通道中所有的回叫用戶端，而第二個 BroadcastMessage 方法則是只傳遞訊息給它第一個參數 ChannelName 指定的通道中由第二個參數 CallbackId 指定的回叫用戶端，最後這個兩個 BroadcastMessage 方法傳遞給用戶端的訊息則由第二個參數 Msg 封裝。

瞭解了如何使用 BroadcastMessage 方法之後，我們就可以看看如何把 DataSnap 伺服器中於 TMemo 元件中輸入的訊息傳遞給回叫用戶端。現在為主表單中的 TMemo 元件實作 OnChange 事件處理函式如下:

```
001   procedure TForm17.mmMessageChange(Sender: TObject);
```

```
002    var
003      vMessage : TJSONString;
004    begin
005      vMessage := TJSONString.Create(mmMessage.Lines.Text);
006      ServerContainer5.DSServer1.BroadcastMessage(DEMOChannel, vMessage);
007    end;
```

在 005 行我們把輸入於 TMemo(mmMessage)中的資訊以 TJSONString 物件封裝，然後在 006 行藉由呼叫 ServerContainer 中的 TDSServer 元件的 BroadcastMessage 方法傳遞給所有註冊的用戶端。但誰是註冊的用戶端呢?請看 BroadcastMessage 的第一個參數 DEMOChannel，這代表 DataSnap 伺服器會傳遞資訊給所有在 DEMOChannel 通道中註冊的用戶端。而 DEMOChannel 是一個通道的名稱，我們在 DataSnap 伺服器中定義它如下:

```
const
  DEMOChannel = 'DemoChannel';
```

因此用戶端只要使用這個名稱向伺服器註冊回叫通道的話，就可以讓 DataSnap 伺服器回叫用戶端，當然也用戶端可以先向伺服器查詢已經定義在 DataSnap 伺服器中的回叫通道名稱，或是由用戶端自行在 DataSnap 伺服器中建立指定名稱的回叫通道。

版權所有 請勿翻印

由於回叫用戶端是向 DataSnap 伺服器中指定名稱的回叫通道註冊，而且每一個用戶端都使用一個特定的回叫識別 ID 來代表，因此我們也可以藉由 TDSServer 元件來查詢某一個名稱的回叫通道中所有註冊的用戶端回叫識別 ID。

此範例 DataSnap 伺服器主表單中的『列出所有回叫識別』按鈕的 OnClick 事件處理函式就可以在主表單上方的 TListBox 中列出特定回叫通道中所有的用戶端回叫識別 ID，下面就是它的 OnClick 實作程式碼:

```
001    procedure TForm17.Button1Click(Sender: TObject);
002    var
003      aIdList : TList<String>;
004      sId : String;
005    begin
006      aIdList := ServerContainer5.DSServer1.GetAllChannelCallbackId(DEMOChannel);
007      try
008        for sId in aIdList do
009          ListBox1.Items.Add(sId);
010      finally
```

```
011        aIdList.Free;
012      end
013    end;
```

GetAllChannelCallbackId 方法會回傳 TList<String>型態的執行結果，其中即包含了所有在此通道名稱中註冊的用戶端回叫識別 ID，因此在 006 行藉由 TDSServer 元件呼叫 GetAllChannelCallbackId 之後，就可以取得到在 DEMOChannel 中註冊的識別 ID，接著從 008 行到 012 行就把這些識別 ID 顯示在主表單的 TListBox 中。

現在請編譯並且執行此範例 DataSnap 伺服器，接著我們就可以實作回叫用戶端了，首先讓我們先說明如何建立 Windows 應用程式型態的用戶端，接著再說明如何建立瀏覽器型態的用戶端。

## 開發回叫 DataSnap 用戶端

在專案管理員中再建立一個 VCL Form 應用程式專案，並且在其中放入 TSQLConnection，TDSClientCallbackChannelManager，TMemo，兩個 TButton 元件，如下圖所示。其中 TSQLConnection 是連結上一小節的範例 DataSnap 伺服器，而 TDSClientCallbackChannelManager 則是使用來向 DataSnap 伺服器註冊回叫用戶端的元件。

版權所有 請勿翻印

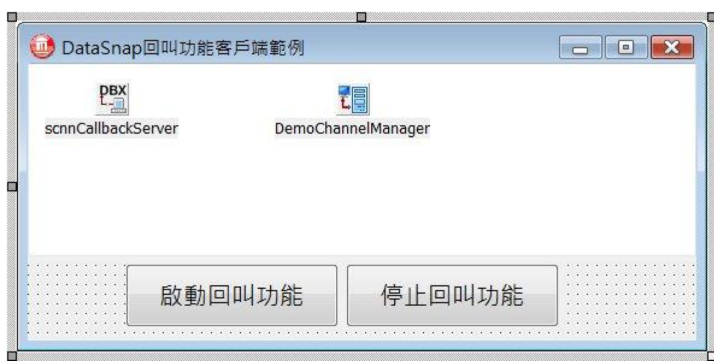



圖 8 DataSnap 回叫伺服器主表單使用的元件

接著設定 TDSClientCallbackChannelManager 特性值如下:

| 特性名稱 | 特性值 |
|---|---|
| ChannelName | DemoChannel |
| CommunicationProtocol | tcp/ip |
| DSHostname | localhost |
| DSPort | 211 |
| Name | DemoChannelManager |

設定 TDSClientCallbackChannelManager 的 ChannelName 特性值為 DemoChannel 是因為前面範例 DataSnap 伺服器使用的通道名稱就是 DemoChannel，而且前面範例 DataSnap 伺服器是支援 TCP/IP 通訊協定和使用 211 通信埠，因此我們需要設定 TDSClientCallbackChannelManager 相對應的特性值，下圖顯示了在物件檢視器中設定 TDSClientCallbackChannelManager 元件的特性值:

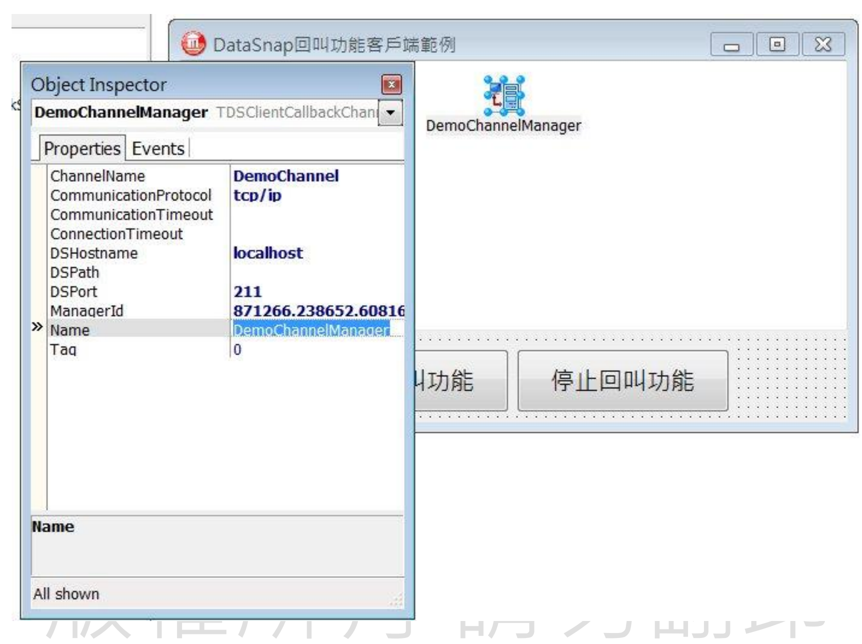



圖 9 設定 TDSClientCallbackChannelManager 元件的特性值

設定好 TDSClientCallbackChannelManager 元件之後，我們就可以看看用戶端主表單中的『啟動回叫功能』按鈕的實作程式碼了:

```
001   procedure TfmMainForm.btnStartClick(Sender: TObject);
002   begin
003     SetupTask;
004     EnableDisableButtons(False, True);
005     DemoChannelManager.RegisterCallback(callbackId, TDemoCallback.Create)
006   end;
007
008   procedure TfmMainForm.SetupTask;
009   begin
010     if not scnnCallbackServer.Connected then
011     begin
012       scnnCallbackServer.Connected := True;
013     end;
014     callbackId := DateTimeToStr(Now);
```

```
015     Self.Caption := callbackId;
016   end;
```

在 btnStartClick 事件處理函式中先於 003 行呼叫 SetupTask 方法以開啟 TSQLConnection 元件連結範例 DataSnap 伺服器，並且在 014 行根據目前的時間建立一個獨特的識別 ID，callbackId。最後在 005 行呼叫 TDSClientCallbackChannelManager 元件的 RegisterCallback 方法向範例 DataSnap 伺服器註冊這個回叫用戶端。

TDSClientCallbackChannelManager 元件的 RegisterCallback 方法原型定義如下:

```
function RegisterCallback(const CallbackId: String; const Callback: TDBXCallback): boolean;
overload;
```

RegisterCallback 接受兩個參數，第一個是每一個回叫用戶端的識別 ID，第二個參數則是型態為 TDBXCallback 的物件，在前面的小節中我們已經解釋過 TDBXCallback 類別，因此在這裡我們需要建立一個從 TDBXCallback 衍生類別的物件，在這個衍生類別中我們需要複載虛擬方法 Execute，如此一來範例 DataSnap 伺服器就可以藉由呼叫複載虛的擬方法 Execute 來呼叫到用戶端。

版權所有 請勿翻印

由於我們在上面的 005 行是傳遞 TDemoCallback 物件給 RegisterCallback 方法，因此我們需要在這個範例用戶端應用程式中定義和實作 TDemoCallback 類別，它需要從 TDBXCallback 繼承下來:

```
type
  TDemoCallback = class(TDBXCallback)
  public
    constructor Create;
    function Execute(const Arg: TJSONValue): TJSONValue; override;
  end;
```

TDemoCallback 需要實作虛擬方法 Execute，當 DataSnap 伺服器回叫用戶端時就會執行虛擬方法 Execute。由於在這個範例中我們希望範例 DataSnap 伺服器在 TMemo 中輸入的資訊能夠立刻顯示在用戶端，因此我們實作虛擬方法 Execute 如下:

```
001   function TDemoCallback.Execute(const Arg: TJSONValue): TJSONValue;
002   var
003     sDemoMessage : String;
004   begin
005     Result := TJSONTrue.Create;
```

```
006
007     if (Arg is TJSONString) then
008     begin
009       sDemoMessage := TJSONString(Arg).Value;
010       TThread.Synchronize(nil,
011                          procedure
012                          begin
013                            fmMainForm.mmDemoMessage.Lines.Text := sDemoMessage;
014                          end
015                          );
016     end;
017   end;
```

當 DataSnap 伺服器藉由回叫功能呼叫用戶端複載的 Execute 時，DataSnap 伺服器可以把伺服端傳遞到用戶端的資訊封裝在 Execute 的參數 Arg 中。由於在前面的範例 DataSnap 伺服器是把在伺服端 TMemo 中的資訊封裝成 TJSONString 傳遞到用戶端，因此在 007 行先判斷伺服端傳遞來的是否是 TJSONString 型態，如果是的話，就 00 行取出伺服端傳遞來的資訊，接著由於我們需要把這個資訊顯示在用戶端主表單中的 TMemo 元件中，因此我們藉由呼叫 TThread 類別的類別方法 Synchronize 來更新用戶端的使用者介面(這是因為用戶端使用者介面的主執行緒和在背景的回叫執行緒是不同的，因此需要使用 Synchronize 來讓背景回叫執行緒更新主執行緒中的元件)。由於 Synchronize 定義了如下的原型:

版權所有 請勿翻印

```
class procedure TThread.Synchronize(AThread: TThread; AMethod: TThreadMethod);
```

因此在上面的 010 行中我們直接使用匿名方法來更新主表單的 TMemo 元件。

最後當我們不再需要讓用戶端被回叫時，可以呼叫 TDSClientCallbackChannelManager 元件的 UnregisterCallback 方法並且傳遞用戶端識別 ID，例如下面的程式碼就是主表單中『停止回叫功能』按鈕 OnClick 事件處理函式的實作程式碼:

```
procedure TfmMainForm.btnStopClick(Sender: TObject);
begin
  EnableDisableButtons(True ,False);
  DemoChannelManager.UnregisterCallback(callbackId);
end;
```

現在編譯並且執行此範例回叫用戶端，從下圖我們可以看到，點選範例 DataSnap 伺服器中的『列出所有回叫識別』按鈕就可以在上方的 TListBox 中

列出用戶端傳遞來的識別 ID，而且只要用戶端應用程式點選了主表單中的『啟動回叫功能』按鈕註冊回叫用戶端，那麼我們在範例 DataSnap 伺服器的 TMemo 中輸入的任何資訊就會立刻顯示在用戶端應用程式的 TMemo 元件中，證明了回叫功能是成功的。

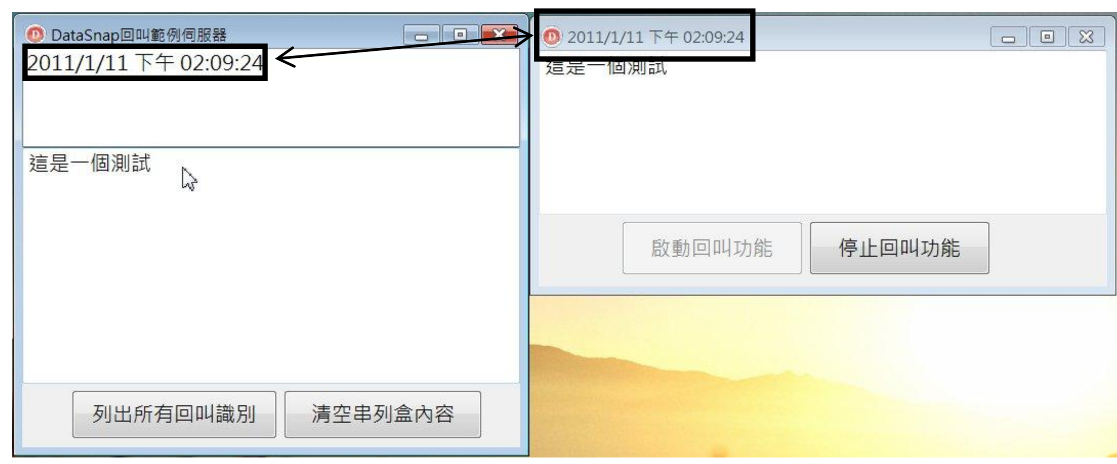



圖 10 在範例 DataSnap 伺服器中輸入的資訊可以立刻藉由回叫功能顯示在用戶端

當然，用戶端可以註冊多個不同的回叫通道，在同一個回叫通道中也可以註冊多個識別 ID，例如讓我們修改剛才的範例用戶端應用程式，加入另外一個 TMemo 元件，以及一個 TEdit 元件讓使用者可以輸入不同的用戶端識別 ID 來註冊:

版權所有 請勿翻印

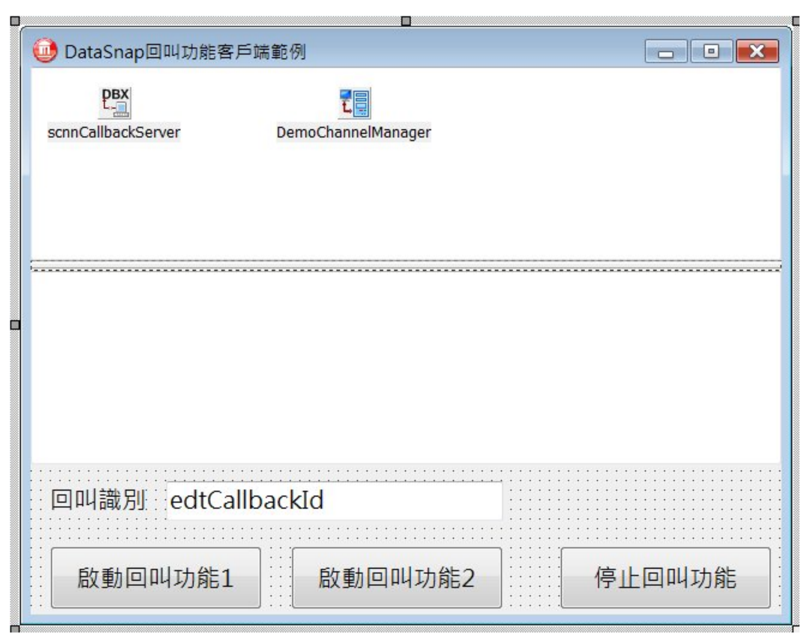



圖 11 修改用戶端應用程式的主表單

接著在『啟動回叫功能 1』和『啟動回叫功能 2』按鈕中都使用下面的程式碼實作，現在用戶端識別 ID 就由使用者輸入而不是使用目前的日期:

```
procedure TfmMainForm.btnStartClick(Sender: TObject);
begin
  SetupTask;
  EnableDisableButtons(False, True, True);
  DemoChannelManager.RegisterCallback(edtCallbackId.Text, TDemoCallback.Create);
  aIdList.Add(edtCallbackId.Text);
end;
```

編譯並且執行新的用戶端應用程式，在下面的圖形中我們可以看到筆者在用戶端應用程式中註冊了兩個用戶端識別 ID，分別是『用戶端識別 ID1』和『用戶端識別 ID2』，而範例 DataSnap 伺服器也可以列出這兩個不同的用戶端識別 ID，在用戶端註冊了兩個不同的識別 ID 之後，範例 DataSnap 伺服器可以藉由同時回叫這兩個不同的用戶端回叫物件。例如下圖就顯示了當我們在範例 DataSnap 伺服器的 TMemo 中輸入了訊息之後，這些訊息會立刻顯示在用戶端應用程式中兩個不同的 TMemo 元件之中:

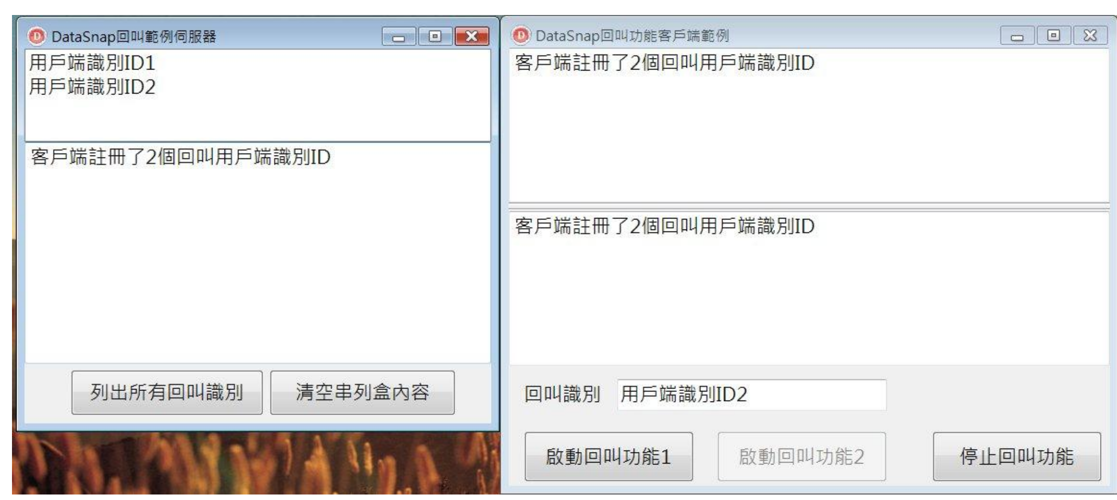



圖 12 用戶端在同一個回叫通道中註冊了兩個識別 ID

如何? DataSnap 10.3 的回叫機制是不是更強大了?不過故事還沒結束，接下來我們將繼續討論如何讓不同的用戶端都能夠藉由回叫機制來溝通，以及如何開發以瀏覽器為基礎的回叫用戶端，這些是更精彩的主題。

## 不同用戶端藉由回叫功能溝通

雖然一個用戶端應用程式可以藉由註冊一個回叫通道並且在回叫通道中註冊多個回叫 ID，但另外一個非常實用的回叫功能就是如何讓不同的用戶端能夠藉由回叫來相互溝通。DataSnap XE 的回叫功能也支援不同的用戶端藉由回叫機制來相互傳遞任何的資料，在說明如何實作之前，讓我們總結一下 DataSnap 框架回叫機制可由用戶端註冊變動的部份包含了:

➢ 通道名稱

➢ 用戶端識別

➢ 回叫識別

因此要讓不同的用戶端能夠藉由回叫功能互動，那麼用戶端只要掌握上述的三個變動的即可。例如用戶端 1 要和用戶端 2 藉由回叫互動，用戶端 1 只需要知道用戶端 2 存在的通道名稱，用戶端 2 的用戶端識別以及用戶端 2 的回叫識別，那麼用戶端 1 就可以回叫用戶端 2 來傳遞任何資訊了。

現在就讓我們看看如何實作不同用戶端藉由回叫功能相互溝通。

## 修改 DataSnap 伺服器

回到範例回叫專案群組，開啟範例 DataSnap 伺服器的主表單，在主表單上方加入一個 **TListBox** 以顯示所有用戶端識別資訊，另外再加入兩個 **TButton**，分別是『列出所有客戶端識別』以及『清除所有客戶端識別』，如下圖所示:

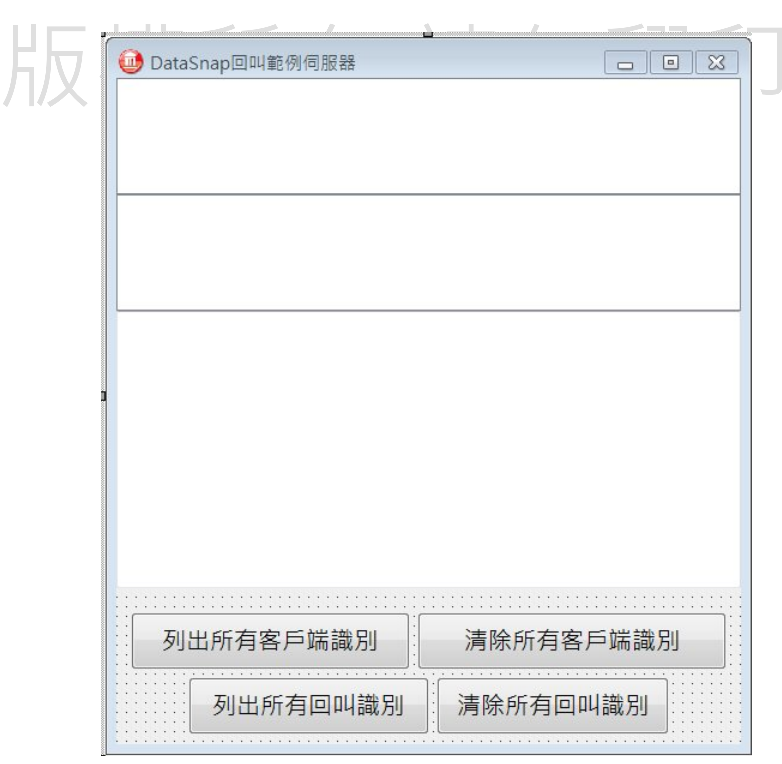



圖 13 修改範例 DataSnap 伺服器的主表單

要取得所有連結到 DataSnap 伺服器的用戶端識別資訊，我們可以呼叫 TDSServer 元件的 GetAllChannelClientId 方法，GetAllChannelClientId 能夠回傳在一個特定的回叫通道中所有註冊的用戶端識別，它的原型如下:

```
function GetAllChannelClientId(const ChannelName: String): TList<String>;
```

我們只需要傳遞特定的回叫通道給 GetAllChannelClientId 即可從回傳的 TList 物件中取得所有註冊的用戶端識別，在這個範例中我們只需要傳入 DEMOChannel 即可。因此『列出所有客戶端識別』按鈕的 OnClick 事件處理函式實作如下:

```
procedure TForm17.btnListAllClientIdsClick(Sender: TObject);
var
  aIdList : TList<String>;
  sId : String;
begin
  aIdList := ServerContainer5.DSServer1.GetAllChannelClientId(DEMOChannel);
  try
    for sId in aIdList do
      lbAllClientIds.Items.Add(sId);
  finally
    aIdList.Free;
  end;
end;
```

現在就可以修改範例 DataSnap 用戶端應用程式了。

## 修改 DataSnap 用戶端應用程式

開啟範例 DataSnap 用戶端應用程式的主表單，在其中加入兩個 TEdit 元件以便讓使用者輸入這個用戶端的用戶端識別以及回叫識別，再置入一個『回叫客戶端』按鈕以及一個 TMemo 元件，當我們在新的 TMemo 元件中輸入任何資訊時，再點選『回叫客戶端』按鈕就可以把新的 TMemo 元件中的資訊藉由回叫功能傳遞給另外一個用戶端應用程式。但我們如何指定要回叫那一個用戶端應用程式呢?這就由新加入的兩個 TComboBox 來指定了，我們可以在這兩個 TComboBox 中輸入要溝通的另外一個用戶端的用戶端識別以及回叫識別，那麼『回叫客戶端』按鈕的 OnClick 事件處理函式就可以藉由回叫功能來回叫另外一個用戶端應用程式了，此時範例用戶端應用程式的主表單如下所示:

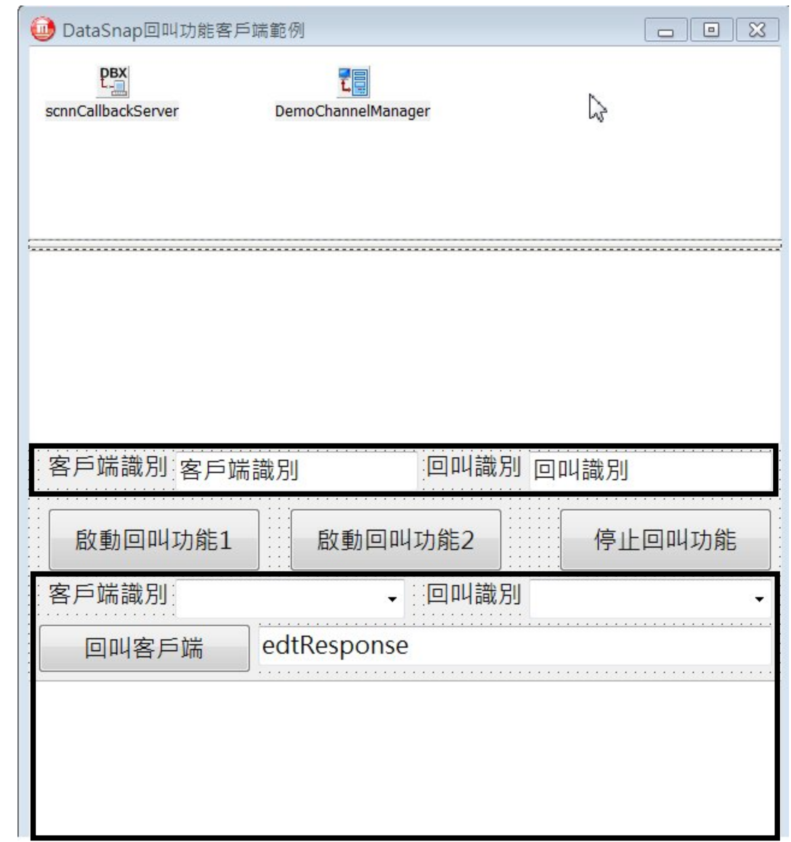



圖 14 修改範例 DataSnap 用戶端應用程式的主表單

版權所有 請勿翻印

接下來就解決的問題是一個用戶端應用程式如何能夠回叫另外一個用戶端應用程式呢? 這可以使用 TDSAdminClient 類別的 NotifyCallback 方法。TDSAdminClient 是位於 DSProxy 程式單元中的類別，在 XE 版中因應強化的回叫功能而加入了數個新的方法，其中的 NotifyCallback 方法可以回叫指定回叫通道中特定用戶端識別以及回叫識別的用戶端，它的原型宣告如下:

```
function NotifyCallback(ChannelName: string; ClientId: string; CallbackId: string; Msg: TJSONValue; out Response: TJSONValue): Boolean;
```

NotifyCallback 接受數個參數，下面的表格說明了每一個參數的目的:

| 參數名稱 | 說明 |
| --- | --- |
| ChannelName | 回叫通道名稱 |
| ClientId | 用戶端識別 |
| CallbackId | 回叫識別 |
| Msg | 傳遞的資訊 |
| Response | 被回叫用戶端的回傳資訊 |

因此我們只需要為每一個用戶端應用程式在註冊時加入一個獨特的用戶端識別，如此一來我們就可以藉由 NotifyCallback 讓任何兩個用戶端或是多個用

戶端相互溝通了。那麼我們如何設定用戶端識別? 還記得我們在用戶端應用程式中使用了 TDSClientCallbackChannelManager 元件來註冊用戶端的回叫資訊嗎? TDSClientCallbackChannelManager 擁有一個 ManagerId 特性，這個 ManagerId 特性就可以做為用戶端的識別，因此讓我們修改用戶端中『啟動回叫功能』按鈕的 OnClick 事件處理函式如下:

```
001   procedure TfmMainForm.btnStartClick(Sender: TObject);
002   begin
003     SetupTask;
004     AddIdsToComboBox(edtClientId.Text, edtCallbackId.Text);
005     EnableDisableButtons(False, True, True);
006     DemoChannelManager.ManagerId := edtClientId.Text;
007     DemoChannelManager.RegisterCallback(edtCallbackId.Text, TDemoCallback.Create);
008     aIdList.Add(edtCallbackId.Text);
009   end;
```

在 004 行我們把使用者於 edtClientId 元件輸入的用戶端識別加入到 cbClientIds 中，並且把使用者於 edtCallbackId 元件輸入的回叫識別加入到 cbCallbackIds 中:

```
procedure TfmMainForm.AddIdsToComboBox(aClientId, aCallbackId: String);
begin
  cbClientIds.Items.Add(aClientId);
  cbCallbackIds.Items.Add(aCallbackId);
end;
```

接著在 006 行設定 TDSClientCallbackChannelManager 元件的 ManagerId 特性為使用者於 edtClientId 元件輸入的用戶端識別，最後才於 007 行呼叫 RegisterCallback 註冊這個用戶端的回叫資訊。一旦設定了用戶端識別，其他用戶端就可以使用這個用戶端識別來和這個用戶端進行溝通。

完成了上述的修改之後，我們就可以實作主表單中『回叫客戶端』按鈕的 OnClick 事件處理函式了:

```
001   procedure TfmMainForm.btnBroadcastToClientClick(Sender: TObject);
002   var
003     LClient: TDSAdminClient;
004     LMessage: TJSONString;
005     LResponse: TJSONValue;
006     LConnection: TDBXConnection;
007   begin
```

```
008     LConnection := scnnCallbackServer.DBXConnection;
009     LClient := TDSAdminClient.Create(LConnection, False);
010     try
011       LMessage := TJSONString.Create(Format('呼叫通道: %s, 回叫識別: %s, 客戶端識別: %s,
回叫訊息: %s',
012         [DemoChannelManager.ChannelName, cbCallbackIds.Text, cbClientIds.Text,
mmChannelCallbacks.Text]));
013       try
014         LClient.NotifyCallback(DemoChannelManager.ChannelName, cbClientIds.Text,
cbCallbackIds.Text, LMessage, LResponse);
015         try
016           if LResponse <> nil then
017             edtResponse.Text := Format('客戶端回應: %s', [LResponse.ToString])
018           else
019             edtResponse.Text := Format('客戶端回應: %s', ['nil']);
020         finally
021           LResponse.Free;
022         end;
023       finally
024         LMessage.Free;
025       end;
026     finally
027       LClient.Free;
028     end;
029   end;
```

首先我們在 009 行先建立 TDSAdminClient 物件，在 010 行建立要傳遞給其他用戶端應用程式的以 TJSONString 封裝的訊息，接著在 014 行就藉由 NotifyCallback 方法呼叫其他的用戶端，請注意我們傳入的資訊，首先參數是回叫通道名稱，這可以從 TDSClientCallbackChannelManager 元件的 ChannelName 特性值取得，第 2 個參數是用戶端識別，我們傳入主表單中由使用者在 TComboBox 元件 cbClientIds 的 Text 特性值，第 3 個參數是用戶端回叫識別，我們傳入主表單中由使用者在 TComboBox 元件 cbCallbackIds 的 Text 特性值，第 4 個參數是這個用戶端要傳遞給其他用戶端的資訊，我們傳入在 011 行封裝的 TJSONString 物件 LMessage，最後一個參數則必須傳入型態為 TJSONValue 的物件以接受其他用戶端回傳的執行結果。

現在請編譯並且執行 2 份不同的範例用戶端應用程式，如下所示:

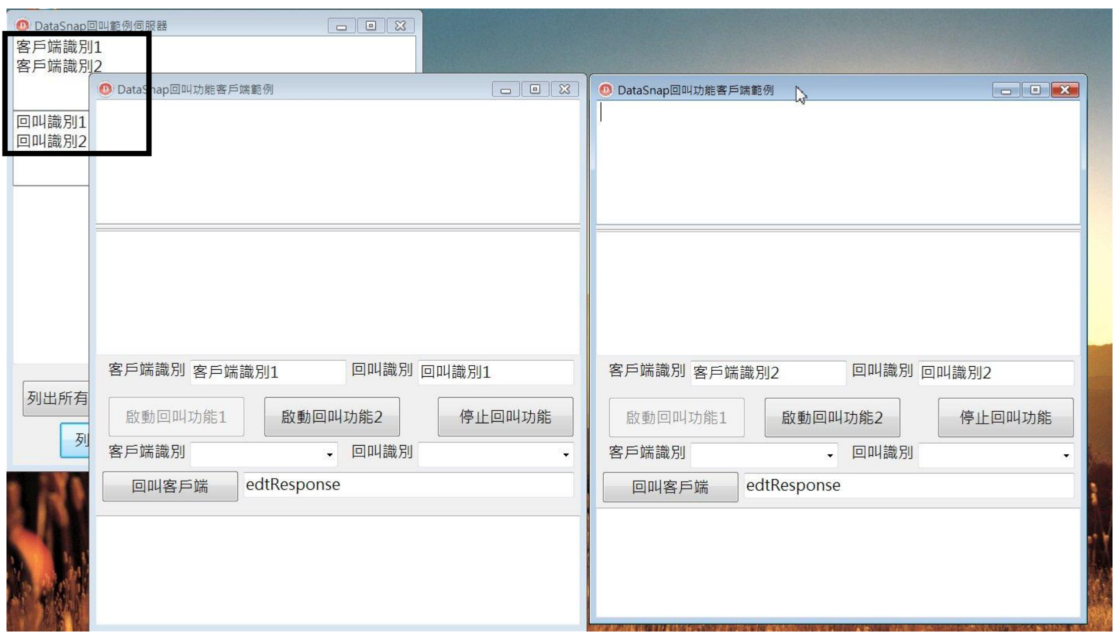



圖 15 啟動範例 DataSnap 伺服器以及兩個用戶端應用程式，並且讓不同的用戶端應用程式註冊不同的用戶端識別

讓我們在第一個用戶端應用程式輸入其用戶端識別為『客戶端識別 1』，其回叫識別為『回叫識別 1』，接著在第二個用戶端應用程式中輸入其用戶端識別為『客戶端識別 2』，其回叫識別為『回叫識別 2』。之後我們如果點選範例 DataSnap 伺服器中來『列出所有客戶端識別』按鈕和『列出所有回叫識別』按鈕，那麼就可以如上圖左上方看到，範例 DataSnap 伺服器果然可在 DemoChannel 這個回叫通道中找到這些資訊。

版權所有 請勿翻印

現在請在用戶端應用程式 1 中的兩個 TComboBox 元件輸入『客戶端識別 2』和『回叫識別 2』讓用戶端應用程式 1 回叫用戶端應用程式 2。同樣的請在請在用戶端應用程式 2 中的兩個 TComboBox 元件輸入『客戶端識別 1』和『回叫識別 1』讓用戶端應用程式 1 回叫用戶端應用程式 1。

之後在用戶端應用程式 1 中下方的 TMemo 元件中輸入任何的資訊，並且點選『回叫客戶端』按鈕，那麼我們立刻可以在用戶端應用程式 2 的上方 TMemo 元件中看到由用戶端應用程式 1 中傳遞來的資訊。同樣的，如果我們在用戶端應用程式 2 中下方的 TMemo 元件中輸入任何的資訊，並且點選『回叫客戶端』按鈕，那麼我們立刻可以在用戶端應用程式 1 的上方 TMemo 元件中看到由用戶端應用程式 2 中傳遞來的資訊，此時這兩個用戶端應用程式看起來如下所示:

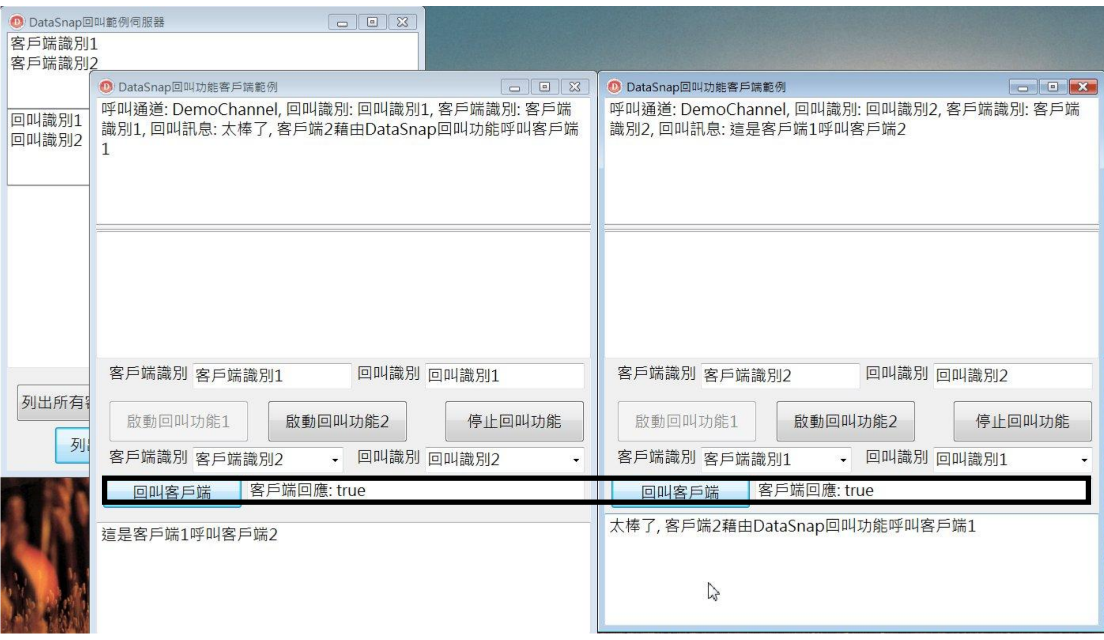



圖 16 兩個不同的用戶端應用程式果然可藉由回叫功能相互溝通了

我們可以看到藉由 DataSnap 的回叫功能，我們果然可以讓不同的用戶端應用程式都藉由回叫通道來溝通並且相互傳遞任何的資訊。此外請注意上圖中兩個用戶端都相互回傳『客戶端回應:true』的訊息，這是因為我們在前面實作用戶端的回叫函式時，用戶端的回叫函式都回傳 TJSONTrue 物件。由於 NotifyCallback 可以回傳回叫用戶端的執行結果，因此如果我們希望回叫函式回傳其他型態的資訊，那麼我們可以修改兩個範例用戶端回叫函式如下:

版權所有 請勿翻印

```
001   function TDemoCallback.Execute(const Arg: TJSONValue): TJSONValue;
002   var
003     sDemoMessage : String;
004   begin
005   //  Result := TJSONTrue.Create;
006     Result := TJSONString.Create('成功呼叫客戶端');
007
008     if (Arg is TJSONString) then
009     begin
010       sDemoMessage := TJSONString(Arg).Value;
011       TThread.Synchronize(nil,
012                       procedure
013                       begin
014                         fmMainForm.mmDemoMessage.Lines.Text := sDemoMessage;
015                       end
016                       );
```

```
017     end;
018   end;
019
020   { TDemoCallback2 }
021
022   constructor TDemoCallback2.Create;
023   begin
024
025   end;
026
027   function TDemoCallback2.Execute(const Arg: TJSONValue): TJSONValue;
028   var
029     sDemoMessage : String;
030   begin
031   //  Result := TJSONTrue.Create;
032     Result := TJSONString.Create('成功呼叫客戶端');
033
034     if (Arg is TJSONString) then
035     begin
036       sDemoMessage := TJSONString(Arg).Value;
037       TThread.Synchronize(nil,
038                         procedure
039                         begin
040                           fmMainForm.mmDemoMessage2.Lines.Text := sDemoMessage;
041                         end
042                         );
043     end;
044 end;
```

在上面的兩個回叫函式中我們修改回傳 TJSONString 封裝的資訊而不是單純的 TJSONTrue 物件，這是果然是因為 NotifyCallback 回傳的型態是 TJSONValue，因此任何從 TJSONValue 類別繼承下來的類別物件都可以做為回叫函式的執行回傳結果。

現在如果我們再次執行兩個用戶端應用程式，那麼就可以看到類似如下的結果:

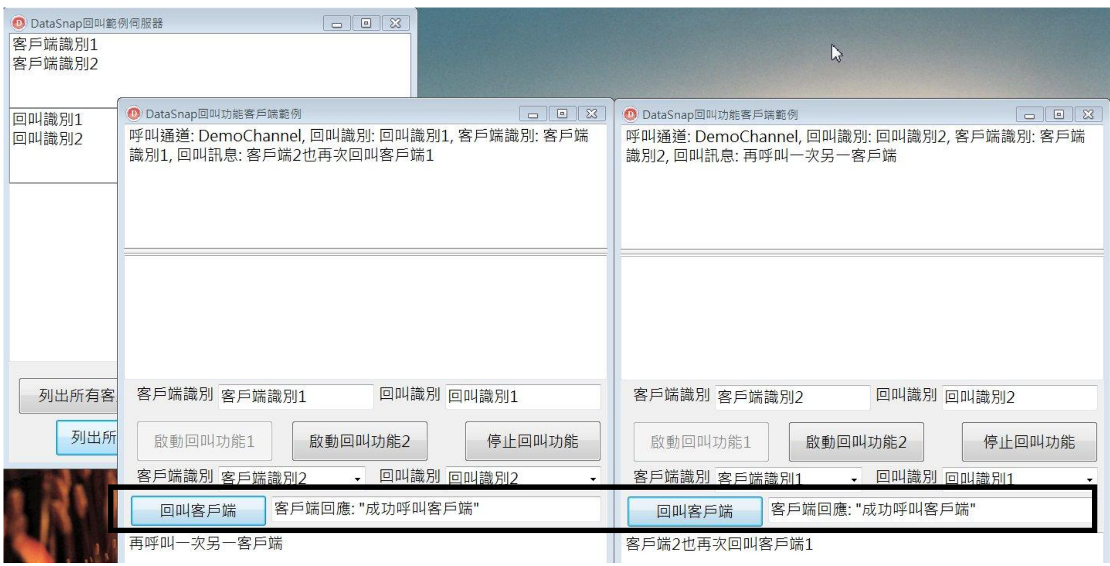



圖 17 不同用戶端的回叫函式可傳遞執行結果回其他用戶端應用程式

從上圖中我們可以看到不同的用戶端果然可以藉由回叫函式傳遞任何的資訊回其他的用戶端應用程式。

現在您應該已經充分的瞭解了 DataSnap 的回叫功能了，也許您可以自己試著完成一個小功課，請繼續修改範例 DataSnap 伺服器和用戶端應用程式讓用戶端應用程式可以註冊不同的回叫通道而不像前面的範例應用程式只使用固定的 DemoChannel 這個回叫通道，如何? 應該不算太難而且應該很有趣吧。

版權所有 請勿翻印

## 開發輕薄型回叫 DataSnap 用戶端

DataSnap 10.3 除了可以讓 Delphi/C++Builder 的用戶端參與回叫機制之外，也提供了支援 JavaScript 用戶端的能力，這意謂任何支援 JavaScript 的框架，開發工具或是程式語言都可以參與 DataSnap 的回叫機制，例如 Java 的用戶端，.NET 的用戶端或是 Ruby/PHP 等的用戶端，這讓 DataSnap XE 的回叫機制變得非常的實用。在前面的章節中我們已經細節的說明了如何使用 DataSnap 的回叫機制以及開發了數個 Delphi 的範例，在本小節中我們將進一步說明如何使用 JavaScript 來參與 DataSnap 10.3 的回叫機制。

首先讓我們建立一個支援 HTTP/HTTPS 的 DataSnap 伺服器:

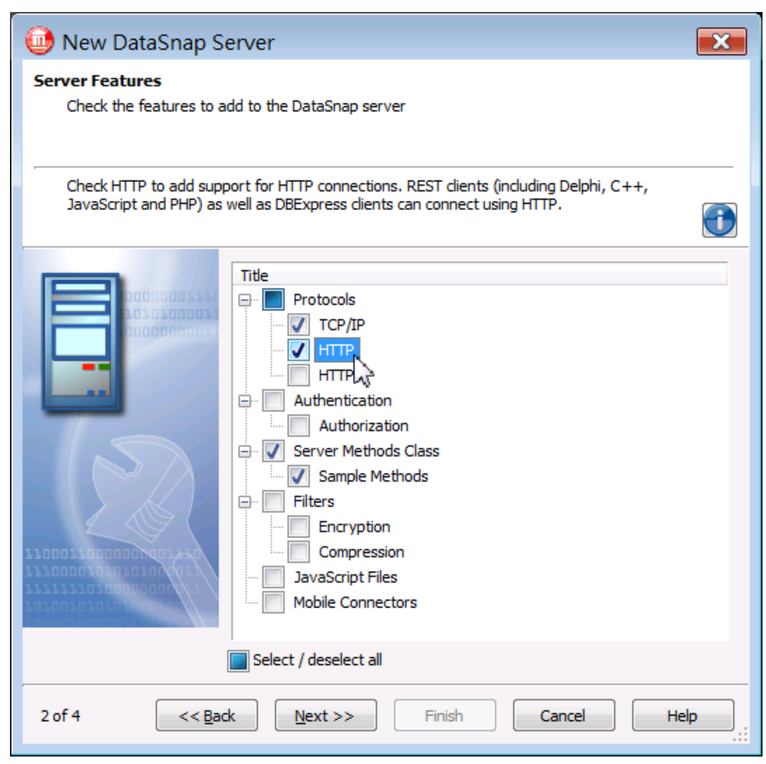



圖 18 建立支援 HTTP/HTTPS 和 TCP 的 DataSnap 伺服器

接著在 ServerContainer 中放入如下的元件:

版權所有 請勿翻印

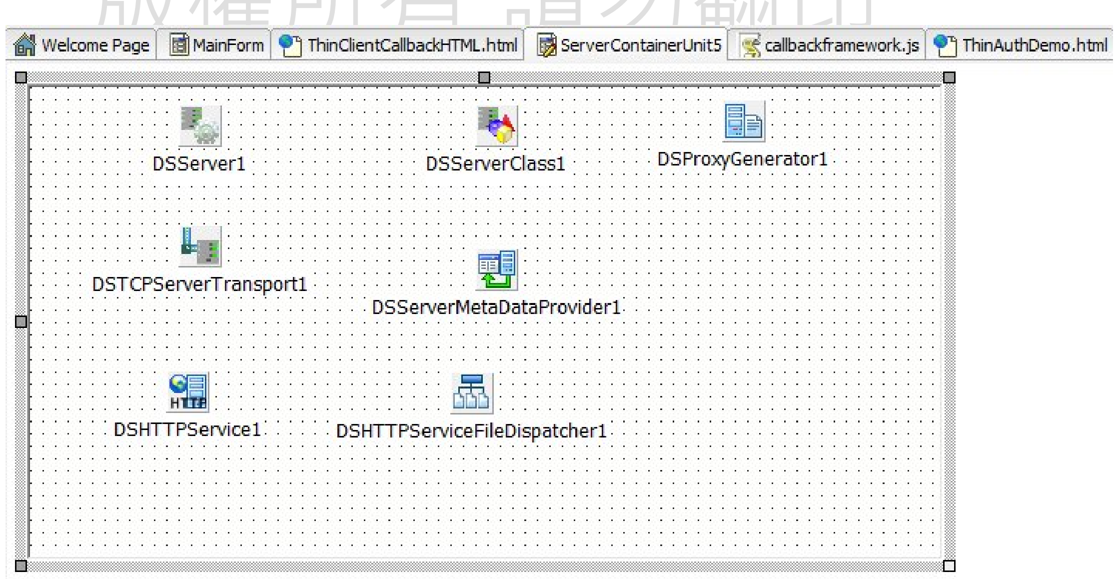



圖 19 在伺服器的 ServerContainer 加入相關的元件以支援瀏覽器的呼叫

並且設定每一個元件的特性值如下:

TDSServerMetaDataProvider 元件:

| 特性 | 特性值 |
|---|---|
| Server | DSServer1 |

TDSHTTPServiceFileDispatcher 元件:

| 特性 | 特性值 |
|---|---|
| Service | DSHTTPService1 |
| RootDirectory | E:\QComm\Book2\Demos\DataSnapChapter03\ThinClientCallbackDemos\Client\ |

TDSHTTPServiceFileDispatcher 元件的 RootDirectory 特性是指定稍後使用 JavaScript 參與回叫的 HTML 檔案的目錄所在地。

TDSProxyGenerator 元件:

| 特性 | 特性值 |
|---|---|
| MetaDataProvider | DSServerMetaDataProvider1 |
| TargetDirectory | E:\QComm\Book2\Demos\DataSnapChapter03\ThinClientCallbackDemos\Client\ |
| TargetUnitName | serverfunctions.js |
| Writer | Java Script REST |

TDSProxyGenerator 元件的 TargetDirectory 是指稍後由 DataSnap 自動產生的 JavaScript 檔案的儲存目錄，這個自動產生的 JavaScript 檔案就是 TargetUnitName 特性值，這也就是說 TDSProxyGenerator 會在目錄 E:\QComm\Book2\Demos\DataSnapChapter03\ThinClientCallback Demos\Client\ 中產生一個名為 serverfunctions.js 的檔案，serverfunctions.js 中的JavaScript 程式碼封裝了 DataSnap 伺服器提供的服務。

版權所有 請勿翻印

```
procedure TForm18.btnListAllClientIdsClick(Sender: TObject);
var
  aIdList : TList<String>;
  sId : String;
begin
  aIdList := ServerContainer5.DSServer1.GetAllChannelClientId(DEMOChannel);
  try
    for sId in aIdList do
      lbAllClientIds.Items.Add(sId);
  finally
    aIdList.Free;
  end;
end;

procedure TForm18.mmCallbackMessagesChange(Sender: TObject);
var
```

```
  vMessage : TJSONString;
begin
  vMessage := TJSONString.Create(mmCallbackMessages.Lines.Text);
  ServerContainer5.DSServer1.BroadcastMessage(DEMOChannel, vMessage);
end;
```

為了讓 TDSProxyGenerator 元件在稍後範例 HTML 檔案被存取時能夠自動產生 serverfunctions.js，因此我們需要在 TDSHTTPServiceFileDispatcher 元件的 BeforeDispatch 事件中撰寫如下的程式碼。下面的程式碼是當目前用戶端的 HTML 檔案在瀏覽器中被存取時，檢查的目錄中是否已經包含了 serverfunctions.js，如果沒有的話就呼叫 TDSProxyGenerator 元件的 Write 方法自動產生 serverfunctions.js。

```
procedure TServerContainer5.DSHTTPServiceFileDispatcher1BeforeDispatch(
  Sender: TObject; const AFileName: string; AContext: TDSHTTPContext;
  Request: TDSHTTPRequest; Response: TDSHTTPResponse; var Handled: Boolean);
begin
  if (SameFileName(ExtractFileName(AFileName), DSProxyGenerator1.TargetUnitName) and not
FileExists(AFileName)) then
    DSProxyGenerator1.Write;
end;
```

例如下圖就是稍後在瀏覽器中執行範例 HTML 檔案 ThinClientCallbackHTML.html 後便會在範例目錄 E:\QComm\Book2\Demos\DataSnapChapter03\ThinClientCallback Demos\Client\中產生為 serverfunctions.js:

e:\QComm\Book2\Demos\DataSnapChapter03\ThinClientCallbackDemos\Client*.*

| Name | Ext | Size | ↓Date |
|---|---|---|---|
| .. | | <DIR> | 2011/05/03 18:30 |
| __history | | <DIR> | 2011/05/03 17:57 |
| serverfunctions | js | 27.9 k | 2011/05/03 18:30 |
| ThinClientCallbackHTML | html | 2.0 k | 2011/05/03 17:57 |
| serverfunctioninvoker | js | 8.6 k | 2010/11/04 03:50 |
| serverfunctionexecutor | js | 24.8 k | 2010/11/04 03:50 |
| base64 | js | 3.3 k | 2010/08/28 06:00 |
| callbackframework | js | 16.3 k | 2010/08/28 06:00 |
| connection | js | 656 b | 2010/08/28 06:00 |
| json-min | js | 2.8 k | 2010/08/28 06:00 |

圖 20 範例目錄中的檔案以及由範例伺服器自動產生的 serverfunctions.js

現在於 Delphi 整合發展環境中建立一個新的 HTML 檔案，然後設計如下的圖形使用者介面:

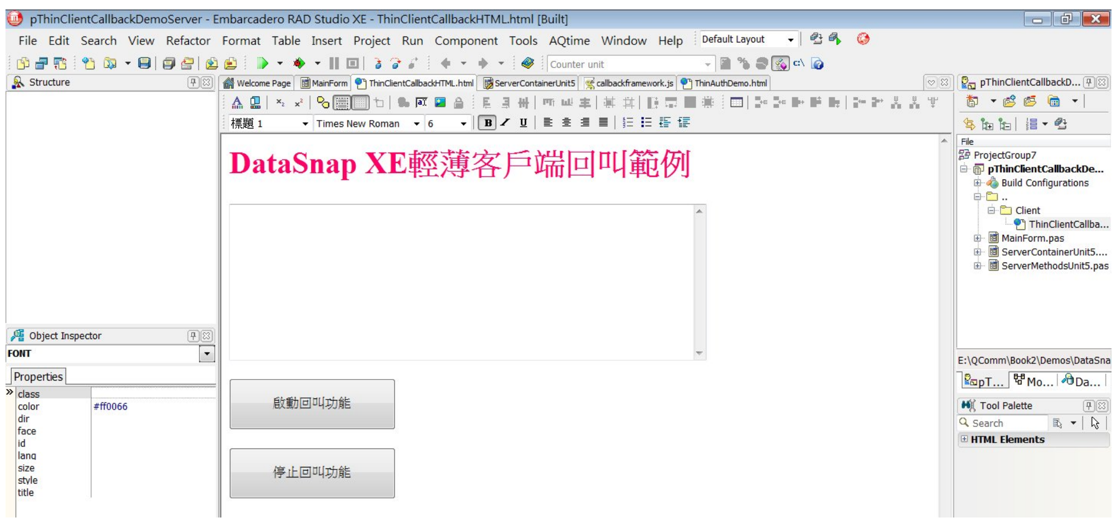



圖 21 使用 Delphi IDE 建立範例 HTML 以準備在瀏覽器中使用回叫功能

接著在 HTML 檔案的程式碼中撰寫如下的程式碼:

```
001    <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
002    <html>
003      <head>
004        <title>DataSnap XE 輕薄客戶端認證範例
005    </title>
006    <meta HTTP-EQUIV="Content-Type" CONTENT="text/html; CHARSET=UTF-8"><meta
http-equiv="cache-control" content="no-cache">
007        <script type="text/javascript" src="base64.js"></script>
008        <script type="text/javascript" src="json-min.js"></script>
009        <script type="text/javascript" src="serverfunctionexecutor.js"></script>
010        <script type="text/javascript" src="ServerFunctions.js"></script>
011        <script type="text/javascript" src="connection.js"></script>
012        <script type="text/javascript" src="callbackframework.js"></script>
013        <script type="text/javascript">
014
015        function startCallback()
016        {
017          var connectionInfo = {"host":"localhost","port":"8089"};
018          var channel = new ClientChannel("jsc1", "DemoChannel", connectionInfo);
019          var callback = new ClientCallback(channel, "uid2", function(jsonValue)
020                            {
021                              if (jsonValue != null && jsonValue.created == null)
022                              {
023                                document.getElementById("mmCallback").value = jsonValue;
```

```
024                          }
025                          return true;
026                        });
027         channel.connect(callback);
028       }
029
030       function stopCallback()
031       {
032         if (channel != null)
033         {
034           channel.disconnect();
035         }
036
037
038       }
039
040       </script>
041     </head>
042     <body>
043     <div>
044        <h1><font color="#ff0066">DataSnap XE 輕薄客戶端回叫範例
045   </font></h1>
046        <form onsubmit="startCallback(); return false;">
047          <textarea rows="10" cols="60" id="mmCallback"></textarea><br><br />
048          <input id="btncallServer" type="submit" value="啟動回叫功能" style="WIDTH:
217px; HEIGHT: 67px" size="34" onclick="StartCallback" />
049        </form>
050        <form onsubmit="stopCallback(); return false;">
051          <input id="btncallServer" type="submit" value="停止回叫功能" style="WIDTH:
217px; HEIGHT: 67px" size="34" onclick="StopCallback" />
052        </form>
053     </div>
054     </body>
055   </html>
```

在上面的程式碼中，當使用者在瀏覽器中點選『啟動回叫功能』按鈕後就會呼叫 startCallback 函式，startCallback 首先在 017 行建立 connectionInfo 資訊設定呼叫的伺服器位址以及 DataSnap 伺服器使用的通信埠。018 行建立 ClientChannel 物件並且傳入前面小節已經說明的參數，019 行建立

ClientCallback 物件並且註冊用戶端 JavaScript 的回叫函式，這個用戶端回叫函式接受 DataSnap 伺服器輸入的訊息並且同步顯示在用戶端的瀏覽器之中，就如同前面的 4-2 節的範例一樣，只是在這裡我們是使用 JavaScript 和瀏覽器做為回叫的用戶端。

現在啟動瀏覽器並且載入範例 HTML 檔案 ThinClientCallbackHTML.html，接著在 DataSnap 伺服器中點選『列出所有客戶端識別』按鈕的話，就可以在 DataSnap 伺服器的主表單中看到用戶端瀏覽器使用 JavaScript 註冊的通道 ID『jsc1』，此時如果在 DataSnap 伺服器的主表單的 TMemo 元件中訊息的話，就可以立刻在瀏覽器中看到這些訊息，因為 DataSnap 伺服器會藉由回叫機制立刻通知用戶端，如下圖所示:

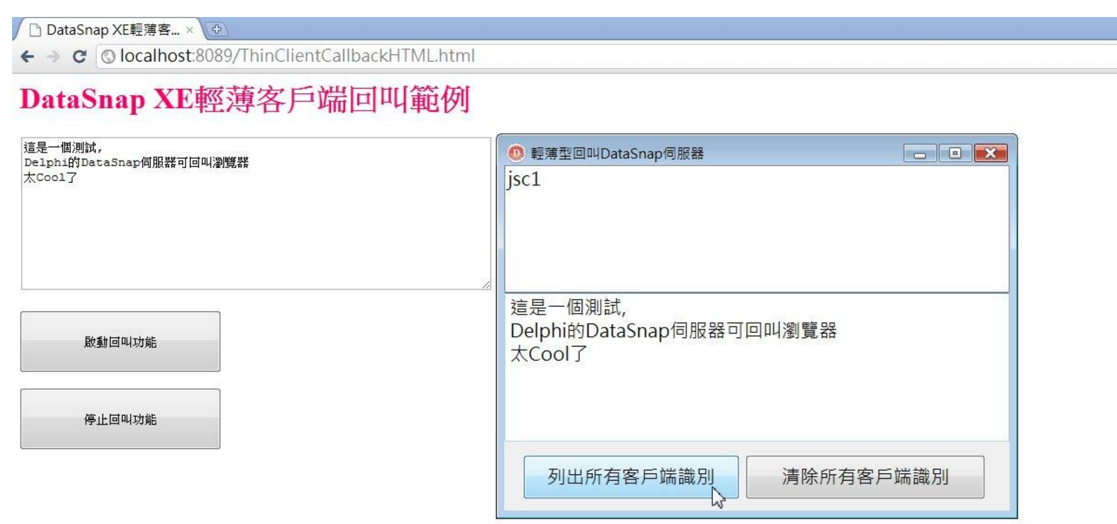



圖 22 範例 DataSnap 伺服器果然可藉由回叫機制回叫在瀏覽器中執行的 JavaScript 程式碼

如何 DataSnap 10.3 是回叫機制是不是很強大? 藉由 JavaScript 開發人員甚至可以撰寫手機的用戶端應用程式並且讓 DataSnap 伺服器藉由回叫機制通知用戶端任何的資訊，Cool!

## 4-3 結論

DataSnap 10.3 在原有的回叫機制下持續的增加功能，DataSnap 10.3 的回叫功能供支援了通道和 JavaScript 用戶端的能力，除了 Delphi/C++Builder 的用戶端之外，也允許所有支援 JavaScript 的用戶端都能夠參與和使用 DataSnap 的回叫機制，讓 DataSnap 的回叫機制可提供跨平台從而能夠建立更為複雜，實際和先進的回叫架構。

# 第5章 使用DataSnap過濾器

讀者現在應該瞭解 DataSnap 的 JSON 是是使用字串型態來傳遞資料的，因此所有非字串型態的資料都必須轉換為字串的型態，雖然如此一來在處理上比較簡單，但這也造成了其他的問題，例如一些敏感性的資料如果使用字串型態來傳遞的話就不太實際。DataSnap 從 2010 開始為了解決這種問題因此加入了過濾器的機制，讓開發人員在傳遞特殊的資料時可以藉由過濾器來進行額外的處理，例如在傳遞資料出去時先加密，並且在接受到資料之後再進行解密，在 DataSnap 2010 即提供了壓縮和解壓縮的過濾器，到了 DataSnap 2011 又增加了內建的加密/解密的過濾器，本章的目的即在於討論如何使用 DataSnap 的過濾器。

版權所有 請勿翻印

使用 DataSnap 過濾器非常的簡單，Delphi 2010 開始即內建了一個壓縮過濾器，可以有效的壓縮使用 TCP/IP 通訊協定的資料傳遞。讓我們先說明如何使用這個內建的過濾器，稍後我們再深入的說明如何開發客製化過濾器。

## 5-1 使用內建的過濾器

DataSnap 10.3 版一共提供了三個過濾器可供開發人員使用，如果開發人員仍然覺得不夠或是有特殊的需求，那麼也可以撰寫客製化過濾器來使用。由於使用 DataSnap 過濾器非常的簡單，因此本節就以一個範例來說明如何使用 DataSnap 的壓縮/解壓縮過濾器。

在下面的範例中我們將使用一個包含圖形的資料表來展示使用壓縮/解壓縮過濾器的效果。

### 5-1-1 建立 DataSnap 過濾器伺服器

首先在 Delphi 10.3 中建立一個 DataSnap Server 專案，先開啟專案中的 ServerMethodUnit 程式單元，然後從 Data Explorer 頁次中拖曳範例資料表 BIOLIFE 到 ServerMethodUnit 程式單元中，IDE 便會自動產生

TSQLConnection 和 TSQLDataSet 元件連結到範例資料表 BIOLIFE，接著放入 TDataSetProvider 元件連結到程式單元中的 TSQLDataSet，如下所示:

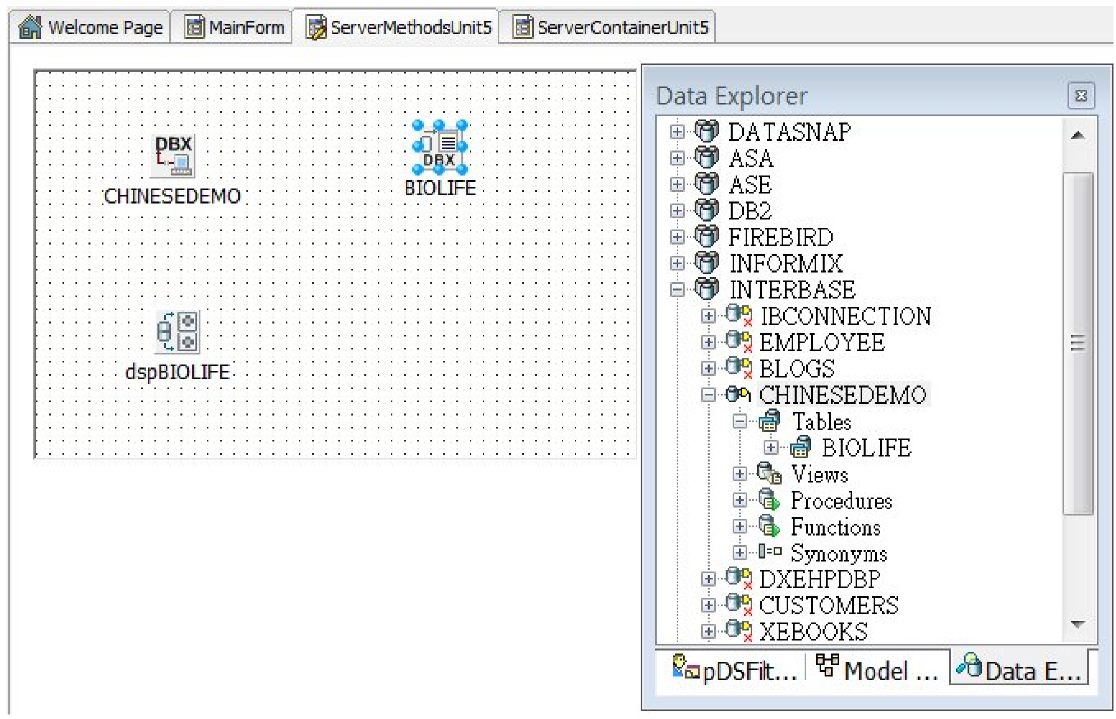



圖 1 在 DataSnap 伺服器的 ServerMethodUnit 中加入 dbExpress 元件連結到範例資料表 BIOLIFE

版權所有 請勿翻印

接著開啟專案中的 ServerContainerUnit 程式單元，點選其中的 TDSTCPServerTransport 元件，接著在物件檢視器中點選它的 Filters 特性，在顯示的特性值編輯器中點選上方的 Add New 按鈕加入一個新的過濾器，點選特性值編輯器中新加入的 TTransportFilterItem 過濾器，再次點選物件檢視器，選擇 FilterId 特性就可以從下拉盒中看到 DataSnap 提供了 PC1，RSA 以及 ZlibCompression 三個過濾器可供使用，如下圖所示:

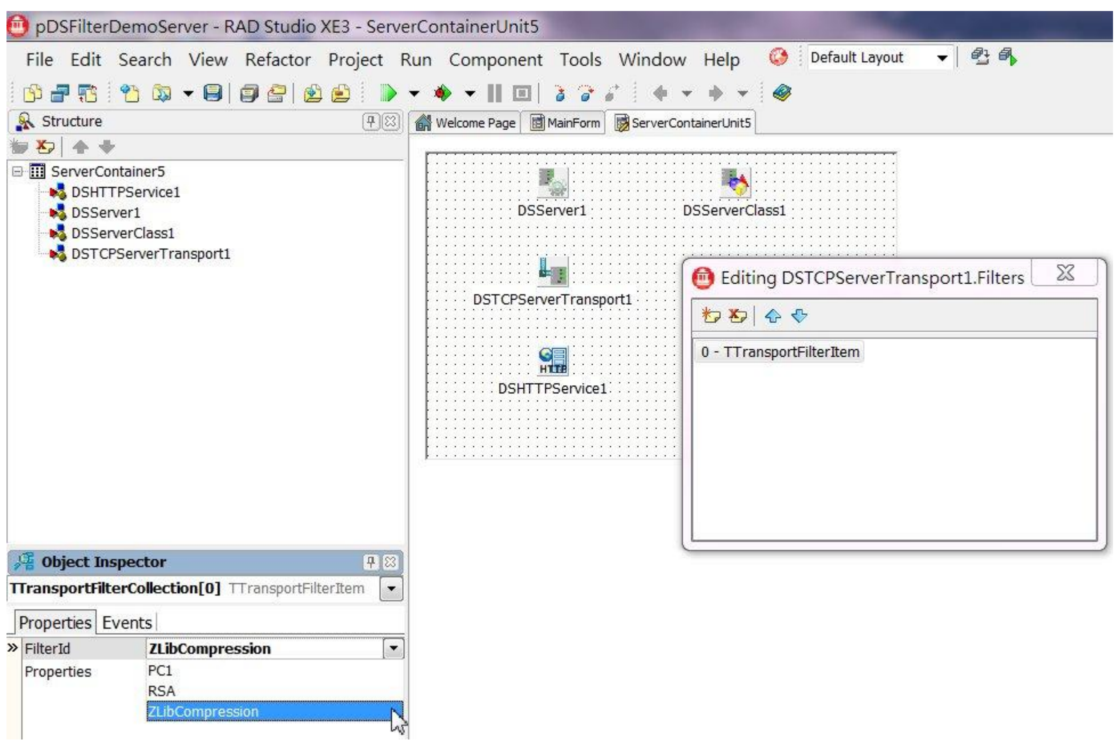



圖 2 在 ServerContainerUnit 程式單元中點選 TDSTCPServerTransport 元件加入使用過濾器

在這裡讓我們選擇使用 ZlibCompression 過濾器，如此一來就完成了使用 DataSnap 過濾器的步驟了，ZlibCompression 過濾器就是 DataSnap 內建的壓縮過濾器，在加入了 ZlibCompression 過濾器之後，編譯並且執行範例 DataSnap 伺服器，現在 DataSnap 伺服器就自動提供了壓縮 JSON 資料的能力。

版權所有 請勿翻印

## 5-1-2 建立使用 DataSnap 過濾器的用戶端應用程式

在專案群組中建立一個 VCL Form 應用程式專案，建立一個 DataSnap Client Module，連結到上一小節的範例 DataSnap 伺服器，IDE 便會在 DataSnap Client Module 中產生 TSQLConnection 元件，接著在其中放入 TDSProviderConnection 和 TClientDataSet 元件，如下圖所示:

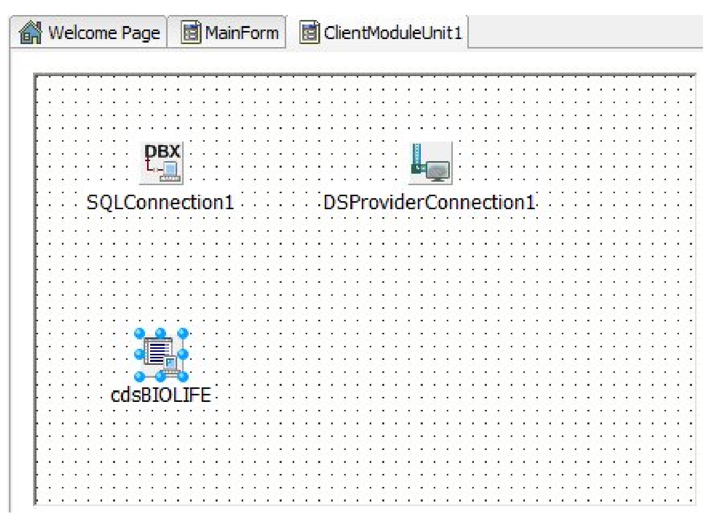



圖 3 在 ClientModuleUnit 程式單元中加入 TDSProviderConnection 和 TClientDataSet 元件

設定 TDSProviderConnection 元件的特性值如下:

| 特性 | 特性值 |
|---|---|
| SQLConnection | SQLConnection1 |
| ServerClassName | TServerMethods5 |
| Name | DSProviderConnection1 |

版權所有 請勿翻印

設定 TDSProviderConnection 元件的特性值如下:

| 特性 | 特性值 |
|---|---|
| RemoteServer | DSProviderConnection1 |
| ProviderName | DspBIOLIFE |
| Name | cdsBIOLIFE |

開啟主表單，在主表單中加入下面的元件並且連結 DataSnap Client Module 中的 cdsBIOLIFE 以準備顯示範例資料表中的資料。

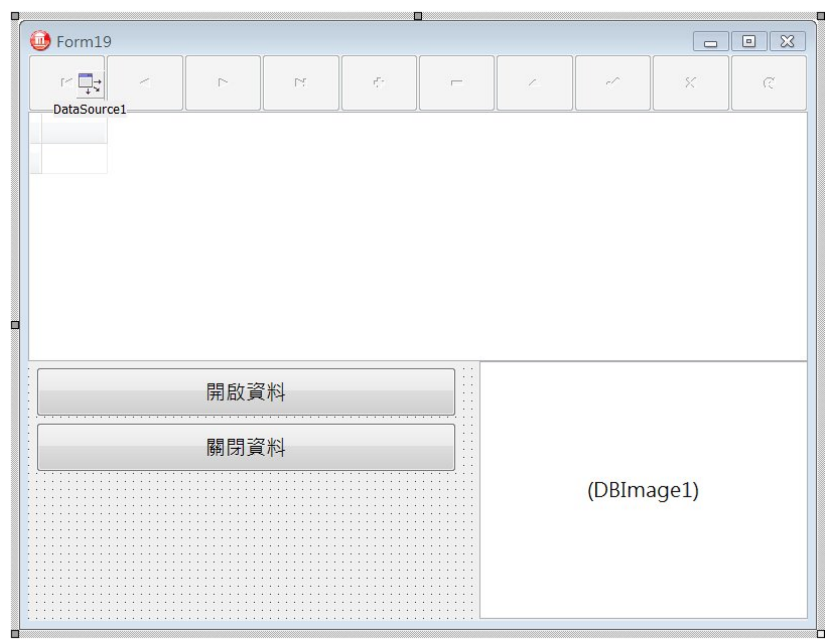



圖 4 範例用戶端應用程式的主表單

接著開啟主表單的原始程式碼，因為我們要在用戶端應用程式中加入解壓縮資料的能力，這非常的簡單，我們只要在用戶端應用程式的主表單中加入使用 DBXCompressionFilter 程式單元即可，例如下面就是用戶端應用程式加入 DBXCompressionFilter 程式單元的程式碼:

版權所有 請勿翻印

```
implementation

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,Dialogs, Grids,
DBGrids, ExtCtrls, DBCtrls, DB, DbxCompressionFilter, StdCtrls;
```

現在編譯並且執行用戶端應用程式，並且讓我們使用 TCP Viewer 來觀察使用壓縮過濾器之前的情形以及使用壓縮過濾器之後的效果。

下圖是 TCP Viewer 顯示範例 DataSnap 應用系統使用壓縮過濾器之前的情形，從下圖中我們可以看到在 DataSnap 伺服器和用戶端應用程式之間傳遞的資料當然是使用字串的型態，所有傳遞的資料都一清二楚，同時請讀者注意下圖右邊顯示了從伺服器傳遞到用戶端的資料量(1093368 位元組)以及從用戶端傳遞到伺服端的資料量(2326 位元組)，由於範例資料表中包含了圖形的資料，因此圖形資料在傳遞時必須轉換為 Base64 的字串型態的資料，所以造成了伺服端和用戶端之間必須傳遞大量的資料。

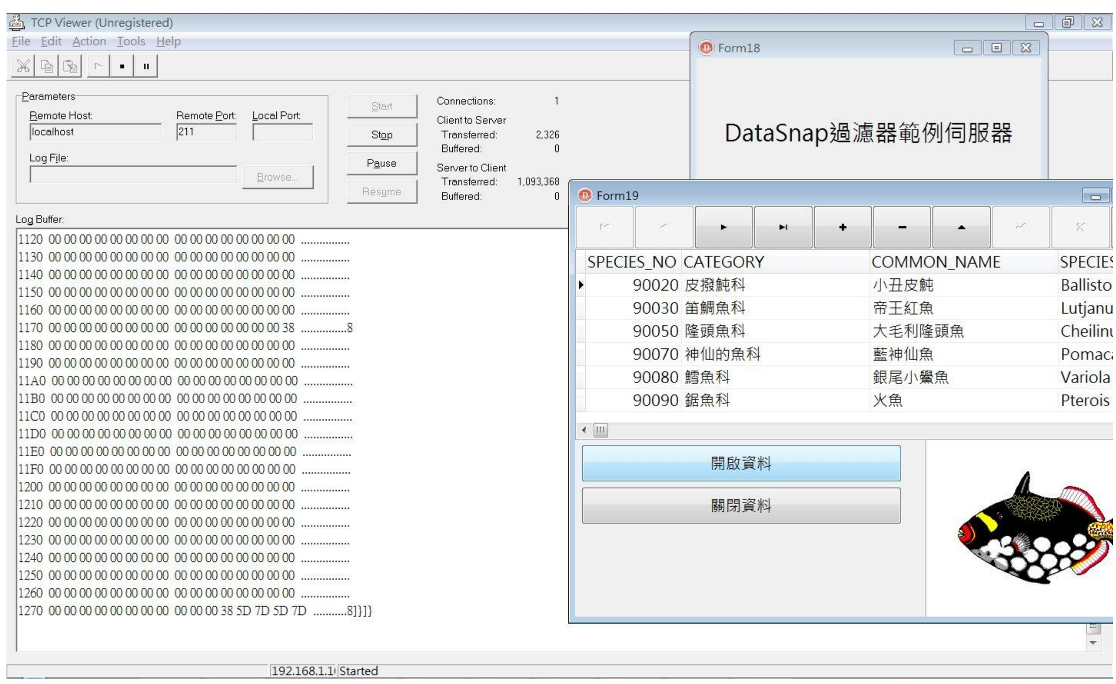



圖 5 不使用 DataSnap 壓縮過濾器傳遞包含圖形資料表的結果

而下圖則是使用壓縮過濾器之後的效果:

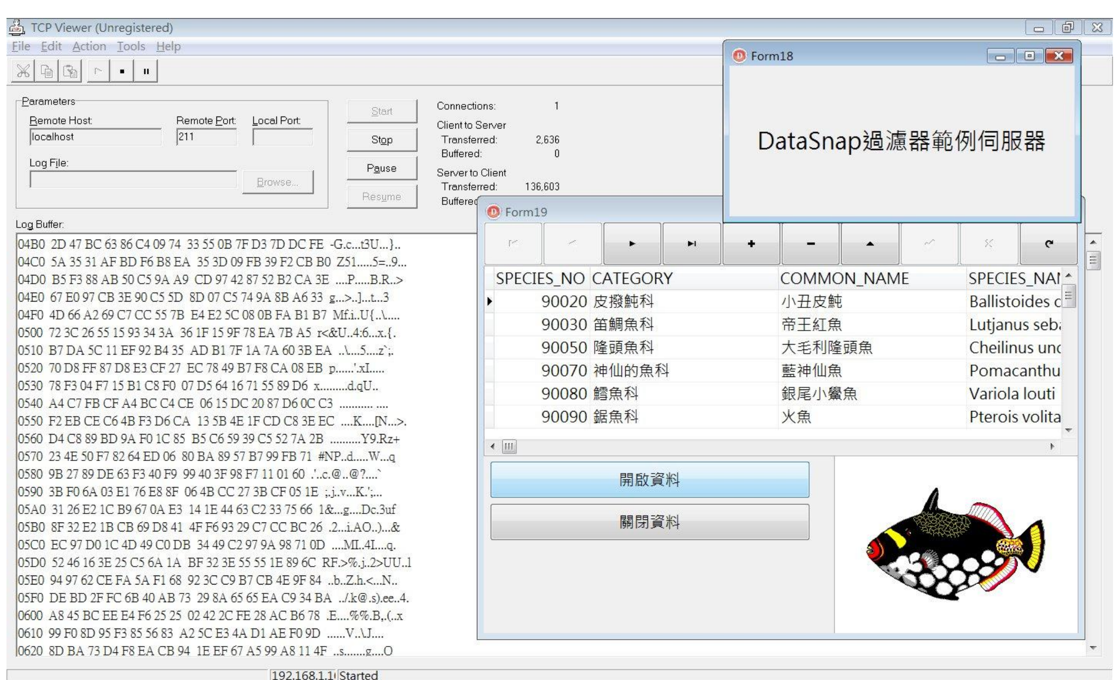



圖 6 使用 DataSnap 壓縮過濾器傳遞包含圖形資料表的結果

我們可以很明顯的看到使用了壓縮過濾器之後，從伺服器傳遞到用戶端的資料量減少到 136603 位元組，可見到壓縮過濾器非常有效的減少了伺服器和用戶端之間的資料傳遞量，這不但可以增加分散式應用程式的執行速度，也可以增加支援的用戶端的數量。

如何? 使用過濾器是不是又簡單，又有明顯的效果? 不過 DataSnap 2010 只提供了一個內建的過濾器實在太少，到了 DataSnap 10.3 雖然增加到了三個，但開發人員仍然可能需要使用客製化的過濾器，好在 DataSnap 過濾器架構在設計時就考慮到了允許讓開發人員能夠自行開發過濾器並且內嵌到 DataSnap 之中，下一小節將討論如何開發客製化過濾器並且使用在 DataSnap 的分散式應用系統中。

## 5-2 開發客製化過濾器

要開發客製化過濾器，開發人員必須從 TTransportFilter 類別衍生子代類別並且實作 TTransportFilter 類別中相關的虛擬方法，下面的表單說明了開發人員需要實作的虛擬方法:

| 函式名稱 | 回傳型態 | 說明 |
|---|---|---|
| GetParameters | TDBXStringArray | 回傳所有的參數 |
| GetUserParameters | TDBXStringArray | 回傳使用者可改變的參數 |
| ProcessInput | TBytes | 使用客製化程式碼正向處理傳遞的資料流 |
| ProcessOutput | TBytes | 使用客製化程式碼反向處理傳遞的資料流 |
| Id | UnicodeString | 過濾器的 ID |
| GetParameterValue | UnicodeString | 取得特定名稱的參數值 |
| SetParameterValue | Boolean | 設定特定名稱的參數值 |

版權所有 請勿翻印

瞭解了需要實作那些虛擬方法之後，我們就可以開始動手開發一個客製化過濾器了。在本文中筆者將撰寫一個非常簡單的加密/解密過濾器，其實這個加密/解密過濾器只是在傳遞資料時和一個字串進行 xor 的動作，到了另一端再次 xor 相同的字串而已，當然這個範例加密/解密過濾器只是為了說明如何開發客製化過濾器，如果讀者需要加密/解密的功能的話，請直接使用 DataSnap 10.3 中提供的 PC1 或是 RSA 過濾器。

首先讓我們宣告範例 TTransportEncryptFilter 類別從 TTransportFilter 類別繼承下來並且複載相關必要的虛擬方法:

```
TTransportEncryptFilter = class(TTransportFilter)
 private
   FEncrypt: TSimpleEncryptor;
   FParameters: TDictionary<String, String>;
 protected
   function GetParameters: TDBXStringArray; override;
```

```
  function GetUserParameters: TDBXStringArray; override;


 public
  function GetParameterValue(const ParamName: UnicodeString): UnicodeString;
    override;
  function SetParameterValue(const ParamName: UnicodeString;
    const ParamValue: UnicodeString): Boolean; override;
  constructor Create; override;
  destructor Destroy; override;
  function ProcessInput(const Data: TBytes): TBytes; override;
  function ProcessOutput(const Data: TBytes): TBytes; override;
  function Id: UnicodeString; override;
 end;
```

TTransportEncryptFilter 類別將使用 TSimpleEncryptor 進行字串 xor 的運算，在 TSimpleEncryptor 類別中實作了兩個方法，Encrypt 和 Decrypt，其實這兩個方法的實作程式碼是一樣的，只是為了說明方便分別實作成 Encrypt 和 Decrypt 以便讓讀者瞭解。Encrypt 和 Decrypt 接受 TBytes 型態的參數，這個參數在 Encrypt 方法中是代表傳遞出去的資料，Encrypt 方法使用程式碼加密之後再把加密過的資料以 TBytes 型態回傳。

版權所有 請勿翻印

而 Decrypt 的參數則是代表接受來的資料，Decrypt 方法必須使用程式碼加以解密以還原資料。

```
 TSimpleEncryptor = class
 protected


 public
  function Encrypt(const Data: TBytes): TBytes;
  function Decrypt(const Data: TBytes): TBytes;
  constructor Create;
 end;
```

下面是這兩個方法的實作程式碼，讀者可以看到這兩個方法的實作程式碼是樣的，它們都接受的參數以'DexterHighlanderTiburonWeaver'這個鍵值字串進行 xor 的運算:

```
const
 EncryptKey = 'DexterHighlanderTiburonWeaver';


constructor TSimpleEncryptor.Create;
```

```
begin
  inherited Create;
end;

function TSimpleEncryptor.Decrypt(const Data: TBytes): TBytes;
var
  i: Integer;
  idx: Integer;

begin
  Result := Data;
  idx := 0;
  for i := 0 to Length(Data) - 1 do
  begin
    Result[i] := Byte(Chr(Ord(Data[i]) xor Ord(EncryptKey[idx])));
    Inc(idx);
    if (idx > Length(EncryptKey)) then
      idx := 0;
  end;
end;

function TSimpleEncryptor.Encrypt(const Data: TBytes): TBytes;
var
  i: Integer;
  idx: Integer;

begin
  Result := Data;
  idx := 0;
  for i := 0 to Length(Data) - 1 do
  begin
    Result[i] := Byte(Chr(Ord(Data[i]) xor Ord(EncryptKey[idx])));
    Inc(idx);
    if (idx > Length(EncryptKey)) then
      idx := 0;
  end;
end;
```

下面則是 TTransportEncryptFilter 類別的實作程式碼，讀者可以看到在在建構函式中建立了 TSimpleEncryptor 物件，並且分別在 ProcessInput 虛擬方法中呼叫 TSimpleEncryptor 物件的 Encrypt 方法加密傳遞的資料並且在 ProcessOutput 虛擬方法中呼叫 TSimpleEncryptor 物件的 Decrypt 方法以解密資料:

```
function TTransportEncryptFilter.GetUserParameters: TDBXStringArray;
begin
  SetLength(Result, 1);
  Result[0] := EncryptKey;
end;

function TTransportEncryptFilter.GetParameters: TDBXStringArray;
begin
  SetLength(Result, 1);
  Result[0] := EncryptKey;
end;

function TTransportEncryptFilter.GetParameterValue
  (const ParamName: UnicodeString): UnicodeString;
begin
  FParameters.TryGetValue(ParamName, Result);

  if ( ParamName = EncryptKey ) and ( Result = '' ) then
     Result := '0';
end;

function TTransportEncryptFilter.SetParameterValue
  (const ParamName, ParamValue: UnicodeString): Boolean;
begin
  FParameters.AddOrSetValue(ParamName, ParamValue);
  Result := True;
end;



constructor TTransportEncryptFilter.Create;
begin
  inherited Create;
```

```
  FParameters := TDictionary<String, String>.Create;
  FEncrypt := TSimpleEncryptor.Create;
end;

destructor TTransportEncryptFilter.Destroy;
begin
  FreeAndNil(FParameters);
  FreeAndNil(FEncrypt);
  inherited Destroy;
end;

function TTransportEncryptFilter.ProcessInput(const Data: TBytes): TBytes;
begin
  OutputDebugString(PWideChar('Encrypted - ' + Stringof(Data)));
  Result := FEncrypt.Encrypt(Data);
end;

function TTransportEncryptFilter.ProcessOutput(const Data: TBytes): TBytes;
begin
  OutputDebugString(PWideChar('Decrypted - ' + Stringof(Data)));
  Result := FEncrypt.Decrypt(Data);
end;

function TTransportEncryptFilter.Id: UnicodeString;
begin
  Result := EncryptFilterName;
end;
```

最後我們需要在 **initialization** 部份註冊這個客製化過濾器並且在 **finalization** 部份解除註冊客製化過濾器:

```
initialization

TTransportFilterFactory.RegisterFilter(EncryptFilterName,
  TTransportEncryptFilter);

finalization
```

```
TTransportFilterFactory.UnregisterFilter(EncryptFilterName);
```

## 使用客製化過濾器

OK，回到範例伺服器，開啟 ServerContainer 程式單元並且在它的 OnCreate 事件處理函式中使用 TDSTCPServerTransport 的 AddFilter 方法加入我們的客製化過濾器:

```
procedure TServerContainer1.DataModuleCreate(Sender: TObject);
var
  i : Integer;
begin
  i := DSTCPServerTransport1.Filters.AddFilter(uEncryptFilter.EncryptFilterName);
  for i := 0 to DSTCPServerTransport1.Filters.Count - 1 do
    Form1.lbFilters.Items.Add(DSTCPServerTransport1.Filters.GetFilter(i).Id);
end;
```

當然我們也需要在 ServerContainer 程式單元的 uses 句子中加入包含客製化過濾器的程式單元 uEncryptFilter:

```
implementation

uses Windows, ServerMethodsUnit1, MainForm, uEncryptFilter;
```

現在執行 DataSnap 伺服器，我們就可以看到伺服器顯示它已經找到了我們的客製化過濾器:

版權所有 請勿翻印

圖 7 範例客製化 DataSnap 過濾器伺服器

接著開啟用戶端應用程式，在主表單中也加入客製化過濾器的程式單元 uEncryptFilter，編譯並且執行用戶端應用程式，再使用 TCP Viewer 觀察傳遞的資料，我們果然看到資料現在都經過加密了：

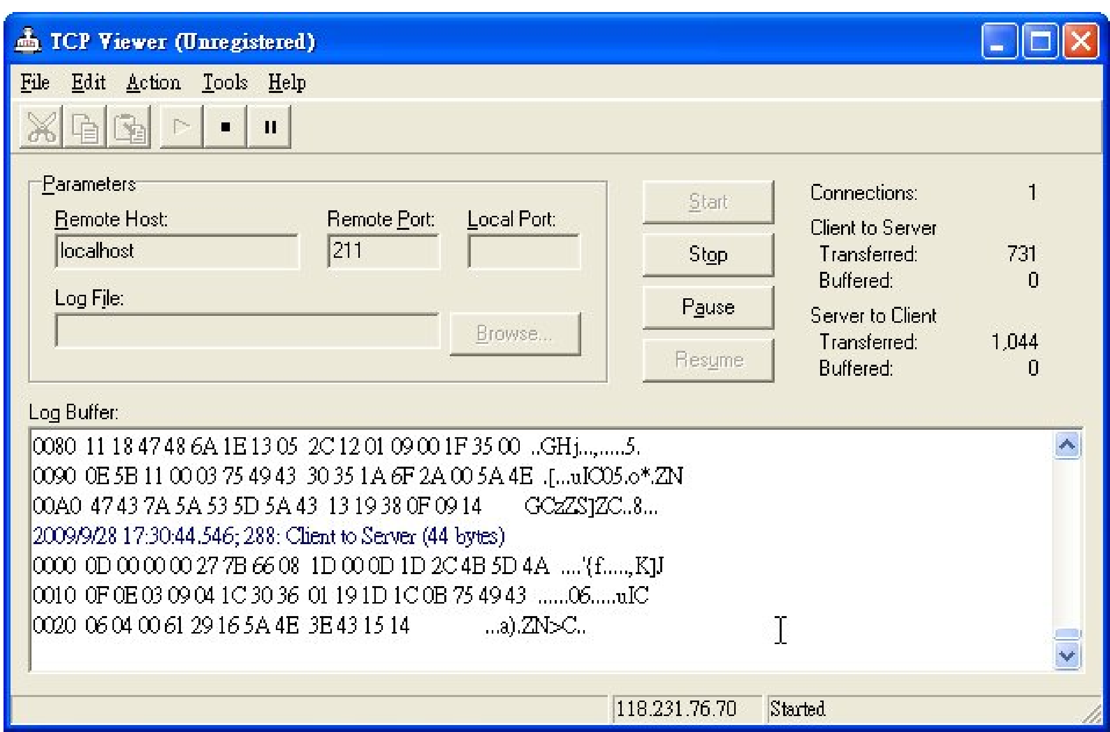


圖 8 使用 TCP Viewer 觀察的結果

我們可以看到傳遞的資料量增加了，但是用戶端仍然可以正確的接受到資料:

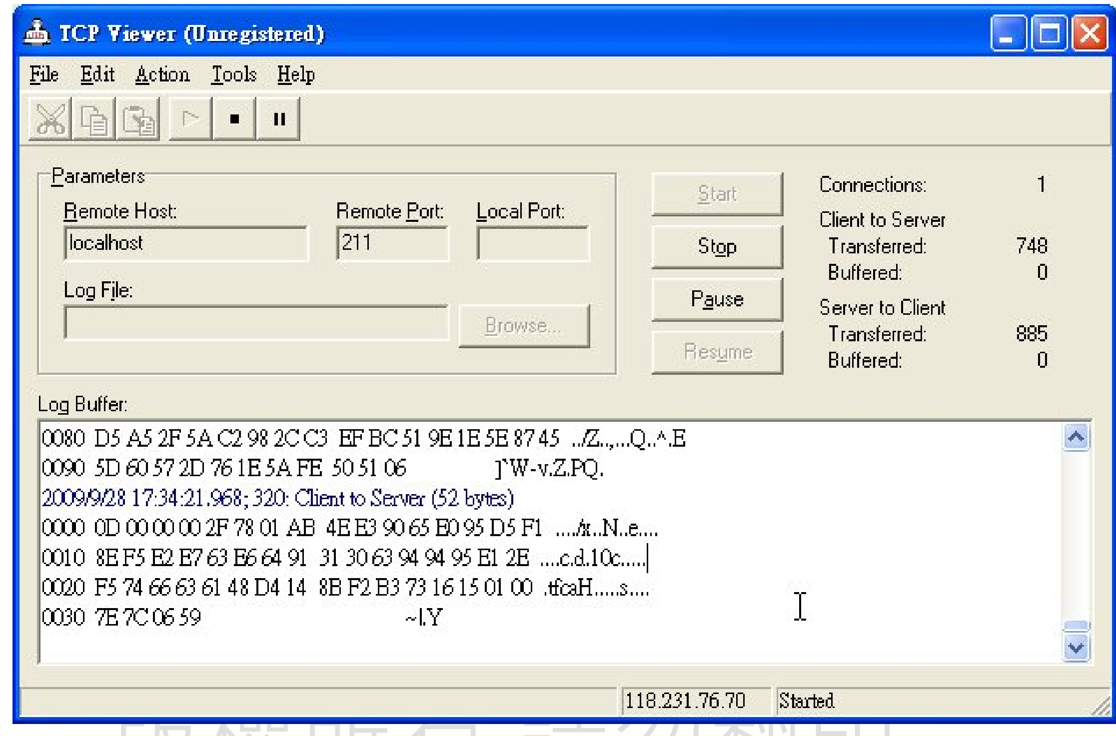
圖 9 範例用戶端應用程式

當然我們也可以同時使用兩個過濾器，享受加密又壓縮的好處，下圖是伺服器同時支援了兩個過濾器:

版權所有 請勿翻印

圖 10 範例客製化 DataSnap 過濾器伺服器

如果我們再次使用 TCP Viewer，就可以看到下圖，享受加密又壓縮的好處，因為資料加密了而且資料傳遞量又減少了。




版權所有 請勿翻印

圖 11 使用 TCP Viewer 觀察的結果

現在您應該瞭解了如何使用 DataSnap 的過濾器功能以及如何開發客製化過濾器，接著讓我們再討論 DataSnap 10.3 中新增的請求過濾器(Request Filter)。

## 5-3 使用 DataSnap 的請求過濾器

所謂的請求過濾器是 DataSnap 在 10.3 版時加入的新型態的過濾器，它主要的目的是在用戶端呼叫伺服端取得資料時，在請求的 URL 中加入特定的過濾器以便過濾伺服端回傳的資料，例如用戶端可以使用請求過濾器讓伺服端只回傳部份的資料，或是特定範圍的資料，以減少不必要的資料佔據網路頻寬。

由於請求過濾器是使用在請求的 URL 中，因此 DataSnap 伺服器必須支援 HTTP/HTTPS 通訊協定，或著是 DataSnap REST 型態的伺服器。那用戶端呢? 讀者可能會想由於用戶端是藉由 URL 使用請求過濾器，因此用戶端必須是瀏覽器型態的用戶端，對嗎? 嗯原則上是正確的，因為瀏覽器可以直接使用 URL 向伺服器請求服務，但由於 DataSnap 10.3 加入了新的元件 TDSRestConnection，因此即使是一般的 Windows 用戶端也可以藉由 URL

使用請求過濾器，稍後我們會說明如何使用 TDSRestConnection 和請求過濾器，在那之前，我們需要先解釋什麼是請求過濾器。

## 5-3-1 請求過濾器的種類

DataSnap 10.3 提供了兩個基本的請求過濾器:

| 請求過濾器 | 說明 |
|---|---|
| SubString (ss) | SubString 請求過濾器允許用戶端擷取部份由伺服端回傳的字串(String)或是串列流(Stream)。例如如果用戶端只需要擷取伺服端回傳的字串的前 10 個字元，或是從位置第 20 到第 50 個的串列流位元組。 |
| Table (t) | 和 SubString 非常的類似，但 Table 請求過濾器允許用戶端擷取由伺服端回傳的部份資料集中的資料，例如從第 15 筆到第 20 筆的資料。 |

SubString 和 Table 請求過濾器都提供了三個函式讓用戶端控制如何擷取資料，下面的表格說明了 SubString 請求過濾器的函式:

版權所有 請勿翻印

| SubString (ss) 請求過濾器函式 | 說明 |
|---|---|
| count 函式(c) | count 可控制從伺服端回傳到用戶端的字元數或是串列流的位元組。count 函式接受一個參數，這個參數即是回傳的字元數或是位元組，而伺服端回傳的字元或是位元組的起始值是從 0 開始。例如如果用戶端只需要擷取伺服端回傳的前 10 個字元，那麼我們可以使用如下的格式:<br>ss.c=10<br>其中 ss 就是 SubString 請求過濾器<br>c 即指 count 函式<br>10 即是 count 函式的參數值，代表擷取前 10 個字元<br>例如下面的 URL<br>http://localhost:8080/datasnap/rest/TServerMethods1/GetDescription?ss.c=10<br>即是呼叫 GetDescription 方法並且只擷取回傳字串的前 10 個字元 |
| offset 函式(o) | Offset 函式可跳過指定數目的字元或是位元組，只擷取隨後的資料。offset 函式接受一個參數，這個參數即是需要跳過的字元數或是位元組。例如如果用戶端不需要伺服器回傳字串的前 10 個字元或是串列流的前 10 個位元組，那麼我們可以使用如下的格式: |

| | |
|---|---|
| | ss.o=10<br>其中 ss 就是 SubString 請求過濾器<br>o 即指 offset 函式<br>10 即是 offset 函式的參數值，代表跳過前 10 個字元<br>例如下面的 URL<br>http://localhost:8080/datasnap/rest/TServerMethods1/GetDescription?ss.o=10<br>即是呼叫 GetDescription 方法並且只擷取第 10 個字元之後的字串資料 |
| range 函式(r) | range 函式接受兩個參數值，第一個參數值是指要從那一個位移位置開始擷取資料，第二個參數值是指要擷取幾個字元數或是位元組。例如，如果用戶端只需要從第 10 個開始的字元並且存取其後的 5 個字元，那麼我們可以使用如下的格式:<br>ss.r=10, 5<br>其中 ss 就是 SubString 請求過濾器<br>r 即指 range 函式<br>10 即是 range 函式的第一個參數值，代表從第 10 個字元開始擷取資料，5 即是 range 函式的第 2 個參數值，代表從第 10 個字元開始擷取 5 個字元的資料，<br>例如下面的 URL<br>http://localhost:8080/datasnap/rest/TServerMethods1/GetDescription?ss.r=10,5<br>即是呼叫 GetDescription 方法並且只擷取第 10 個字元之後的 5 個字元的資料 |

版權所有 請勿翻印

下面的表格說明了 Table 請求過濾器的函式，它的使用方法和 SubString 非常的類似:

| Table (t) 請求過濾器函式 | 說明 |
|---|---|
| count 函式 | count 可控制從伺服端回傳到用戶端的記錄數。count 函式接受一個參數，這個參數即是回傳記錄數，而伺服端回傳的記錄數起始值是從 0 開始。例如如果用戶端只需要擷取伺服端回傳的前 10 筆資料，那麼我們可以使用如下的格式:<br>t.c=10<br>其中 t 就是 Table 請求過濾器<br>c 即指 count 函式<br>10 即是 count 函式的參數值，代表擷取前 10 筆資料<br>例如下面的 URL |

| | |
|---|---|
| | http://localhost:8080/datasnap/rest/TServerMethods1/GetEmployee?t.c=10<br>即是呼叫 GetEmployee 方法並且只擷取回傳 Employee 資料表中的的前 10 筆資料 |
| offset 函式 | Offset 函式可跳過指定數目記錄，只擷取隨後的資料。offset 函式接受一個參數，這個參數即是需要跳過的記錄數。例如如果用戶端不需要伺服器回傳字串的前 10 筆資料，那麼我們可以使用如下的格式:<br>t.o=10<br>其中 t 就是 Table 請求過濾器<br>o 即指 offset 函式<br>10 即是 offset 函式的參數值，代表跳過前 10 筆資料<br>例如下面的 URL<br>http://localhost:8080/datasnap/rest/TServerMethods1/GetEmployee?t.o=10<br>即是呼叫 GetEmployee 方法並且只擷取第 10 個字元之後的字串資料 |
| Range 函式 | range 函式接受兩個參數值，第一個參數值是指要從那一個位移位置開始擷取資料，第二個參數值是指要擷取幾筆資料。例如，如果用戶端只需要從第 10 筆開始的資料並且存取其後的 5 筆資料，那麼我們可以使用如下的格式:<br>t.r=10, 5<br>其中 t 就是 Table 請求過濾器<br>r 即指 range 函式<br>10 即是 range 函式的第一個參數值，代表從第 10 筆資料開始擷取資料，<br>5 即是 range 函式的第 2 個參數值，代表從第 10 筆資料後開始擷取 5 筆資料，<br>例如下面的 URL<br>http://localhost:8080/datasnap/rest/TServerMethods1/GetEmployee?t.r=10,5<br>即是呼叫 GetEmployee 方法並且只擷取第 10 筆資料之後的 5 筆資料 |

版權所有 請勿翻印

## 5-3-2 使用請求過濾器

在前一小節說明 SubString 和 Table 請求過濾器的表格中已經列出了如何使用它們的範例，例如下圖就是在瀏覽器中使用 Table 請求過濾器:

http://localhost:8080/datasnap/rest/TServerMethods1/GetEmployee?t.r=10,5

的結果:

localhost:8080/dat...

localhost:8080/datasnap/rest/TServerMethods1/GetEmployee?t.r=10,5

{"result":[{"table":[["EMP_NO",5,0,0,0,2,0,0,false,false,0,false,false],["FIRST_NAME",1,1,0,15,16,0,0,false,false,0,false,false],["LAST_NAME",1,2,0,20,21,0,0,false,false,0,false,false],
["PHONE_EXT",1,3,0,4,5,0,0,false,false,0,false,false],["HIRE_DATE",24,4,0,0,16,0,0,false,false,0,false,false],["DEPT_NO",1,5,0,3,4,0,0,false,false,0,false,false],
["JOB_CODE",1,6,0,5,6,0,0,false,false,0,false,false],["JOB_GRADE",5,7,0,0,2,0,0,false,false,0,false,false],["JOB_COUNTRY",1,8,0,15,16,0,0,false,false,0,false,false],
["SALARY",8,9,0,2,34,10,0,false,false,0,false,false],["FULL_NAME",1,10,0,37,38,0,0,false,false,0,false,false]],"EMP_NO":[24,28,29,34,36],"FIRST_NAME":
["Pete","Ann","Roger","Janet","Roger"],"LAST_NAME":["Fisher","Bennet","De Souza","Baldwin","Reeves"],"PHONE_EXT":["888","5","288","2","6"],"HIRE_DATE":["1990-09-12.0","1991-02-01.0","19
18.0","1991-03-21.0","1991-04-25.0"],"DEPT_NO":["671","120","623","110","120"],"JOB_CODE":["Eng","Admin","Eng","Sales","Sales"],"JOB_GRADE":[3,5,3,3,3],"JOB_COUNTRY":
["USA","England","USA","USA","England"],"SALARY":["81810.19","22935","69482.63","61637.81","33620.63"],"FULL_NAME":["Fisher, Pete","Bennet, Ann","De Souza, Roger","Baldwin, Janet","Re
Roger"]}]}

圖 12 在瀏覽器中使用 Table 請求過濾器

但是除了瀏覽器之外，對於原生 Windows 應用程式的用戶端來說請求過濾器也是非常有用的功能，因為請求過濾器可以在伺服端就過濾用戶端需要的資料，如此一來就可以減少網路傳遞的資料量，但問題是如何在原生 Windows 應用程式中使用請求過濾器呢? 答案就是使用 TDSRestConnection 元件並且藉由它來產生用戶端的 Proxy 程式碼，再藉由它產生的用戶端的 Proxy 程式碼使用請求過濾器。下面的小節即說明了如何使用 TDSRestConnection 元件和請求過濾器。

## 藉由 TDSRestConnection 元使用請求過濾器

首先建立一個 DataSnap REST Application 專案，在 ServerMethodUnit1 程式單元中使用 dbExpress 元件連結 InterBase 的 Employee 資料表:

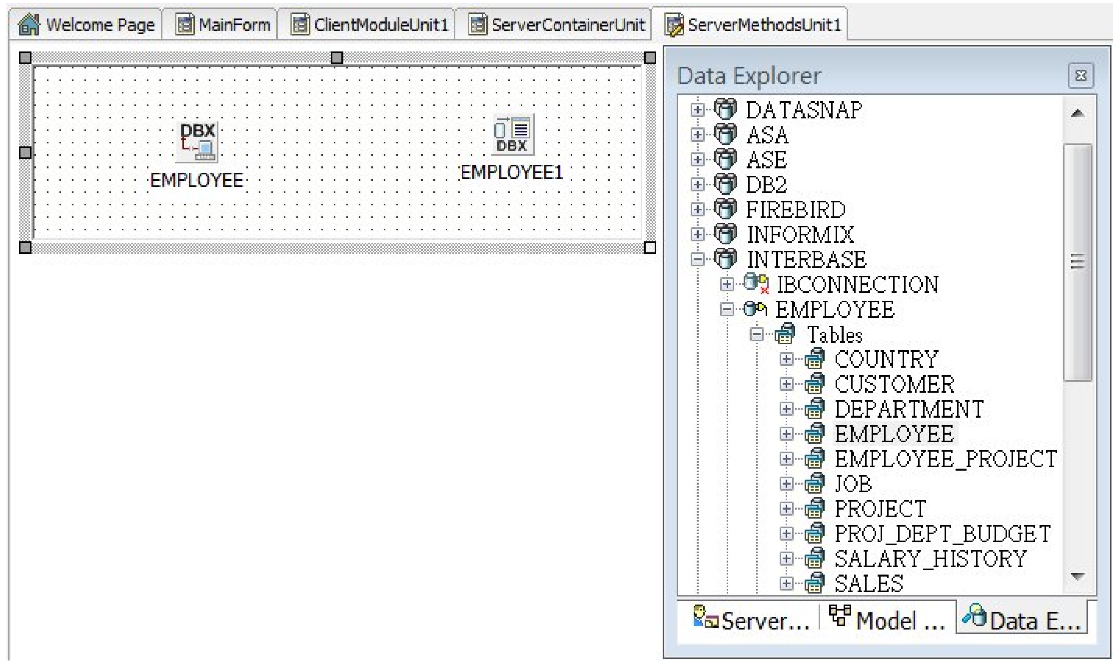



圖 13 在 ServerMethodUnit1 程式單元中使用 dbExpress 元件連結 InterBase 的 Employee 資料表

開啟 ServerMethodUnit1 程式單元的程式碼，並且加入兩個範例方法:

```
public
   function GetDescription : string;
   function GetEmployee : TDataSet;
 end;
```

並且實作這兩個範例方法如下:

```
function TServerMethods1.GetDescription: string;
```

```
begin
  Result := '讓您一次掌握 Embarcadero 最尖端的產品和技術' +
           '改變軟體架構和使用的革新性產品 AppWave' +
           '萬眾矚目和期待的視窗原生開發工具新王者 Delphi 10.3 預覽版';
end;

function TServerMethods1.GetEmployee: TDataSet;
begin
  EMPLOYEE.Connected := True;
  EMPLOYEE1.Active := True;
  Result := EMPLOYEE1;
end;
```

GetDescription 回傳字串的資料以準備稍後使用 SubString 請求過濾器來測試，而 GetEmployee 則是回傳 Employee 資料表中的資料，以準備稍後使用 Table 請求過濾器來測試。

現在編譯並且執行此範例 DataSnap REST 伺服器。

接著在專案群組再建立一個 VCL Form Application 專案，再於其中建立一個 DataSnap Client Module，DataSnap Client Module 會產生一個 TSQLConnection 元件連結 DataSnap REST 伺服器並且產生使用 TSQLConnection 元件連結伺服器封裝的用戶端類別，現在在此 DataSnap Client Module 中加入一個 TDSRestConnection 元件，然後使用滑鼠右鍵選擇 Generate DataSnap client classes，如下圖所示:

版權所有 請勿翻印

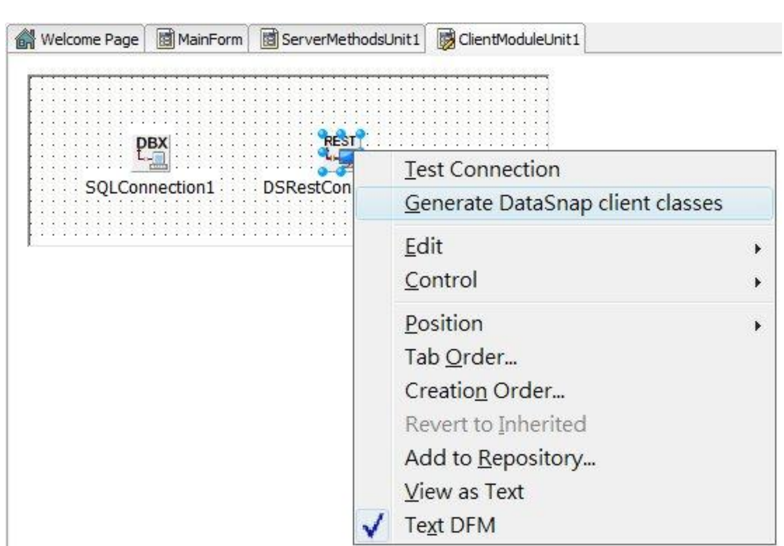



圖 14 使用 TDSRestConnection 產生用戶端 Proxy 類別程式碼

TDSRestConnection 此時便會產生另外一個封裝伺服器服務的用戶端類別程式碼，這個新產生的類別程式碼和 TSQLConnection 元件產生的類別程式碼的差別在於 TDSRestConnection 元件產生的類別程式碼是使用 REST 呼叫方式來呼叫伺服端的服務，這也就是說這個類別程式碼會藉由 REST 的 URL 呼叫伺服端，因此我們也就可以在其中使用請求過濾器。

開啟由 TDSRestConnection 元件產生的類別程式碼，我們可以在其中找到 GetDescription 和 GetEmployee 的宣告原型:

```
function GetDescription(const ARequestFilter: string = ''): string;
function GetEmployee(const ARequestFilter: string = ''): TDataSet;
```

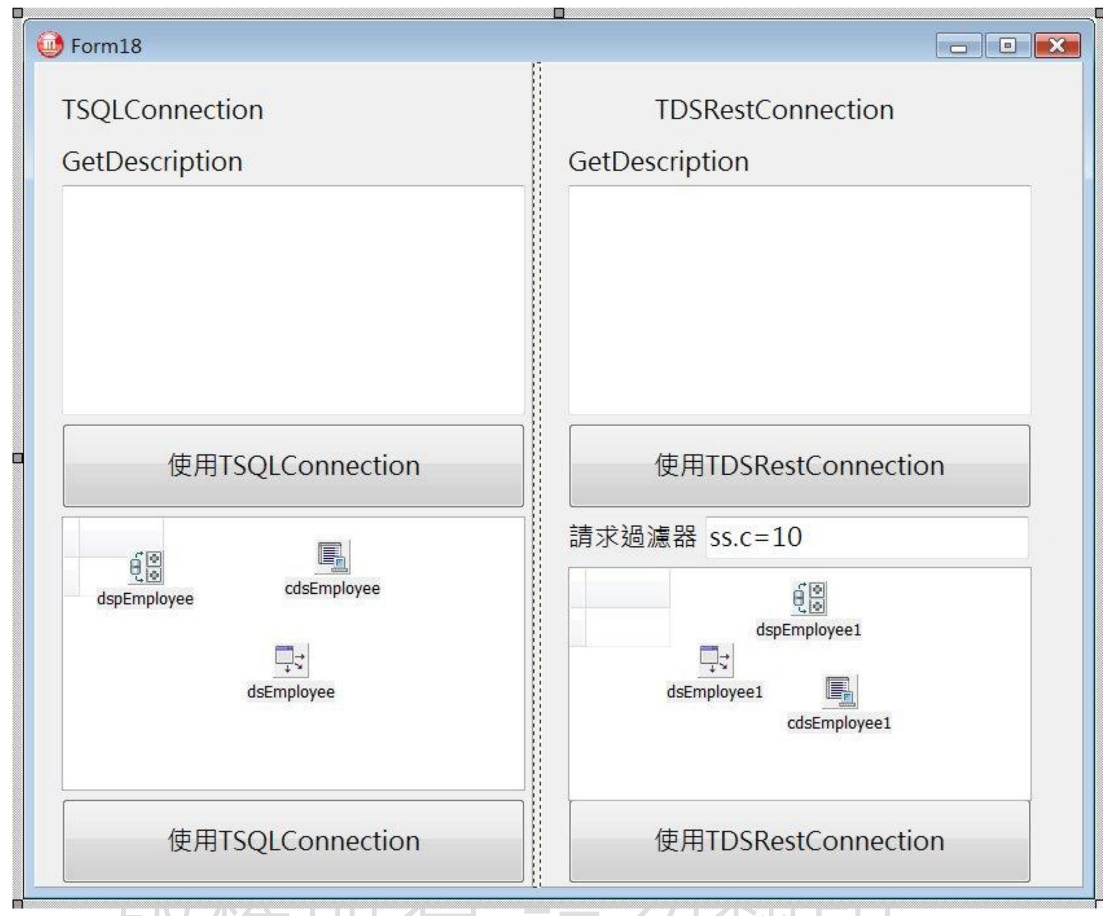
從上面的程式碼中可以看到我們可以藉由參數的方式把請求過濾器傳遞給上述的函式，再由這些函式自動產生正確的 URL 並且向伺服端發出請求。反觀如果我們開啟由 TSQLConnection 產生的類別程式碼，我們可以看到下面的原型宣告，由 TSQLConnection 產生的類別程式碼是無法使用請求過濾器的:

```
function GetDescription: string;
function GetEmployee: TDataSet;
```

這個原因當然是因為 TSQLConnection 是使用 dbExpress 呼叫伺服器的服務，而不是使用 REST 呼叫慣例。

版權所有 請勿翻印

OK，現在我們就可以開始測試由 TSQLConnection 產生的類別程式碼以及由 TDSRestConnection 元件產生的類別程式碼，現在讓我們設計範例用戶端應用程式的主表格如下:




版權所有 請勿翻印

圖 15 範例用戶端主表單

接著使用下面的程式碼藉由 TSQLConnection 元件呼叫伺服端的 GetDescription 方法:

```
procedure TForm18.Button1Click(Sender: TObject);
var
  aServer : TServerMethods2Client;
begin
  aServer := ClientModule1.ServerMethods2Client;
  mmNoRF.Lines.Text := aServer.GetDescription;
end;
```

再使用下面的程式碼藉由 TDSRestConnection 元件呼叫伺服端的 GetDescription 方法，由於藉由 TDSRestConnection 可使用請求過濾器，因此我們在呼叫 GetDescription 時把主表格中 TEdit 元件的 Text 特性值傳入做為請求過濾器:

```
procedure TForm18.Button2Click(Sender: TObject);
var
  aRSServer : TServerMethods1Client;
```

```
begin
  aRSServer := TServerMethods1Client.Create(ClientModule1.DSRestConnection1);
  try
    mmRF.Lines.Text := aRSServer.GetDescription(edtFilter.Text);
  finally
    aRSServer.Free;
  end;
end;
```

編譯並且執行範例用戶端應用程式，從下圖中我們可以看到使用 TSQLConnection 呼叫伺服器的 GetDescription 取得了所有的字串內容，但使用 TDSRestConnection 呼叫伺服器時則可以使用請求過濾器，在這裡我們使用了 ss.c=10 代表只存取前 10 個字元，從執行結果來看執行結果果然是正確的，伺服器只會傳遞 10 個字元到用戶端而不會傳遞整個字串。

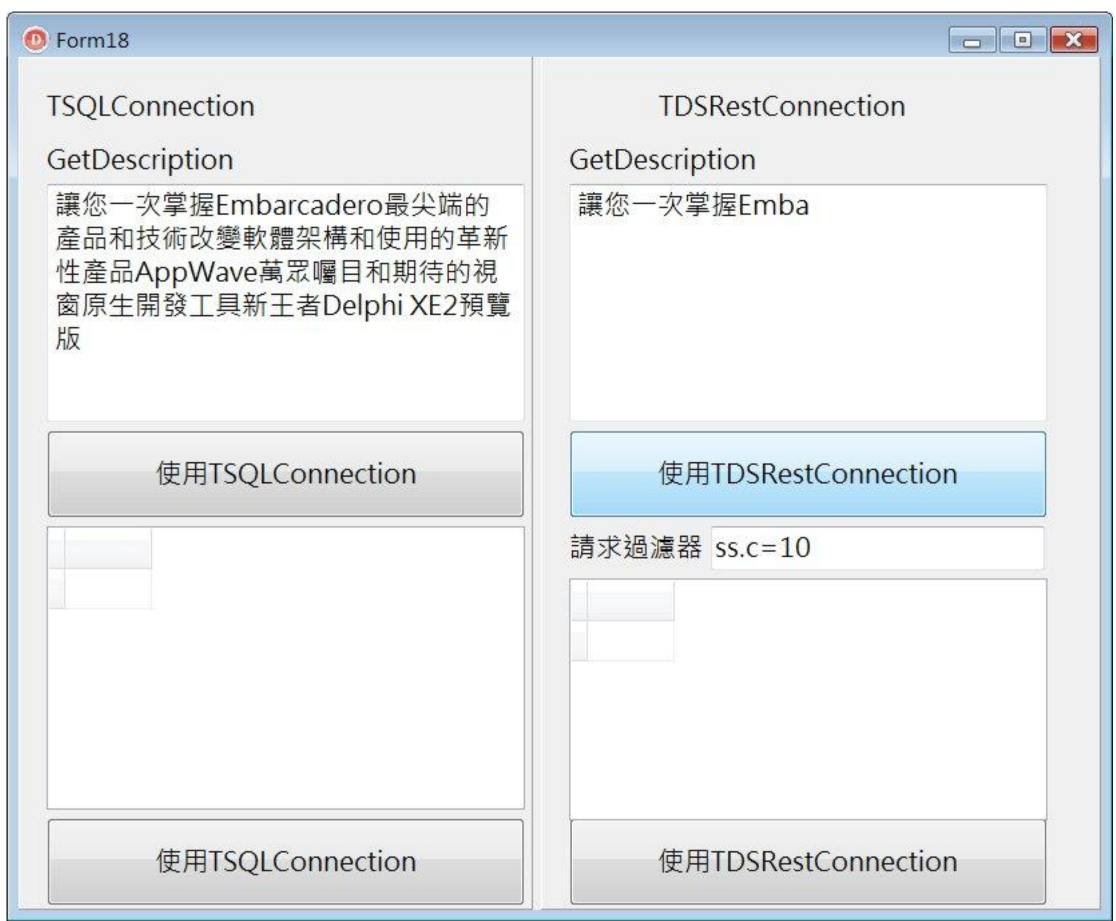



圖 16 使用 SubString 請求過濾器的執行結果

再讓我們呼叫 GetEmployee 存取資料表的資料看看請求過濾器的效果。同樣的先讓我們使用 TSQLConnection 呼叫伺服器的 GetEmployee 方法:

```
procedure TForm18.Button3Click(Sender: TObject);
```

```
var
  aServer : TServerMethods2Client;
  aDataSet : TDataSet;
begin
  aServer := ClientModule1.ServerMethods2Client;
  cdsEmployee.Active := False;
  aDataSet := aServer.GetEmployee;
  dspEmployee.DataSet := aDataSet;
  cdsEmployee.Active := True;
end;
```

再使用 TDSRestConnection 呼叫伺服器的 GetEmployee 方法，並且傳入使用者在主表單中使用的請求過濾器:

```
procedure TForm18.Button4Click(Sender: TObject);
var
  aRSServer : TServerMethods1Client;
  aDataSet : TDataSet;
begin
  aRSServer := TServerMethods1Client.Create(ClientModule1.DSRestConnection1);
  try
    cdsEmployee.Active := False;
    aDataSet := aRSServer.GetEmployee(edtFilter.Text);
    dspEmployee.DataSet := aDataSet;
    cdsEmployee.Active := True;
  finally
    aRSServer.Free;
  end;
end;
```

編譯並且執行範例用戶端應用程式，這次讓我們使用請求過濾器 t.c=3，代表只存取 Employee 資料表前 3 筆的資料：

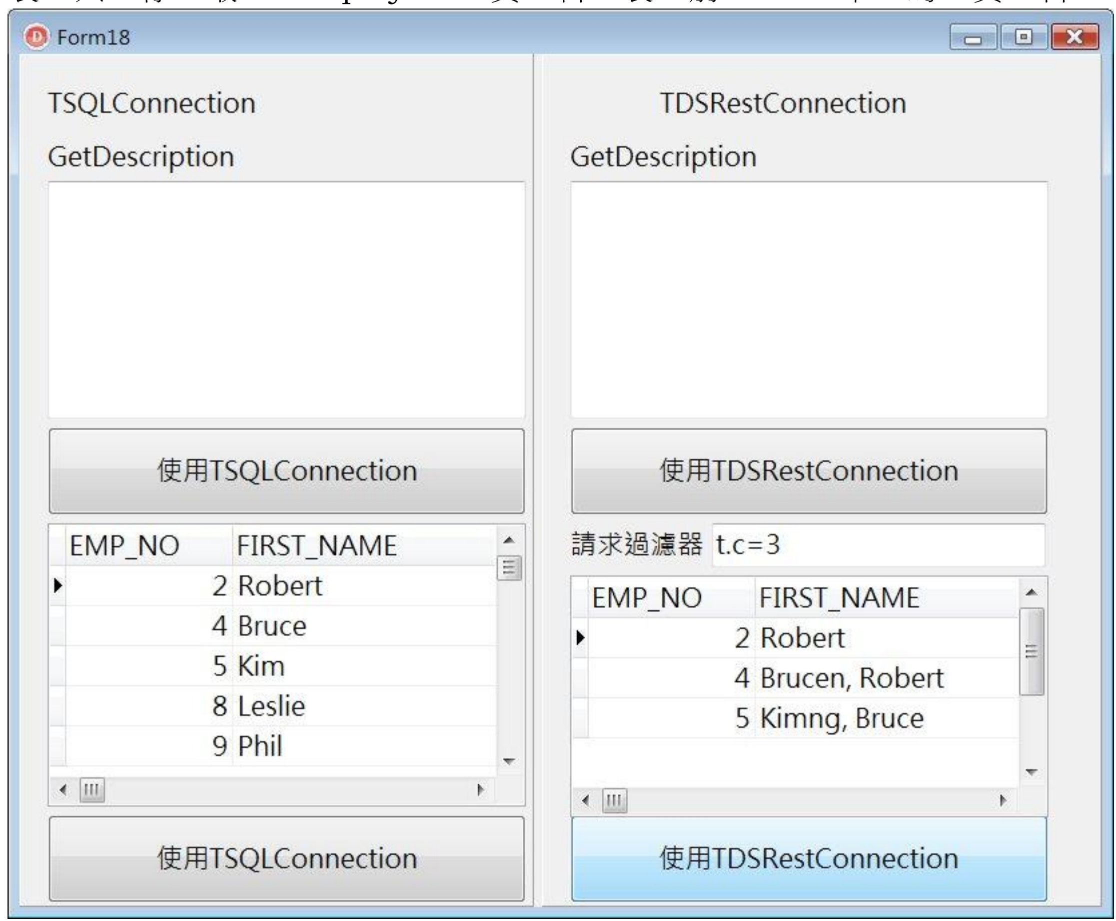



圖 17 使用 Table 請求過濾器的執行結果

從上面的執行結果可以看到使用 TSQLConnection 果然取得了 Employee 資料表所有的資料，而右方使用 t.c=3 請求過濾器的也的確只存取 3 筆資料，伺服端的確只傳遞前 3 筆的資料回用戶端。

## 5-4 結論

DataSnap 的過濾器功能允許開發人員控制傳遞 JSON 資料的實際格式，開發人員可以進行加/解密以保護以文字形式傳遞資料的 JSON 格式，而 DataSnap 10.3 新加入的請求過濾器則允許開發人員在用戶端即可控制伺服端需要傳遞到用戶端的資料條件，以消除不必要的資料佔據網路頻寬，或是在用戶端即可使用 URL 來客製化伺服端傳遞到用戶端的資料。當然，開發人員更可以結合 DataSnap 的過濾器和請求過濾器功能更進一步控制傳遞的資料，DataSnap 的過濾器是個非常實用的功能。

# 第6章 DataSnap生命週期和管理功能

在前面討論的章節中建立的 DataSnap 伺服端的服務類別都是使用 Server 生命週期型態的伺服器，除了 Server 型態之外，還有 Session 型態以及 Invocation 型態。每一種不同的 DataSnap 伺服端的服務型態都擁有不同的特性，本章將討論 DataSnap 伺服端服務的生命週期的意義。

版權所有 請勿翻印

## 6-1 DataSnap 伺服端服務的生命週期

從 DataSnap XE 之後便允許開發人員建立不同生命週期的 DataSnap 伺服端的服務，在 10.3 中提供了三種不同的生命週期，開發人員可以在 TDSServerClass 元件的 LifeCycle 特性中設定，下面的表格說明了每一種生命週期的意義:

| 特性值 | 說明 |
| --- | --- |
| Server | 在整個 DataSnap 伺服器中只會建立一個服務類別物件以服務所有的用戶端，只有當 DataSnap 伺服器結束時才會釋放此服務類別物件 |
| Session | 在 DataSnap 伺服器中會為每一個連結的用戶端建立一個專屬的服務類別物件服務此用戶端，一旦用戶端結束或是關閉 TSQLConnection 的連結，此服務類別物件便會被釋放 |
| Invocation | 在 DataSnap 伺服器中每當用戶端執行一次請求時，在 DataSnap 伺服器便會為這個請求建立一個服務類別物件服務此用戶端請 |

| | 求，當請求執行結束後，DataSnap 伺服器便會釋放此服務類別物件 |
|---|---|

從上面表格的說明我們可以瞭解，使用 Server 生命週期的伺服端服務類別只會在 DataSnap 伺服器中建立一個服務物件，使用 Session 生命週期的伺服端服務類別則視用戶端使用使用多少 TSQLConnection 元件藉由 DataSnap 驅動程式連結到 DataSnap 伺服器的數目而在 DataSnap 伺服器中建立相對應的服務物件來服務服務，最後使用 Invocation 生命週期的伺服端服務類別則會在每一次用戶端呼叫 DataSnap 伺服器時被建立來服務用戶端，因此被建立和釋放的次數相當巨量。

那麼開發人員應該如何決定使用那一種的生命週期伺服端服務類別呢?這當然時要看伺服端服務類別提供的服務種類，下面的表格簡單的說明了每一種生命週期適用的場景:

| 生命週期 | 說明 |
|---|---|
| Server | 提供所有用戶端公用的服務，由於所有用戶端都使用單一的服端服務類別物件，因此使用這種生命週期的服務物件負荷都比較大，使用這種生命週期的服務物件適合提供快速，簡單，無狀態的服務為主。 |
| Session | 由於這種生命週期型態的服端服務類別物件為會每一個用戶端的連結建立一個專屬的服務物件，因此可提供用戶端無狀態以及有狀態的服務，也可提供長期，負荷較大的服務。 |
| Invocation | 這種生命週期型態的服端服務類別物件只存在於每一個用戶端的呼叫周期，因此適合提供可在背景執行的服務，或是執行資料庫的預儲程序，或是批次處理等和用戶端較無相關的服務。不過由於使用這種生命週期的服務類別會被頻繁的建立和釋放，因此這種服務類別應該儘量精簡，如果需要使用資料庫，那麼也應該搭配使用 dbExpress 的連結池功能以加快服務速度。 |

版權所有 請勿翻印

## 6-1-1 Server 生命週期

由於使用 Server 生命週期的 DataSnap 伺服端物件只有一個並且服務所有的用戶端請求，因此這種型態的伺服端服務類別物件負荷可能很重，開發人員應

該儘量讓每一個用戶端的請求快速完成，以便服務更多的用戶端請求並且減少伺服端服務類別物件的負荷，要設定特定的服務類別為 Server 生命週期，開發人員只需如下圖在物件檢視器中設定 TDSServerClass 元件的 LifeCycle 特性值為 Server 即可:

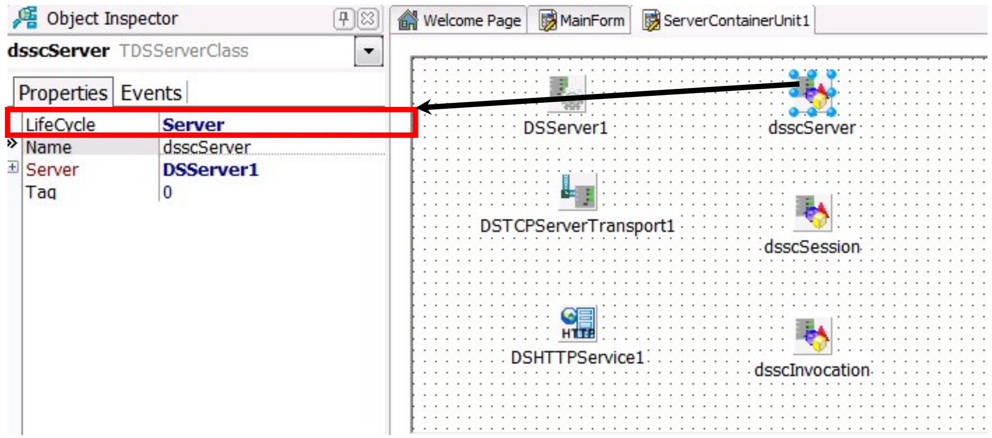



圖 6-1 設定 TDSServerClass 元件的 LifeCycle 特性值為 Server 生命週期

那麼由這個 TDSServerClass 元件輸出的服務類別就會自動使用 Server 生命週期，例如上圖的 dssServer 元件使用了下面的程式碼輸出服務類別 TdssmServer:

版權所有 請勿翻印

```
procedure TServerContainer1.dsscServerGetClass(
  DSServerClass: TDSServerClass; var PersistentClass: TPersistentClass);
begin
  PersistentClass := ServerMethodsUnit1.TdssmServer;
end;
```

因此 TdssmServer 類別就使用了 Server 生命週期，而在 TdssmServer 類別中輸出了兩個方法可讓用戶端查詢郵遞區號，或是使用郵遞區號查詢區域名稱:

```
    function GetZipCode(const SDistrict : String) : Integer;
    function GetDistrictFromZipCode(sZipCode : String) : String;
```

為了加快執行速度以便在更短的時間內服務更多的用戶端請求，因此 TdssmServer 服務類別使用了記憶體資料表來服務用戶端:

```
function TdssmServer.CreateZipCodeTable: TClientDataSet;
begin
  Result := TClientDataSet.Create(nil);

  with Result do
  begin
```

```
    with FieldDefs.AddFieldDef do
    begin
      DataType := ftString;
      Size := 20;
      Name := '地區名';
    end;
    with FieldDefs.AddFieldDef do
    begin
      DataType := ftInteger;
      Name := '郵遞區號';
    end;
    with IndexDefs.AddIndexDef do
    begin
      Fields := '地區名';
      Name := 'idxTD';
    end;
    CreateDataSet;
    IndexDefs.Update;
    IndexName := 'idxTD';
  end;
end;

function TdssmServer.GetDistrictFromZipCode(sZipCode : String) : String;
begin
  Result := cdsZipCode.Lookup('郵遞區號', sZipCode, '地區名');
end;

function TdssmServer.GetZipCode(const SDistrict : String) : Integer;
begin
  Result := cdsZipCode.Lookup('地區名', sDistrict, '郵遞區號');
end;
```

從下圖的範例 DataSnap 應用系統可以看到範例 TdssmServer 服務物件在 DataSnap 伺服器中只被建立了一個物件，這當然是因為使用了 Server 生命週期，而且在不到 1 秒的時間就完成了用戶端的請求，因此 TdssmServer 是很好的使用 Server 生命週期的範例。

圖 6-2 TdssmServer 服務物件快速的完成用戶端的請求

## 6-1-2 Session 生命週期

Session 生命週期是 TDSServerClass 元件內定的生命週期型態，每一個使用 TSQLConnection 元件藉由 DataSnap 驅動程式連結到 DataSnap 伺服器的用戶端都會建立一個專屬的服務物件，因此不同的用戶端並不會相互干擾，要設定特定的服務類別為 Session 生命週期，開發人員只需如下圖在物件檢視器中設定 TDSServerClass 元件的 LifeCycle 特性值為 Session 即可:

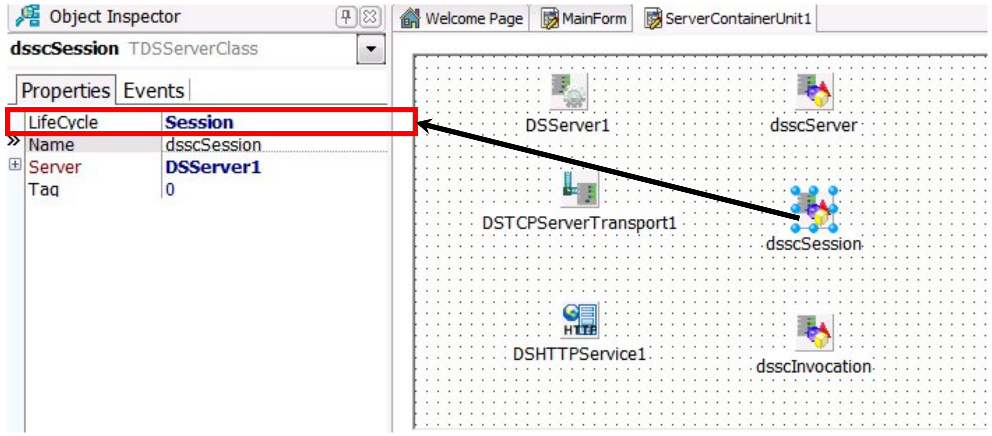



圖 6-3 設定 TdssmSession 元件的 LifeCycle 特性值為 Session 生命週期

由於上圖的 dssSession 元件輸出了 TdssmSession 類別，因此 TdssmSession 服務物件會為每一個用戶端建立一個專屬的物件:

```
procedure TServerContainer1.dsscSessionGetClass(DSServerClass: TDSServerClass;
  var PersistentClass: TPersistentClass);
begin
  PersistentClass := uServerModuleSession.TdssmSession;
end;
```

版權所有 請勿翻印

在 TdssmSession 類別中輸出了一個方法 GetThreadID 讓用戶端呼叫，在 GetThreadID 方法中使用了 dbExpress 元件連結到 InterBase 資料表，擷取其中的書名資訊並且結合伺服端的服務執行緒 ID 回傳給用戶端:

```
001   function TdssmSession.GetThreadID : String;
002   begin
003     DXEHPDBP.Connected := True;
004     try
005       cdsSessionQuery.Active := True;
006       cdsSessionQuery.MoveBy(Random(cdsSessionQuery.RecordCount));
007       Result := cdsSessionQuery.FieldByName('BOOKNAME').AsString + ' : ' +
IntToStr(TThread.CurrentThread.ThreadID);
008     finally
009       cdsSessionQuery.Active := False;
010       DXEHPDBP.Connected := False;
011     end;
012   end;
```

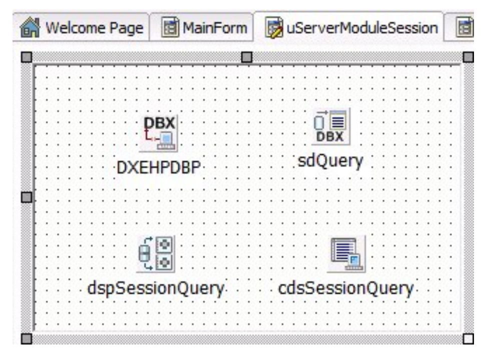



圖 6-4 uServerModuleSession 資料模組使用了 dbExpress 元件連結 InterBase 資料庫

如果現在我們執行數個這個範例 DataSnap 用戶端應用程式，那麼 DataSnap 伺服器會為每一個範例 DataSnap 用戶端應用程式建立一個專屬的 TdssmSession 物件服務用戶端。因此如果此時我們執行許多份的此範例 DataSnap 用戶端應用程式，例如 100 個用戶端應用程式，那麼 DataSnap 伺服器便會在伺服端建立 100 個 TdssmSession 物件，每一個 TdssmSession 物件又需要使用一個 TSQLConnection 連結到資料庫，那麼很快的所有資料庫連結都會被使用完畢而造成錯誤，而且 DataSnap 伺服器的負荷會非常的沈重，因此在使用 Session 或是稍後介紹的 Invocation 生命週期的服務物件如果會連結到資料庫的話，那麼筆者建議一定要開啟 dbExpress 的連結池功能，以便讓所有用戶端分享資料庫連結，而且在撰寫服務方法時，一定要在方法執行完畢之際關閉資料庫連結，以釋放資料庫連結回 dbExpress 的連結池讓其他方法或是其他用戶端重覆使用，例如在上面的 GetThreadID 方法中最後在 010 行關閉了 TSQLConnection 對於資料庫的連結以釋放 InterBase 的連結回 dbExpress 的連結池。

版權所有 請勿翻印

例如下圖是執行數個範例 DataSnap 用戶端在沒有開啟 dbExpress 連結池的情形下，DataSnap 伺服器開啟了數個和 InterBase 的資料庫連結，而且執行的速度大約為 0.15 秒服務每一次的用戶端請求:

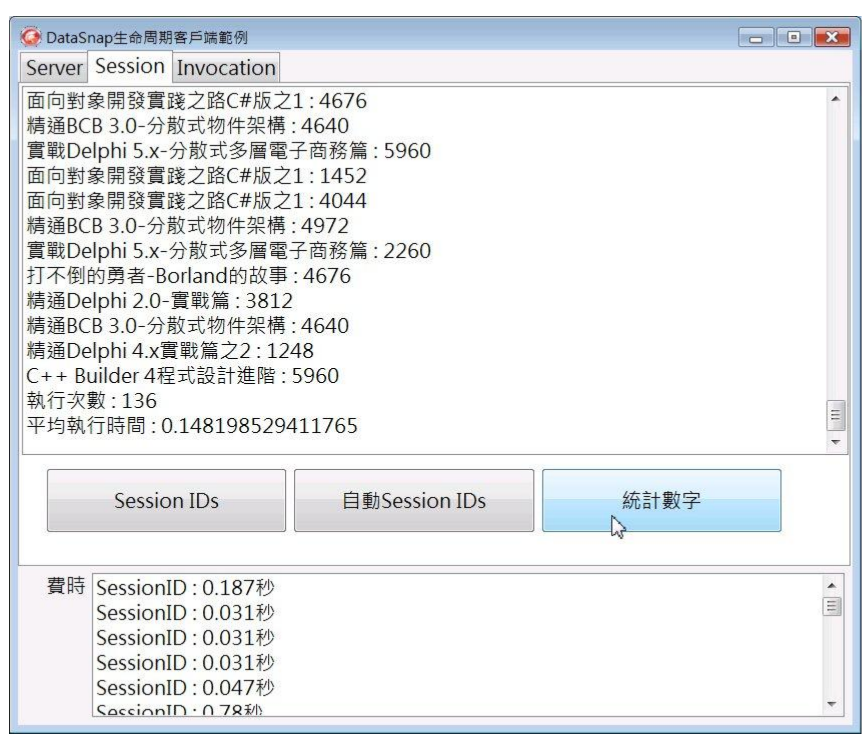



圖 6-5 使用 Session 生命週期的服務物件服務用戶端的請求

現在如果我們開啟圖 6-4 的 TSQLConnection 使用 dbExpress 的連結池功能，如下所示:

版權所有 請勿翻印

圖 6-6 開啟使用 dbExpress 的連結池功能

那麼如果我們再次執行許多範例用戶端 DataSnap 應用程式並且呼叫 GetThreadID 方法，那麼在 DataSnap 伺服器觀察的話，就會發現使用的 InterBase 連結變少了，而且如下圖所示執行效率也增加了，現在平均只需要 0.02 秒左右的執行時間。

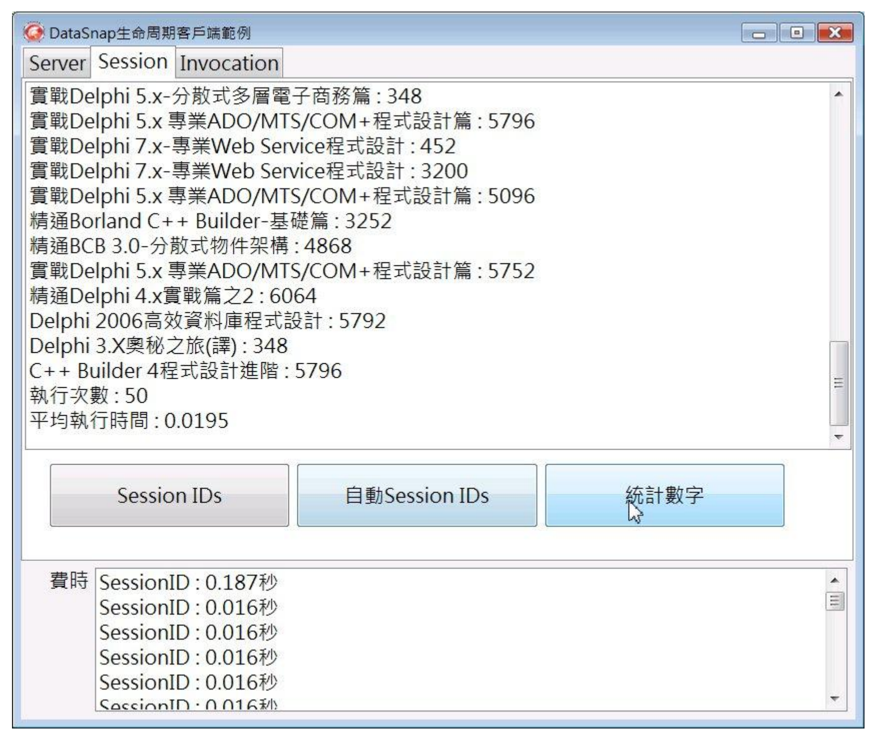



圖 6-7 使用 Session 生命週期的服務物件並且開啟 dbExpress 連結池功能服務用戶端的請求

## 6-1-3 Invocation 生命週期

版權所有 請勿翻印

使用 Invocation 生命週期的伺服端服務物件會在每一個用戶端請求時被建立，服務完用戶端請求之後會被釋放，開發人員只需如下圖在物件檢視器中設定 TDSServerClass 元件的 LifeCycle 特性值為 Invocation 即可:

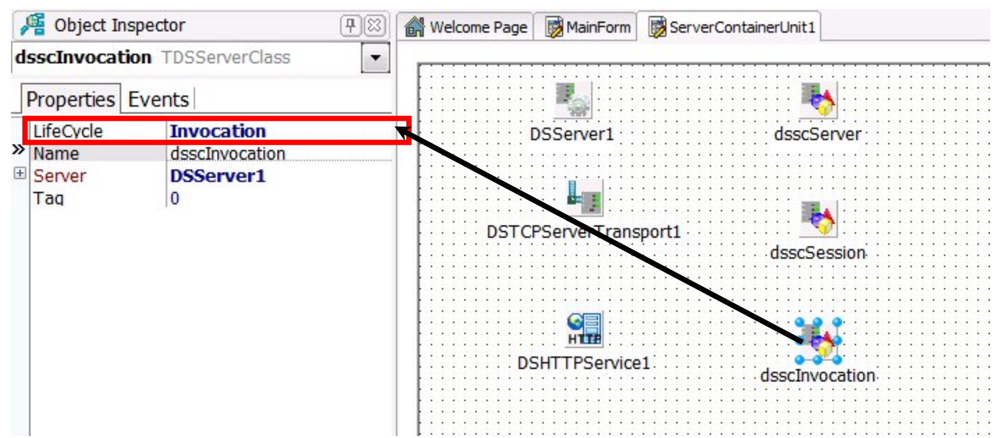



圖 6-8 設定 TdssmInvocation 元件的 LifeCycle 特性值為 Invocation 生命週期

使用 Invocation 生命週期的服務類別由於會被頻繁的建立和釋放，因此適合使用在小型，快速服務和無狀態的應用中，這種服務類別不應該很龐大以避免花費太多的時間在建立和釋放的過程中，如果這種服務類別需要存取資料庫，那麼應該使用 dbExpress 連結池功能，而且也應該在服務方法執行完畢之後立刻釋放資料庫的連結。

不過由於目前 DataSnap 框架在釋放 Invocation 生命週期的服務物件實作程式碼上有臭蟲，因此目前 Invocation 生命週期的服務物件在服務之後不會自動被釋放而造成記憶體漏失的錯誤，因此開發人員目前需要在程式碼中釋放 Invocation 生命週期的服務物件。這個釋放的工作可以藉由 TDSDestroyInstanceEventObject 類別來幫忙:

```
  TDSDestroyInstanceEventObject = class(TDSEventObject)
  public
    constructor Create(const ADbxContext: TDBXContext; const AServer: TDSCustomServer;
const ATransport: TDSServerTransport; const ADbxConnection: TDBXConnection);
  private
    FServerClassInstance: TObject;
  public
    property ServerClassInstance: TObject read FServerClassInstance write
FServerClassInstance;
  end;
```

TDSDestroyInstanceEventObject 類別的 ServerClassInstance 特性就是 Invocation 生命週期的服務物件樣例，因此在 Invocation 生命週期的服務物件服務完用戶端的請求之後，請開發人員在使用 Invocation 生命週期的 TDSServerClass 元件的 OnDestroyInstance 事件中撰寫如下的程式碼以釋放 Invocation 生命週期的服務物件:

版權所有 請勿翻印

```
procedure TServerContainer1.dsscInvocationDestroyInstance(
  DSDestroyInstanceEventObject: TDSDestroyInstanceEventObject);
begin
  DSDestroyInstanceEventObject.ServerClassInstance.Free;
end;
```

當使用 Invocation 生命週期的服務物件服務完用戶端之後，DataSnap 框架會呼叫使用 Invocation 生命週期的 TDSServerClass 元件的 OnDestroyInstance 事件,並且傳遞 TDSDestroyInstanceEventObject 物件給這個事件，因此我們只需要在 OnDestroyInstance 事件中藉由存取 TDSDestroyInstanceEventObject 物件中的 ServerClassInstance 特性值，並且呼叫它的 Free 方法即可。當然，待日後 Embarcadero 修正了這個臭蟲之後就無需上面的程式碼了。

## 6-2 DataSnap 管理功能

在 DataSnap 伺服器中除了開發人員的服務類別之外，DataSnap 框架也提供了一些內建的類別，其中的 DSAdmin 類別是 DataSnap 伺服器中內建的

管理類別，開發人員可以在 DataSnap 伺服器中呼叫 DSAdmin 類別以存取它的服務，此外 DSAdmin 也可以被 DataSnap 用戶端呼叫來存取其提供的服務，但是在 DataSnap 用戶端，開發人員是使用 TDSAdminClient 類別來存取 DataSnap 伺服器中的 DSAdmin 服務，下面的表格說明了這兩個類別的使用方式:

| 類別 | 說明 |
|---|---|
| DSAdmin | DataSnap 伺服器中提供管理功能的類別 |
| TDSAdminClient | DataSnap 用戶端類別，用戶端可使用此類別存取 DataSnap 伺服器中 DSAdmin 類別物件提供的服務，通常用戶端的 Proxy 類別會由此類別繼承下來 |

DSAdmin 類別提供了許多的服務方法，下面的表格是 DSAdmin 類別提供的服務函式:

版權所有 請勿翻印

| 函式 | 說明 |
|---|---|
| GetPlatformName | 取得 DataSnap 伺服器執行的平台名稱 |
| ClearResources | 清除 DataSnap 伺服器配置的資源 |
| FindPackages | 回傳所有 DataSnap 伺服器中的封包(Package)資訊，這些資訊都封裝在回傳的 TDBXReader 物件中 |
| FindClasses | 回傳某特定封包中所有符合搜尋類別樣例的資訊，這些資訊都封裝在回傳的 TDBXReader 物件中 |
| FindMethods | 回傳某特定封包中，某特定類別中所有符合搜尋方法樣例的資訊，這些資訊都封裝在回傳的 TDBXReader 物件中 |
| CreateServerClasses | 在 DataSnap 伺服器中建立服務類別 |
| DropServerClasses | 在 DataSnap 伺服器中刪除服務類別 |
| CreateServerMethods | 在 DataSnap 伺服器中建立服務方法 |
| DropServerMethods | 在 DataSnap 伺服器中刪除服務方法 |
| GetServerClasses | 取得 DataSnap 伺服器中所有的服務類別資訊，這些資訊都封裝在回傳的 TDBXReader 物件中 |
| ListClasses | 取得 DataSnap 伺服器中所有的服務類別資訊，這些資訊都封裝在回傳的 TJSONArray 物件中 |
| DescribeClass | 回傳 DataSnap 伺服器中特定服務類別的敘述資訊，這些資訊都封裝在回傳的 TJSONObject 物件中 |
| ListMethods | 取得 DataSnap 伺服器中所有的服務方法資訊，這些資訊都封裝在回傳的 TJSONArray 物件中 |
| DescribeMethod | 回傳 DataSnap 伺服器中特定服務方法的敘述資 |

| | 訊，這些資訊都封裝在回傳的 TJSONObject 物件中 |
|---|---|
| GetServerMethods | 回傳 DataSnap 伺服器中所有的服務方法資訊，這些資訊都封裝在回傳的 TDBXReader 物件中 |
| GetServerMethodParameters | 回傳 DataSnap 伺服器中所有的服務方法的參數資訊，這些資訊都封裝在回傳的 TDBXReader 物件中 |
| GetConnection | 回傳在傳 DataSnap 伺服器中的 TDBXConnection 物件 |
| GetDatabaseConnectionProperties | 回傳在傳 DataSnap 伺服器中所有的資料庫連結資訊，這些資訊都封裝在回傳的 TDBXReader 物件中 |

從上面的表格可以瞭解許多 DSAdmin 的服務方法都是回傳 TDBXReader 物件或是 JSON 相關類別物件，在前面的章節中已經過如何使用這些物件。

如果在 DataSnap 用戶端需要存取 DSAdmin 的服務的話，那麼開發人員可以使用 TDSAdminClient 類別，下面的表格是 TDSAdminClient 類別提供的服務函式:

| 函式 | 說明 |
|---|---|
| GetPlatformName | 取得 DataSnap 伺服器執行的平台名稱 |
| ClearResources | 清除 DataSnap 伺服器配置的資源 |
| FindPackages | 回傳所有 DataSnap 伺服器中的封包(Package)資訊，這些資訊都封裝在回傳的 TDBXReader 物件中 |
| FindClasses | 回傳某特定封包中所有符合搜尋類別樣例的資訊，這些資訊都封裝在回傳的 TDBXReader 物件中 |
| FindMethods | 回傳某特定封包中，某特定類別中所有符合搜尋方法樣例的資訊，這些資訊都封裝在回傳的 TDBXReader 物件中 |
| GetServerMethods | 回傳 DataSnap 伺服器中所有的服務方法資訊，這些資訊都封裝在回傳的 TDBXReader 物件中 |
| GetServerMethodParameters | 回傳 DataSnap 伺服器中所有的服務方法的參數資訊，這些資訊都封裝在回傳的 TDBXReader 物件中 |
| GetDatabaseConnectionProperties | 回傳在傳 DataSnap 伺服器中所有的資料庫連結資訊，這些資訊都封裝在回傳的 TDBXReader 物件中 |

版權所有 請勿翻印

| | |
|---|---|
| BroadcastToChannel | 傳遞訊息給回叫通道 |
| BroadcastObjectToChannel | 傳遞物件給回叫通道 |
| ListClasses | 取得 DataSnap 伺服器中所有的服務類別資訊，這些資訊都封裝在回傳的 TJSONArray 物件中 |
| DescribeClass | 回傳 DataSnap 伺服器中特定服務類別的敘述資訊，這些資訊都封裝在回傳的 TJSONObject 物件中 |
| ListMethods | 取得 DataSnap 伺服器中所有的服務方法資訊，這些資訊都封裝在回傳的 TJSONArray 物件中 |
| DescribeMethod | 回傳 DataSnap 伺服器中特定服務方法的敘述資訊，這些資訊都封裝在回傳的 TJSONObject 物件中 |
| NotifyCallback | 通知回叫 |
| NotifyObject | 通知回叫物件 |

請讀者比較上面 DSAdmin 和 TDSAdminClient 類別提供的服務，我們可以注意到這兩個類別提供的服務方法並不完全一致，TDSAdminClient 類別並沒有在用戶端提供完整的 DSAdmin 服務方法，因此如果開發人員需要在 DataSnap 用戶端呼叫 TDSAdminClient 類別沒有提供的 DSAdmin 服務方法，那麼開發人員仍然可以使用程式碼從用戶端呼叫 DataSnap 伺服器中的 DSAdmin 服務方法，在稍後的範例中我們會看到如何做到。

版權所有 請勿翻印

許多 DSAdmin 和 TDSAdminClient 類別提供的服務都以回傳 TDBXReader 物件或是 JSON 相關類別物件來代表執行的結果，而在這些回傳的物件中都是由其他一些相關的類別來定義的，例如 DSAdmin 的 GetServerClasses 方法回傳由 TDBXReader 物件代表的所有伺服端服務類別，而在這個回傳的 TDBXReader 物件中的資訊則是由 TDSClassEntity 類別定義的。下面的表格整理了這些相關的類別以及它們的說明:

| 類別 | 說明 |
|---|---|
| TDSClassEntity | 定義伺服端服務類別的資訊 |
| TDSConnectionEntity | 定義伺服端連結的資訊 |
| TDSMethodEntity | 定義伺服端服務方法的資訊 |
| TDSPackageEntity | 定義伺服端服務封包的資訊 |
| TDSProcedureEntity | 定義伺服端服務預儲程序的資訊 |
| TDSProcedureParametersEntity | 定義伺服端服務預儲程序的參數的資訊 |

例如 TDSClassEntity 類別定義了如下的特性，開發人員可以藉由這些特性取得或是瞭解伺服端服務類別的相關資訊:

| 特性 | 說明 |
|---|---|
| PackageName | 類別的封包名稱 |
| ServerClassName | 伺服端服務類別名稱 |
| RoleName | 類別存取角色 |
| LifeCycle | 伺服端服務類別的生命週期 |

例如 DSAdmin 的 GetServerClasses 方法回傳 TDBXReader 物件，其中的資料就是 TDSClassEntity 類別的特性值，因此在呼叫 GetServerClasses 之後，開發人員可以藉由存取這個 TDBXReader 物件的 Value[1].AsString 而得到伺服類別的名稱，也就是上面表格的 ServerClassName 特性值。

讓我們看看如何使用 DSAdmin 類別在 DataSnap 用戶端應用程式中取得 DataSnap 伺服器中相關的服務資訊，下圖是範例應用程式執行的畫面，在 DataSnap 用戶端應用程式中我們可以取得 DataSnap 伺服器中的服務類別以及服務方法，從下圖中我們也可以看到非常有趣和有用的資訊，那就是除了前面小節討論的 3 個生命週期的服務類別之外，請讀者注意的是，在得 DataSnap 伺服器中的 DSAdmin 和 DSMetadat 這兩個內建的服務類別也都是使用 Session 生命週期：

版權所有 請勿翻印

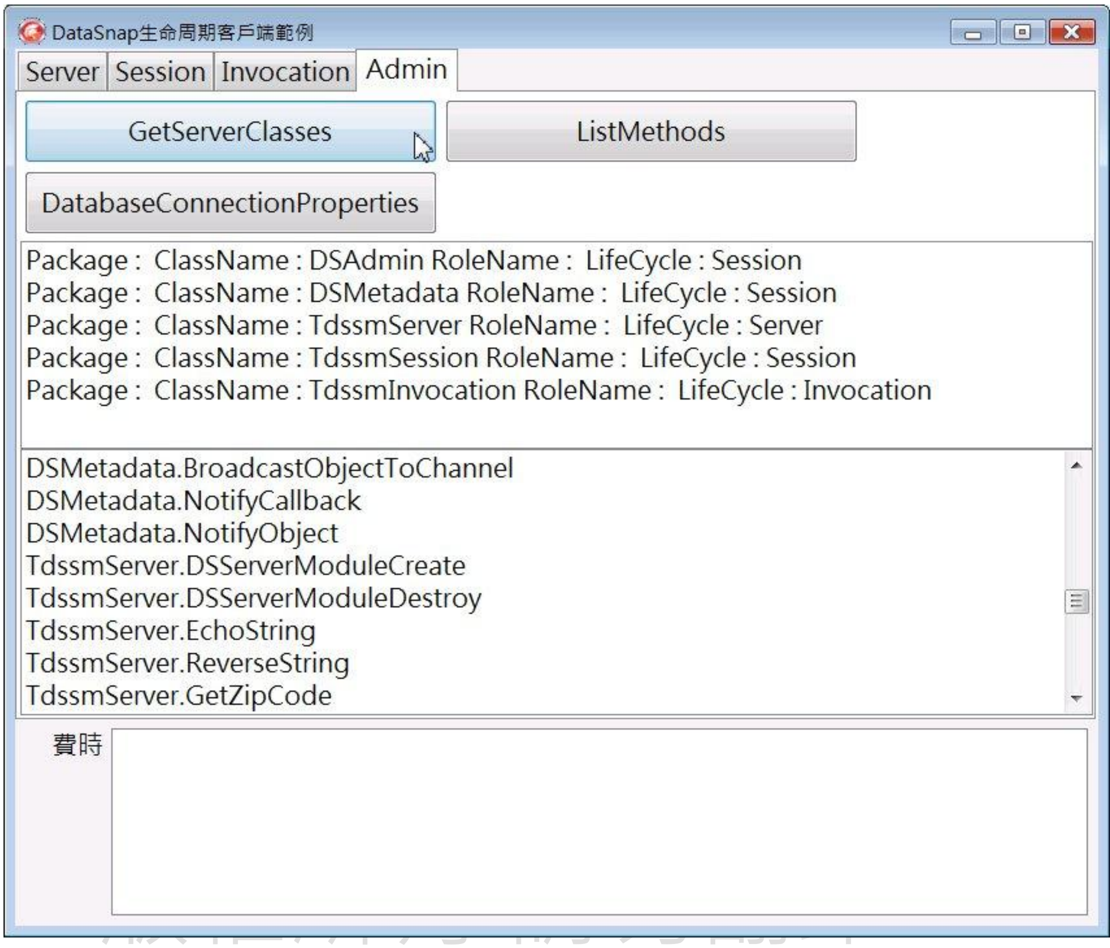



圖 6-9 藉由 DSAdim 類別取得 DataSnap 伺服器中的服務類別以及服務方法

上面畫面的『GetServerClasses』按鈕使用了下面的程式碼呼叫 DataSnap 伺服器中的 DSAdmin 物件的 GetServerClasses 方法以取得伺服端所有的服務類別：

```
001   procedure TForm9.btnGetServerClassesClick(Sender: TObject);
002   var
003     dsAdmin : TDBXCommand;
004     aReader : TDBXReader;
005     sClassData : TStringBuilder;
006   begin
007     if not ClientModule1.SQLConnection1.Connected then
008       ClientModule1.SQLConnection1.Connected := True;
009     dsAdmin := ClientModule1.SQLConnection1.DBXConnection.CreateCommand;
010     sClassData := TStringBuilder.Create;
011     try
012       dsAdmin.CommandType := TDBXCommandTypes.DSServerMethod;
013       dsAdmin.Text := TDSAdminMethods.GetServerClasses;
```

```
014       dsAdmin.Prepare;
015       dsAdmin.ExecuteUpdate;
016       aReader := dsAdmin.Parameters[0].Value.GetDBXReader(true);
017       lbClasses.Clear;
018       while aReader.Next do
019       begin
020         sClassData.Append('Package : ' + aReader.Value[0].AsString);
021         sClassData.Append(' ClassName : ' + aReader.Value[1].AsString);
022         sClassData.Append(' RoleName : ' + aReader.Value[2].AsString);
023         sClassData.Append(' LifeCycle : ' + aReader.Value[3].AsString);
024         lbClasses.Items.Add(sClassData.ToString);
025         sClassData.Clear;
026       end;
027     finally
028       sClassData.Free;
029       dsAdmin.Free;
030     end;
031   end;
```

版權所有 請勿翻印

在上面的程式碼中展示了如何在 DataSnap 用戶端呼叫位於 DataSnap 伺服器中的 DSAdmin 物件，使用的方法就是先在 009 行建立 TDBXCommand 物件，在 012 行設定此 TDBXCommand 物件執行的命令是呼叫 DataSnap 伺服器中的方法，013 行設定 TDBXCommand 物件執行的命令是 TDSAdminMethods.GetServerClasses，而 TDSAdminMethods.GetServerClasses 則是定義為 'DSAdmin.GetServerClasses'，也就是呼叫 DSAdmin 類別的 GetServerClasses 方法。

在 016 行取得回傳的 TDBXReader 物件之後，其中的資料定義就是前面已經介紹的 TDSClassEntity 類別的特性值，因此 018 行之後就可以取得相關的資料並且顯示在元件中了。

下面則是如何取得伺服端服務類別的程式碼，它和上面的程式碼非常的類似，同樣是使用 TDBXCommand 呼叫 DSAdmin 類別的 GetServerMethods 方法：

```
001   procedure TForm9.btnListMethodsClick(Sender: TObject);
002   var
003     dsAdmin : TDBXCommand;
004     aReader : TDBXReader;
```

```
005   begin
006     if not ClientModule1.SQLConnection1.Connected then
007       ClientModule1.SQLConnection1.Connected := True;
008     dsAdmin := ClientModule1.SQLConnection1.DBXConnection.CreateCommand;
009     try
010       dsAdmin.CommandType := TDBXCommandTypes.DSServerMethod;
011       dsAdmin.Text := TDSAdminMethods.GetServerMethods;
012       dsAdmin.Prepare;
013       dsAdmin.ExecuteUpdate;
014       aReader := dsAdmin.Parameters[0].Value.GetDBXReader(true);
015       lbAdmin.Clear;
016       while aReader.Next do
017       begin
018         lbAdmin.Items.Add(aReader.Value[1].AsString + '.' +
aReader.Value[2].AsString);
019       end;
020     finally
021       dsAdmin.Free;
022     end;
end;
```

014 行取得的 TDBXReader 物件，其中的資料定義就是下表的 TDSMethodEntity 類別的特性值：

| 特性 | 說明 |
| --- | --- |
| MethodAlias | 服務方法的別名 |
| ServerClassName | 伺服端服務類別名稱 |
| ServerMethodName | 服務方法名稱 |
| RoleName | 類別存取角色 |

因此 016 行之後就可以藉由這些特性值取得服務類別名稱和服務方法名稱並且顯示在畫面中，當然我們也可以直接顯示 TDSMethodEntity 的 MethodAlias 特性值，例如下圖就是修改上面 018 行為：

```
lbAdmin.Items.Add('MethodAlias : ' + aReader.Value[0].AsString);
```

之後的執行結果：

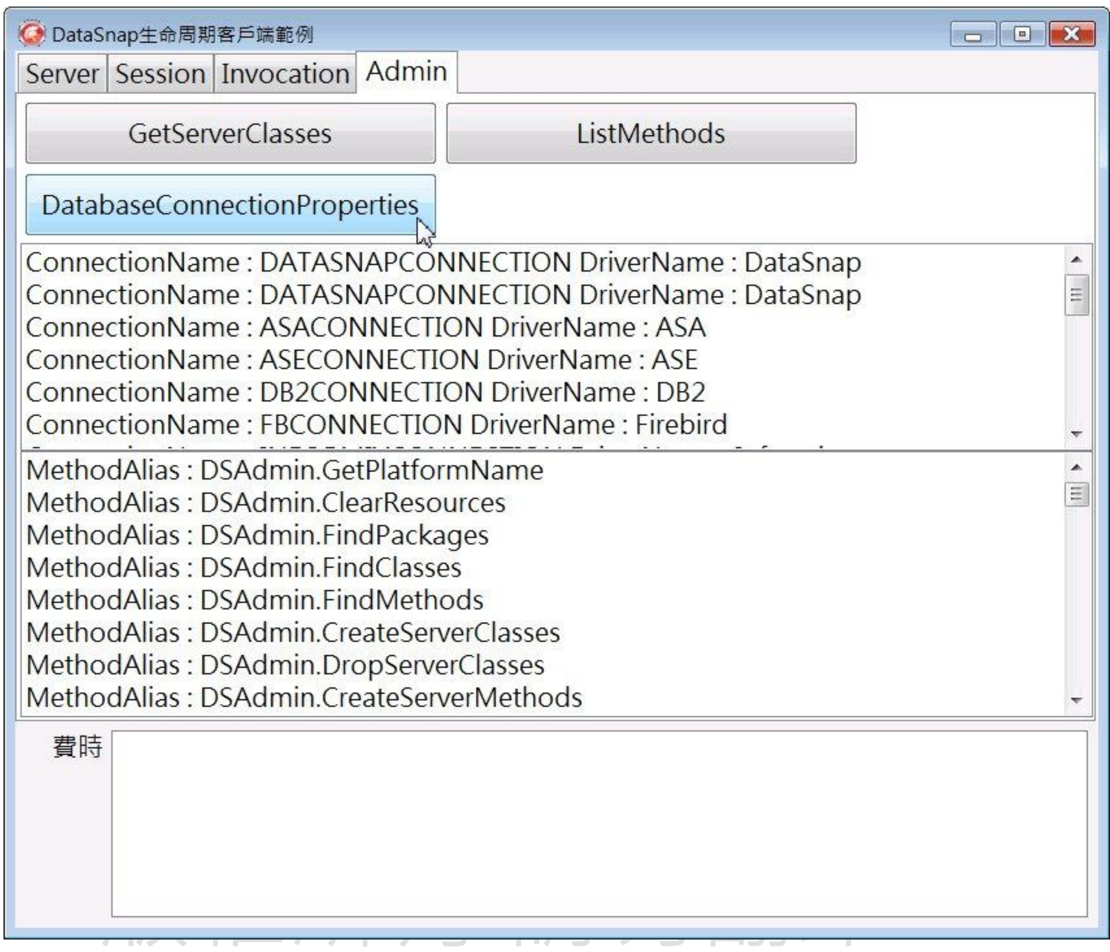



圖 6-10 藉由 DSAdim 類別取得 DataSnap 伺服器中的連結資訊以及服務方法別名

同樣的下面的程式碼呼叫 DSAdmin 類別的 GetDatabaseConnectionProperties 方法取得所有 DataSnap 伺服端的資料庫連結資訊：

```
001   procedure TForm9.btnDatabaseConnectionsClick(Sender: TObject);
002   var
003     dsAdmin : TDBXCommand;
004     aReader : TDBXReader;
005     sClassData : TStringBuilder;
006   begin
007     if not ClientModule1.SQLConnection1.Connected then
008       ClientModule1.SQLConnection1.Connected := True;
009     dsAdmin := ClientModule1.SQLConnection1.DBXConnection.CreateCommand;
010     sClassData := TStringBuilder.Create;
011     try
012       dsAdmin.CommandType := TDBXCommandTypes.DSServerMethod;
013       dsAdmin.Text := TDSAdminMethods.GetDatabaseConnectionProperties;
```

```
014       dsAdmin.Prepare;
015       dsAdmin.ExecuteUpdate;
016       aReader := dsAdmin.Parameters[0].Value.GetDBXReader(true);
017       lbClasses.Clear;
018       while aReader.Next do
019       begin
020         sClassData.Append('ConnectionName : ' + aReader.Value[0].AsString);
021         sClassData.Append(' DriverName : ' + aReader.Value[2].AsString);
022         lbClasses.Items.Add(sClassData.ToString);
023         sClassData.Clear;
024       end;
025     finally
026       sClassData.Free;
027       dsAdmin.Free;
028     end;
end;
```

這些資料庫連結資訊是由 TDSConnectionEntity 類別定義的：

| 特性 | 說明 |
|---|---|
| ConnectionName | 伺服端連結名稱 |
| ConnectionProperties | 伺服端連結特性 |
| DriverName | 驅動程式名稱 |
| DriverProperties | 驅動程式特性 |

版權所有 請勿翻印

在 DataSnap 用戶端也可以使用 TDSAdminClient 類別來呼叫 DataSnap 伺服器提供的管理服務，例如下面的程式碼直接藉由 DataSnap 用戶端應用程式中的 TdssmServerClient 呼叫遠端 DSAdmin 的 DescribeClass 方法，這是因為 TdssmServerClient 是由 TDSAdminClient 類別繼承下來：

```
001   procedure TForm9.btnDescribeClassClick(Sender: TObject);
002   var
003     aCM: TClientModule1;
004     aServer : TdssmServerClient;
005     jsonObj : TJSONObject;
006   begin
007     aCM := TClientModule1.Create(Self);
008     try
009       aServer := aCM.dssmServerClient;
010       jsonObj := aServer.DescribeClass('TdssmServer');
```

```
011        lbClasses.Clear;
012        lbClasses.Items.Add(jsonObj.ToString);
013      finally
014        aCM.Free;
015      end;
016    end;
```

下面是上面程式碼的執行結果：

圖 6-11 藉由 TDSAdminClient 類別呼叫 DataSnap 伺服器中 DSAdmin 的 DescribeClass 方法

現在讀者應該瞭解了如何使用 DSAdmin 和 TDSAdminClient 呼叫 DataSnap 伺服器中的服務方法了。

版權所有 請勿翻印

## 6-3 結論

本章討論了 DataSnap 伺服器中服務類別的生命週期以及應該在什麼情形下使用那一種生命週期，開發人員應該充分瞭解用戶端的需求而選擇使用最適合的生命週期。

最後本章討論了如何在 DataSnap 伺服端以及用戶端呼叫和使用 DataSnap 伺服器提供的管理服務，開發人員可以分別藉由 DSAdmin 和 TDSAdminClient 這兩個類別來存取 DataSnap 伺服器的管理服務功能。

# 第7章 開發移動式DataSnap用戶端

手機和移動式設備的開發在現今似乎變得愈來愈重要，因此許多應用系統都需要能夠把手機和移動式設備整合到現有的系統之中做為新的用戶端，在 RAD Studio 10.3 中提供了 Mobile Connector 的功能，允許開發人員開發 iPhone，Android 和黑莓機的 DataSnap 用戶端，讓主流手機的使用者也可以藉由手機連結到 DataSnap 伺服器以存取 DataSnap 伺服器提供的服務，本章的內容就在說明如何藉由 DataSnap Mobile Connector 的功能開發手機的 DataSnap 用戶端應用程式。

版權所有 請勿翻印

## 7-1 DataSnap Mobile Connector

10.3 推出 DataSnap Mobile Connector 技術的目的是為了讓手機客戶端能夠非常容易的連結到 Windows 平台的 DataSnap 伺服器取得服務，如此一來就能夠讓原本的 Midas 分散式系統或是最新的 DataSnap 分散式系統和移動式客戶端整合在一起。

目前由於不同的手機客戶端必須使用不同的程式語言和技術來開發，因此開發人員如果需要整合數個不同的手機客戶端和 Midas/DataSnap 分散式系統，那麼將會是非常辛苦的工作，而 DataSnap Mobile Connector 正好解決了這個問題，因為 DataSnap Mobile Connector 藉由可自動產生不同手機客戶端的程式碼並且統一使用 JSON/REST 的技術讓不同的手機客戶端連結到 Midas/DataSnap 分散式系統。

例如對於 iOS 客戶端，DataSnap Mobile Connector 可自動產生 Object C 客戶端程式碼，對於 Android 和 BlackBerry 客戶端，DataSnap Mobile Connector1 可自動產生 Java 客戶端程式碼，接著開發人員就可以使用這些用

戶端程式碼使用 JSON/REST 存取 Midas/DataSnap 伺服端服務。此外由 DataSnap Mobile Connector 產生的不同用戶端的程式碼都包含了一樣的 JSON/REST 類別函式庫，因此所有不同手機客戶端都可使用 DataSnap Mobile Connector 自動產生的 JSON/REST 類別函式庫。

下圖說明了使用 DataSnap Mobile Connector 技術之後整合 Midas/DataSnap 分散式系統和各種不同移動式客戶端的架構，讀者可以注意到在這新的分散式架構中都使用 JSON/REST 來存取服務，不過在企業內部如果為了效率的考量，那麼在企業內部仍然可以使用 TCP/IP 來存取 Midas/DataSnap 服務。

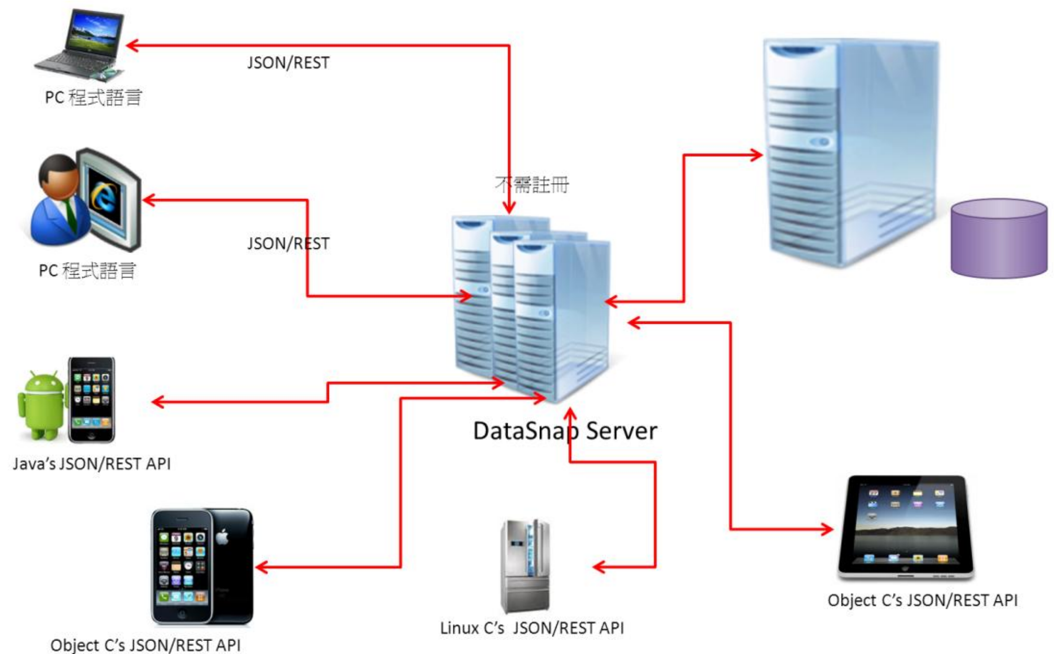



圖 1 整合分散式和移動用戶端架構

例如下圖展示如果沒有 DataSnap Mobile Connector 技術的話要如何整合 Midas/DataSnap 分散式系統和手機客戶端，我們可以看到在不同的手機客戶端需要使用不同的程式語言以及不同的 JSON/REST 函式庫:

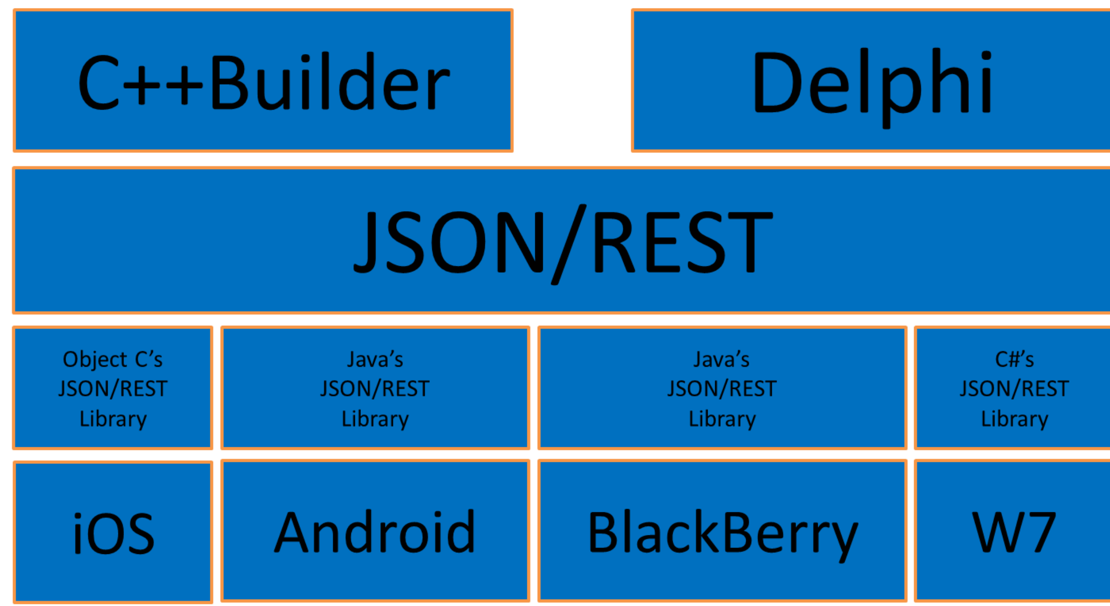



圖 2 不使用 DataSnap Mobile Connector 整合分散式和移動用戶端架構

版權所有 請勿翻印

而下圖則展示了使用 DataSnap Mobile Connector 的好處，除了 DataSnap Mobile Connector 可以自動產生手機客戶端程式碼之外，各種不同的手機客戶端都可以使用相同的 JSON/REST 函式庫存取 Midas/DataSnap 分散式系統:

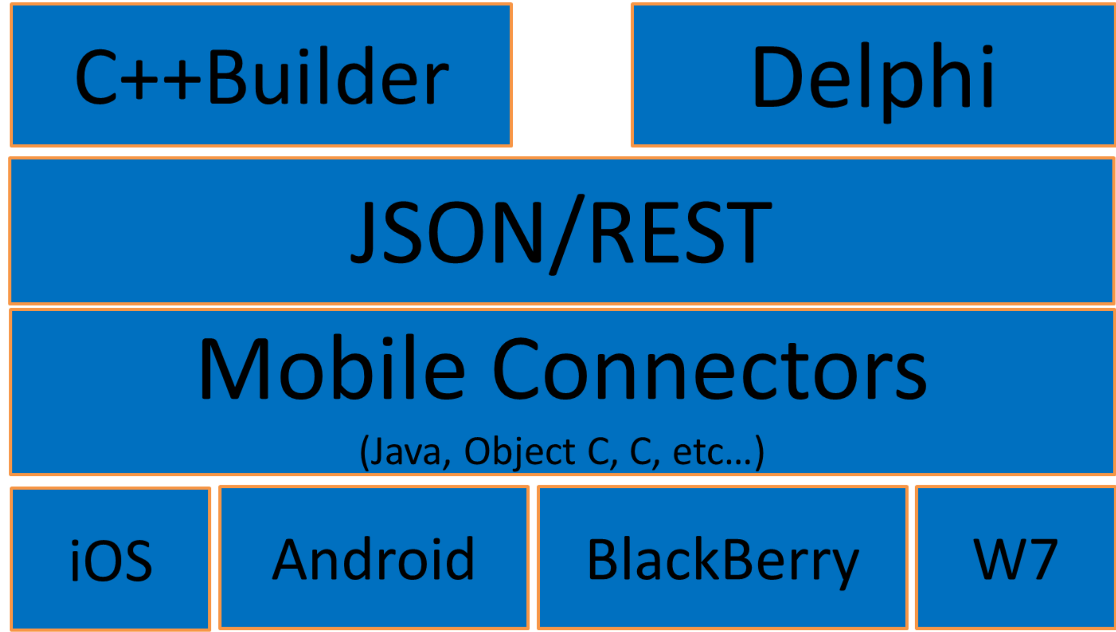



圖 3 使用 DataSnap Mobile Connector 整合分散式和移動用戶端架構

版權所有 請勿翻印

在 XE2 Update 4 中 Delphi 進一步的強化了 DataSnap 10.3 的功能，對於 iOS 客戶端 DataSnap 10.3 直接提供了 Free Pascal 的客戶端程式碼和 JSON/REST 函式庫，因此對於 iOS 客戶端在 Update 4 之後開發人員可以直接使用最熟悉的 Pascal 程式碼來整合 iOS 和 Midas/DataSnap 分散式系統而無需再使用 Object C 了，如下圖所示:

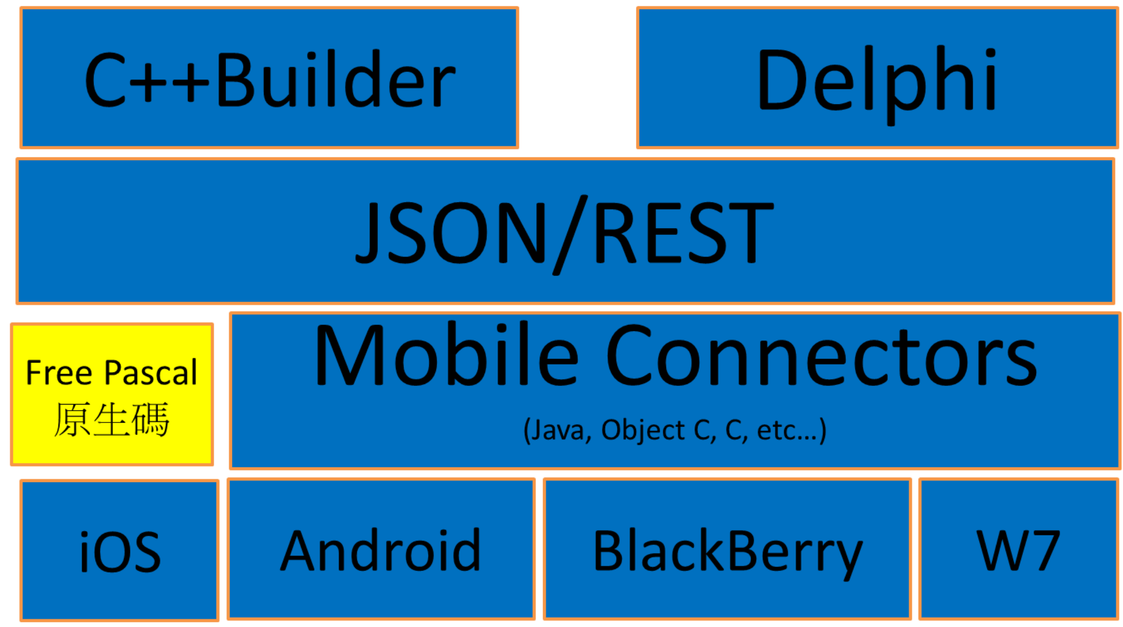



圖 4 Update 4 可直接使用 Pascal 程式碼整合分散式和移動用戶端架構

版權所有 請勿翻印

當然從 Update 4 這個發展跡象來看，DataSnap Mobile Connector 會持續的進化到允許開發人員直接使用 Delphi 程式碼來整合分散式和移動用戶端，如下圖所示:

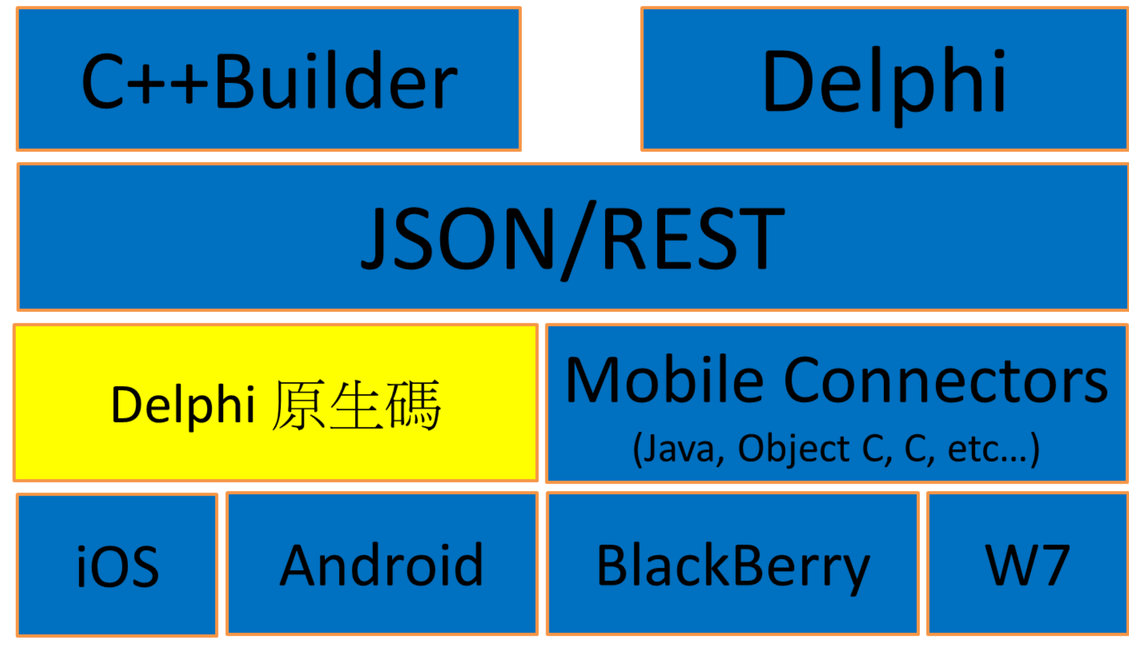



圖 5 DataSnap Mobile Connector 會持續的進化

版權所有 請勿翻印

瞭解了 DataSnap Mobile Connector 工作原理和架構之後接下來當然就是說明如何使用 DataSnap Mobile Connector 了。

## 7-2 開發 Android 用戶端

要讓 Android 客戶端連結到 DataSnap 伺服器，開發人員必須執行下列的步驟:

- 從 DataSnap 伺服器取得 Mobile Connector 的 Java 客戶端程式碼，這份程式碼不但可以讓 Android 的 Java 程式碼連結到 DataSnap 伺服器，更重要的是其中包含了所有 DataSnap 伺服器中的服務方法，可讓 Android 的 Java 客戶端程式碼直接呼叫 DataSnap 伺服器。
- 把 Mobile Connector 的 Java 客戶端程式碼匯入到 Eclipse For Android 中，再使用 Java 呼叫 DataSnap 伺服器。

接下來就讓我們使用一個實際的範例來說明如何使用 Mobile Connector。

## 7-2-1 建立 DataSnap 伺服器

要讓手機用戶端能夠連結到 DataSnap 伺服器，在建立 DataSnap 伺服器時必須加入支援 Mobile Connectors 的功能，因此先讓我們建立一個新的範例 DataSnap REST 應用程式，如下所示:

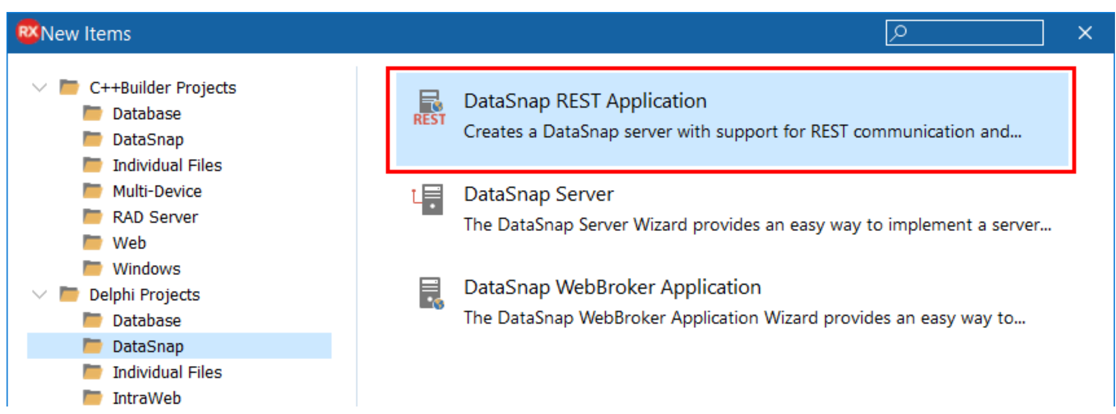



圖 6 建立範例 DataSnap 伺服器

接著在下一步中勾選支援 Mobile Connectors 功能，如下所示:

版權 印

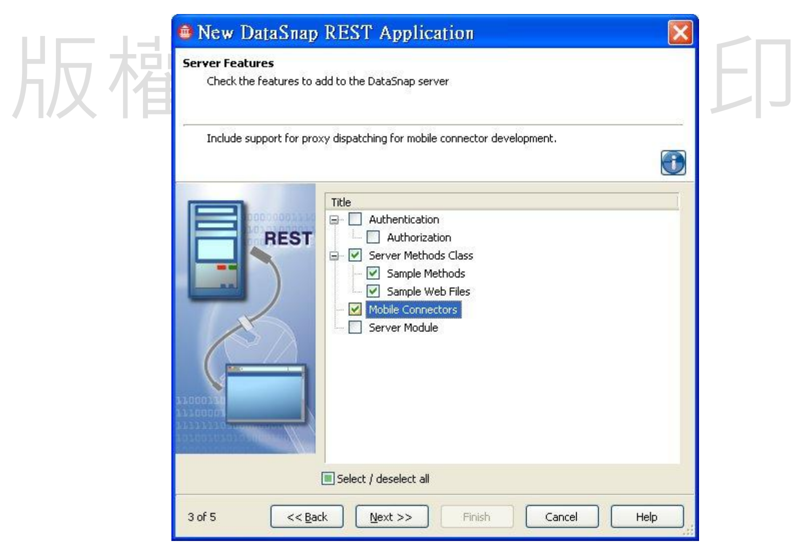



圖 7 範例 DataSnap 伺服器選擇支援 Mobile Connectors 技術

請接著繼續完成建立範例 DataSnap 伺服器的其他步驟，在完成建立範例 DataSnap 伺服器之後，如果開啟 WebModule 程式單元，那麼可以看到如下的元件。由於在前面的步驟中勾選支援 Mobile Connectors 技術，因此在此程式單元中會加入 DSProxyDispatcher 元件，這個元件可以讓用戶端藉由使用特定的 URL 來產生支援特定手機用戶端的 Mobile Connectors 原始程式:

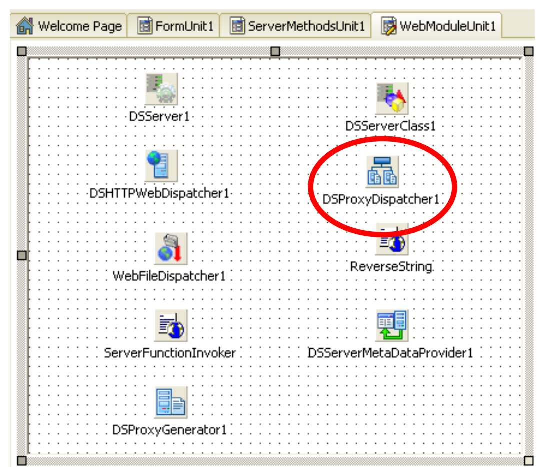



圖 8 範例 DataSnap 伺服器選擇支援 Mobile Connectors 技術

如果現在讀者開啟 WebModule 程式單元的原始程式檔，會看到它加入了支援 Android(Datasnap.DSProxyJavaAndroid)，iOS(Datasnap.DSProxyObjectiveCiOS)等手機用戶端的程式單元:

```
uses
  System.SysUtils, System.Classes, Web.HTTPApp, Datasnap.DSHTTPCommon,
  Datasnap.DSHTTPWebBroker, Datasnap.DSServer,
  Web.WebFileDispatcher, Web.HTTPProd,
  DSAuth,
  Datasnap.DSProxyDispatcher, Datasnap.DSProxyJavaAndroid,
  Datasnap.DSProxyJavaBlackBerry, Datasnap.DSProxyObjectiveCiOS,
  Datasnap.DSProxyCsharpSilverlight,
  Datasnap.DSProxyFreePascal_iOS,
  Datasnap.DSProxyJavaScript, IndyPeerImpl, Datasnap.DSClientMetadata,
  Datasnap.DSCommonServer;
```

現在如果我們編譯而且執行範例 DataSnap 伺服器，那麼就可以使用瀏覽器藉由特定的 URL 來取得支援特定手機用戶端的 Mobile Connectors 程式碼，接著就可以使用這些支援特定手機用戶端的 Mobile Connectors 程式碼來連結並且存取 DataSnap 伺服器中的服務。例如下圖是在瀏覽器中使用:

```
http://localhost:8080/proxy/java_android.zip
```

這個 URL 取得 Android 用戶端的 Mobile Connectors 程式碼:

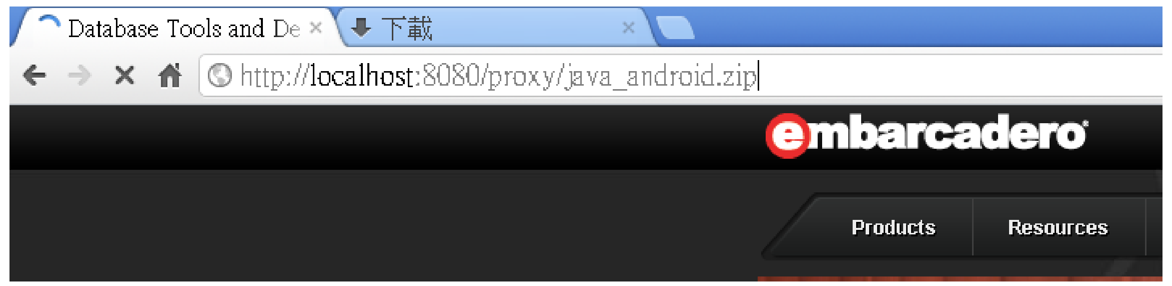



圖 9 在瀏覽器中使用特定手機的 URL 取得特定手機的 Mobile Connectors 用戶端程式碼

在瀏覽器中使用了上述的 URL 之後，範例 DataSnap 伺服器就會傳送 Android 用戶端的 Mobile Connectors 程式碼(以 ZIP 檔案壓縮)到瀏覽器中，讓瀏覽器下載。

下面的表格整理了 Mobile Connectors 技術對應不同手機用戶端必須使用的 URL:

| 手機用戶端 | URL |
|---|---|
| Android | http://伺服器位址:伺服器通信埠/proxy/java_android.zip |
| iOS 4 | http://伺服器位址:伺服器通信埠/proxy/java_android.zip |
| BlackBerry | http://伺服器位址:伺服器通信埠/proxy/java_android.zip |
| Win 7 Mobile | http://伺服器位址:伺服器通信埠/proxy/java_android.zip |
| iOS 5 | http://伺服器位址:伺服器通信埠/proxy/freepascal_ios50.zip |

版權所有 請勿翻印

如果我們解壓縮下載的 java_android.zip 就可以看到其中包含的檔案如下，這些檔案都是 Java 原始程式，可以讓開發人員在 Eclipse For Android 使用以連結到 DataSnap 伺服器。

h:\QComm\PulsarBooks\Book2\Demos\NewDemos\MobileConnectorDemo1\proxy\java_android\com\embarcadero\javaandroid*.*

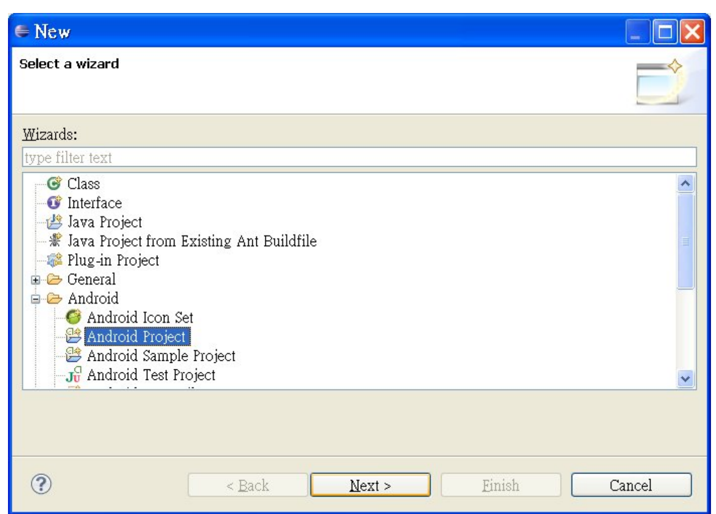
| Name | Ext | Size | ↓Date | Attr |
|---|---|---|---|---|
| .. | | <DIR> | 2011/11/06 13:58 | ---- |
| Base64 | java | 4.3 k | 2011/09/20 05:55 | -a-- |
| DBXCallback | java | 606 b | 2011/09/20 05:55 | -a-- |
| DBXDataTypes | java | 4.2 k | 2011/09/20 05:55 | -a-- |
| DBXDefaultFormatter | java | 9.8 k | 2011/09/20 05:55 | -a-- |
| DBXException | java | 895 b | 2011/09/20 05:55 | -a-- |
| DBXJSONTools | java | 14.3 k | 2011/09/20 05:55 | -a-- |
| DBXParameter | java | 1.5 k | 2011/09/20 05:55 | -a-- |
| DBXTools | java | 2.3 k | 2011/09/20 05:55 | -a-- |
| DBXValue | java | 12.9 k | 2011/09/20 05:55 | -a-- |
| DBXValueType | java | 4.3 k | 2011/09/20 05:55 | -a-- |
| DBXWritableValue | java | 11.2 k | 2011/09/20 05:55 | -a-- |
| DSAdmin | java | 36.3 k | 2011/09/20 05:55 | -a-- |
| DSAdminRestClient | java | 541 b | 2011/09/20 05:55 | -a-- |
| DSCallbackChannelManager | java | 13.3 k | 2011/09/20 05:55 | -a-- |
| DSHTTPRequestType | java | 441 b | 2011/09/20 05:55 | -a-- |
| DSRESTCommand | java | 4.5 k | 2011/09/20 05:55 | -a-- |
| DSRESTConnection | java | 20.3 k | 2011/09/20 05:55 | -a-- |
| DSRESTParamDirection | java | 787 b | 2011/09/20 05:55 | -a-- |
| DSRESTParameter | java | 1.6 k | 2011/09/20 05:55 | -a-- |
| DSRESTParameterMetaData | java | 1.3 k | 2011/09/20 05:55 | -a-- |

圖 10 java_android.zip 中包含的 Mobile Connectors 原始程式

取得了 Android 用戶端的 Mobile Connectors 原始程式之後，接下來就可以使用它來開發 Android 用戶端的 App 了。

## 7-2-2 建立 Android 用戶端

啟動 Eclipse For Android 並且建立 Android 專案:




版權所有 請勿翻印

圖 11 使用 Eclipse For Android 建立 Android 專案

接著選擇使用 Android 2.1 SDK:

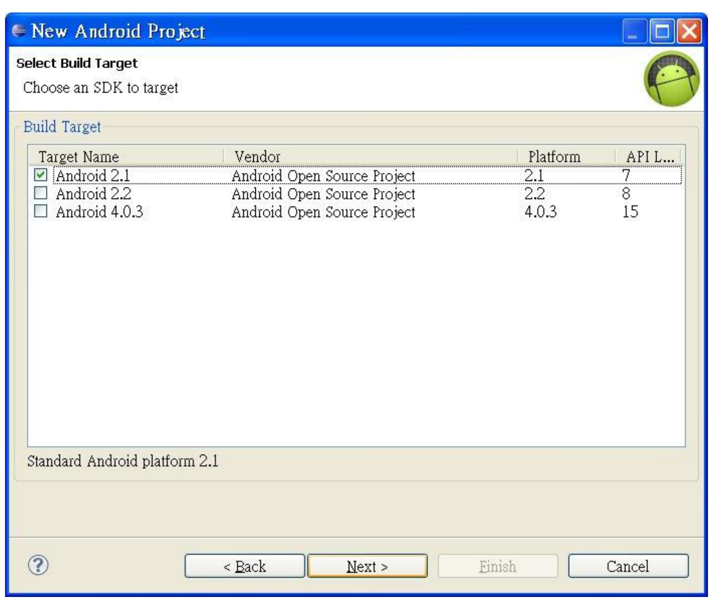



圖 12 選擇使用 Android 2.1 SDK

在建立完成 Android 專案後，請匯入 java_android.zip 到專案中，並且匯入到專案的 src 目錄中:

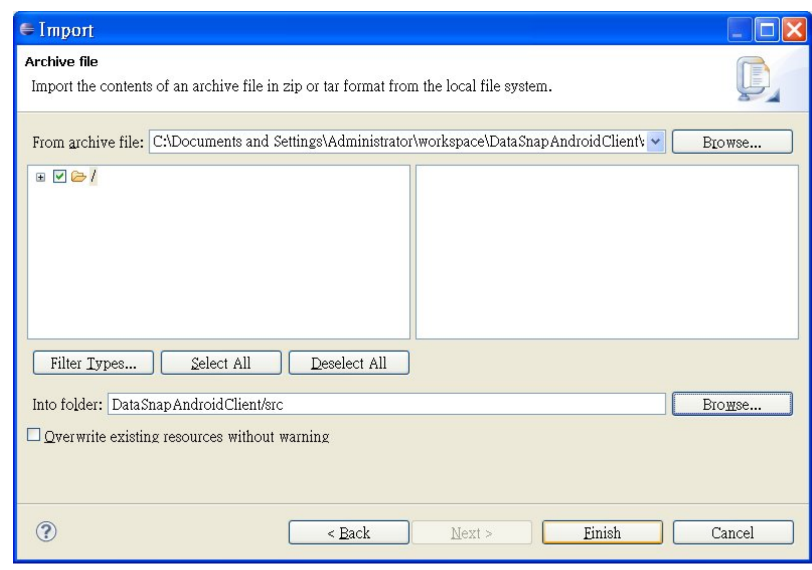
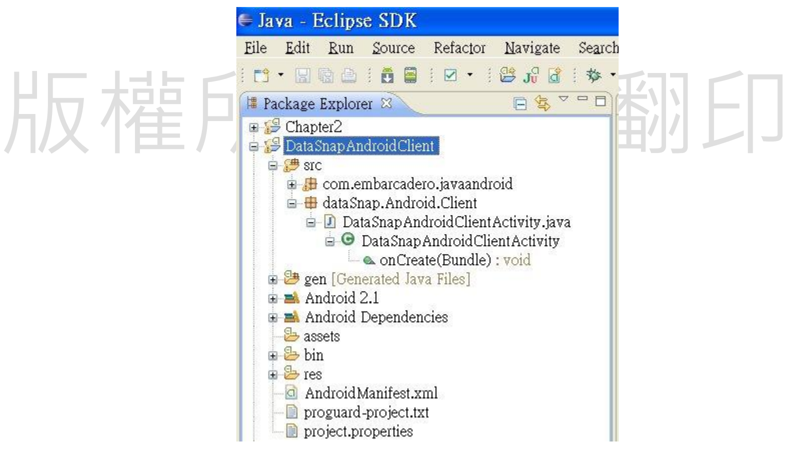



圖 13 使用輸入功能把 java_android.zip 檔案匯入到 Android 專案的 src 目錄中

在匯入完成之後如果開啟 src 節點就可以看到 com.embarcadero.javaandroid 封包出現在 src 節點下:



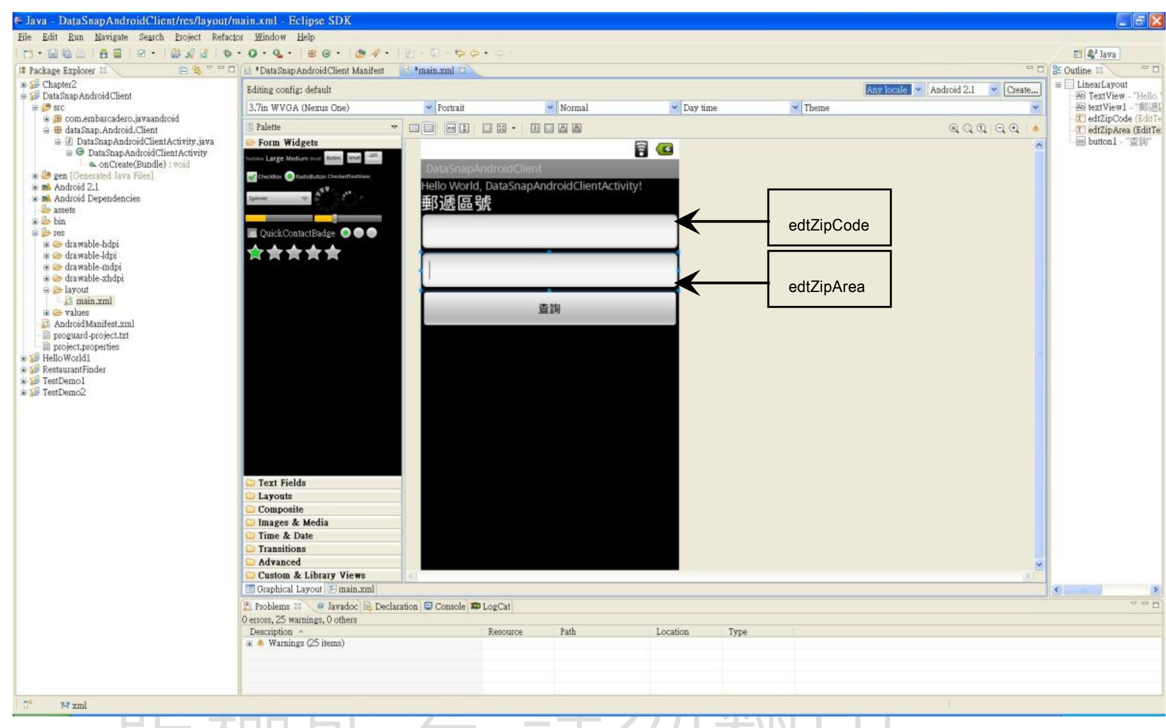

版權所 翻印

圖 14 匯入 java_android.zip 檔之後 dataSnap.Android.Client 出現在 src 目錄中

為了讓 Android 能夠連結 DataSnap 伺服器，我們必須開啟 INTERNET 存取權限，請編輯專案中的 Manifest.xml 檔案，加入 INTERNET 存取的使用者權限，如下所示:

圖 15 修改 Android 專案的 Mainfest.xml 檔，加入 INTERNET 存取的使用者權限

最後讓我們修改使用者介面，請使用滑鼠雙擊專案中 res/layout 節點之下的 main.xml 檔，此時 Eclipse 便會顯示視覺化設計介面，請在主表單中加入一個 Label，2 個 EditText 和一個 Button 元件，修改後的 main.xml 如下所示:




版權所有 請勿翻印

圖 16 修改 Main.xml 設計使用者介面

加入的第 1 個 EditText 是做為使用者輸入查詢的郵遞區號之用，而第 2 個 EditText 是做為顯示查詢的結果，因此請右擊第 1 個 EditText，設定它的 ID 為 edtZipCode，右擊第 2 個 EditText，設定它的 ID 為 edtZipArea，最後右擊 Button，設定它的 Text 為 “查詢” ，它的 ID 為 btnQuery。

設計完需要的介面之後就可以開始撰寫實作程式碼了，請雙擊 DataSnapAndroidClientActivity.java 開啟原始程式碼檔，並且實作如下的程式碼：

```
001   package dataSnap.Android.Client;
002
003   import com.embarcadero.javaandroid.DSProxy.TServerMethods1;
004   import com.embarcadero.javaandroid.DSRESTConnection;
005
006   import android.app.Activity;
007   import android.os.Bundle;
008   import android.view.View;
009   import android.view.View.OnClickListener;
```

```
010   import android.widget.Button;
011   import android.widget.EditText;
012
013   public class DataSnapAndroidClientActivity extends Activity  implements
OnClickListener{
014       Button btnQuery;
015       EditText edtZipCode;
016       EditText edtZipArea;
017       /** Called when the activity is first created. */
018       @Override
019       public void onCreate(Bundle savedInstanceState) {
020          super.onCreate(savedInstanceState);
021          setContentView(R.layout.main);
022
023          btnQuery = (Button) findViewById(R.id.btnQuery);
024          edtZipCode = (EditText) findViewById(R.id.edtZipCode);
025          edtZipArea = (EditText) findViewById(R.id.edtZipArea);
026
027          btnQuery.setOnClickListener(this);
028       }
029       @Override
030       public void onClick(View arg0) {
031            // TODO Auto-generated method stub
032            DSRESTConnection conn = new DSRESTConnection();
033            conn.setHost("172.16.137.136");
034            conn.setPort(8080);
035            conn.setProtocol("http");
036            TServerMethods1 proxy = new TServerMethods1(conn);
037            try
038            {
039
     edtZipArea.setText(proxy.GetDistrictFromZipCode(edtZipCode.getText().toString()))
;
040            }
041            catch (Exception ex)
042            {
043                 ex.printStackTrace();
044            }
```

```
045         }
046     }
```

在前面匯入的java_android.zip中有2個最重要的java檔是讓開發人員使用 REST 連結 DataSnap 伺服器，並且可用 java 程式碼呼叫 DataSnap 伺服器中輸出的服務方法。其中的 DSRESTConnection.java 檔中提供了 DSRESTConnection 類別讓 Android 的 java 客戶端可藉由 REST 技術連結 DataSnap 伺服器，而其中的 DSProxy.java 則封裝了 DataSnap 伺服器中輸出的服務方法。

因此在上面的程式碼中 032 行先建立 DSRESTConnection 物件，033 行設定 DataSnap 伺服器的位址，034 行設定 DataSnap 伺服器使用的通訊埠，035 行再設定使用的連結通訊協定，036 行就能夠建立去裝在 DSProxy.java 中的 TServerMethods1 類別物件並且呼叫 DataSnap 伺服器中輸出的服務方法了。

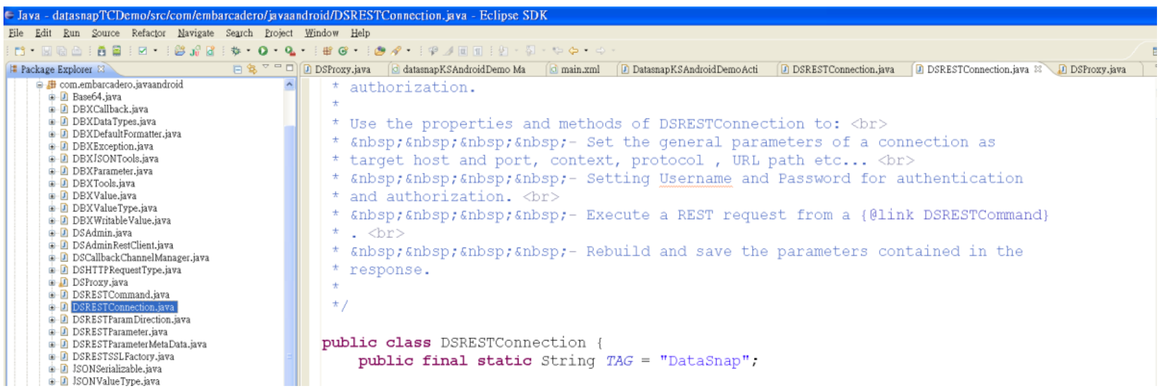



圖 17 Mobile Connectors 自動產生的 DSRESTConnection.java 可使用 REST 連結 DataSnap 伺服器

接著為範例專案建立一個執行時期組態設定，如下所示：

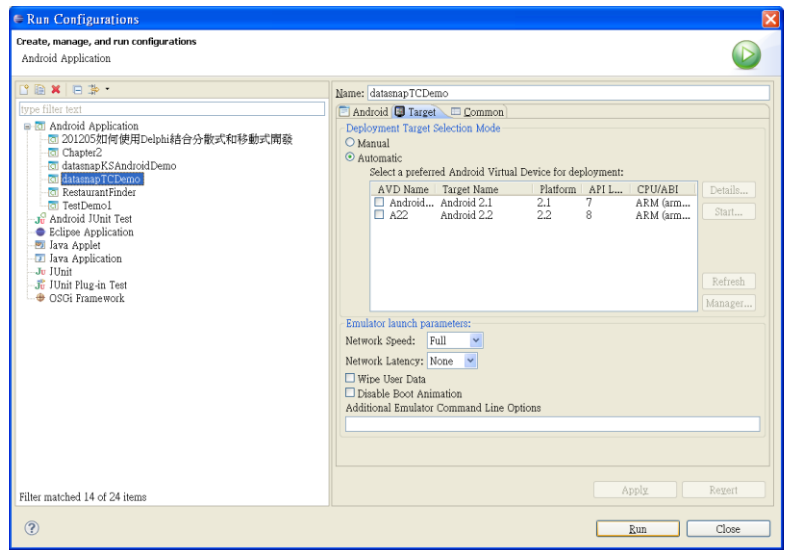


圖 18 為範例專案建立執行時期組態設定

最後使用建立的執行時期組態設定執行範例專案，Eclipse 便會啟動 Android Simulator 並且載入執行範例專案，請在範例程式中輸入一個台北的郵遞區號並且點選按鈕查詢，便會看到 Android 客戶端的確呼叫 DataSnap 伺服器進行查詢，最後查詢結果就傳遞回 Android 客戶端，並且顯示在使用者介面中，如下所示：

圖 19 在 Simulator 中執行 Android App 成功的連結 DataSnap 伺服器並且呼叫其中的服務

藉由 Mobile Connectors 我們果然可以非常容易的整合 Android 客戶端到 DataSnap 的分散式系統中，接下來讓我們繼續說明如何整合 iOS 的客戶端。

## 7-3 開發 iOS 用戶端

10.3 提供了原生 iOS 的開發能力和環境，比 XE2 提供的 iOS 開發功能進步太多了。要在 10.3 中開發 DataSnap iOS 客戶端 App，請先在 IDE 中建立一個 Multi-Device Application 專案，如下所示：

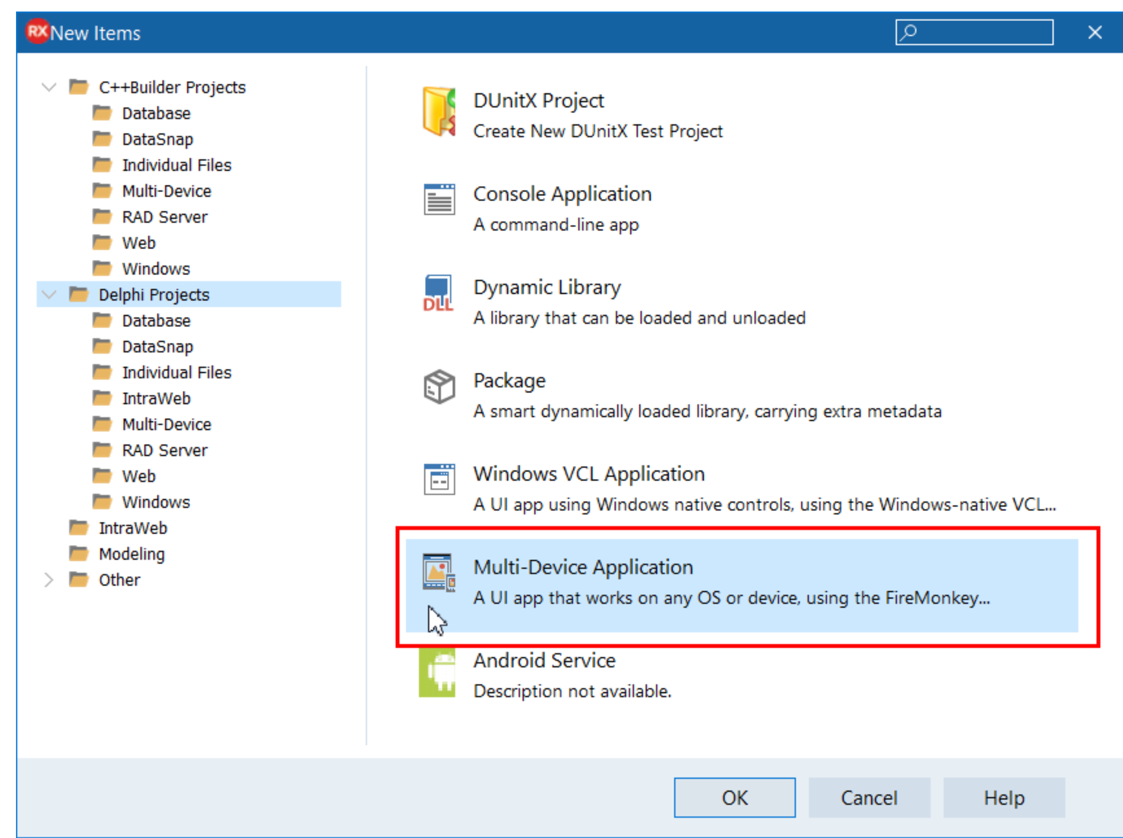



圖 20 建立 iOS 專案

版權所有 請勿翻印

接著在眾多的專案樣版中讓我們選擇最簡單的 Blank Application 樣板，如下所示：

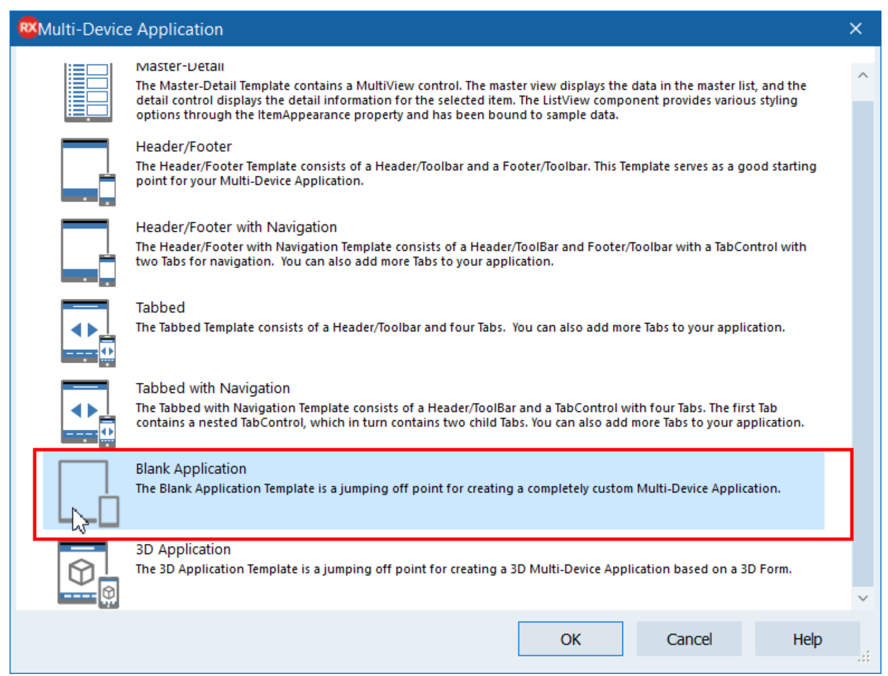



圖 21 選擇使用 Blank Application 樣板

接著在此 iOS App 的主表單中使用 TListBox，TLabel，TButton 和 TEdit 等元件設計如下的使用者介面，最後再放入 TSQLConnection 元件準備連接範例 DataSnap 伺服器，如下所示：

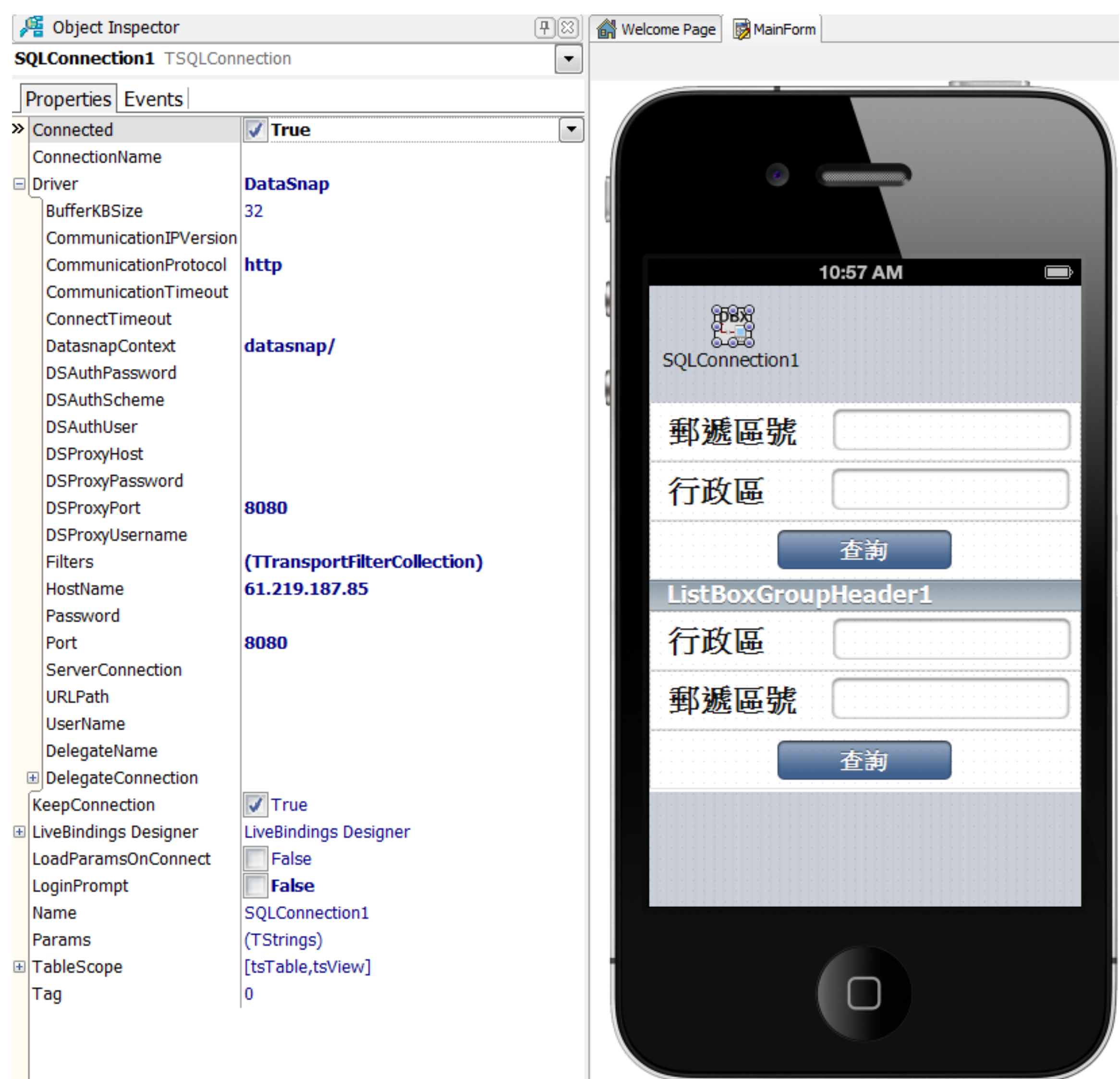



圖 22 設計主介面,並且使用 TSQLConnection 元件連結範例 DataSnap 伺服器

在物件檢視器中設定 TSQLConnection 元件的特性值:

| 特性名稱 | 特性值 |
|---|---|
| Driver | DataSnap |
| HostName | 範例 DataSnap 伺服器 IP 位址 |
| Port | 8080 |
| CommunicationProtocol | http |
| LoginPrompt | False |

我們需要藉由TSQLConnection元件設定範例DataSnap伺服器的所在地並且使用 DataSnap 驅動程式來連結。

在設定了上述的特性值之後，請再設定 TSQLConnection 元件的 Active 特性值為 True 讓 TSQLConnection 元件真正連結到範例 DataSnap 伺服器。

接著使用滑鼠右擊 TSQLConnection 元件，從突顯示選單中選擇"Generate DataSnap client classes"選項以便在 iOS 專案中產生可呼叫範例 DataSnap 伺服器服務的客戶端 Delphi 程式碼，如下所示：

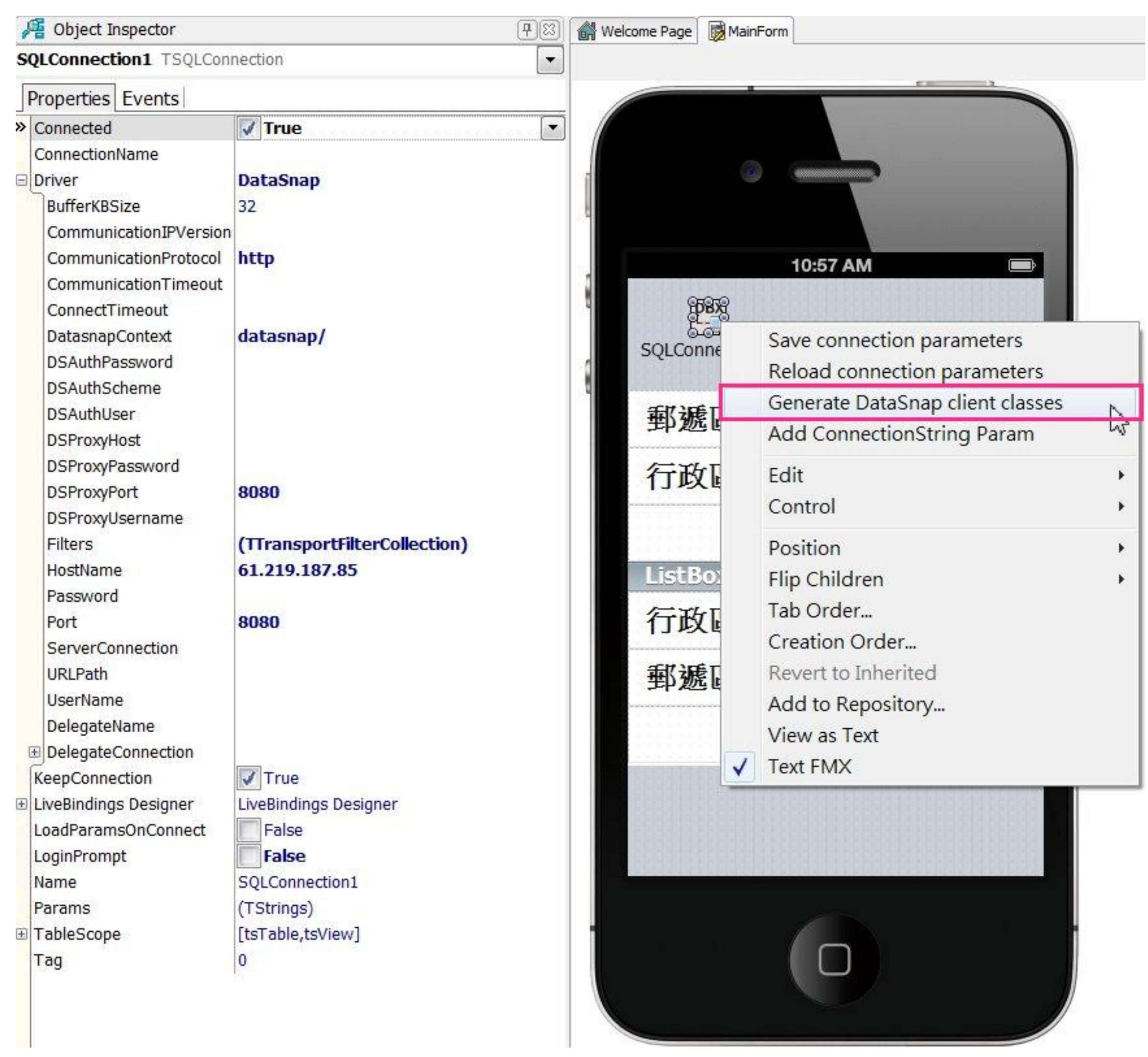



圖 23 藉由 TSQLConnection 元件自動產生客戶端連結程式碼

點選之後 Delphi 便會產生一個新的程式單元，請儲存此程式單元。在此程式單元中有一個名為 TServerMethods1Client 的類別，我們在客戶端便可以藉由這個類別來呼叫範例 DataSnap 伺服器中的服務，例如在下面的 TServerMethods1Client 的類別宣告中我們就可以看到我們需要呼叫的 GetDistrictFromZipCode 和 GetZipCode 這 2 個方法：

```
  TServerMethods1Client = class(TDSAdminClient)
  private
    FDSServerModuleDestroyCommand: TDBXCommand;
    FDSServerModuleCreateCommand: TDBXCommand;
    FEchoStringCommand: TDBXCommand;
    FReverseStringCommand: TDBXCommand;
    FGetDistrictFromZipCodeCommand: TDBXCommand;
    FGetZipCodeCommand: TDBXCommand;
  public
    constructor Create(ADBXConnection: TDBXConnection); overload;
    constructor Create(ADBXConnection: TDBXConnection; AInstanceOwner: Boolean); overload;
    destructor Destroy; override;
    procedure DSServerModuleDestroy(Sender: TObject);
    procedure DSServerModuleCreate(Sender: TObject);
    function EchoString(Value: string): string;
    function ReverseString(Value: string): string;
    function GetDistrictFromZipCode(sZipCode: string): string;
    function GetZipCode(SDistrict: string): Integer;
  end;
```

有了 TServerMethods1Client 類似之後就非常容易了，請在 iOS 主表單中使用剛才儲存的 TServerMethods1Client 類別程式單元，然後在主表單的 2 個按鈕的 OnClick 事件處理函式中撰寫如下的程式碼:

```
procedure TForm50.btnQDistrictClick(Sender: TObject);
var
  ss : TServerMethods1Client;
begin
  ss := TServerMethods1Client.Create(SQLConnection1.DBXConnection);
  try
    edtQDistrict.Text := ss.GetDistrictFromZipCode(edtZipDistrict.Text);
  finally
    ss.Free;
  end;
end;


procedure TForm50.btnQZipClick(Sender: TObject);
var
  ss : TServerMethods1Client;
```

```
begin
  ss := TServerMethods1Client.Create(SQLConnection1.DBXConnection);
  try
    edtQZip.Text := ss.GetZipCode(edtDistrictZip.Text).ToString;
  finally
    ss.Free;
  end;
end;
```

上面的程式碼只是簡單的建立 TServerMethods1Client 物件然後分別呼叫 GetDistrictFromZipCode 和 GetZipCode 這 2 個方法。

請編譯此範例 iOS App 並且分別在 iOS Simulator 和 iOS 實機中執行，您應該就可以像下面的畫面一樣成功的使用 iOS 呼叫遠端的範例 DataSnap 伺服器了。

版權

印

圖 24 範例 iOS App 成功的在 iOS Simulator 中呼叫遠端的範例 DataSnap 伺服器

圖 25 範例 iOS App 成功的在 iOS 5 實機中呼叫遠端的範例 DataSnap 伺服器

## 7-4 結論

藉由 Mobile Connectors 技術，開發人員能夠很快速，方便的開發 4 種移動平台的客戶端並且整合到 DataSnap 分散式架構中，很快的開發人員就能夠使用 Delphi 和 FireMonkey 開發 Android 平台任何種類的 App。

如是要開發 iOS 的客戶端那麼 10.3 已經提供原生的 iOS 開發能力，開發 DataSnap 的 iOS 客戶端 App 就像開發原生的 Windows 客戶端一樣的方便了。

# 第8章 DataSnap監督功能

Delphi 10.3 在 DataSnap 方面也增加了一些功能，執行效率也比 XE 版進行了一些最佳化的改善。10.3 的 DataSnap 新功能主要是增加許多監督的功能，10.3 允許 DataSnap 伺服端和用戶端能夠在執行時期取得更多對方的資訊以及對方的執行狀態，以便處理突發的狀態。例如 10.3 允許 DataSnap 伺服端能夠在 DataSnap 用戶端突然斷線時能夠取得這樣的資訊並且釋放用戶端在 DataSnap 伺服端配置的資源。

本章即將介紹 DataSnap 10.3 增加的新功能，以便讓讀者瞭解如何使用這些新功能來改善或是強化 DataSnap 應用系統。

版權所有 請勿翻印

## 8-1 DataSnap 10.3 新增監督功能

DataSnap 在 10.3 版中進行最多的改善功能應該就是為 DataSnap 伺服端和用戶端加入了大量的監督，管理功能，以便讓開發人員能夠取得更多的控制和較詳細的執行狀態資料，首先讓我們討論如何在 DataSnap 伺服端取得用戶端的資訊。

### 8-1-1 DataSnap 伺服端監督功能

DataSnap 10.3 為伺服器加入了取得 DataSnap 用戶端的連結資訊類別，藉由 TDBXClientInfo 記錄，DataSnap 伺服器可以取得連結用戶端的 IP 地址，使用的通訊協定等資訊，下面即是 TDBXClientInfo 的定義:

```
TDBXClientInfo = record
  IpAddress: String;
  ClientPort: String;
  Protocol: String;
  AppName: String;
end;
```

下面的表格說明了 TDBXClientInfo 記錄中特性值的意義:

| 特性名稱 | 說明 |
|---|---|
| IpAddress | 用戶端的連結 IP 地址 |
| ClientPort | 用戶端使用的連結通信埠 |
| Protocol | 用戶端使用的通訊協定 |
| AppName | 用戶端的應用程式名稱(當用戶端是 Web 用戶端才有這項資訊) |

DataSnap 伺服器要取得 TDBXClientInfo 中的資訊可以藉由 TDSConnectEventObject 物件，當 DataSnap 用戶端連結到 DataSnap 伺服器時，會觸發 TDSServer 元件的 OnConnect 事件處理函式，而 TDSServer 元件的 OnConnect 事件處理函式就會接受一個 TDSConnectEventObject 物件的參數。

下面是 TDSConnectEventObject 的定義，我們可以看到它定義了一個型態為 TDBXChannelInfo 的特性，FChannelInfo:

```
 TDSConnectEventObject = class(TDSEventObject)
 public
   constructor Create(const ADbxContext: TDBXContext; const AServer: TDSCustomServer;
const ATransport: TDSServerTransport; const AChannelInfo: TDBXChannelInfo; const
ADbxConnection: TDBXConnection; const AConnectProperties: TDBXProperties);
 private
   FConnectProperties: TDBXProperties;
   FChannelInfo: TDBXChannelInfo;
 Public
```

而 TDBXChannelInfo 的類別定義如下:

```
 TDBXChannelInfo = class
 public
   constructor Create(const AId: Integer);
 protected
   function GetInfo: UnicodeString; virtual;
 private
   FId: Integer;
   FClientInfo: TDBXClientInfo;
 public
   property Id: Integer read FId;
   property Info: UnicodeString read GetInfo;
```

```
  property ClientInfo: TDBXClientInfo read FClientInfo write FClientInfo;
end;
```

我們可以從 TDBXChannelInfo 類別中看到它定義了一個型態為 TDBXClientInfo 的特性 ClientInfo，因此我們只需要在發 TDSServer 元件的 OnConnect 事件處理函式中使用下面的程式碼就可以取得 TDBXClientInfo 中定義的 DataSnap 用戶端資訊:

```
DSConnectEventObject.ChannelInfo.ClientInfo
```

現在讓我們展示如何在 DataSnap 伺服器中顯示連結的 DataSnap 用戶端的資訊。首先建立一個 DataSnap 伺服器專案，並且選擇同時支援 TCP/IP 和 HTTP 兩種通訊協定:

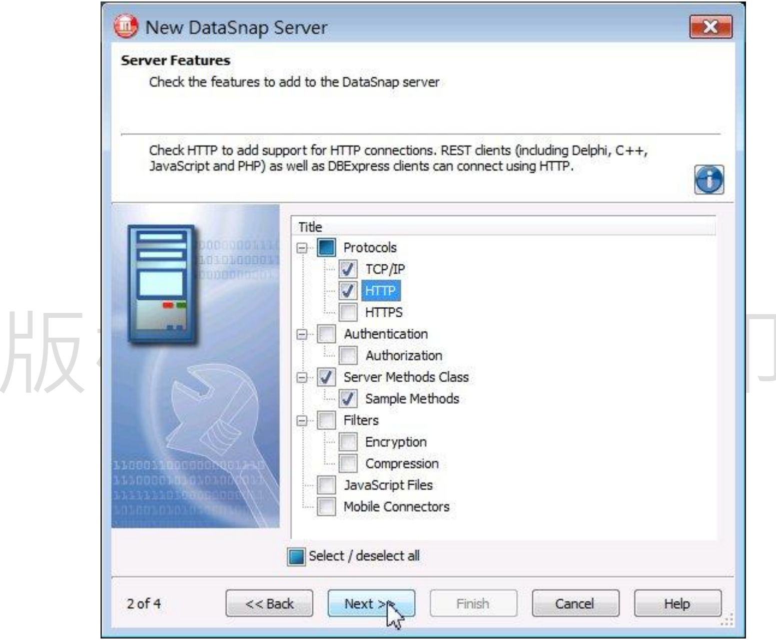



圖 1 建立支援 TCP/IP 和 HTTP 兩種通訊協定的 DataSnap 伺服器

在建立了範例 DataSnap 伺服器之後，開啟 ServerContainerUnit 程式單元，撰寫程式單元中 TDSServer 元件的 OnConnect 事件處理函式如下:

```
001   procedure TServerContainer2.DSServer1Connect(
002     DSConnectEventObject: TDSConnectEventObject);
003   begin
004     ShowClientConnections(DSConnectEventObject);
005   end;
006
007   procedure TServerContainer2.ShowClientConnections(
008     DSConnectEventObject: TDSConnectEventObject);
```

```
009   var
010     sb : TStringBuilder;
011   begin
012     sb := TStringBuilder.Create;
013     try
014       sb.Append('AppName : ' + DSConnectEventObject.ChannelInfo.ClientInfo.AppName);
015       sb.Append(' ');
016       sb.Append('Protocol : ' +
DSConnectEventObject.ChannelInfo.ClientInfo.Protocol);
017       sb.Append(' ');
018       sb.Append('IpAddress : ' +
DSConnectEventObject.ChannelInfo.ClientInfo.IpAddress);
019       sb.Append(' ');
020       sb.Append('ClientPort : ' +
DSConnectEventObject.ChannelInfo.ClientInfo.ClientPort);
021       sClient := sb.ToString;
022       TThread.Synchronize(nil, UpdateClientConnections);
023     finally
024       sb.Free;
025     end;
026   end;
```

在上面的程式碼中藉由 TDSConnectEventObject 物件的 ChannelInfo 特性值取得 TDBXChannelInfo 物件，再藉由 TDBXChannelInfo 物件的 ClientInfo 特性值取得 TDBXClientInfo 記錄，然後就可以擷取其中的 DataSnap 用戶端資訊了。

下面的畫面是執行此範例 DataSnap 伺服器應用程式，並且在用戶端分別執行使用 TCP/IP 通訊協定的 Window DataSnap 用戶端以及使用 HTTP 通訊協定的 Web 用戶端呼叫範例 DataSnap 伺服器中的方法，從範例 DataSnap 伺服器的主表單中我們可以看到範例 DataSnap 伺服器果然可以取得每一個連結 DataSnap 用戶端的相關資訊:

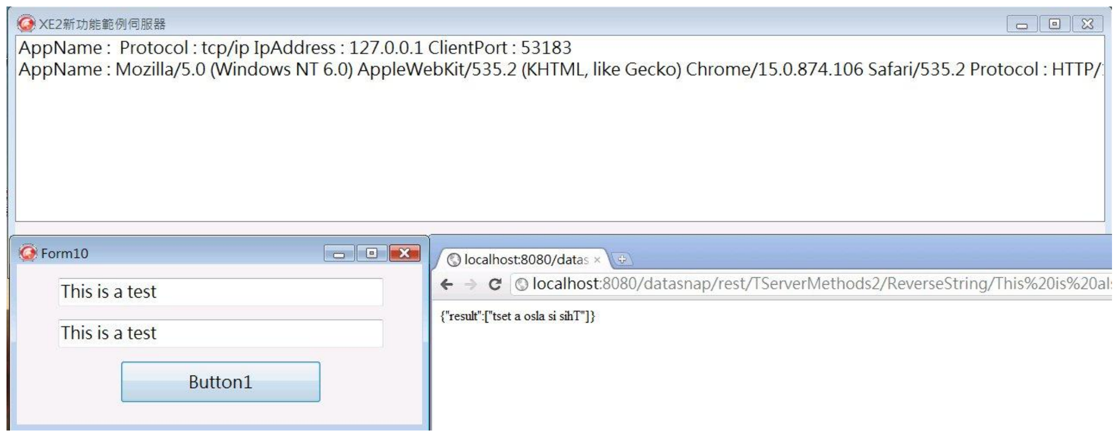



圖 2 範例 DataSnap 伺服器顯示使用 TCP/IP 和 HTTP 通訊協定連結的 DataSnap 用戶端的資訊

## 8-2 DataSnap Session 功能

DataSnap 10.3 在伺服端也提供了 Session 物件的機制，也提供了許多有用的 Session 管理功能。在 DataSnap 框架中，伺服端的 Session 是由 TDSSession 類別定義的，TDSSession 類別提供了許多有用的方法和特性讓開發人員能夠取得服務或是取得重要的資訊，例如取得 Session 狀態，安排 Session 物件定時或是自動執行工作，或是在 Session 物件中暫時儲存資料等。

版權所有 請勿翻印

如果開發人員需要暫時在 Session 物件中儲存資料，那麼開發人員可以使用下面的方法來儲存字串或是客製化類別物件:

| 方法 | 說明 |
|---|---|
| function HasData(Key: String): Boolean | 判斷在 Session 物件中是否存有字串資料 |
| function GetData(Key: String): String | 在 Session 物件中根據傳入的鍵值取得字串資料 |
| procedure PutData(Key, Value: String) | 把鍵值/字串數值儲存到 Session 物件中 |
| procedure RemoveData(Key: String) | 從 Session 物件中根據鍵值移除字串資料 |
| function HasObject(Key: String): Boolean | 判斷在 Session 物件中是否存有物件 |
| function GetObject(Key: String): TObject | 在 Session 物件中根據傳入的鍵值取得物件 |
| function PutObject(Key: String; Value: TObject): Boolean | 把鍵值/物件儲存到 Session 物件中 |
| function RemoveObject(Key: String; InstanceOwner: Boolean = True): TObject | 從 Session 物件中根據鍵值移除物件 |

除了 TDSSession 類別之外，DataSnap 框架也定義了 TDSSessionManager 類別來管理所有的 Session 物件，

TDSSessionManager 使用了 Singleton 設計樣例，因此在整個 DataSnap 伺服器中只有一個 TDSSessionManager 物件。

TDSSessionManager 類別提供了許多讓開發人員可以為特定的 Session 加入監督事件，以便在特定 Session 事件發生時能夠呼叫開發人員定義的事件處理函式。此外 TDSSessionManager 也提供了許多方法讓開發人員能夠拜訪或是管理所有 TDSSessionManager 物件管理的 Session 物件。

現在讓我們來看看如何使用這兩個類別。

假設我們現在有下面的類別，當 DataSnap 用戶端連結到 DataSnap 伺服器時，我們希望建立一個 TMySessionData 物件並且儲存在這個 DataSnap 用戶端的 Session 物件中。

```
type
  TMySessionData = class
  private
    FdtData : longint;
  public
    property myTime : longint read FdtData write FdtData;
    destructor Destroy; override;
  end;

implementation

{ TMySessionData }

{ TMySessionData }

destructor TMySessionData.Destroy;
begin
  fdtData := 0;
  inherited;
end;
```

現在讓我們開啟 8-1 小節的範例 DataSnap 伺服器，在 TServerMethods2 類別中加入兩個新的方法 GetSessionData 和 StoreSessionData，如下所示:

```
001   procedure TServerMethods2.GetSessionData(const key: String;
002     out dtDateTime: longint);
003   var
```

```
004     session : TDSSession;
005     sessionData : TMySessionData;
006   begin
007     session := TDSSessionManager.GetThreadSession;
008     sessionData := TMySessionData(session.GetObject(key));
009     dtDateTime := sessionData.myTime;
010   end;
011
012   procedure TServerMethods2.StoreSessionData(dtDateTime : longint; out sKey : String);
013   var
014     session : TDSSession;
015     sessionData : TMySessionData;
016   begin
017     session := TDSSessionManager.GetThreadSession;
018     sessionData := TMySessionData.Create;
019     sessionData.myTime := dtDateTime;
020     sKey := session.SessionName;
021     if (session.PutObject(sKey, sessionData)) then
022       TThread.Synchronize(nil, UpdateDataStatus);
023   end;
```

在範例 DataSnap 用戶端中先呼叫 DataSnap 伺服器的 StoreSessionData 方法以儲存參數 dtDateTime 的數值，並且從參數 sKey 中得到 DataSnap 伺服器回傳的鍵值(其實就是這個 DataSnap 用戶端的 Session 物件的 Id 值)。而 StoreSessionData 在 017 行先藉由 TDSSessionManager 類別的類別方法 GetThreadSession 取得目前這個 DataSnap 用戶端專屬的 Session 物件，在 018 行建立 TMySessionData 物件，019 行把參數 dtDateTime 儲存到 TMySessionData 物件的 myTime 特性中，最後在 020 行把 Session 物件的 SessionName 特性值儲存到參數 sKey 中。請注意，由於參數 sKey 是定義為 out 型態的參數，因此這個參數值會回傳回用戶端。

而上面的 GetSessionData 方法則接受用戶端傳遞來的鍵值(Session Id)，008 行呼叫 Session 物件的 GetObject 方法以鍵值取得對應的儲存物件，由於 GetObject 回傳的是 TObject，因此 008 行再轉變型態為 TMySessionData 即可。

由於在前面我們建立了一個 TMySessionData 物件並且儲存在 Session 物件，那麼如果希望在 Session 關閉或是結束時能夠自動釋放 TMySessionData 物件的話，那麼要如何做呢? 我們可以藉由向 Session 物件註冊一個回叫函式，

在 Session 物件發生特定的事件時自動呼叫我們的事件處理函式。在目前的 DataSnap 框架中為 Session 定義了如下的兩個狀態:

```
TDSSessionEventType = (SessionCreate, SessionClose);
```

SessionCreate 代表目前 Session 物件被建立,SessionClose 則代表目前 Session 物件在關閉或是結束狀態。

由於所有的 Session 物件都是由 TDSSessionManager 單一物件管理的,因此我們可以向這個 TDSSessionManager 物件註冊回叫事件,讓 TDSSessionManager 在建立或是結束 Session 時呼叫我們的事件處理函式。要向 TDSSessionManager 註冊回叫事件,我們可以呼叫 TDSSessionManager 的 AddSessionEvent 方法:

```
procedure AddSessionEvent(Event: TDSSessionEvent);
```

AddSessionEvent 接受一個型態為 TDSSessionEvent 的事件參數,而 TDSSessionEvent 則定義如下:

```
TDSSessionEvent = reference to procedure(Sender: TObject;
                           const EventType: TDSSessionEventType;
                           const Session: TDSSession);
```

版權所有 請勿翻印

因此我們就可以撰寫一個原型為 TDSSessionEvent 的函式並且傳遞給 AddSessionEvent 方法。現在就讓我們撰寫一個回叫事件並且向 TDSSessionManager 物件註冊。

現在開啟範例 DataSnap 伺服器的 TServerContainer2 程式單元,在它的 OnCreate 事件處理函式中呼叫 AddSessionListener 方法。AddSessionListener 方法在 008 行藉由 TDSSessionManager 類別的類別特性 Instance 取得 DataSnap 伺服器中唯一的 TDSSessionManager 物件,接著呼叫它的 AddSessionEvent 方法,並且傳遞行 009 行開始的匿名程序做為參數,這個匿名程序的原型和 TDSSessionEvent 定義的是相符合的:

```
001   procedure TServerContainer2.DataModuleCreate(Sender: TObject);
002   begin
003     AddSessionListener;
004   end;
005
006   procedure TServerContainer2.AddSessionListener;
007   begin
008     TDSSessionManager.Instance.AddSessionEvent(
009       procedure(Sender: TObject;
```

```
010              const EventType: TDSSessionEventType;
011              const Session: TDSSession)
012     begin
013       case EventType of
014         SessionCreate :
015           begin
016             Inc(iSessionNumber);
017             TThread.Synchronize(nil, UpdateSessionNumber);
018           end;
019         SessionClose  :
020           begin
021             ReleaseAnySessionObject(Session);
022           end;
023       end;
024     end);
025   end;
026
027   procedure TServerContainer2.ReleaseAnySessionObject(const Session: TDSSession);
028   begin
029     if (Session.HasObject(Session.SessionName)) then
030     begin
031       Session.RemoveObject(Session.SessionName, True);
032     end;
033   end;
```

這個匿名程序中會根據它的 EventType 參數值來判斷目前 Session 物件的狀態，如果發現目前 Session 即將結束，那麼就在 021 行呼叫 ReleaseAnySessionObject 方法來釋放 TMySessionData 物件。

現在如果我們執行範例 DataSnap 伺服器和範例 DataSnap 用戶端以及使用瀏覽器存取範例 DataSnap 伺服器，那麼我們可以看到如下的執行結果，不論是 Windows DataSnap 用戶端或是瀏覽器用戶端宦都可以儲存 TMySessionData 物件到每一個用戶端專屬的 Session 物件中並且從 DataSnap 伺服器取得 Session Id。而當用戶端直接結束時，也能夠藉由匿名回叫程序自動釋放 TMySessionData 物件。

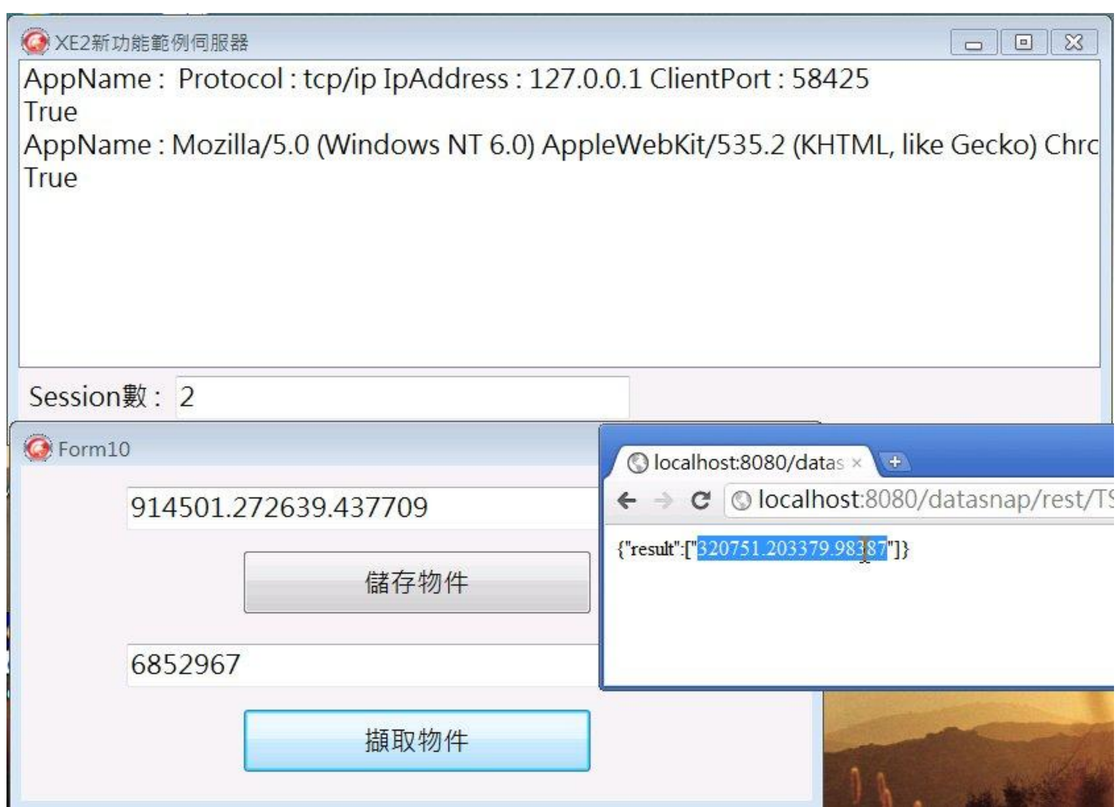



圖 3 DataSnap 用戶端應用程式能夠儲存物件在 DataSnap 伺服器的 Session 之中

> 注意，前面討論的是告訴讀者如何使用 TDSSessionManager 和 TDSSession 類別，即使讀者在 Session 物件中儲存了客製化物件，但沒有使用前面的方法釋放客製化物件時，DataSnap 框架在結束 Session 物件時也會自動釋放所有儲存在 Session 物件中的客製化物件

## 8-3 TCP 連結監督功能

當使用 DataSnap 開發分散式應用系統時，DataSnap 伺服器可以服務使用各種不同通訊協定連結的用戶端，包括了使用 HTTP/HTTPS 和 TCP/IP 的用戶端。對於 HTTP/HTTPS 用戶端，DataSnap 伺服器可以在完成用戶端的請求之後即釋放用戶端在伺服端配置的資源，但對於使用 TCP/IP 連結的用戶端應用程式而言，這個連結是具狀態，且可能一直保持連結的狀態，因此在服務 TCP/IP 的用戶端時，DataSnap 伺服端無法自動在用戶端完成服務請求時即釋放資源，必須等待 TCP/IP 的用戶端安全的離線之後才能夠釋放此用戶端在伺服端配置的資源。

但如果使用 TCP/IP 連結的用戶端由於某種原因而斷線時，DataSnap 伺服端可能並不知道 TCP/IP 連結的用戶端斷線了，因此無法釋放伺服端的資源，最後造成 DataSnap 伺服器無法負荷而異常終止執行。

為了解決這個問題，DataSnap 10.3 特別加強對於使用 TCP/IP 連結的用戶端的監督功能，希望能夠讓 DataSnap 伺服器能夠掌握不正常 TCP/IP 用戶端的連線和斷線狀況，以便適當的釋放伺服端的資源。因此，開發人員如果瞭解 DataSnap 10.3 的 TCP 連結監督功能，再結合下一小節討論的 KeepAlive 功能，那麼就可以開發出處理 DataSnap 用戶端不正常斷線的狀況。

DataSnap 框架中的 TDSTCPServerTransport 元件提供了 OnConnect 和 OnDisconnect 兩個事件處理函式來通知 DataSnap 伺服器使用 TCP/IP 的 DataSnap 用戶端連線的斷線的事件,開發人員可以在這兩個事件處理函式執行程式碼以處理相對應的工作，下面是這兩個事件處理函式的原型:

```
property OnConnect: TDSTCPConnectEvent read FTDSTCPConnectEvent write FTDSTCPConnectEvent;
property OnDisconnect: TDSTCPDisconnectEvent read FTDSTCPDisconnectEvent write
FTDSTCPDisconnectEvent;
```

其中的 OnConnect 事件特別重要，因為它可提供豐富的資訊，讓 DatsSnap 伺服器能夠記錄用戶端的資訊，以便處理不正常斷線的狀況。OnConnect 事件是宣告為如下的函式原型型態:

版權所有 請勿翻印

```
TDSTCPConnectEvent = procedure(Event: TDSTCPConnectEventObject) of object;
```

TDSTCPConnectEvent 函式接受一個型態為 TDSTCPConnectEventObject 的 Event 參數，在 TDSTCPConnectEventObject 中宣告了使用 TCP/IP 用戶端的連線資訊，其中有兩個重要的特性:Connection 和 Channel。TDSTCPConnectEventObject 是一個記錄型態，它的宣告原型如下:

```
TDSTCPConnectEventObject = record
private
  FConnection: TObject;
  FChannel: TDSTCPChannel;
public
  constructor Create(AConnection: TObject; AChannel: TDSTCPChannel);
  property Connection: TObject read FConnection;
  property Channel: TDSTCPChannel read FChannel;
end;
```

下面表格說明了 Connection 和 Channel 特性的意義:

| 特性 | 說明 |
| --- | --- |
| Connection | 用戶端的連結物件，在目前 10.3 中是 TIdTCPConnection 型態的物件 |
| Channel | 代表此連結的通道物件，是型態為 TDSTCPChannel 的物件 |

其中代表此連結通道的特性 Channel 是 TDSTCPChannel 的物件，而 TDSTCPChannel 類別中則包含了下一小節即將討論的 KeepAlive 相關的函式，此外 TDSTCPChannel 類別中也包含了此連結用戶端的 SessionId。

因此開發人員可以使用下面的步驟管理使用 TCP/IP 連結的 DataSnap 用戶端應用程式:

- 在 TDSTCPServerTransport 的 OnConnect 事件處理函式中記錄此連結 DataSnap 用戶端的連結資訊，以處理正常和不正常的斷線狀況。例如使用 TObjectDictionary 記錄 Connection 和 Channel 特性，或是使用 TDictionary 記錄 SessionId 和 Channel 特性

- 在 DataSnap 用戶端應用程式不正常使用 DataSnap 伺服器的服務時，藉由記錄的 Channel 特性來強迫切斷 DataSnap 用戶端的連結

版權所有 請勿翻印

- 在使用 KeepAlive 功能時，當使用 TCP/IP 連結的 DataSnap 用戶端不正常斷線時，切斷 DataSnap 用戶端的連結並且釋放 DataSnap 用戶端在伺服端配置的資源

在下面的小節中我們將示範如何完成上述的前 2 項工作，至於上述第 3 項的工作將在下一節『KeepAlive 功能』中說明。

## 8-3-1 使用 TCP 連結監督功能

讓我們繼續使用前面小節的範例 DataSnap 伺服器做為說明，此時我們需要為範例 DataSnap 伺服器建立下面的功能:

1. 建立一個資料結構以記錄使用 TCP/IP 通訊協定連結到 DataSnap 伺服器的 DataSnap 用戶端應用程式相關的 TIdTCPConnection 和 TDSTCPChannel 物件

2. 當需要強迫斷線 DataSnap 用戶端應用程式時，需要找到此 DataSnap 用戶端應用程式被記錄的 TIdTCPConnection 和 TDSTCPChannel 物件

3. 呼叫 TDSTCPChannel 物件的 Close 方法強迫切斷 DataSnap 用戶端應用程式的連結

首先讓我們為的範例 DataSnap 伺服器中的 ServerContainer 程式單元中的 TDSTCPServerTransport 元件建立一個 OnConnect 事件處理函式，以記錄使用 TCP/IP 通訊協定連結到 DataSnap 伺服器的 DataSnap 用戶端應用程式相關的 TIdTCPConnection 和 TDSTCPChannel 物件:

```
001   procedure TServerContainer2.DSTCPServerTransport1Connect(
002     Event: TDSTCPConnectEventObject);
003   begin
004     System.TMonitor.Enter(FConnections);
005     try
006       FConnections.Add(TIdTCPConnection(Event.Connection), Event.Channel);
007     finally
008       System.TMonitor.Exit(FConnections);
009     end;
010     AddConnectionToList(TIdTCPConnection(Event.Connection), Event.Channel);
011     TThread.Synchronize(nil, UpdateTCPMonitorInfo);
012   end;
```

在 006 行使用了 FConnections 物件記錄 DataSnap 用戶端應用程式相關的 TIdTCPConnection 和 TDSTCPChannel 物件。

而 FConnections 則是宣告為 TObjectDictionary 的型態的物件，如下所示:

```
  FConnections: TObjectDictionary<TIdTCPConnection,TDSTCPChannel>;
```

FConnections 是在 ServerContainer 程式單元的 OnCreate 事件處理函式中建立的:

```
procedure TServerContainer2.DataModuleCreate(Sender: TObject);
begin
  FConnections := TObjectDictionary<TIdTCPConnection,TDSTCPChannel>.Create;
  AddSessionListener;
end;
```

現在回到範例 DataSnap 伺服器的主表單，讓我們在其中加入一個『關閉選擇的 TCP 客戶端』Button 元件以及其下方的 TListBox 元件，為這個 TListBbox 取名為 lbTCPMonitorInfo，如下所示:

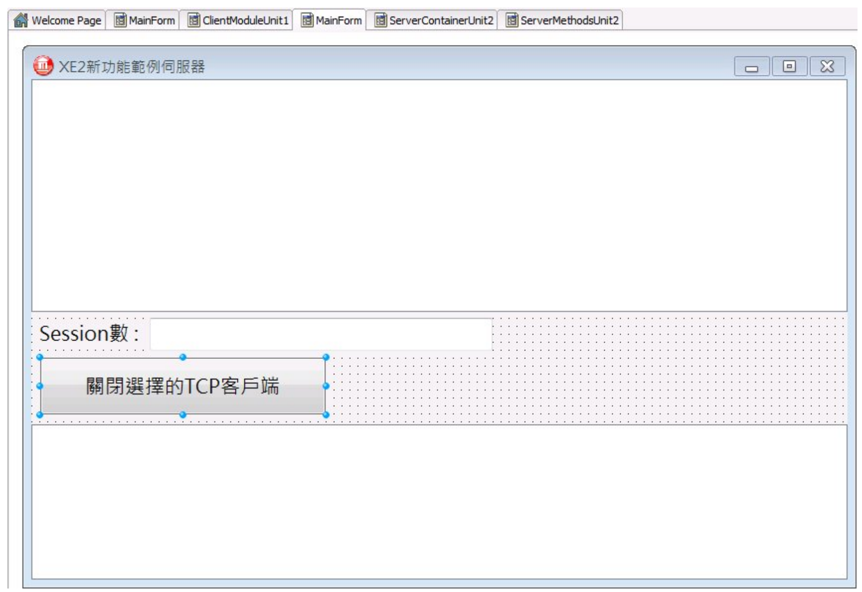



圖 4 在範例 DataSnap 伺服器的主表單中加入『關閉選擇的 TCP 客戶端』Button 元件

現在我們要展示前面的第 2 個步驟，記錄 TIdTCPConnection 和 TDSTCPChannel 物件在 lbTCPMonitorInfo 中，以便讓使用者可以藉由點選 lbTCPMonitorInfo 之中的連結資訊，再點選『關閉選擇的 TCP 客戶端』Button 元件以切斷使用 TCP/IP 通訊協定連結的 DataSnap 用戶端應用程式。

版權所有 請勿翻印

請讀者注意的是，在前面 DSTCPServerTransport1Connect 的事件處理函式中第 010 行呼叫了 AddConnectionToList 方法，而 AddConnectionToList 則是把 TIdTCPConnection 物件藉由呼叫 lbTCPMonitorInfo 的 AddObject 方法把 TIdTCPConnection 物件加入到 TListBox 中:

```
001    procedure TServerContainer2.UpdateTCPMonitorInfo;
002    begin
003      frmMainForm.lbTCPMonitorInfo.Items.AddObject(ConnInfoStr, pConn);
004    end;
005
006    procedure TServerContainer2.AddConnectionToList(Conn: TIdTCPConnection; Channel:
TDSTCPChannel);
007    begin
008      pConn := Conn;
009      if (Conn <> nil) and (Channel <> nil) and (Channel.ChannelInfo <> nil) and
010         (Channel.ChannelInfo.ClientInfo.IpAddress <> EmptyStr) then
011      begin
012        with Channel.ChannelInfo.ClientInfo do
```

```
013       begin
014         ConnInfoStr := Format('%s:%s', [IpAddress, ClientPort]);
015       end;
016     end
017     else
018       ConnInfoStr := '通道資訊錯誤.';
019   end;
```

最後，讓我們實作『關閉選擇的 TCP 客戶端』Button 元件的 OnClick 事件處理函式。當使用者點選了其下方的 lbTCPMonitorInfo 其中某一個連結資訊時，我們可以從其中擷取出已經儲存在這個被選擇的項目中的 TIdTCPConnection 物件，藉由 TIdTCPConnection 物件取得 TDSTCPChannel 物件，最後再呼叫 TDSTCPChannel 物件的 Close 方法即可。

下面是『關閉選擇的 TCP 客戶端』Button 元件的 OnClick 事件處理函式:

```
001   procedure TfrmMainForm.Button1Click(Sender: TObject);
002   var
003     pConn: TIdTCPConnection;
004     connstr : String;
005   begin
006     pConn := GetSelectedConnection;
007     ServerContainer2.DisConnectConnection(pConn);
008     ShowMessage('已切斷 : ' + lbTCPMonitorInfo.Items[lbTCPMonitorInfo.ItemIndex] + '
的連線');
009   end;
010
011   function TfrmMainForm.GetSelectedConnection: TIdTCPConnection;
012   var
013     I, Count, Index: Integer;
014     Obj: TObject;
015   begin
016     Result := nil;
017     Index := -1;
018     Count := lbTCPMonitorInfo.Count;
019
020     if Count > 0 then
021     begin
022       for I := 0 to Count - 1 do
```

```
023       begin
024         if lbTCPMonitorInfo.Selected[I] then
025         begin
026           Index := I;
027           break;
028         end;
029       end;
030
031       if Index > -1 then
032       begin
033         Obj := lbTCPMonitorInfo.Items.Objects[Index];
034         if Obj <> nil then
035           Exit(TIdTCPConnection(Obj));
036       end;
037     end;
038   end;
```

006 行呼叫 GetSelectedConnection 取得在 lbTCPMonitorInfo 被選擇的 TIdTCPConnection 物件，接著 007 行呼叫 ServerContainer 的 DisConnectConnection 程序關閉 TCP/IP 連結。

版權所有 請勿翻印

DisConnectConnection 非常的簡單，因為一旦有了 TIdTCPConnection 物件，就可以藉由 FConnections 物件取得它相關的 TDSTCPChannel 物件了，最後呼叫 TDSTCPChannel 物件的 Close 方法:

```
procedure TServerContainer2.DisConnectConnection(theConnection : TIdTCPConnection);
var
  theChannel : TDSTCPChannel;
begin
  if (theConnection <> nil) then
  begin
    FConnections.TryGetValue(theConnection, theChannel);
    theChannel.Close;
  end;
end;
```

現在執行範例 DataSnap 伺服器，再執行一個範例 DataSnap 用戶端應用程式，接著點選範例 DataSnap 伺服器主表單中 lbTCPMonitorInfo 的連結資訊，再點選『關閉選擇的 TCP 客戶端』Button 元件，我們就可以看到如下的執行結果，範例 DataSnap 伺服器強迫中斷了範例 DataSnap 用戶端應用程式的 TCP/IP 連結:

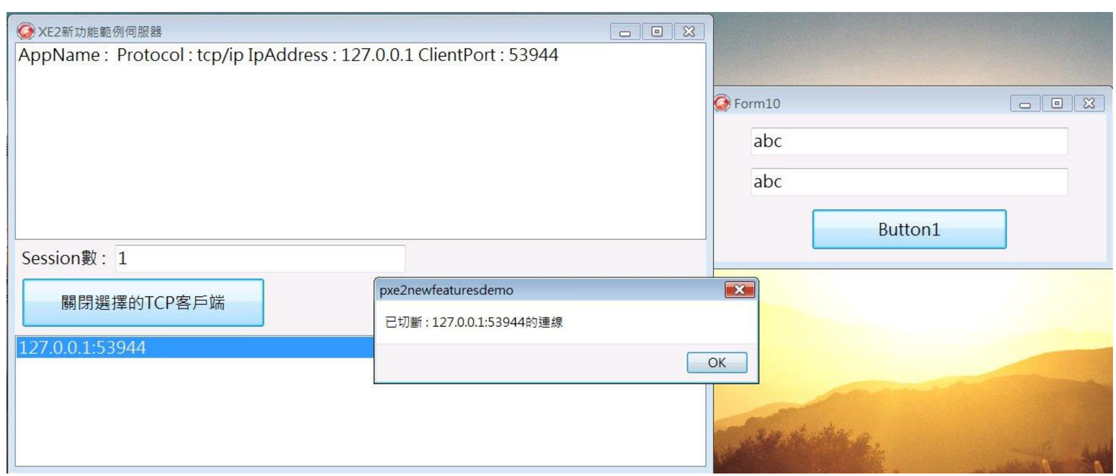



圖 5 在範例 DataSnap 伺服器中選擇要切斷的 DataSnap 用戶端應用程式

現在如果 DataSnap 用戶端應用程式想再存取範例 DataSnap 伺服器就會產生錯誤，如下所示:

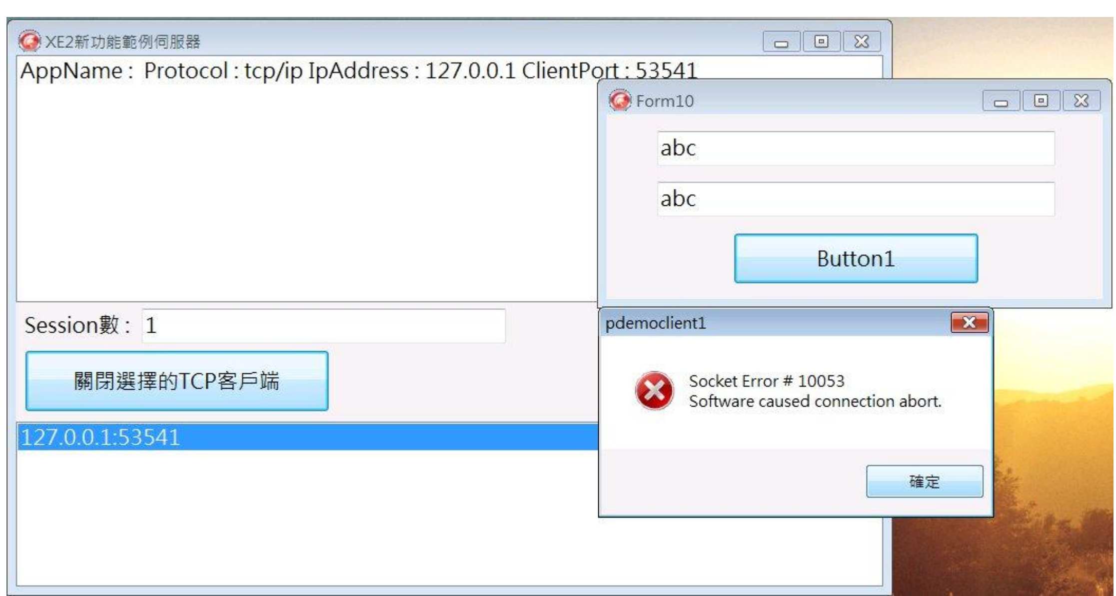



圖 7 在範例 DataSnap 伺服器切斷的 DataSnap 用戶端應用程式之後，如果 DataSnap 用戶端應用程式想再存取範例 DataSnap 伺服器就會產生錯誤

這個範例充分展現了新的 DataSnap 10.3 框架可監督 TCP 連結的功能。

## 8-4 KeepAlive 功能

DataSnap 10.3 框架最重要的功能之一就是加入了 DataSnap 伺服器和 DataSnap 用戶端的互動查詢的功能，藉由 KeepAlive 功能，DataSnap 伺服器可以主動在設定的時間之內查詢 DataSnap 用戶端的連結是否正常，如果 DataSnap 伺服器一直無法查詢到 DataSnap 用戶端，那麼 DataSnap 伺服器

就會主動切斷連結並且釋放 DataSnap 用戶端在 DataSnap 伺服器中配置的資源。

這個功能是藉由 TDSTCPServerTransport 元件新的 3 個相關的 KeepAlive 特性來設定和控制的:

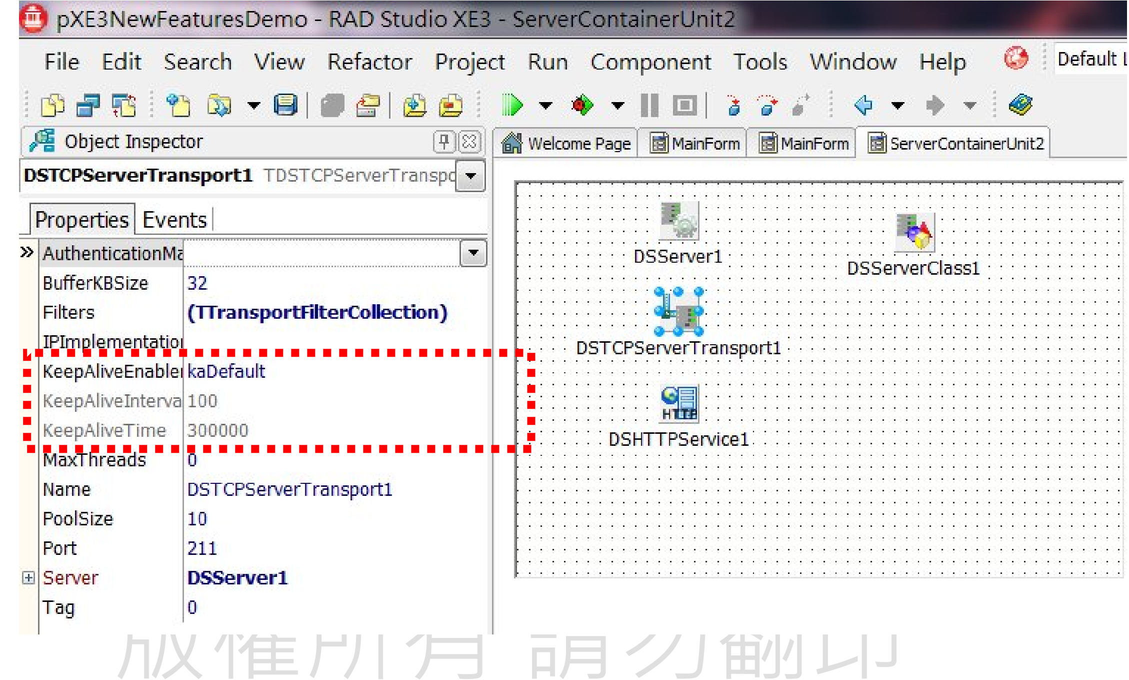



圖 5 使用 TDSTCPServerTransport 元件新的 KeepAlive 相關特性設定 DataSnap 伺服器和 DataSnap 用戶端的互動查詢

下面的表格說明了這 3 個相關的 KeepAlive 特性:

| 特性 | 說明 |
|---|---|
| KeepAliveEnablement | 如何設定 KeepAlive 的狀態 |
| KeepAliveInternal(毫秒) | 只有 KeepAliveEnablement 特性設定為 kaEnabled 時才作用，它代表每次 DataSnap 伺服器查詢 DataSnap 用戶端是否還存在的間隔時間 |
| KeepAliveTime(毫秒) | 只有 KeepAliveEnablement 特性設定為 kaEnabled 時才作用，它代表每次 DataSnap 伺服器查詢 DataSnap 用戶端是否還存在的總時間，如果在這個總時間之內 DataSnap 用戶端都沒有回應，那麼 DataSnap 伺服器就會主動切斷連結並且釋放 DataSnap 用戶端在 DataSnap 伺服器中配置的資源 |

下面的表格說明了 KeepAliveEnablement 特性可以設定的特性值:

| 特性值 | 說明 |
|---|---|
| kaDefault | 使用系統內定的設定 |
| kaDisabled | 關閉 KeepAlive 功能 |
| kaEnabled | 開啟 KeepAlive 功能 |

這整個的執行流程如下所述:

1. 當開發人員設定了 KeepAliveEnablement 特性值為 kaEnabled 之後，DataSnap 10.3 框架的 KeepAlive 功能便開始啟動

2. 當 KeepAlive 功能啟動之後，DataSnap 伺服器便會等待 KeepAliveTime 特性值設定的時間之後查詢 DataSnap 用戶端是否還在線

3. 如果查詢失敗，那麼 DataSnap 伺服器便會等待 KeepAliveInternal 特性值設定的時間之後，再次查詢 DataSnap 用戶端是否還在線

4. DataSnap 伺服器會根據作業系統設定的查詢次數限制，例如 Windows 是查詢 10 次，如果在查詢了作業系統設定的次數之後 DataSnap 用戶端還是沒有回應，那麼 DataSnap 伺服器便會判定 DataSnap 用戶端已經因為某種原因斷線了

版權所有 請勿翻印

5. 接著 DataSnap 伺服器就可以釋放 DataSnap 用戶端的 TCP/IP 連線以及 DataSnap 用戶端在 DataSnap 伺服器中配置的任何資源

現在有了 KeepAlive 功能之後，開發人員就可以避免 DataSnap 用戶端不正常的斷線，再不斷的重新連結 DataSnap 伺服器，造成 DataSnap 伺服器無法釋放先前連結所配置的資源，最後造成 DataSnap 伺服器因為記憶體/資源不足而發生執行錯誤的情形了。

## 8-5 結論

本章討論了 DataSnap 10.3 中重要的新功能，開發人員可以藉由這些新功能更完善，精確的控制 DataSnap 應用系統，以便開發出更安全，穩定的分散式應用系統。

# 開發高效率 **DataSnap** 篇

版權所有 請勿翻印

在本書的" 開發高效率 DataSnap 篇"內容中將討論如何開發高執行效率的 DataSnap 系統，這是因為在本書前面的內容中說明了如何開發基本的 DataSnap 應用系統，但由於從 Delphi 10.3 版本開始 DataSnap 又開始新增功能而且 Embarcadero 也開始調整 DataSnap 的內部實作方式，再加上本書前面的內容並沒有說明如何結合 FireDAC 和 DataSnap 在一起開發，在這些因素相加相乘的影響下本書有必要加入新的章節以便讓讀者瞭解如何開發新一代的 DataSnap 架構以便提供高效率的 DataSnap 伺服器並且連結移動客戶端。

在本書接下來的章節中將討論下面的內容：

- 使用 FireDAC 開發 DataSnap 應用系統
- 開發安全，高效率的 DataSnap 應用系統

當然讀者在前面章節學習到的各種 DataSnap 技術，例如 DataSnap 生命週期，過濾器，回叫機制等都可以繼續使用在本篇新的架構中。在稍後的第 9 章中將詳細說明如何使用 FireDAC 開發 DataSnap 應用系統，在讀者瞭解如何結合 FireDAC 和 DataSnap 之後第 10 章將進一步討論如何開發高效率的 FireDAC+DataSnap 應用系統。

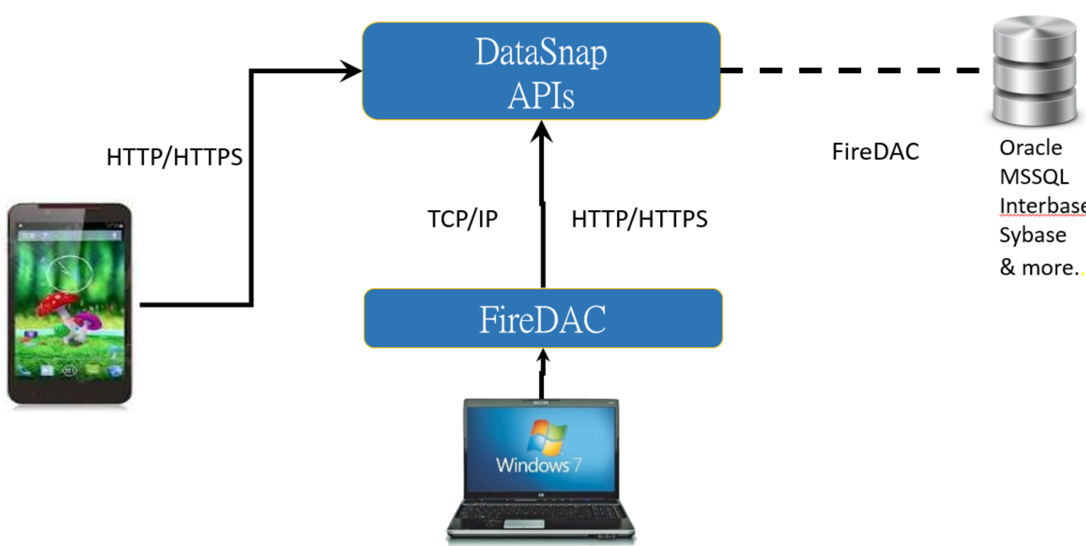
版權所有 請勿翻印

# 第9章 使用FireDAC開發DataSnap應用系統

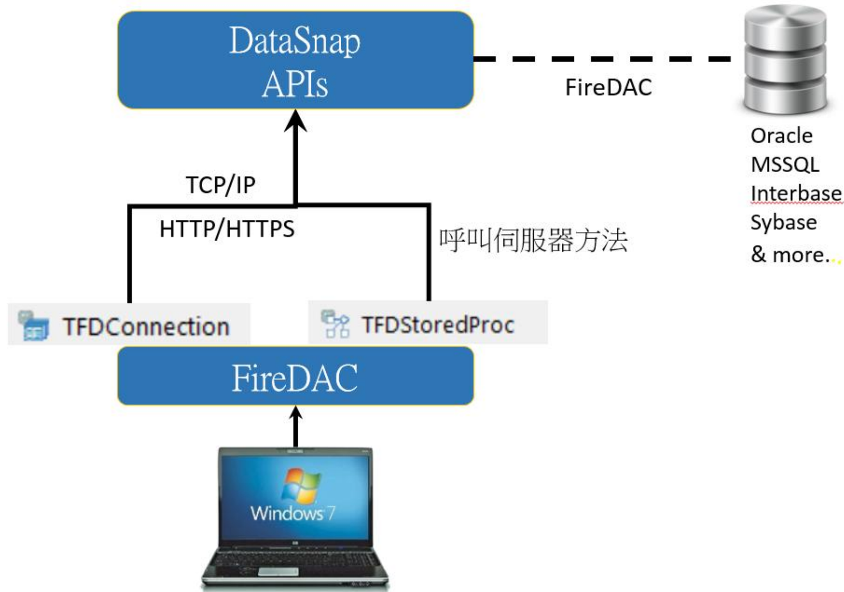
當使用 FireDAC 開發 DataSnap 應用系統時和使用前面章節討論的 dbExpress 有些不同，FireDAC 是把 DataSnap 伺服器當成 API 來呼叫，而不像 dbExpress 使用 IAppServer 介面。

FireDAC 客戶端可使用 TCP/IP 和 HTTP/HTTPS 連結使用 FireDAC 開發的 DataSnap 伺服器，FireDAC 客戶端以 API 的方式呼叫 FireDAC 的 DataSnap 伺服器，如下圖所示：

版權所有 請勿翻印



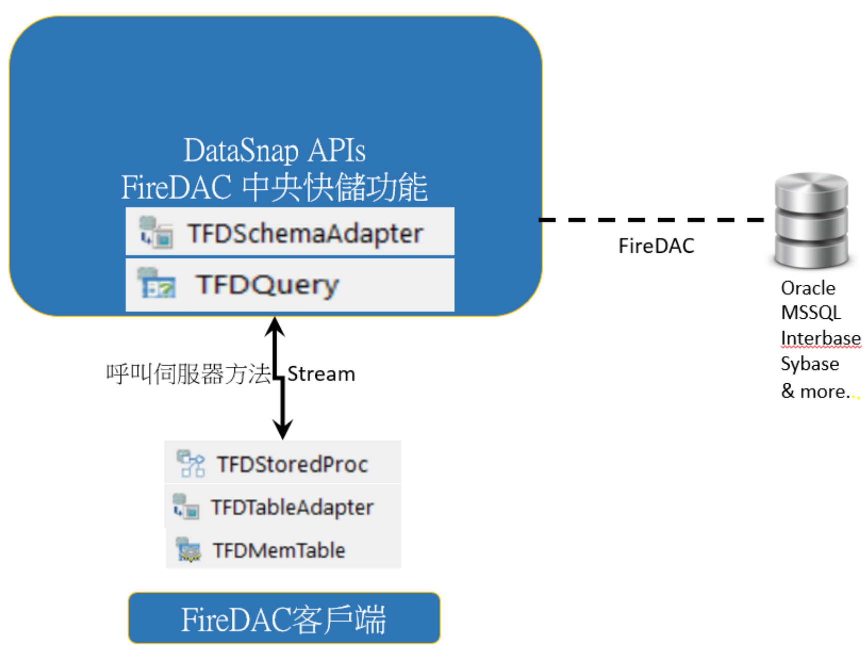

在 FireDAC 客戶端也是使用 TFDConnection 元件連結 FireDAC 的 DataSnap 伺服器並使用 TFDStoredProc 元件呼叫服器的 API，如下圖所示：




如果客戶端呼叫的 API 要回傳資料，那麼 FireDAC 會把資料以 Stream 的格式傳遞資料，如果要對資料進行包含異動的工作(CRUD)，那麼可搭配使用 FireDAC 的中央快儲功能來幫助程式師對資料進行異動。

在 DataSnap 架構中要使用 FireDAC 的中央快儲功能，程式師需要在伺服端使用 TFDSchemaAdapter 元件，並讓 TFDQuery 等元件連結到 TFDSchemaAdapter 元件。而在 FireDAC 客戶端則需使用 TFDTableAdapter 元件把在客戶端 TFDMemTable 元件中的資料從伺服端取回或是從客戶端把異動的資料更新回伺服端，如下圖所示：

版權所有 請勿翻印



因此要使用 FireDAC 開發 DataSnap 系統，程式師需要完成下列的步驟：

1. 開發使用 FireDAC 的 DataSnap 伺服器，並使用 FireDAC 中央快儲功能
2. 開發使用 FireDAC 的客戶端，使用 TFDConnection 連結伺服器
3. 使用 TFDStoredProc 元件呼叫服器的 API
4. 處理資料資料流(Stream Data)
5. 使用 TFDSchemaAdapter 元件把資料儲存到客戶端的 TFDMemTable 中
6. 使用 TFDStoredProc 元件呼叫服器的 API 把異動資料回傳給 FireDAC 的 DataSnap 伺服器，再藉由 FireDAC 中央快儲功能把資料更新回資料庫

在接下來的內容中將使用一個範例一步一步的說明如何開發一個 FireDAC DataSnap 架構可查詢資料並可異動多個資料表的資料。

## 9-1 開發可查詢的 FireDAC DataSnap 系統

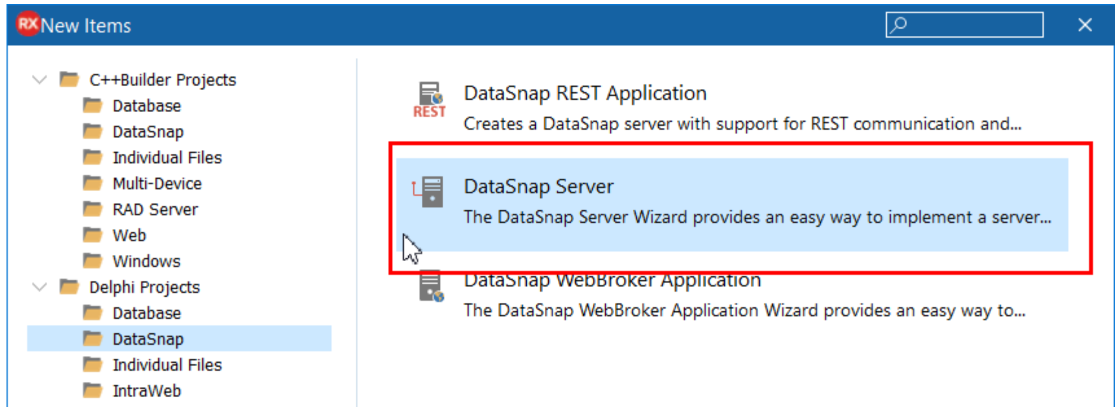
版權所有 請勿翻印

在本小節中將使用範例資料表 TBLTAIPEIHOTELS 來開發一個能在客戶端 PC 和手機中查詢台北市旅館資料的 DataSnaph 系統架構，在讀者瞭解如何使用 FireDAC 開發單一查詢的 DataSnap 系統架構之後，下一小節再說明多資料表的 CRUD 應用架構。

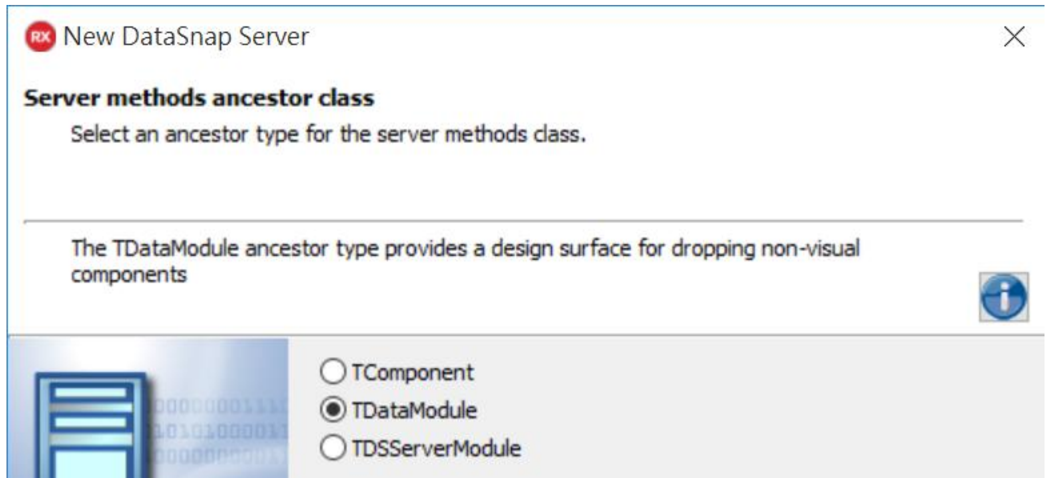
### 9-1-1 開發 DataSnap 伺服器

為了同時讓 PC 和手機客戶端都能使用 DataSnap 伺服器，我們可以使用下列任何種類的 DataSnap 架構，在這裡先讓我們使用下面的 DataSnap Server 做為範例 DataSnap 伺服器，稍後再說明如何使用 RESTful 架構的 DataSnap Server。

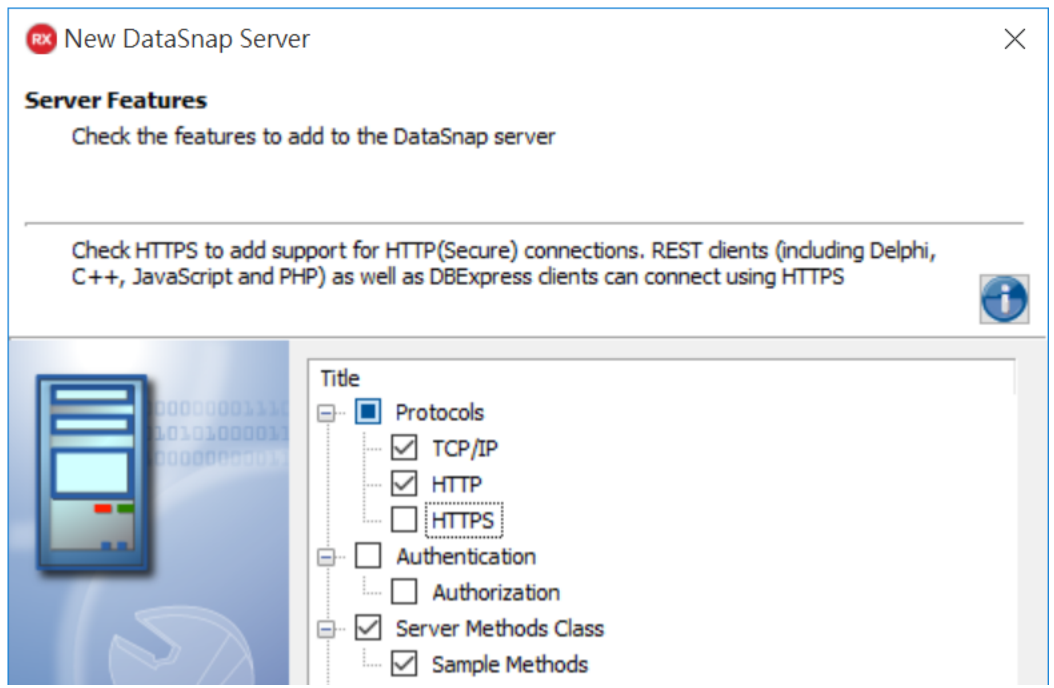
請在 IDE 下方的 New Items 對話盒中選擇如下的 DataSnap Server 圖像：




由於我們不需要使用 IAppServer 介面，因此可以選擇使用 TDataModule 做為伺服端 API 的類別：



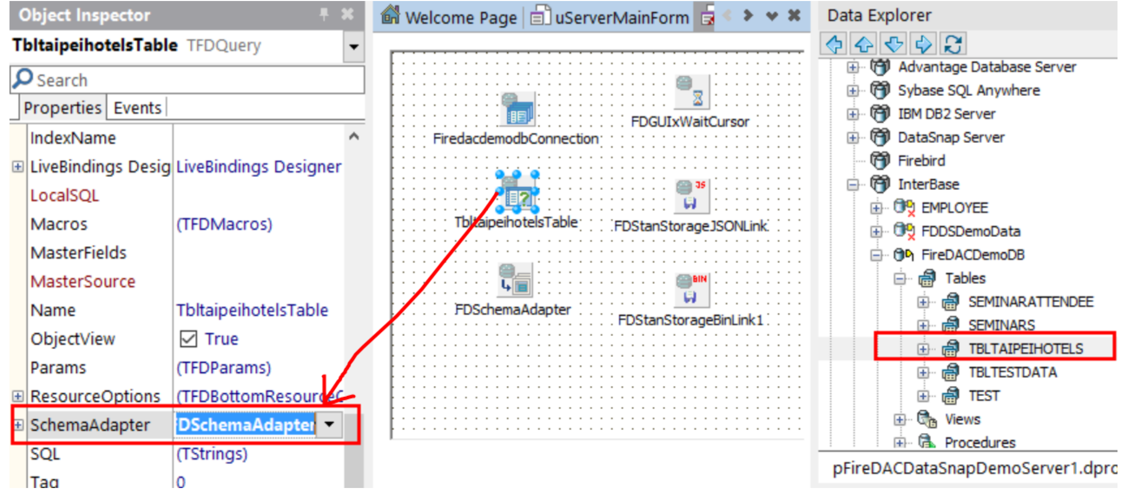

接著在下一個話盒中可選擇要支援的功能，例如我們可以讓此範例 DataSnap 伺服器同時支援 TCP/IP 和 HTTP 通訊協定：



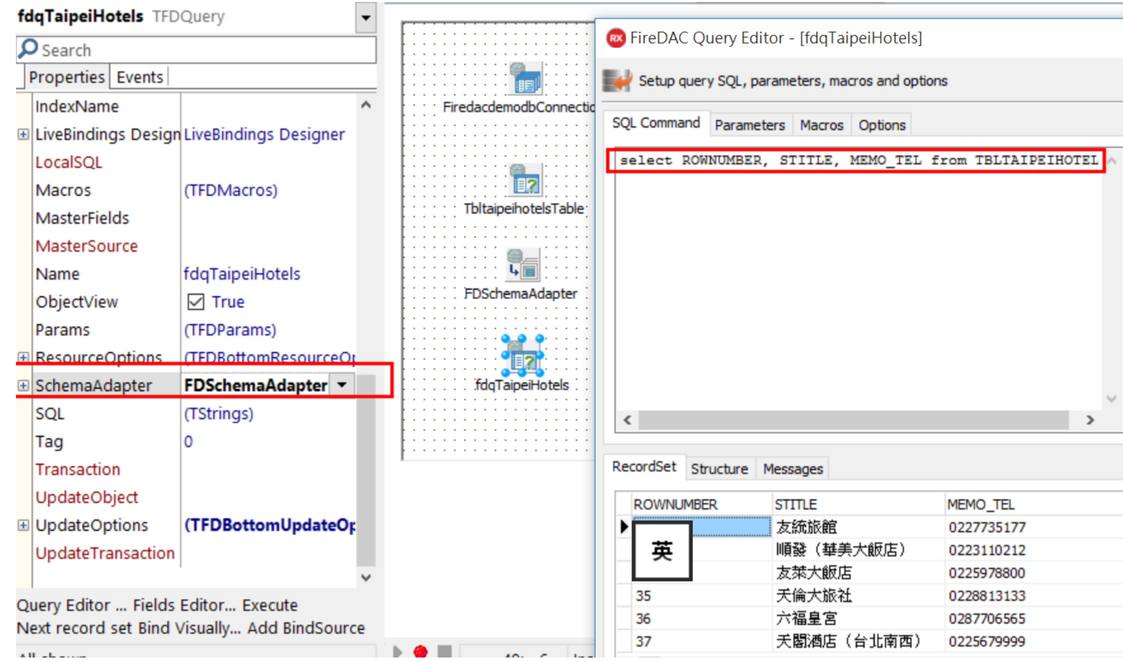

在 IDE 建立好了此範例專案後我們可以開啟專案中的 Server Method 程式單元，從 IDE 的 Data Explorer 中拖曳範例資料庫中的 TBLTAIPEIHOTELS 這個包含台北市旅館資料的資料表到資料模組中，並在其中加入如下其他的元件，由於我們要使用 FireDAC 的中央快儲功能，因此也加入了下方名為 ”FDSchemaAdapter” 的 TFDSchemaAdapter 元件，並且要設定 TbltaipeihotelsTable 元件的 SchemaAdapter 特性為 FDSchemaAdapter。

版權所有 請勿翻印

至於 FDStanStorageJSONLink 和 FDStanStorageBinLink1 元件則是為了讓 FireDAC 可處理 2 進位(TCP/IP)和 JSON(HTTP)格式的資料：



接著再放入另外一個 TFDQuery 元件: fdqTaipeiHotels，fdqTaipeiHotels 元件是為了讓客戶端查詢旅館名稱和電話資料使用的元件：




fdqTaipeiHotels 使用了如下的 SQL 命令：

版權所有 請勿翻印

```
select ROWNUMBER, STITLE, MEMO_TEL from TBLTAIPEIHOTELS
```

由於在前面我們選擇使用 TDataModule 做為伺服端 API 的類別，因此我們需要使用{$MethodInfo ON}/{$MethodInfo OFF}這對編譯器指令輸出 TDataModule 中的公共方法讓客戶端可以呼叫，並在其中加入下方 008 行的 GetTaipeiHotels()方法讓客戶端查詢旅館資訊。在 FireDAC 中要在伺服端和客戶端傳遞資料，我們只需要傳遞 TStream 型態的資料即可：

```
001    {$MethodInfo ON}
002    TsmFireDACDataSnapDemo1 = class(TDataModule)
003     ...
004    private
005      { Private declarations }
006    public
007      { Public declarations }
008      function GetTaipeiHotels: TStream;
009    end;
010    {$MethodInfo OFF}
```

最後我們需要實作 GetTaipeiHotels()方法，它的實作程式碼非常簡單，003 行先建立一個 TMemoryStream 物件，再開啟 fdqTaipeiHotels 元件取得資料，007 行把 fdqTaipeiHotels 元件中的資料集物件藉由 TMemoryStream 類別的 SaveToStream()方法把它拷貝到 TMemoryStream 物件中，在 008 行要把 TMemoryStream 物件中的資料串列流位置重置到開始的位置，最後把 TMemoryStream 物件回傳到客戶端即可：

```
001   function TServerMethods1.GetTaipeiHotels: TStream;
002   begin
003     Result := TMemoryStream.Create;
004     try
005       fdqTaipeiHotels.Close;
006       fdqTaipeiHotels.Open;
007       fdqTaipeiHotels.SaveToStream(Result, TFDStorageFormat.sfBinary);
008       Result.Position := 0;
009     except
010       raise;
011     end;
012   end;
```

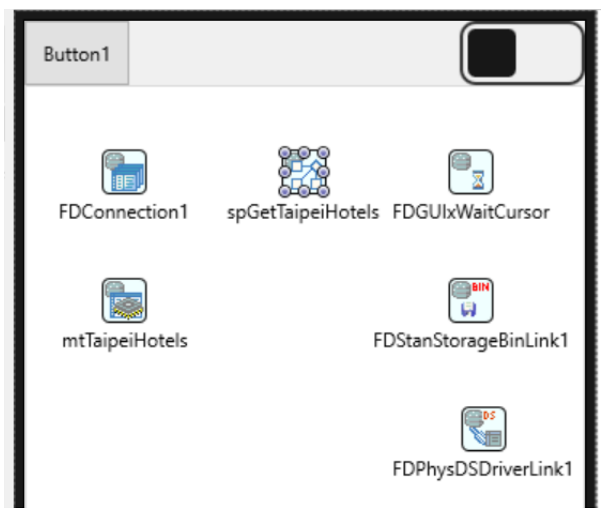
版權所有 請勿翻印

現在先讓我們暫時實作範例 DataSnap 伺服器致此，馬上就開始開發客戶端看看如何讓客戶端使用 FireDAC 從伺服器取得資料。

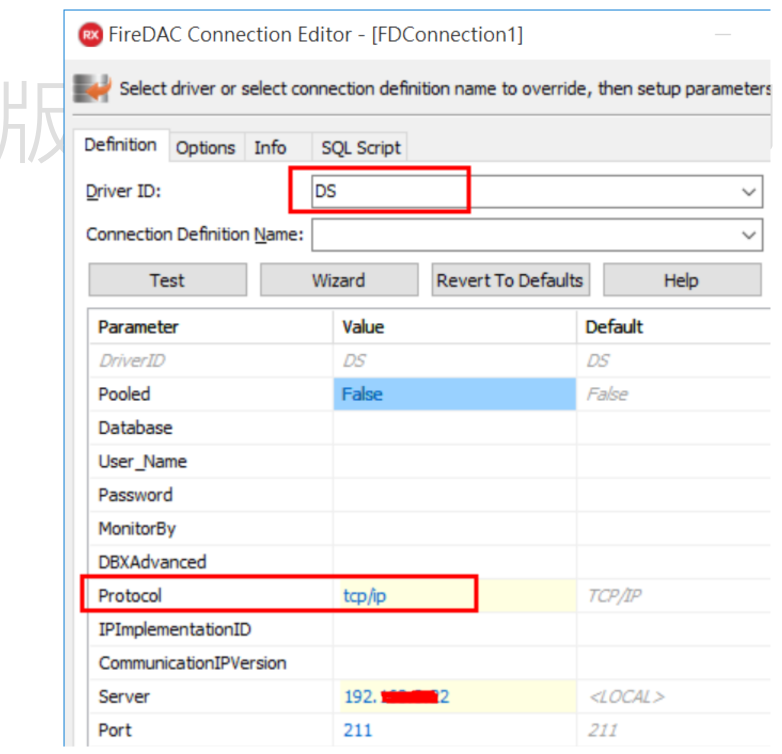
## 9-1-2 開發 DataSnap 客戶端

在使用 FireDAC 呼叫 DataSnap 伺服器的 API 時，程式師可使用 TFDStoredProc 元件，如果 DataSnap 伺服器的 API 回傳包含資料的 TStream 物件，那麼客戶端可以使用 TFDMemTable 元件來還原資料。

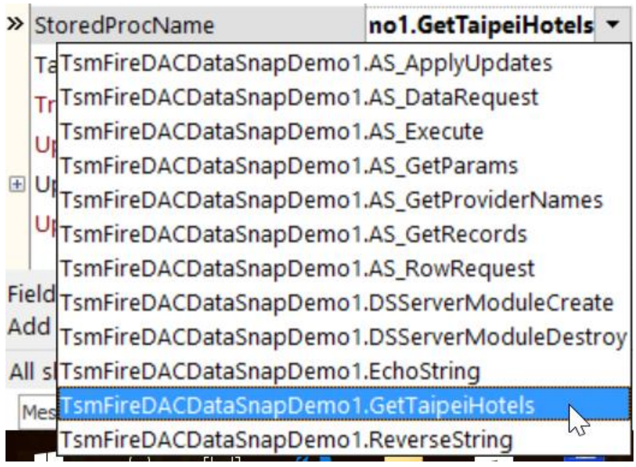
因此請在專案群組中建立一個 Multi-Device Application 客戶端專案，在主表單中加入如下的元件，spGetTheHotel 元件是稍後呼叫 DataSnap 伺服器 GetTaipeiHotels()方法使用的，而 mtTaipeiHotels 元件則是用來還原 GetTaipeiHotels()方法回傳的包含資料集的 TStream 物件。



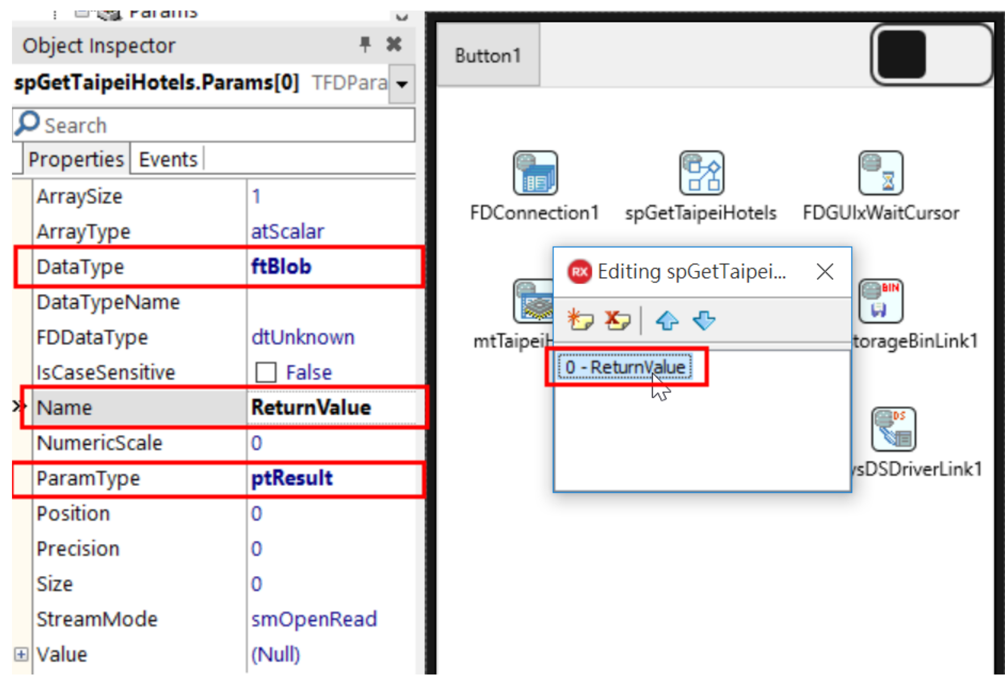

首先需要設定 FireDAC 的 TFDConnection 元件連結到 DataSnap 伺服器，請右擊上面的 FDConnection1 元件，在下面的對話盒中設定它使用的 Driver ID 為 DS，代表要使用 DataSnap 連結程式，再於 Protocol 欄位中設定使用 TCP/IP 並於 Server 欄位中設定 DataSnap 伺服器的 IP 位址：




正確設定 FDConnection1 後就可以點選主表單中的 spGetTheHotel 元件設，在物件檢視器中設定它的 StoredProcName 特性，從其中就可以看到 DataSnap 伺服器提供的服務方法，讓我們選擇要呼叫 GetTaipeiHotels()方法，如下所示：




選擇了 GetTaipeiHotels()方法後 spGetTheHotel 就會取得 DataSnap 伺服器 GetTaipeiHotels()方法的元資料，此時在物件檢視器中點選它的 Params 特性就可以看到如下所示 GetTaipeiHotels()方法會回傳一個參數，它的資料型態是 ftBlob：




現在就可以實作從客戶端向 DataSnap 伺服器查詢資料了，在主表單的 Button1 的 OnClick 事件中呼叫 GetTaipeiHotels()方法取得查詢資料，再呼叫 ShowTaipeiHotels()方法顯示資料：

```
procedure TfmMainForm.Button1Click(Sender: TObject);
begin
```

```
  GetTaipeiHotels;
  ShowTaipeiHotels;
end;
```

在 GetTaipeiHotels()方法 006 行只需要呼叫 spGetTaipeiHotels 的 ExecProc()方法 FireDAC 就可以呼叫遠方 DataSnap 伺服器中指定的服務方法(TsmFireDACDataSnapDemo1.GetTaipeiHotels)，呼叫成功之後 DataSnap 伺服器回傳的結果會儲存在 spGetTaipeiHotels 的第 1 個參數(ReturnValue)中，而且型態是 ftBlob。因此在 007 行建立一個 TStringStream 物件並把 spGetTaipeiHotels 的第 1 個參數內容做為建構元參數，如此一來 TStringStream 物件的內容就是回傳的結果，010 行把 TStringStream 物件包含的資料流位置設定到起始位置，012 行再使用 mtTaipeiHotels 元件的 LoadFromStream()方法從 TStringStream 物件中讀取資料流並還原成資料集物件即可：

```
001   procedure TfmMainForm.GetTaipeiHotels;
002   var
003     LStringStream: TStringStream;
004   begin
005     lStart := Now;
006     spGetTaipeiHotels.ExecProc;
007     LStringStream :=
TStringStream.Create(spGetTaipeiHotels.Params[0].asBlob);
008     try
009       if (LStringStream <> nil) then
010       begin
011         LStringStream.Position := 0;
012         mtTaipeiHotels.LoadFromStream(LStringStream,
TFDStorageFormat.sfBinary);
013       end;
014     finally
015       LStringStream.Free;
016       lEnd := Now;
017     end;
018   end;
```

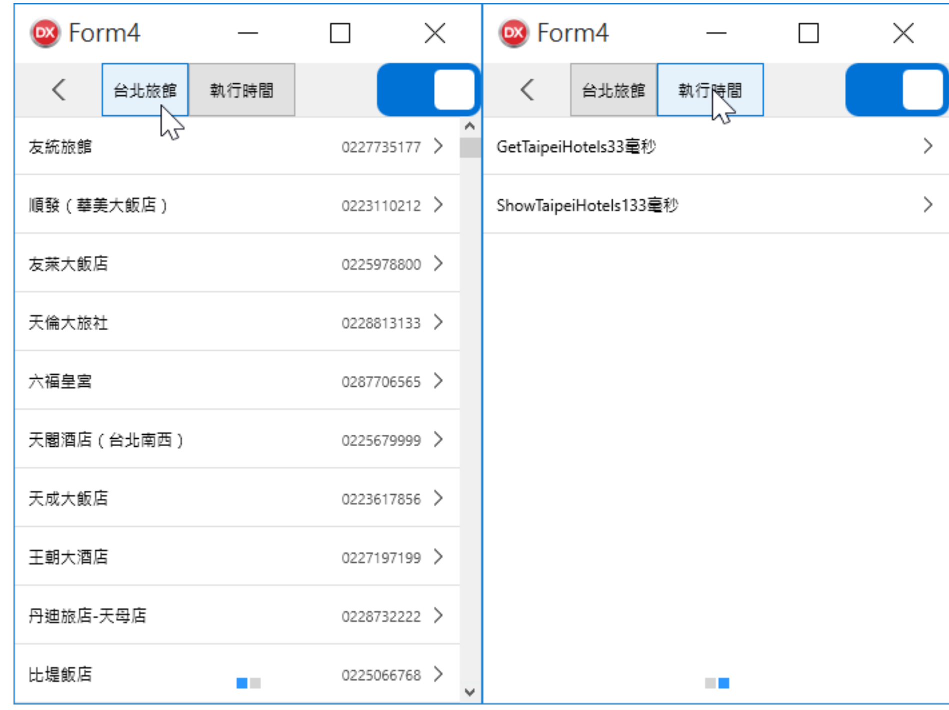
版權所有 請勿翻印

接著 ShowTaipeiHotels()方法就可以使用我們已經熟悉的方法把資料從 TFDMemTable 元件中顯示在主表單的 TListView 元件中：

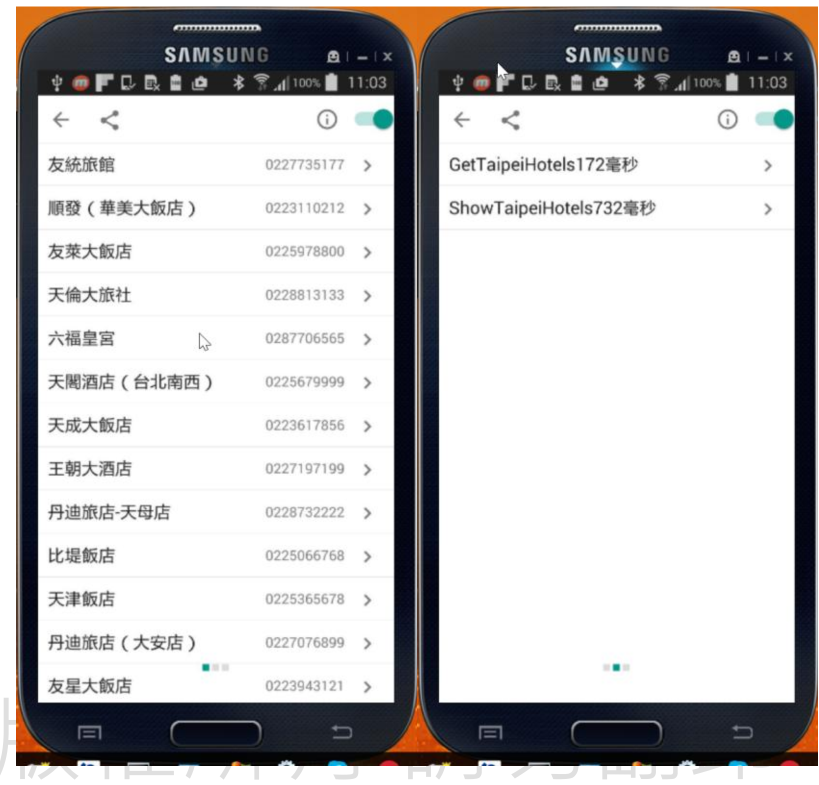
```
procedure TfmMainForm.ShowTaipeiHotels;
var
  lvi : TListViewItem;
begin
  lStart := Now;
  lvTaipeiHotels.Items.Clear;
  mtTaipeiHotels.First;
  while (not mtTaipeiHotels.Eof) do
  begin
    lvi := lvTaipeiHotels.Items.Add;
    lvi.Text := mtTaipeiHotels.FieldByName('STITLE').AsString;
    lvi.Detail := mtTaipeiHotels.FieldByName('MEMO_TEL').AsString;
    mtTaipeiHotels.Next;
  end;
  lEnd := Now;
end;
```

版權所有 請勿翻印

現在如果編譯並執行此範例客戶端就可以看到它在筆者的 Windows 10 中以 Win32 程式執行並從 DataSnap 伺服器成功查詢到台北市旅館資料：



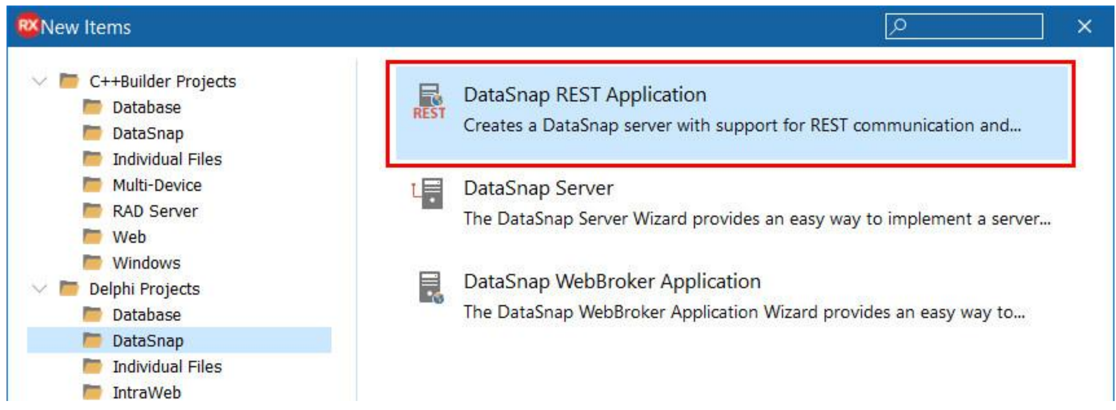
如果此時把此客戶端部署到筆者的 Samsung S4 手機中可以可以看到也能正常執行，而且從筆者的虛擬 Windows 10 的 DataSnap 伺服器中取得 396 筆旅館資料也只要 172 毫秒：



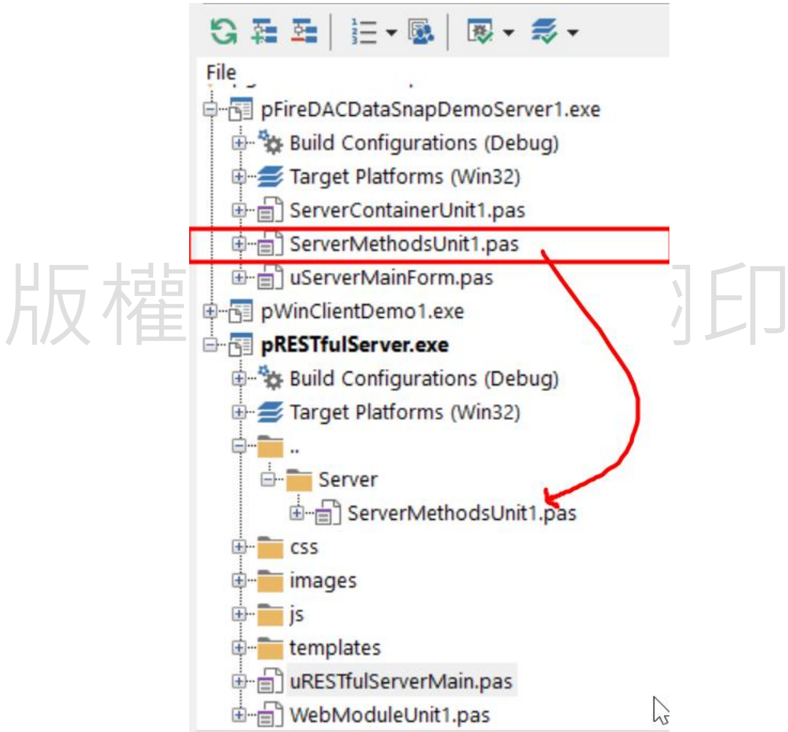

## 9-1-3 開發 RESTful DataSnap 伺服器

使用 TCP/IP 連結 DataSnap 伺服器和客戶端僅限於客戶端是使用 Delphi 或是 C++Builder 開發的，如果我們希望客戶端可以其他語言或是工具開發的，那麼我們可以使用 RESTful 架構，讓 DataSnap 伺服器使用 HTTP/HTTPS 通訊協定並使用 JSON 格式傳遞資料。

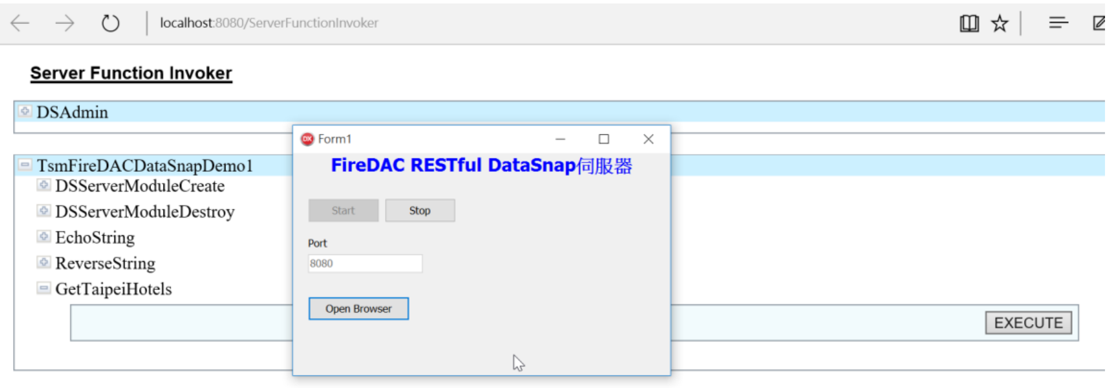
由於在前面建立 DataSnap 伺服器時伺服器提供的服務方法是實作在 TDataModule 中，因此也可以使用在 RESTful 的 DataSnap 伺服器中。因此請在專案群組中再建立一個如下所示的 DataSnap REST Application 專案：




把此專案以 pRESTfulServer 名稱儲存，接著在此專案中加入前面第 1 個 DataSnap 伺服器專案中的 SeverMethodsUnit1 程式單元到此新專案中：




現在我們只需要編譯此新專案執行即可：




**Executed:** TsmFireDACDataSnapDemo1.GetTaipeiHotels

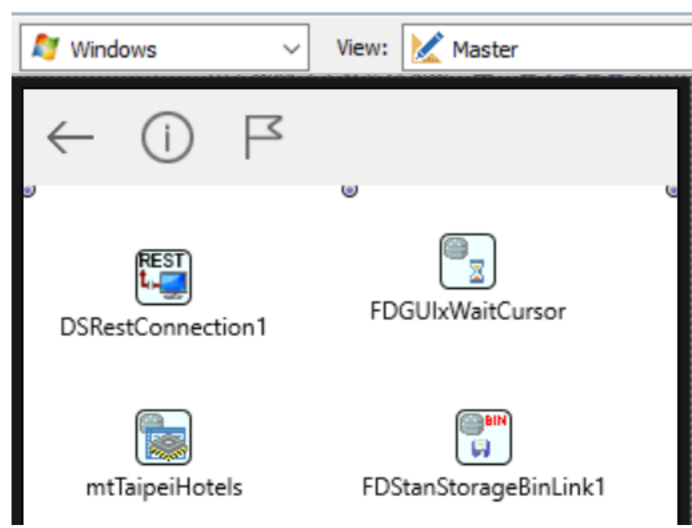
"ADBS\u000f\u0000ô\u0003\fk\u0000\u0000ÿ\u0000\u0001\u0000\u0001ÿ\u0002ÿ\u0003\u0004\u0000\u001e\u0000\u0000
\u0000f\u0000d\u0000q\u0000T\u0000a\u0000i\u0000p\u0000e\u0000i\u0000H\u0000o\u0000t\u0000e\u0000l\u0000s\u0000\u0005\u0000
\u001e\u0000\u0000


由於這是一個 RESTful DataSnap 伺服器，因此從上圖可以看到筆者使用 Windows 10 的 Edge 瀏覽器呼叫 ShowTaipeiHotels() 方法並取得資料 (Unicode 格式)，接下來就可以開發一個手機客戶端來查詢資料了。

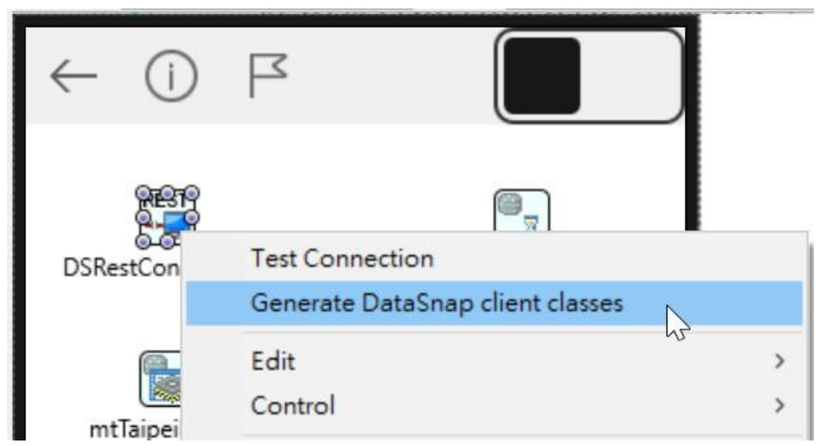
版權所有 請勿翻印

## 9-1-4 開發 RESTful 手機客戶端

再於專案群組中建立一個 Muli-Device Applicartion 專案，要使用 RESTful 架構連結 RESTful DataSnap 伺服器，我們可以使用 TDSRestConnection 元件，因此在主表單中加入 TDSRestConnection 和 TFDMemTable 元件：



右擊 TDSRestConnection 元件在突顯示選單中選擇 Generate DataSnap client classe 選項：




TDSRestConnection 元件便會產生伺服端服務的類別，GetTaipeiHotels() 方法便在其中：

```
TsmFireDACDataSnapDemo1Client = class(TDSAdminRestClient)
private
  ...
public
  ...
  function EchoString(Value: string; const ARequestFilter: string = ''): string;
  function ReverseString(Value: string; const ARequestFilter: string = ''): string;
  function GetTaipeiHotels(const ARequestFilter: string = ''): TStream;
  function GetTaipeiHotels_Cache(const ARequestFilter: string = ''):
IDSRestCachedStream;
end;
```

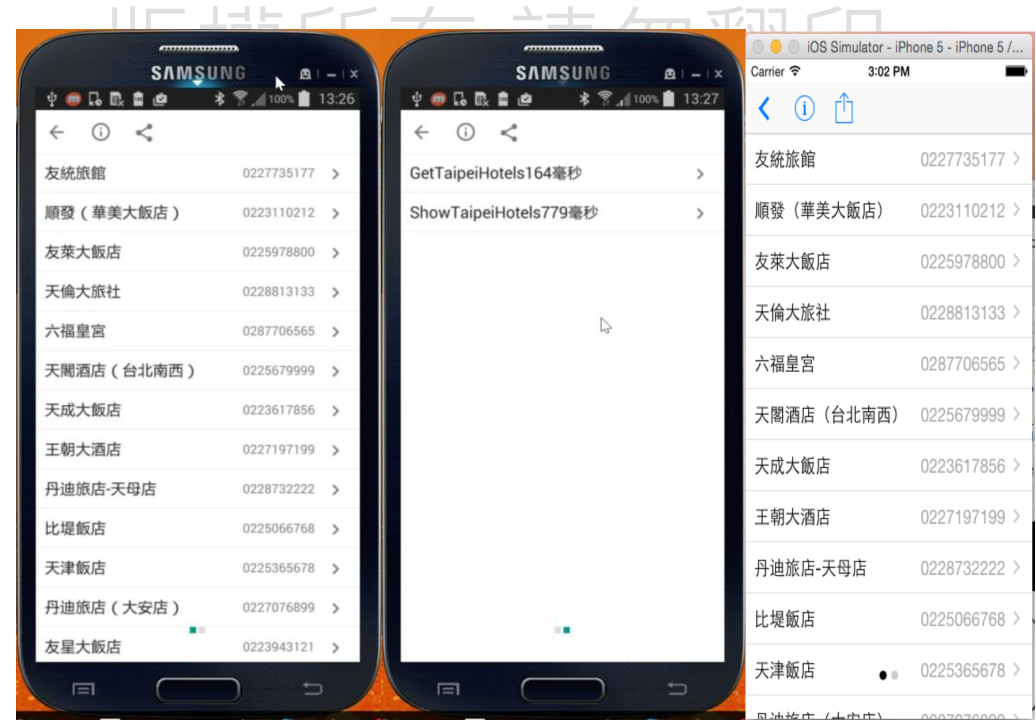
版權所有 請勿翻印

接著使用如下的程式碼呼叫 RESTful DataSnap 伺服器中的服務方法，請注意下面的程式碼和前面幾乎一樣，除了在 008 行是建立剛才自動產生的 TsmFireDACDataSnapDemo1Client 物件並把 TDSRestConnection 元件傳入做為建構元的參數：

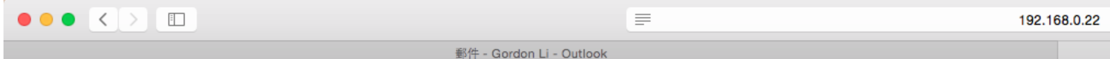
```
001   procedure TfmMainForm.GetTaipeiHotels;
002   var
003     LStream: TStream;
004     aServer : TsmFireDACDataSnapDemo1Client;
005   begin
006     lStart := Now;
```

```
007
008     aServer := TsmFireDACDataSnapDemo1Client.Create(Self.DSRestConnection1);
009     try
010       LStream := aServer.GetTaipeiHotels();
011       if (LStream <> nil) then
012       begin
013         LStream.Position := 0;
014         mtTaipeiHotels.LoadFromStream(LStream, TFDStorageFormat.sfBinary);
015       end;
016     finally
017       LStream.Free;
018       lEnd := Now;
019       aServer.Free;
020     end;
021   end;
```

編譯並執行此範例 RESTful 客戶端，我們可以看到下面的結果畫面，這個範例 RESTful 客戶端可以成功執行在 Android 和 iPhone Simulator 中：




由於這是個 RESTful 的架構，因此筆者也可以使用 Mac OSX 中的 Safari 瀏覽器呼叫範例 RESTful DataSnap 伺服器並取得查詢結果(Unicode 格式)：




## 9-1-5 如何更新旅館資料

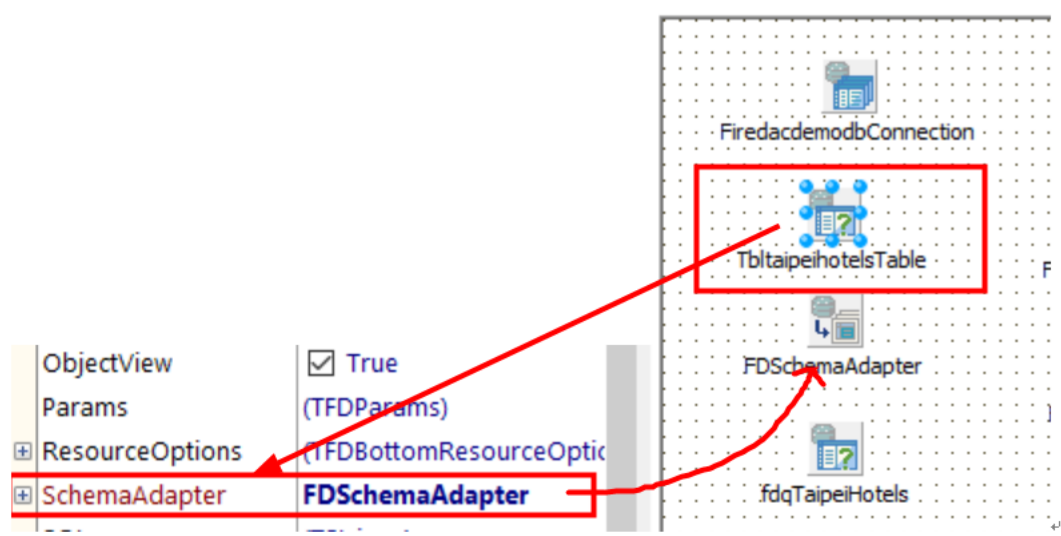
那麼是否可以在 PC 或是手機端對資料進行異動呢？當然可以，而且也很簡單，接下來讓我們繼續修改範例讓我們在手機端可修改旅館資料。

### 修改 FireDAC DataSnap 伺服器

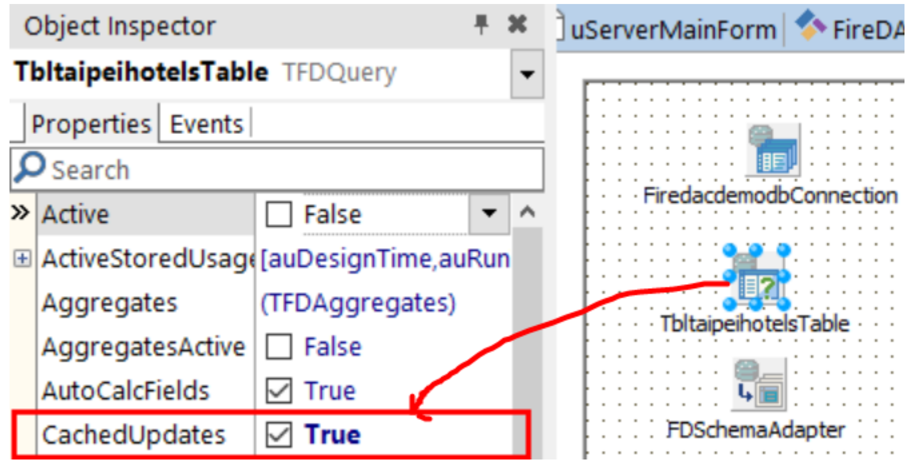
FiriDAC 的中央快儲功能可讓程式師在 DataSnap 架構中對資料進行 CRUD 的工作，現在我們想讓稍後的手機端可在遠端修改旅館資料，因此我們需要先在伺服器中加入可把資料異動回資料庫的服務方法。

版權所有 請勿翻印

回到第 1 個範例 DataSnap 伺服器在 ServerMethodsUnit1 程式單元確定 TbltaipeihotelsTable 元件的 SchemaAdapter 設定為資料模組中的 FDSchemaAdapter：




而且一定要設定 TbltaipeihotelsTable 元件使用快儲功能，即設定它的 CachedUp[dates 特性值為 True：




設定 TbltaipeihotelsTable 元件的 SQL 特性值為如下的 SQL 命令：

```
SELECT * FROM TBLTAIPEIHOTELS where ROWNUMBER = :ROWNUMBER
```

現在讓我們加入一個修改資料的功能，當使用者查詢了旅館資料後可在 TListView 元件中點選任一旅館然後我們允許使用修改旅館名稱，電話或是價格資訊。因此在 DataSnap 伺服器需要 2 個新的方法，GetTheHotel()可取得使用者在 TListView 元件中點選的旅館，而 PostHotel()方法則可把手機端異動的資訊更新回資料庫中：

版權所有 請勿翻印

```
  function GetTaipeiHotels: TStream;
  function GetTheHotel(const sHotelID : String) : TStream;
  function PostHotel(AStream: TStream) : Integer;
```

GetTheHotel() 方法實作很簡單，只是把客戶端傳來的旅館 ID 帶入 TbltaipeihotelsTable 元件的 SQL 特性值的參數中，再開啟 TbltaipeihotelsTable 元件並存入 TMemoryStream 物件中再回傳到客戶端：

```
function TsmFireDACDataSnapDemo1.GetTheHotel(const sHotelID: String): TStream;
begin
  Result := TMemoryStream.Create;
  try
    TbltaipeihotelsTable.Close;
    TbltaipeihotelsTable.Params.ParamByName('ROWNUMBER').Value := sHotelID;
    TbltaipeihotelsTable.Open;
    FDSchemaAdapter.SaveToStream(Result, TFDStorageFormat.sfBinary);
    Result.Position := 0;
  except
    raise;
```

```
  end;
end;
```

PostHotel()方法是把客戶端異動的資料更新回資料庫，在稍後實作手機端時會看到手機是把客戶端 TFDMemTable 元件的 Delta 傳回伺服器，因此 PostHotel()方法先在 027 行呼叫 CopyStream()方法把客戶端傳來的資料流先拷貝到 TMemoryStream 物件中，028 行重置資料流位置到起始位置，031 行藉由 TFDSchemaAdapter 類別的 LoadFromStream()方法把資料流還原到 TbltaipeihotelsTable 元件中，最後在 032 行呼叫 TFDSchemaAdapter 類別的 ApplyUpdates()方法，TFDSchemaAdapter 即會根據傳來的 Delta 和元資料自動產生 SQL 命令把異動的資料更新回資料庫：

```
001   function CopyStream(const AStream: TStream): TMemoryStream;
002   var
003     LBuffer: TBytes;
004     LCount: Integer;
005   begin
006     Result := TMemoryStream.Create;
007     try
008       SetLength(LBuffer, 1024 * 32);
009       while True do
010       begin
011         LCount := AStream.Read(LBuffer, Length(LBuffer));
012         Result.Write(LBuffer, LCount);
013         if LCount < Length(LBuffer) then
014           break;
015       end;
016     except
017       Result.Free;
018       raise;
019     end;
020   end;
021
022   function TsmFireDACDataSnapDemo1.PostHotel(AStream: TStream) : Integer;
023   var
024     LMemStream: TMemoryStream;
025     LErrors: Integer;
026   begin
```

版權所有 請勿翻印

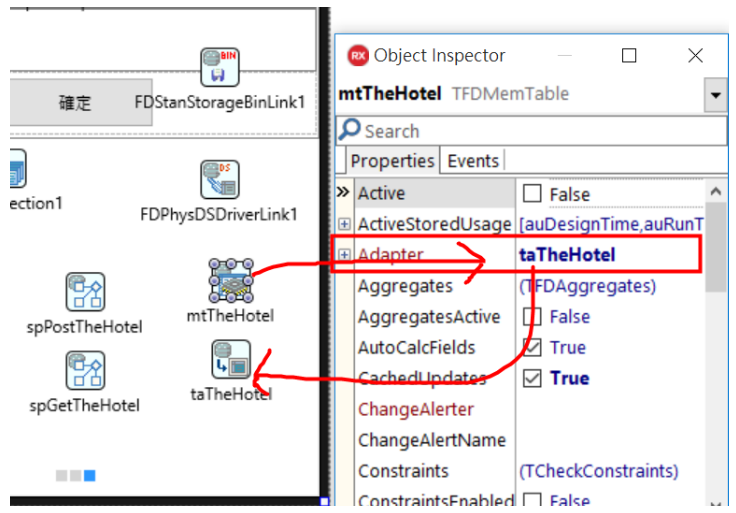
```
027      LMemStream := CopyStream(AStream);
028      LMemStream.Position := 0;
029
030      try
031        FDSchemaAdapter.LoadFromStream(LMemStream, TFDStorageFormat.sfBinary);
032        Result := FDSchemaAdapter.ApplyUpdates;
033      finally
034        LMemStream.Free;
035      end;
036    end;
```

## 修改 FireDAC DataSnap 手機客戶端

再回到前面第 2 個範例客戶端專案，在主表單中加入 **spGetTheHotel** 元件呼叫伺服器中的 **GetTheHotel()** 方法取得使用者點選的旅館並還原在 **mtTheHotel** 元件中。而 **taTheHotel** 是 **TFDTableAdapter** 元件，它必須連結到 **taTheHotel** 元件以使用中央快儲功能，最後 **spPostTheHotel** 元件則是把客戶端異動的資料藉由呼叫伺服器中的 **PostHotel** 方法更新資料：

版權所有 請勿翻印

接著在物件檢視器中設定 **taTheHotel** 的 **Adapter** 特性值為 **taTheHotel** 元件：




現在就可以開始實作客戶端程式碼，首先在 TListView 元件的 OnItemClick 事件中呼叫 GetTheHotel()方法向伺服器查詢使用者在 TListView 元件中點選的旅館，再呼叫 DisplayTheHotel()方法把點選的旅館顯示出來並準備編輯：

```
001   procedure TfmMainForm.lvTaipeiHotelsItemClick(const Sender: TObject;
002     const AItem: TListViewItem);
003   begin
004     if (GetTheHotel(AItem.Text) ) then
005     begin
006       DisplayTheHotel;
007       EditTheHotel;
008     end;
009   end;
```

版權所有 請勿翻印

由於 GetTheHotel()方法使用了和前面說明 GetTaipeiHotels()方法一樣的技巧，因此就不再贅述了，請讀者自行參考範例程式碼。

這裡的重點是當使用者修改完旅館資料並點選主表單中的”確定”按鈕時，它呼叫 PostTheHotel()方法把資料更新回資料庫：

```
procedure TfmMainForm.Button4Click(Sender: TObject);
begin
  PostTheHotel;
end;
```

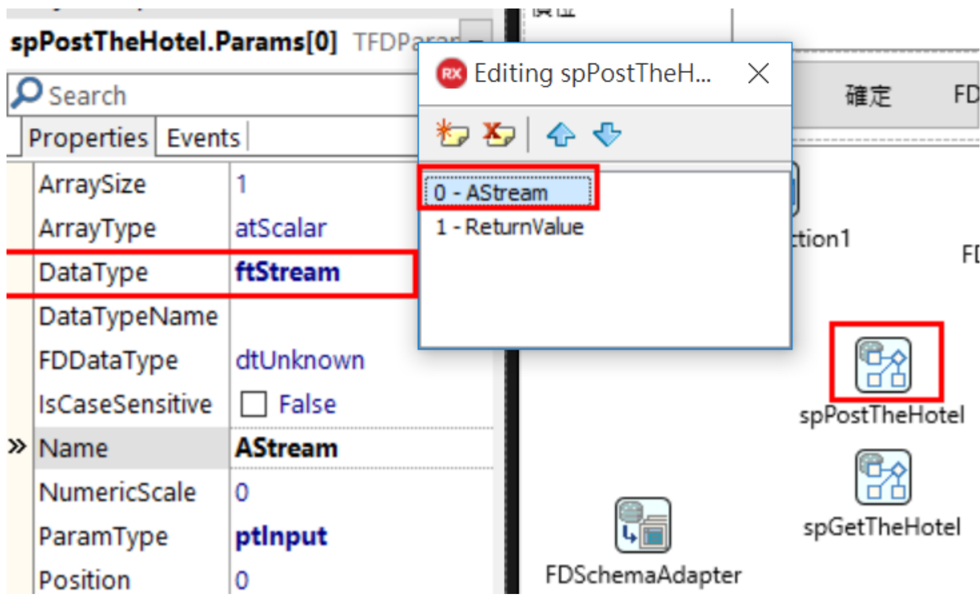
PostTheHotel()方法先把使用者輸入的資料更新回 mtTheHotel 元件中，再於 016 行建立一個 TMemoryStream 物件，018 行非常重要，它設定

FDSchemaAdapter 的 ResourceOptions. StoreItems 為[siDelta, siMeta]，這代表在客戶端是把 mtTheHotel 元件的 Delta 回傳，而 siMeta 代表要儲存資料集的元資料，以便伺服器中的 FDSchemaAdapter 元件可以根據元資料來產生更新資料的 SQL 命令。接著把 TMemoryStream 物件中的資料流寫入 spPostTheHotel 元件的參數中，最後藉由 spPostTheHotel 元件呼叫伺服器中的 PostHotel()方法：

```
001   procedure TfmMainForm.PostTheHotel;
002   var
003     LMemStream: TMemoryStream;
004     I: integer;
005     LDataSet: TDataSet;
006   begin
007     lStart := Now;
008
009     mtTheHotel.Edit;
010     mtTheHotel.FieldByName('ROWNUMBeR').Value :=
mtTheHotel.FieldByName('ROWNUMBeR').Value;
011     mtTheHotel.FieldByName('STITLE').Value := edtTitle.Text;
012     mtTheHotel.FieldByName('MEMO_TEL').Value := edtPhone.Text;
013     mtTheHotel.FieldByName('MEMO_COST').AsString := edtPrice.Text;
014     mtTheHotel.Post;
015
016     LMemStream := TMemoryStream.Create;
017     try
018       FDSchemaAdapter.ResourceOptions.StoreItems := [siDelta, siMeta];
019       FDSchemaAdapter.SaveToStream(LMemStream, TFDStorageFormat.sfBinary);
020       LMemStream.Position := 0;
021       spPostTheHotel.Params[0].asStream:= LMemStream;
022       spPostTheHotel.ExecProc;
023     except
024       On E: Exception do
025         raise Exception.Create(E.Message);
026     end;
027     lEnd := Now;
028   end;
```

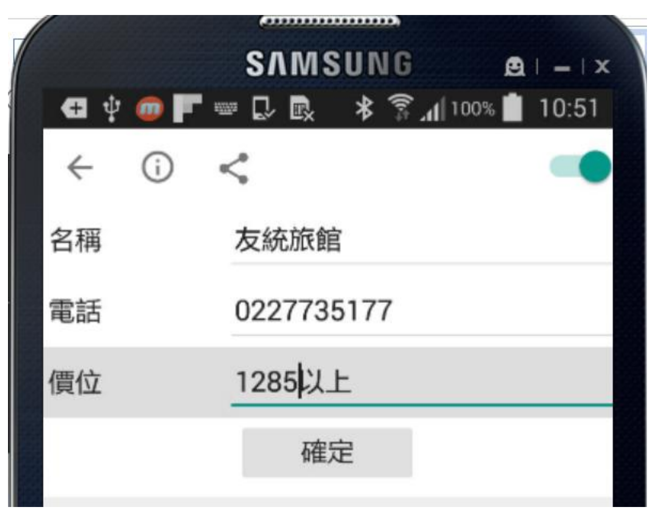
版權所有 請勿翻印

在這裡要注意的是 spPostTheHotel 元件的第 1 個參數的資料型態一定要設定為 ftStream 如下所示，否則資料集無法以正確的格式回傳回伺服器，如此一來也就無法正確更新回資料庫中。




現在就可以再次編譯手機客戶端執行，從下圖中可以看到我們在 S4 中點選了”友統旅館”，並在編輯頁面中修改了它的價位資料：

版權所有 請勿翻印




再點選”確定”按鈕把資料更新回去，從下圖可以看到速度很理想：

而且異動的資料果然成功的由 DataSnap 伺服器藉由 FireDAC 的中央快儲功能更新回 InterBase 了：

uServerMainForm | FireDACDemoDB: View TBLTAIPEIHOTELS

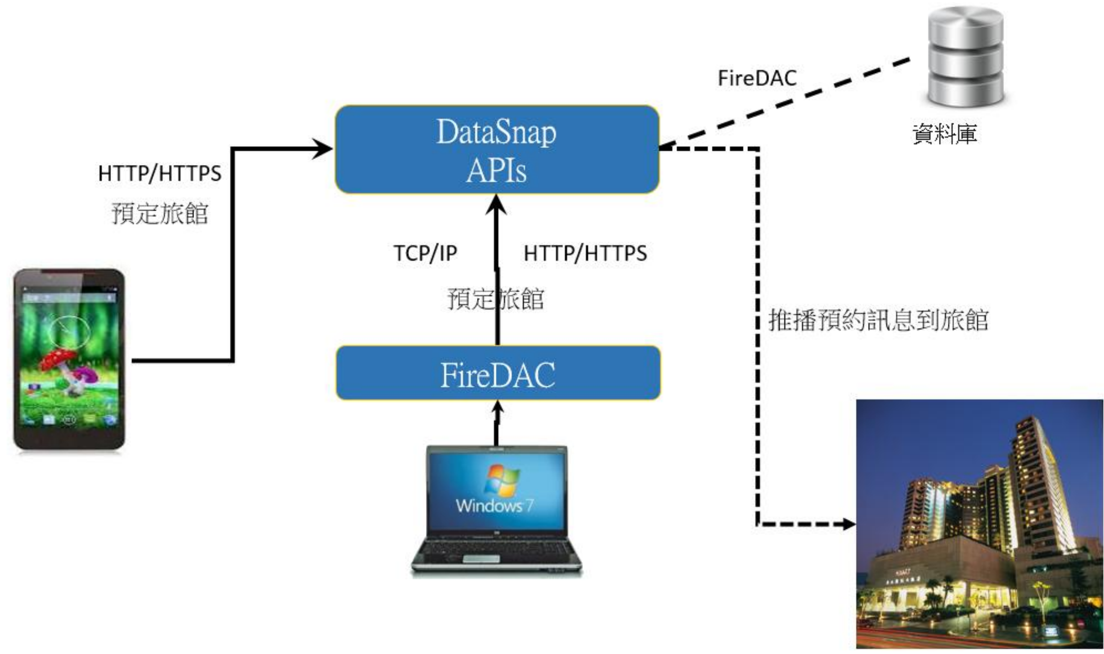
SELECT * FROM TBLTAIPEIHOTELS

| ROWNUMBER | STITLE | MEMO_COST |
|---|---|---|
| 33 | 友統旅館 | 1285以上 |
| 47 | 友華賓館（貝斯特旅店） | 1000以上 |
| 34 | 友萊大飯店 | 1550以上 |
| 38 | 天成大飯店 | 4500以上 |
| 42 | 天津飯店 | 1500以上 |
| 35 | 天倫大旅社 | 1000以上 |
| 57 | 天閣酒店 | 7600以上 |

## 9-2 開發可異動多資料表的 FireDAC DataSnap 系統

在 9-1 小節中本書說明了如何在 FireDAC 的 DataSnap 架構中更新一個資料表的資料，在本小節將說明如何在 FireDAC 的 DataSnap 架構中更新 Master/Detail 的資料。讓我們試著在前面的範例 DataSnap 架構中加入可讓使用者在查詢到想要的旅館後可使用手機預定旅館的應用。

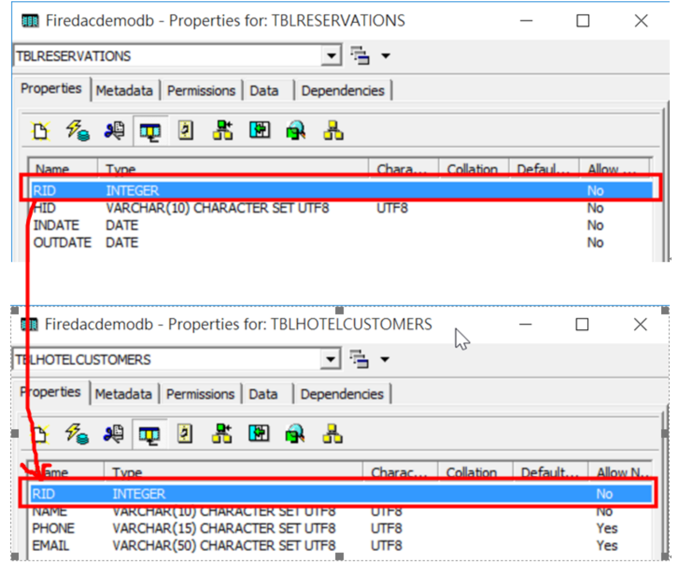
例如下圖展示了本小節將實作的架構，我們甚至可以再結合推播功能把 DataSnap 伺服器中預定的資料直接推播到預定旅館的實際資訊系統：



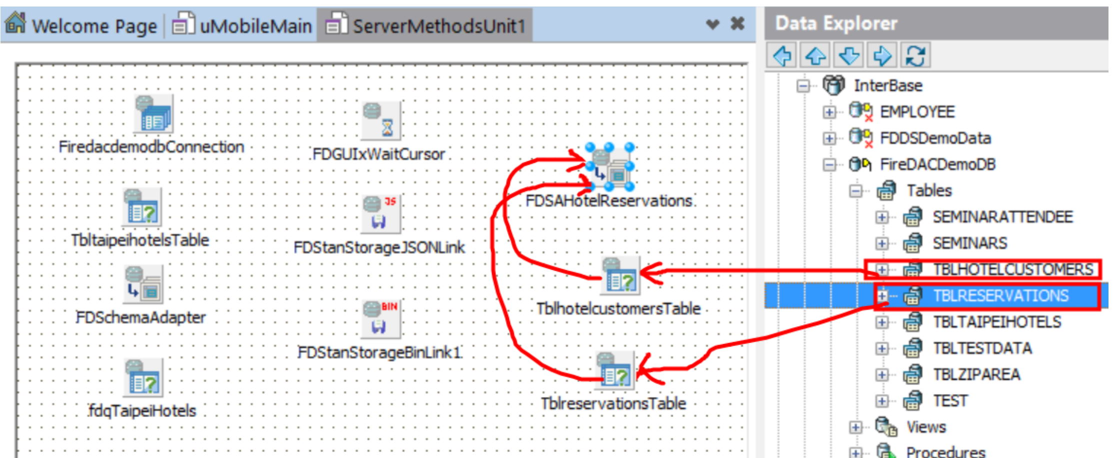

## 9-2-1 修改 FireDAC DataSnap 伺服器

在範例資料庫中 FIREDACDEMODB.GDB 中有 2 個簡單的範例資料表 TBLRESERVATIONS 和 TBLHOTELCUSTOMERS，TBLRESERVATIONS 將儲存預定旅館資訊而 TBLHOTELCUSTOMERS 則儲存預定客戶資料，這 2 個資料表是藉由 RID 這個欄位關連，如下所示：

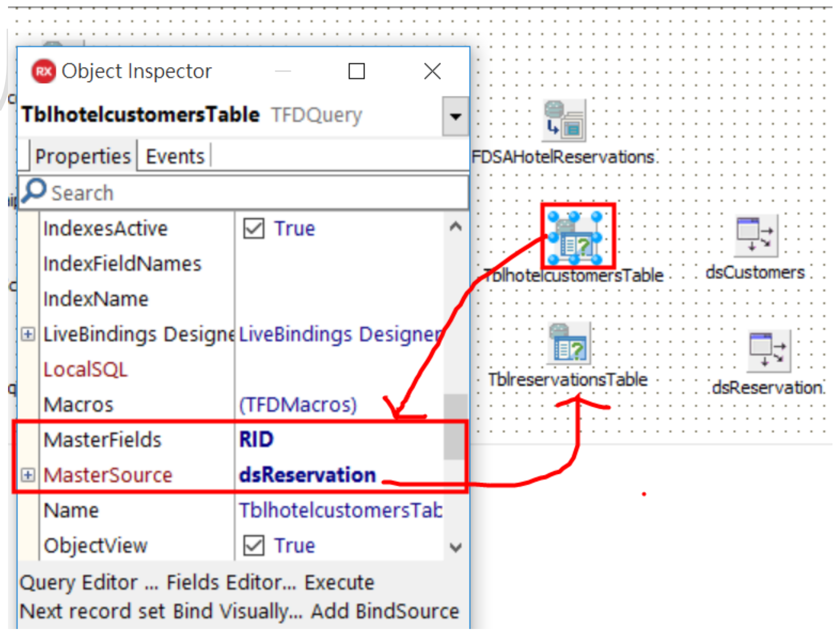
版權所有 請勿翻印




因此請回到範例 DataSnap 伺服器專案，開啟 ServerMethodsUnit1 程式單元，放入 TFDSchemaAdapter 元件 FDSAHotelReservations，再從 Data Explorer 中拖曳這 2 個資料表到其中，並設定拖入的 2 個 TFDQuery 元件的 SchemaAdapter 特性值為 FDSAHotelReservations，如下所示：




再放入 2 個 TDataSource 元件分別連結到拖入的 2 個 TFDQuery 元件，並把 TblhotelcustomersTable 元件的 MasterSource 和 MasterFields 設定如下以便和 TblreservationsTable 形成 Master/Detail 的關係，如下所示：




由於在稍後使用 FireDAC 的中央快儲功能時我們只是希望 FireDAC 把這 2 個資料表的元資訊傳遞到客戶端以便客戶端的 TFDMemoryTable 能夠在客戶端新增資料，因此為了避免把所有這 2 個資料表的預定資料傳遞到客戶端，我們在 TblreservationsTable 的 SQL 特性值中使用有參數的 SQL 命令：

```
SELECT * FROM TBLRESERVATIONS where RID = :RID
```

接著在伺服器加入如下的 2 個服務方法：

```
function PostHotelReservation(AStream: TStream) : Integer;
function GetReservation(const iRID : Integer) : TStream;
```

GetReservation()方法的目的是讓客戶端呼叫並把 TBLRESERVATIONS 和 TBLHOTELCUSTOMERS 這 2 個資料表的元資訊傳遞回客戶端，在稍後實作客戶端會看到客戶端會傳遞 -1 給 GetReservation() 方法讓 TblreservationsTable 不會把任何預定資料傳回而只是傳回元資訊：

```
001   function TsmFireDACDataSnapDemo1.GetReservation(const iRID : Integer):
TStream;
002   begin
003     Result := TMemoryStream.Create;
004     try
005       TblreservationsTable.Close;
006       TblreservationsTable.Params[0].Value := iRID;
007       TblreservationsTable.Open;
008       TblhotelcustomersTable.Close;
009       TblhotelcustomersTable.Open;
010       FDSAHotelReservations.SaveToStream(Result, TFDStorageFormat.sfBinary);
011       Result.Position := 0;
012     except
013       raise;
014     end;
015   end;
```

版權所有 請勿翻印

PostHotelReservation()方法則是把客戶端傳遞來的預定資料藉由中央快儲功能把預定資料更新回 TBLRESERVATIONS 和 TBLHOTELCUSTOMERS 這 2 個資料表。要如此做很簡單，由於 TblreservationsTable 和 TblhotelcustomersTable 元件把經連結到 FDSAHotelReservations，因此只需要在下面的 008 行呼叫 FDSAHotelReservations 的 LoadFromStream()方法把客戶端傳來的異動資料(Delta)分別讀入 TblreservationsTable 和 TblhotelcustomersTable 元件，再於 009 行呼叫 FDSAHotelReservations 的 ApplyUpdates()方法就可以把資料更新回 TblreservationsTable 和 TblhotelcustomersTable 元件連結的 TBLRESERVATIONS 和 TBLHOTELCUSTOMERS 這 2 個資料表：

```
001   function TsmFireDACDataSnapDemo1.PostHotelReservation(AStream: TStream):
Integer;
```

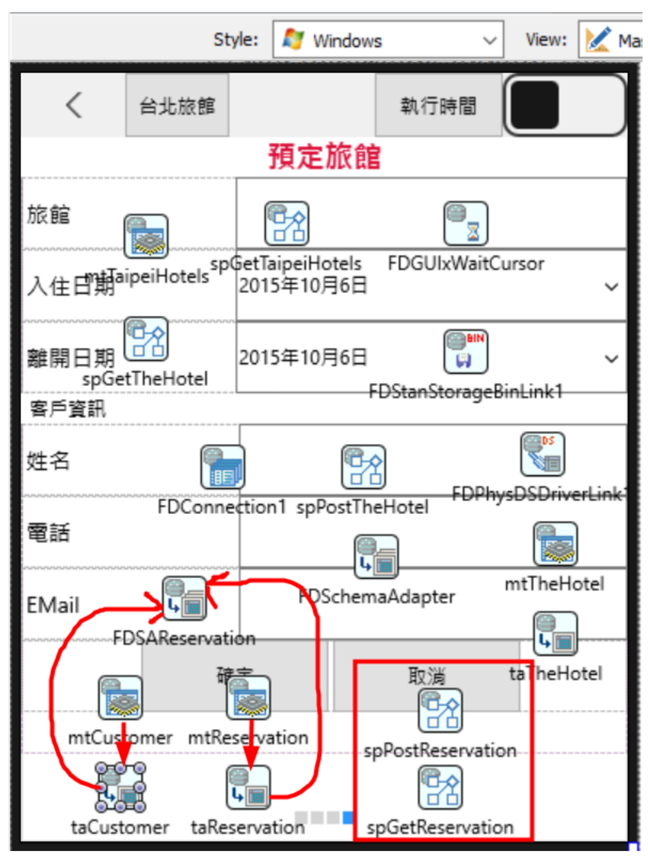
```
002   var
003     LMemStream: TMemoryStream;
004   begin
005     LMemStream := CopyStream(AStream);
006     LMemStream.Position := 0;
007     try
008       FDSAHotelReservations.LoadFromStream(LMemStream,
TFDStorageFormat.sfBinary);
009       Result := FDSAHotelReservations.ApplyUpdates;
010     finally
011       LMemStream.Free;
012     end;
013   end;
```

請重新編譯並執行此範例 DataSnap 伺服器。

## 9-2-2 修改手機客戶端

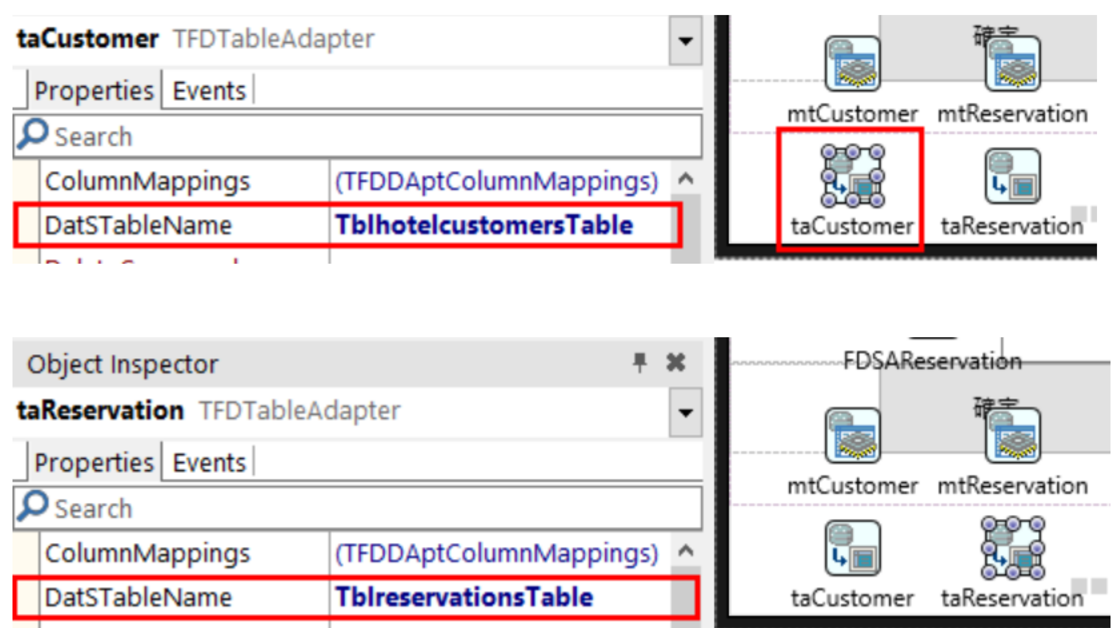
回到範例客戶端，設計如下可輸入預定資訊的 UI，並放入 FDSAReservation 這個 TFDSchemaAdapter 元件，並讓 2 個 TFDMemTable 元件分別連結到 2 個 TFDTableAdapter 元件，再把 2 個 TFDTableAdapter 元件連結到 FDSAReservation。而 spGetReservation 是準備呼叫伺服器的 GetReservation() 方法而 spPostReservation 則是呼叫伺服器的 PostHotelReservation()方法：

版權所有 請勿翻印

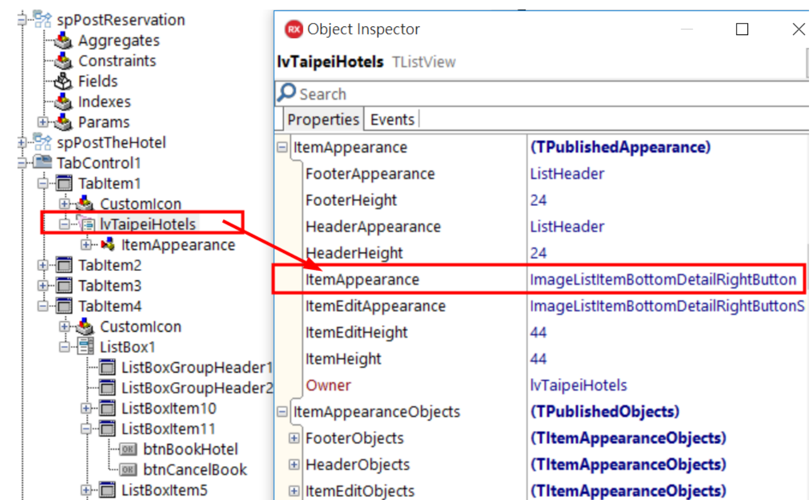



版權所有 請勿翻印

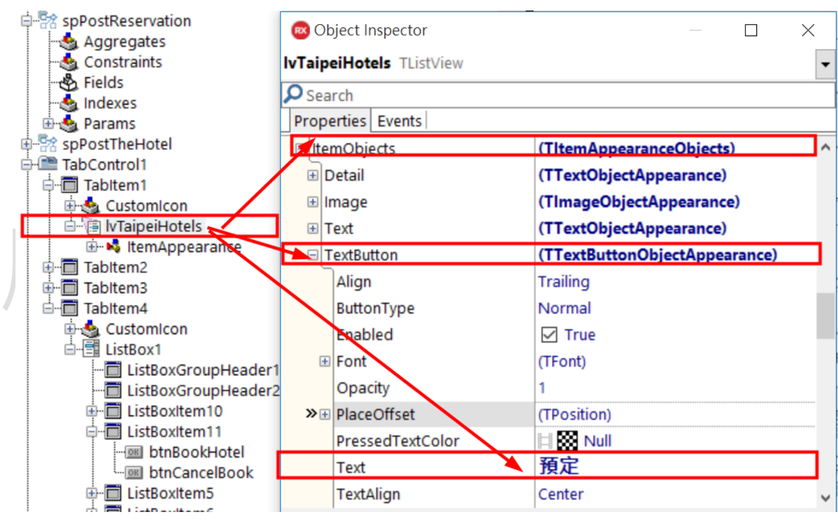
上面的 taCustomer 和 taReservation 的 DatSTableName 特性值必須設定為伺服器 ServerMethodsUnit1 程式單元中的 2 個 TFDQuery 元件的名稱：




接著修改主表單中的 TListView 元件，設定它的 ItemAppearance 子特性值為 ImageListItemBottomDetailRightButton 以便加入一個點選按鈕：




再設定它的 ItemObjects.TextButton.Text 子特性值為”預定”：




在新加入的”確定”按鈕中實作如下的程式碼：

```
procedure TfmMainForm.lvTaipeiHotelsButtonClick(const Sender: TObject;
  const AItem: TListItem; const AObject: TListItemSimpleControl);
begin
  if (FListItemTextButtonClicked) then
  begin
    DoBookHotel(lvTaipeiHotels.Items[AItem.Index].Text);
    TabControl1.ActiveTab := TabItem4;
  end;
end;
```

DoBookHotel()方法呼叫 GetReservation()方法從 DataSnap 伺服器取得 TBLRESERVATIONS 和 TBLHOTELCUSTOMERS 這 2 個資料表的元資料以

便據此設定客戶端的 2 個 TFDMemTablem 元件。PostDataToMTable()方法把使用者輸入的預定資訊更新到客戶端的 TFDMemTablem 元件中，最後 PostHotelReservation() 方法呼叫 DataSnap 伺服器的 PostHotelReservation()服務方法把預定資訊寫入資料庫中：

```
procedure TfmMainForm.btnBookHotelClick(Sender: TObject);
begin
  GetReservation;
  PostDataToMTable;
  PostHotelReservation;
end;
```

GetReservation()方法使用的技巧在前而已經說明過，由於它的目的只是取得元資料，因此在 006 行傳入-1 以避免取得任何真的的預定資料：

```
01   procedure TfmMainForm.GetReservation;
002   var
003     LStringStream: TStringStream;
004   begin
005     lStart := Now;
006     spGetReservation.Params[0].Value := -1;
007     spGetReservation.ExecProc;
008     LStringStream := TStringStream.Create(spGetReservation.Params[1].asBlob);
009     try
010       if (LStringStream <> nil) then
011       begin
012         LStringStream.Position := 0;
013         FDSAReservation.LoadFromStream(LStringStream,
TFDStorageFormat.sfBinary);
014       end;
015     finally
016       LStringStream.Free;
017       lEnd := Now;
018     end;
019   end;
```

版權所有 請勿翻印

在上面的 013 行執行完畢後，客戶端的 2 個 TFDMemTablem 元件就設好元資料(欄位架構和資訊)。

PostHotelReservation()方法在 008~014 行是確定它連結的 2 個 TFDMemTablem 元件完成寫入預定資料的動作，016~027 行藉 spPostReservation 元件呼叫 DataSnap 伺服器的 PostHotelReservation()方法把資料寫入資料庫中：

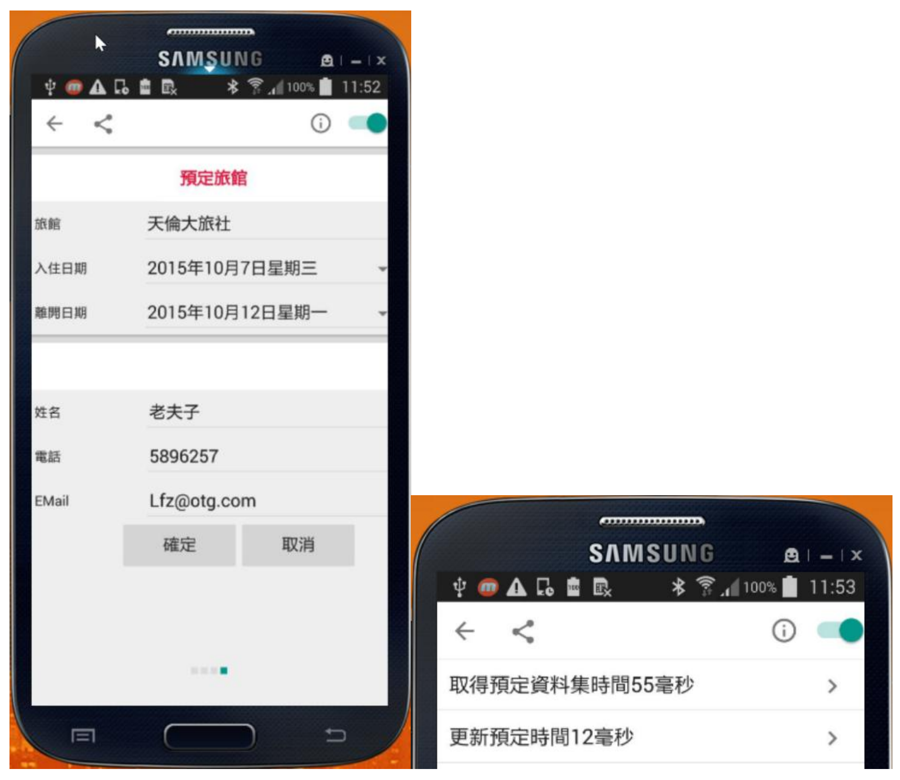
```
001   procedure TfmMainForm.PostHotelReservation;
002   var
003     LMemStream: TMemoryStream;
004     I: integer;
005     LDataSet: TDataSet;
006   begin
007     lStart := Now;
008     for I := 0 to FDSAReservation.Count - 1 do
009     begin
010       LDataSet := FDSAReservation.DataSets[I];
011       if LDataSet <> nil then
012         if LDataSet.State in dsEditModes then
013           LDataSet.Post;
014     end;
015
016     LMemStream := TMemoryStream.Create;
017     try
018       FDSAReservation.ResourceOptions.StoreItems := [siDelta, siMeta];
019       FDSAReservation.SaveToStream(LMemStream, TFDStorageFormat.sfBinary);
020       LMemStream.Position := 0;
021       spPostReservation.Params[0].asStream:= LMemStream;
022       spPostReservation.ExecProc;
023     except
024       On E: Exception do
025         raise Exception.Create(E.Message);
026     end;
027     lEnd := Now;
028   end;
```

版權所有 請勿翻印

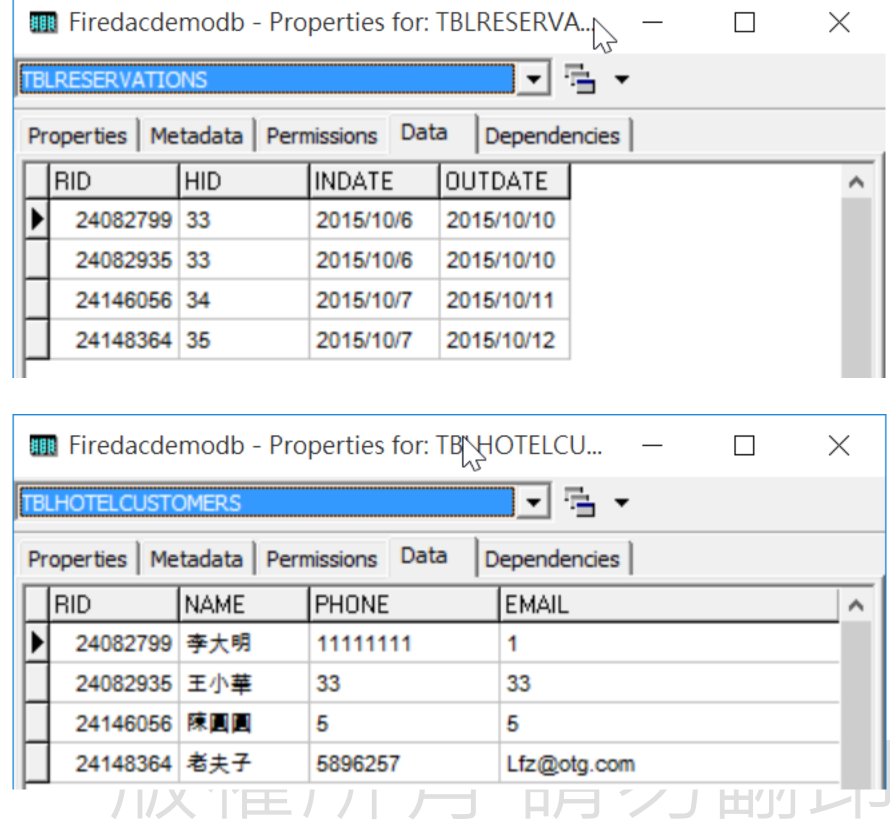
現在編譯並執行客戶端，下圖是範例 App 在 S4 手機中執行的畫面，點選任一旅館右方的”預定”按鈕：

可以在下方左圖的 UI 輸入預定資訊再點選”確定”按鈕後就可以把資料藉由範例 DataSnap 伺服器寫回 InterBase 資料庫中，從下方右圖可以看到執行速度非常理想，在客戶端從範例 DataSnap 伺服器取得元資料只需要 55 毫秒，從客戶端把輸入的預定資料藉由範例 DataSnap 伺服器寫回 InterBase 資料庫只需要 12 毫秒：

版權所有 請勿翻印



從下方 InterBase 資料庫的管理工具 IBConsole 中可以看到資料果然正確寫入到了 InterBase 資料庫中：




從本小節的說明中可以看到使用 FireDAC 的中央快儲功能在 DataSnap 架構中處理多資料表資料異動的應用時非常的方便。

## 9-2-3 使用 JSON 更新資料

在上一小節中使用的技巧是 FireDAC 的中央快儲功能，除此之外也很有很多其他的方法，例如直接使用 JSON。我們可以在客戶端根據使用者輸入的預定資料建立一個 JSON 物件然後傳遞給範例 DataSnap 伺服器，再由範例 DataSnap 伺服器解析其中的資料再寫回 InterBase 資料庫。本小節就說明如何使用 JSON 的方式把客戶端預定資料更新回資料庫。

讓我們把客戶端的預定資料封裝成一個 JSON 物件，並把預定客戶封裝成一個 JSON 陣列內嵌在 JSON 物件中，例如如下的一個 JSON 物件範例：

```
{"RID":"24172071","HID":"36","INDATE":"2015/10/7 下午
06:27:12","OUTDATE":"2015/10/7 下午 06:27:12","客戶":["李
JSON","0933000000","ljson@hotmail.com"]}
```

## 修改範例 DataSnap 伺服器

首先在範例 DataSnap 伺服器中加入一個新的方法 PostHotelReservationByJSON()：

```
function PostHotelReservationByJSON(const sData : String) : Boolean;
```

它接受一個字串的參數，此字串參數的內容就如上面說明的 JSON 物件。

PostHotelReservationByJSON()方法需要解析傳入的 JSON 物件並取得正確的資料更新回資料庫中。因此 056~088 行就是藉由新的 JSON 框架解析 SON 物件並新增資料到 TblreservationsTable 和 TblhotelcustomersTable 元件中：

```
001   function TsmFireDACDataSnapDemo1.PostHotelReservationByJSON(const sData :
String): Boolean;
002   var
003     sr : TStringReader;
004     jr : TJSONTextReader;
005     iPos : Integer;
006     iRID : Integer;
007
008     procedure HandleReservationField;
009     var
010       sField : String;
011       dt : TDateTime;
012     begin
013       sField := jr.Value.ToString;
014       if (sField = 'RID') then
015       begin
016         iRID := jr.ReadAsInteger;
017         TblreservationsTable.FieldByName(sField).Value := iRID;
018       end
019       else
020         if (sField = 'HID') then
021           TblreservationsTable.FieldByName(sField).Value := jr.ReadAsString
022         else
023         begin
024           if ( (sField = 'INDATE') or (sField = 'OUTDATE') ) then
```

版權所有 請勿翻印

```
025         begin
026           dt := StrToDateTime(jr.ReadAsString);
027           TblreservationsTable.FieldByName(sFieldValue).Value := dt;
028         end;
029       end;
030   end;
031
032   procedure HandleCustomerField;
033   var
034     sFieldValue : String;
035   begin
036     sFieldValue := jr.Value.ToString;
037     case iPos of
038       0 :
039         TblhotelcustomersTable.FieldByName('NAME').Value := sFieldValue;
040       1 :
041         TblhotelcustomersTable.FieldByName('PHONE').Value := sFieldValue;
042       2 :
043         TblhotelcustomersTable.FieldByName('EMAIL').Value := sFieldValue;
044     end;
045   end;
046
047 begin
048   Result := True;
049   FiredacdemodbConnection.StartTransaction;
050
051   if (not TblreservationsTable.Active) then
052     TblreservationsTable.Open();
053   if (not TblhotelcustomersTable.Active) then
054     TblhotelcustomersTable.Open();
055
056   try
057       sr := TStringReader.Create(sData);
058       jr := TJSONTextReader.Create(sr);
059       TblreservationsTable.Insert;
060       try
061         jr.Rewind;
062         while (jr.Read) do
```

版權所有 請勿翻印

```
063          begin
064            case jr.TokenType of
065              TJsonToken.PropertyName:
066              begin
067                HandleReservationField;
068              end;
069              TJsonToken.StartArray:
070              begin
071                TblhotelcustomersTable.Insert;
072                TblhotelcustomersTable.FieldByName('RID').Value := iRID;
073                iPos := 0;
074                while (jr.Read) do
075                begin
076                  case jr.TokenType of
077                    TJsonToken.String:
078                      HandleCustomerField;
079                  end;
080                  Inc(iPos);
081                end;
082              end;
083            end;
084          end;
085        finally
086          sr.Free;
087          jr.Free;
088        end;
089
090      TblhotelcustomersTable.Post;
091      TblreservationsTable.Post;
092      TblhotelcustomersTable.ApplyUpdates(0);
093      TblreservationsTable.ApplyUpdates(0);
094      FiredacdemodbConnection.Commit;
095    except
096      FiredacdemodbConnection.Rollback;
097      Result := False;
098    end;
099  end;
```

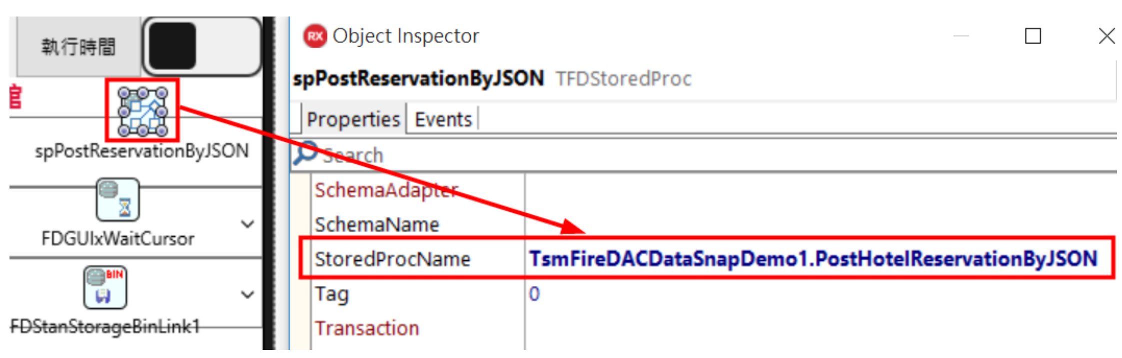
版權所有 請勿翻印

090~099 行呼叫 2 個元件的 ApplyUpdates()方法實際把資料寫回資料庫中。

請再次編譯和執行此範例 DataSnap 伺服器。

## 修改範例客戶端

在客戶端加入一個新 TFDStoredProc 元件 spPostReservationByJSON 並在物件檢視器中設定它呼叫範例 DataSnap 伺服器的 PostHotelReservationByJSON()方法：




版權所有 請勿翻印

再加入一個新的” 確定(JSON)”按鈕並實作如下的程式碼：

```
procedure TfmMainForm.btnPostByJSONClick(Sender: TObject);
var
  sJSONData : String;
begin
  sJSONData := CreateReservationJSONObject;
  PostHotelReservationByJSON(sJSONData);
end;
```

它先呼叫 CreateReservationJSONObject()方法根據使用者輸入的資料建立如前所述的 JSON 物件並儲存在 sJSONData 中，再呼叫 PostHotelReservationByJSON()方法傳遞給範例 DataSnap 伺服器。

CreateReservationJSONObject()方法使用新的 JSON 框架建立 JSON 物件(請參考”Delphi 開發手冊”一書)：

```
001    function TfmMainForm.CreateReservationJSONObject : String;
002    var
003      sw : TStringWriter;
```

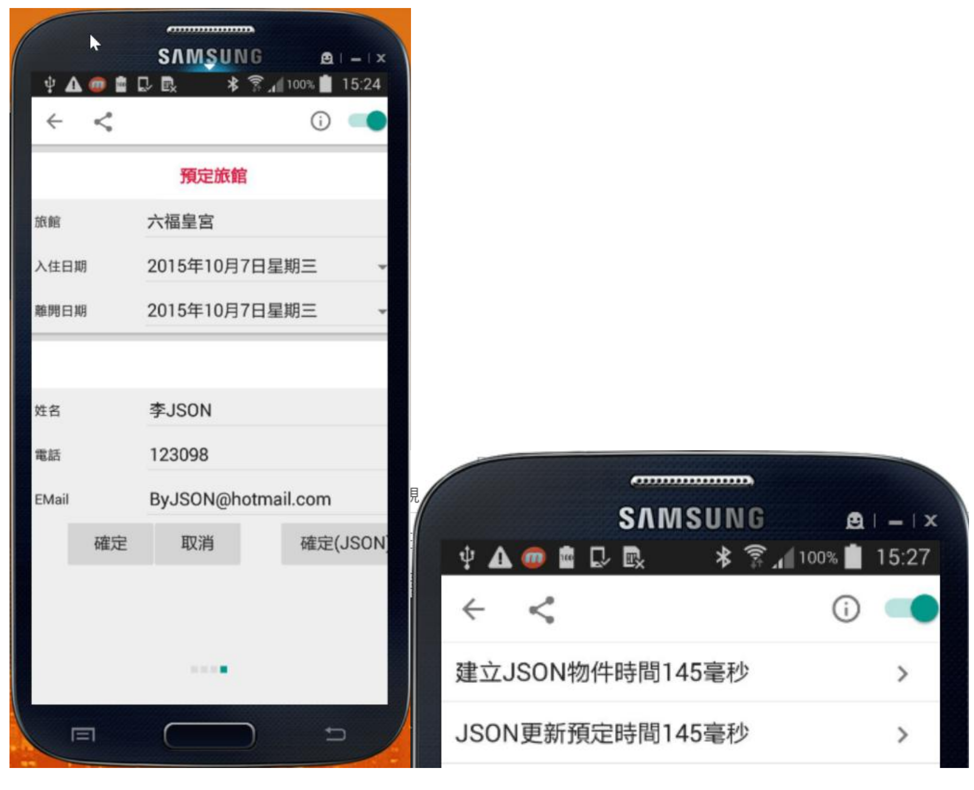
```
004      jtw : TJsonTextWriter;
005      joBuilder : TJSONObjectBuilder;
006      sRowNumber : String;
007    begin
008      sRowNumber := GetHotelRowNumber(edtBookName.Text);
009      sw := TStringWriter.Create;
010      jtw := TJsonTextWriter.Create(sw);
011      joBuilder := TJSONObjectBuilder.Create(jtw);
012      try
013        joBuilder.BeginObject
014         .Add('RID', SecondOfTheYear(Now).ToString())
015         .Add('HID',sRowNumber)
016         .Add('INDATE', DateTimeToStr(dedtIn.DateTime))
017         .Add('OUTDATE', DateTimeToStr(dedtOut.DateTime))
018         .BeginArray('客戶')
019           .Add(edtCustomerName.Text)
020           .Add(edtCustomerPhone.Text)
021           .Add(edtCustomerEMail.Text)
022         .EndAll;
023      finally
024        Result := sw.ToString;
025        joBuilder.Free;
026        jtw.Free;
027        sw.Free;
028      end;
029    end;
```

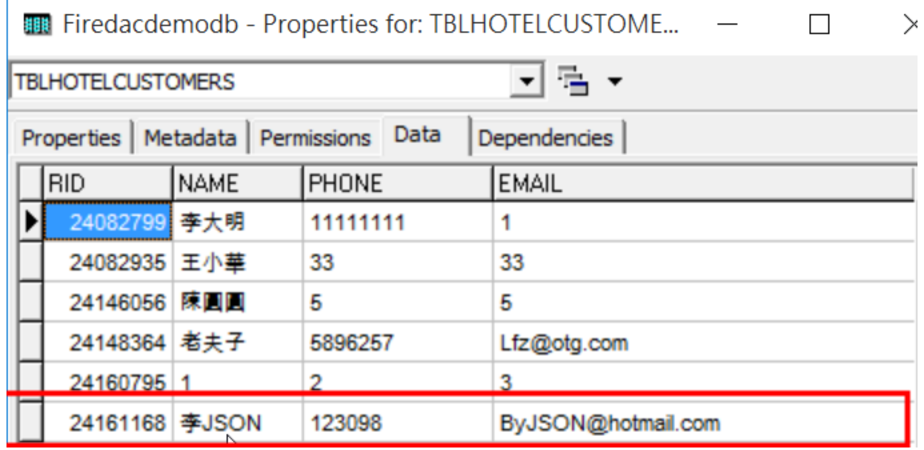
版權所有 請勿翻印

PostHotelReservationByJSON()方法非常簡單，只是執行 ExecProc()方法把 JSON 物件遞給範例 DataSnap 伺服器：

```
procedure TfmMainForm.PostHotelReservationByJSON(const sJSONData : String);
begin
  spPostReservationByJSON.Params[0].AsString:= sJSONData;
  spPostReservationByJSON.ExecProc;
end;
```

編譯和執行此範例手機 App，從下圖 S4 手機畫面可看到客戶端成功藉由使用 JSON 把預定資料更新回 InterBase 資料庫，而且速度也不錯：




| RID | NAME | PHONE | EMAIL |
|---|---|---|---|
| 24082799 | 李大明 | 11111111 | 1 |
| 24082935 | 王小華 | 33 | 33 |
| 24146056 | 陳圓圓 | 5 | 5 |
| 24148364 | 老夫子 | 5896257 | Lfz@otg.com |
| 24160795 | 1 | 2 | 3 |
| 24161168 | 李JSON | 123098 | ByJSON@hotmail.com |

如果讀者仔細比較前 2 小節使用 FireDAC 中央快儲和使用 JSON 處理資料的方式，會發現 FireDAC 中央快儲的執行速度比使用 JSON 更快，不過使用 JSON 的好處是可以其他開發工具或是程式語言寫的客戶端也可以呼叫範例 DataSnap 伺服器。

## 9-3 使用 TFDJSONDataSets 功能

在 Delphi XE5 Update 2 中加入了一個非常重要的 FireDAC 和 DataSnap 功能，那就是新的 TFDJSONDataSets 相關類別，TFDJSONDataSets 相關類別隨後不斷的強化功能並在 10.3 的版本加入了壓縮資料的功能讓執行速度更快速。

那 TFDJSONDataSets 相關類別到底是做什麼用的呢？簡單的說 TFDJSONDataSets 相關類別可以結合 DataSnap 開發使用 JSON 和 RESTful 的系統架構，讓程式師可以使用比較簡單的方式開發 CRUD 的 App。在上一小節說明使用 JSON 技術開發 DataSnap 系統時，程式師都需要執行下面的工作：

- 客戶端需自行解析內容
- 伺服端和客戶端都需自行處理 CRUD
- 直接回傳 TDataSet 回客戶端的話，客戶端需解析 TDataSet 的 JSON 內容

例如在上一小節中客戶端就需要把旅館預定資料封裝成 JSON 物件再傳遞給 DataSnap 伺服器，而伺服器又需要再解析 JSON 物件。

但如果使用 TFDJSONDataSets 相關類別，那這些類別可提供如下的功能：

- 可在單一呼叫中處理多個資料集(DataSet)
- 可和 TFDMemTable 共同使用
- 可直接在單一呼叫中回傳客戶端的多個資料異動(Delta)
- 非常容易在伺服端處理 CRUD 的資料
- 可使用客製化格式讀取或是異動資料

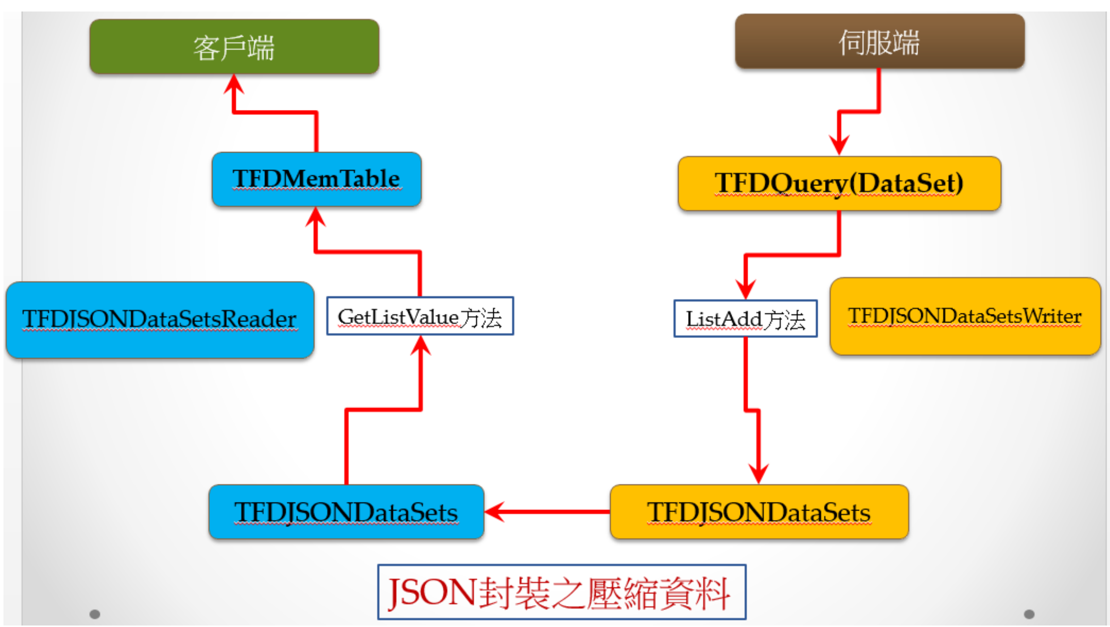
版權所有 請勿翻印

從上面的說明可以瞭解，由於 TFDJSONDataSets 相關類別可在一次網路的回來呼叫中處理多個資料集，因此它們可以減少網路的回來次數而增加執行效率，另外 TFDJSONDataSets 相關類別又可壓縮資料也可增加執行效率，最後 TFDJSONDataSets 相關類別即可提供資料集物件和 JSON 物件之間的轉換和解析，因此也可以免除程式師需要自行處理 JSON 內容的工作，進而加快程式師的開發速度。

TFDJSONDataSets 相關類別是使用 Reader/Writer 設計樣例，程式師使用 Writer 把資料集物件寫入 TFDJSONDataSets 中，藉由 TFDJSONInterceptor 轉成 JSON 再傳遞出去。在接收方再使用 TFDJSONInterceptor 把 JSON 轉回 TFDJSONDataSets，最後再使用 Reader 把資料集物件讀出。一般來說程式師只需要使用下表的 4 個類別即可：

| 類別 | 說明 |
| --- | --- |
| TFDJSONDataSets | FireDAC 使用的 JSON DataSet，其中可包含多個 DataSet 物件 |
| TFDJSONDataSetsWriter | 使用此類別把 TDataSet 寫入 TFDJSONDataSets |
| TFDJSONDataSetsReader | 使用此類別把 TDataSet 從 TFDJSONDataSets 中讀回 |
| TFDJSONInterceptor | 使用此類別把 TFDJSONDataSets 和 JSON 格式之間做轉換 |

使用 TFDJSONDataSets 相關類別的方式如同下圖所示：




在伺服端使用 TFDJSONDataSetsWriter 類別的類別方法 ListAdd()把 TFDQuery 物件寫入 TFDJSONDataSets 物件，再傳遞到另一端，而客戶端則可使用 TFDJSONDataSetsReader 類別的類別方法 GetListValue()或是 GetListValueByName()把資料集物件讀出再載入 TFDMemTable 物件中即可存取其中的資料。

TFDJSONDataSetsWriter 類別的類別方法 ListAdd()定義如下：

```
   class procedure ListAdd(const ADataList: TFDJSONDataSets; const AName: string;
const ADataSet: TFDAdaptedDataSet); overload;
   class procedure ListAdd(const ADataList: TFDJSONDataSets; const ADataSet:
TFDAdaptedDataSet); overload;
```

它接受 1 個 TFDJSONDataSets 物件，一個字串名稱參數和一個 TFDAdaptedDataSet 物件。這代表在 TFDJSONDataSets 中使用下列的方式儲存資料集物件：

```
Name : DataSet
```

例如如果我們使用 TFDJSONDataSets 物件來處理前面旅館預定資料的應用，那麼在 TFDJSONDataSets 物件中我們可儲存如下的 2 對資料：

```
'預定資訊': TblreservationsTable 和'客戶' : TblhotelcustomersTable
```

因此可使用如下的程式碼：

```
TFDJSONDataSetsWriter. ListAdd (aFDJSONDataSets ,'預定資訊': TblreservationsTable);
TFDJSONDataSetsWriter.ListAdd (aFDJSONDataSets ,'客戶' : TblhotelcustomersTable);
```

而如果我們是想查詢台北市旅館資料，那那麼在 TFDJSONDataSets 物件中我們可儲存如下的資料：

```
'台北市旅館': fdqTaipeiHotels
```

版權所有 請勿翻印

由於台北市旅館資料只有一個資料集物件，因此可使用上面第 2 個 ListAdd() 類別方法：

因此可使用如下的程式碼：

```
TFDJSONDataSetsWriter. ListAdd (aFDJSONDataSets , fdqTaipeiHotels);
```

而當 TFDJSONDataSets 物件傳遞到客戶端後，客戶端就可以使用 TFDJSONDataSetsReader 類別的類別方法 GetListValue() ／ GetListValueByName()讀出 TFDAdaptedDataSet：

```
   class function GetListValue(const ADataList: TFDJSONDataSets; I: Integer):
TFDAdaptedDataSet; static;
   class function GetListValueByName(const ADataList: TFDJSONDataSets;const AName:
string): TFDAdaptedDataSet; static;
```

因此使用如下的程式碼即可讀回上面的北市旅館資料：

```
TFDJSONDataSetsReader.GetListValue (aFDJSONDataSets ,0);
```

使用如下的程式碼即可讀回上面的旅館預定資料和客戶資料：

```
TFDJSONDataSetsReader.GetListValue (aFDJSONDataSets ,0);
TFDJSONDataSetsReader.GetListValue (aFDJSONDataSets ,1);
```

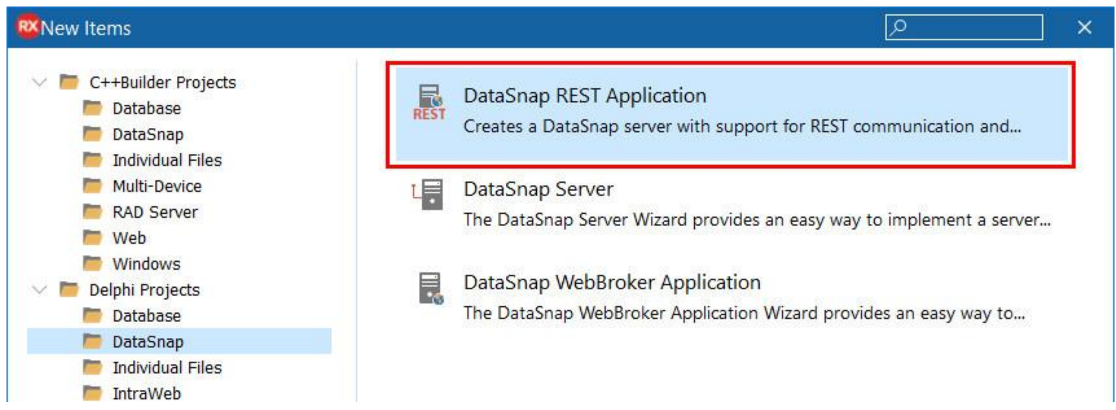
或是：

```
TFDJSONDataSetsReader. GetListValueByName (aFDJSONDataSets , '預定資訊');
TFDJSONDataSetsReader. GetListValueByName (aFDJSONDataSets , '客戶');
```

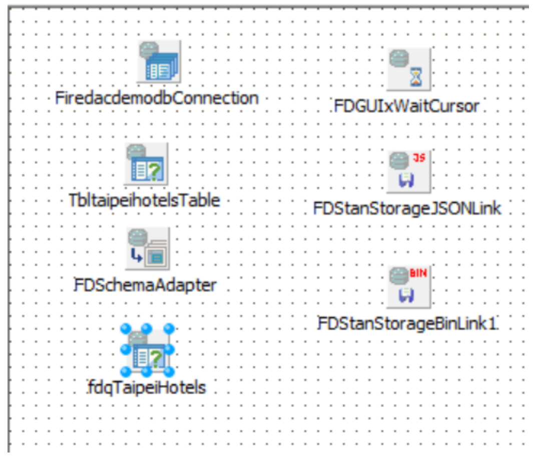
瞭解了如何使用 TFDJSONDataSets 相關類別後就可以讓我們使用開發查詢台北市旅館資訊的 RESTful 系統了。

## 9-3-1 開發 RESTful DataSnap 伺服器

首先建一個 DataSnap REST Application 專案：




再於 ServerMethodsUnit1 中加入如前面範例 DataSnap 伺服器一樣的元件：




接著在 ServerMethodsUnit1 程式單元的 public 部分同樣宣服務方法 GetTaipeiHotels()：

```
Public
  { Public declarations }
…
  function GetTaipeiHotels : TFDJSONDataSets;
```

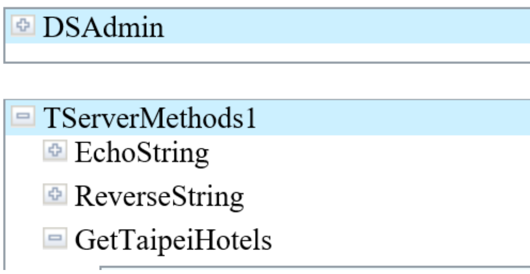
再實作如下：

```
001   function TServerMethods1.GetTaipeiHotels: TFDJSONDataSets;
002   begin
003     fdqTaipeiHotels.Active := False;
004     Result := TFDJSONDataSets.Create;
005     TFDJSONDataSetsWriter.ListAdd(Result, fdqTaipeiHotels);
006   end;
```

GetTaipeiHotels()方法於 003 行先把 fdqTaipeiHotels 元件關閉，004 行建立回傳的 TFDJSONDataSets 物件，最後於 005 行呼叫 TFDJSONDataSetsWriter 類別的類別方法 ListAdd()把 fdqTaipeiHotels 寫入要回傳的 TFDJSONDataSets 物件中即可。

現在執行此範例伺服器，開啟瀏覽器就可以如下呼叫 GetTaipeiHotels()方法，這當然是因為此範例伺服器是 RESTful 伺服器且回傳 JSON 資料，從下圖下半部分可看到 GetTaipeiHotels() 方法以 JSON 物件封裝回傳 fdqTaipeiHotels 元件中的資料集物件：

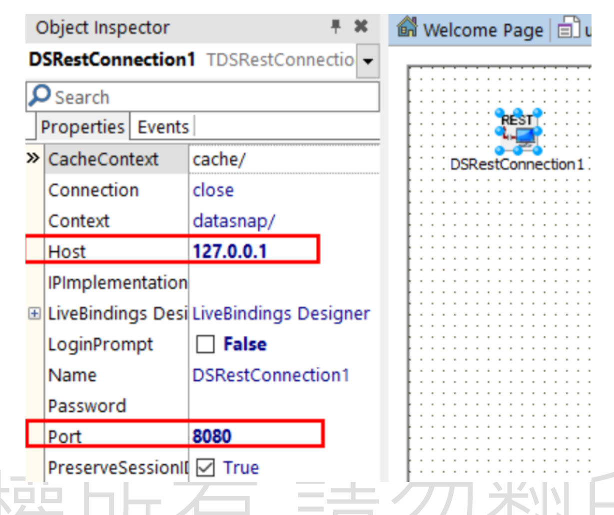
版權所有 請勿翻印




**Executed:** TServerMethods1.GetTaipeiHotels

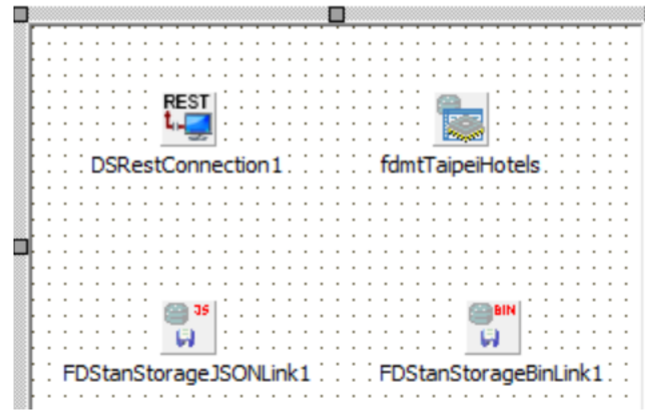
{"result":{"type":"Data.FireDACJSONReflect.TFDJSONDataSets","id":1,"fields":{"FDataSets"
\r\njxj0cUm/q+2S3Jw6PhZzRQOCkwHt+SUafdj9XJGQpHji3d1Hs8OZCnckP1xlojhCLoLyC1I
\r\nhtJGpFUNeLRXaXIZEHI5TXBv8aqbm2bU87UryYNsUk/NkXpUrcMcAwt1qI5Fd0lY7Na4

現在於剛才的範例專案中再建立一個新的 Multi-Device Applicatio 專案，於其中再建立一個資料模組並在其中加入一個 TDSRestConnection 元件，設定它的 Host 和 Port 特性值為剛才的範例 RESTful DataSnap 伺服器的位置使用的 IP 地址和通訊埠。由於筆者是在筆者的機器中執行範例 RESTful DataSnap 伺服器，因此設定 Host 為 127.0.0.1，而 8080 則是 RESTful DataSnap 伺服器使用的預定通訊埠：

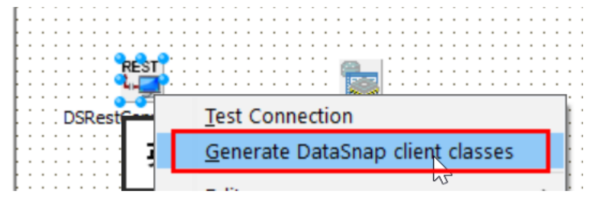



版權所有 請勿翻印

接著再放入 TFDMemTable 和 TFDStanStorageBinLink 以及 TFDStanStorageJSONLink 元件：




再右擊 TDSRestConnection 元件選擇 Generate DataSnap client classes 選項以自動產生客戶端連結範例 RESTful DataSnap 伺服器的程式碼(範例 RESTful DataSnap 伺服器必須先執行)：




在上面步驟自動產生的客戶端程式碼中就可以看到範例 RESTful DataSnap 伺服器提的服務方法 GetTaipeiHotels()：

```
function GetTaipeiHotels(const ARequestFilter: string = ''): TFDJSONDataSets;
```

設計此範例 RESTful 客戶端的主表單和上一個範例客戶一樣，並在”台北旅館”按鈕中先呼叫 GetTaipeiHotels()方法連結範例 RESTful DataSnap 伺服器，呼叫服務方法最得回傳的 TFDJSONDataSets 物件 LDataSetList，再呼叫 DisplayTaipeiHotels()方法顯示回傳的資料：

```
001   procedure TfmMainForm.Button1Click(Sender: TObject);
002   var
003     aServer : TServerMethods1Client;
004     LDataSetList: TFDJSONDataSets;
005   begin
006     aServer := Nil;
007     try
008       LDataSetList := GetTaipeiHotels(aServer);
009       DisplayTaipeiHotels(LDataSetList);
010     finally
011       aServer.Free;
012     end;
013   end;
```

版權所有 請勿翻印

客戶端的 GetTaipeiHotels()方法先於 004 行建立遠端伺服器物件再於 005 行呼叫它的 GetTaipeiHotels()方法並取得回傳結果：

```
001   function TfmMainForm.GetTaipeiHotels(var aServer : TServerMethods1Client) :
TFDJSONDataSets;
002   begin
003     Result := Nil;
004     aServer :=
TServerMethods1Client.Create(dmRESTfulClient.DSRestConnection1);
005     Result := aServer.GetTaipeiHotels();
006   end;
```

DisplayTaipeiHotels()方法顯示了如何從 TFDJSONDataSets 物件取出結果資料集。006 行先關閉資料模組中的 fdmtTaipeiHotels 元件 007 行確定回傳的 TFDJSONDataSets 物件中的確包含了一個 TFDAdaptedDataSet 物件(即伺服端回傳的 fdqTaipeiHotels 元件中的資料集物件)，008 行呼叫 TFDJSONDataSetsReader 類別的類別方法 GetListValue()方法取出資料集物件，再於 009 行呼叫 fdmtTaipeiHotels 元件的 AppendData()方法把此資料集物件加入到 fdmtTaipeiHotels 元件中，最後即可藉由 fdmtTaipeiHotels 元件一一取出其中的回傳台北市旅館資料並顯示出來：

```
001   procedure TfmMainForm.DisplayTaipeiHotels(LDataSetList: TFDJSONDataSets);
002   var
003     lvi : TListViewItem;
004     aDataSet : TFDAdaptedDataSet;
005   begin
006     dmRESTfulClient.fdmtTaipeiHotels.Active  := False;
007     Assert(TFDJSONDataSetsReader.GetListCount(LDataSetList) = 1);
008     aDataSet := TFDJSONDataSetsReader.GetListValue(LDataSetList, 0);
009     dmRESTfulClient.fdmtTaipeiHotels.AppendData(aDataSet);
010
011     dmRESTfulClient.fdmtTaipeiHotels.First;
012     while (not dmRESTfulClient.fdmtTaipeiHotels.Eof) do
013     begin
014       lvi := lvTaipeiHotels.Items.Add;
015       lvi.Text :=
dmRESTfulClient.fdmtTaipeiHotels.FieldByName('STITLE').AsString;
016       lvi.Detail :=
dmRESTfulClient.fdmtTaipeiHotels.FieldByName('MEMO_TEL').AsString;
017       dmRESTfulClient.fdmtTaipeiHotels.Next;
018     end;
019   end;
```

版權所有 請勿翻印

現在執行此範例客戶端即可看到類似如下的執行結果，和前面範例不同的是這是一個 RESTful 架構，而資料都是使用 JSON 格式傳遞的。

## 9-3-3 開發 RESTful 多資料表查詢

版權所有 請勿翻印

瞭解了上面討論的內容，那 RESTful 多資料表查詢就很容易實作了，我們只要把多個資料集元件寫入回傳到客戶端的 TFDJSONDataSets 物件即可。為了簡單的說明起見，讓我們實作一個簡單的功能，允許客戶端查詢特定旅館的預定資訊。

### 修改範例 DataSnap 伺服器

先在範例 RESTful DataSnap 伺服器提供 GetHotelReservation()方法，它接受旅館 ID 然後從 TblreservationsTable 和 TblhotelcustomersTable 這 2 個資料表中取出資料，再於下列的 010 和 011 行把這 2 個資料表寫入 TFDJSONDataSets 物件中並回傳到客戶端：

```
001    function TServerMethods1.GetHotelReservation(iHID: String): TFDJSONDataSets;
002    begin
003      TblreservationsTable.Active := False;
004      TblreservationsTable.Params[0].Value := iHID;
005      TblreservationsTable.Open();
006      TblhotelcustomersTable.Active := False;
007      TblhotelcustomersTable.Params[0].Value :=
TblreservationsTable.FieldByName('RID').Value;
```

```
008     TblreservationsTable.Close();
009     Result := TFDJSONDataSets.Create;
010     TFDJSONDataSetsWriter.ListAdd(Result, TblreservationsTable);
011     TFDJSONDataSetsWriter.ListAdd(Result, TblhotelcustomersTable);
012   end;
```

## 修改範例客戶端

在客戶端只需先在 009 行確定伺服端回傳了 2 個資料表物件，再分別取出每一個資料表物件並載入客戶端的 **TFDMemeTable** 物件中即可：

```
001   procedure TfmMainForm.DisplayHotelReservations(LDataSetList:
TFDJSONDataSets);
002   var
003     lvi : TListViewItem;
004     aDataSet : TFDAdaptedDataSet;
005   begin
006     lStart := Now;
007     dmRESTfulClient.mtRHotel.Active  := False;
008     dmRESTfulClient.mtRCustomers.Active := False;
009     Assert(TFDJSONDataSetsReader.GetListCount(LDataSetList) = 2);
010     aDataSet := TFDJSONDataSetsReader.GetListValue(LDataSetList, 0);
011     dmRESTfulClient.mtRHotel.AppendData(aDataSet);
012     aDataSet := TFDJSONDataSetsReader.GetListValue(LDataSetList, 1);
013     dmRESTfulClient.mtRCustomers.AppendData(aDataSet);
014
015     dmRESTfulClient.mtRCustomers.First;
016     while (not dmRESTfulClient.mtRCustomers.Eof) do
017     begin
018       lvi := lvRCustomers.Items.Add;
019       lvi.Text := dmRESTfulClient.mtRCustomers.FieldByName('NAME').AsString;
020       lvi.Detail :=
dmRESTfulClient.mtRCustomers.FieldByName('PHONE').AsString;
021       dmRESTfulClient.mtRCustomers.Next;
022     end;
023     lEnd := Now;
024   end;
```

版權所有 請勿翻印

## 9-3-4 開發 CRUD 功能伺服端

TFDJSONDataSets 相關類別也提供了非常易於使用但功能強大的 RESTful CRUD 的功能，提供 RESTful CRUD 功能的相關類別是 TFDJSONDeltas，TFDJSONDeltasWriter 和 TFDJSONDeltasApplyUpdates。下面的表格說明了這些類別提供的功能：

| 類別 | 說明 |
|---|---|
| TFDJSONDeltas | 包含異動資料(Delta)的類別，程式師把 Delta 寫入 TFDJSONDeltas 物件再把它傳遞給另一方。 |
| TFDJSONDeltasWriter | 使用此類別把 Delta 寫入 TFDJSONDeltas 物件中 |
| TFDJSONDeltasApplyUpdates | 使用此類別把 Delta 真正異動回資料庫中 |

請注意，在使用 TFDJSONDeltasWriter 把 Delta 寫入 TFDJSONDeltas 物件時也是使用"名稱"和 Delta 的格式：

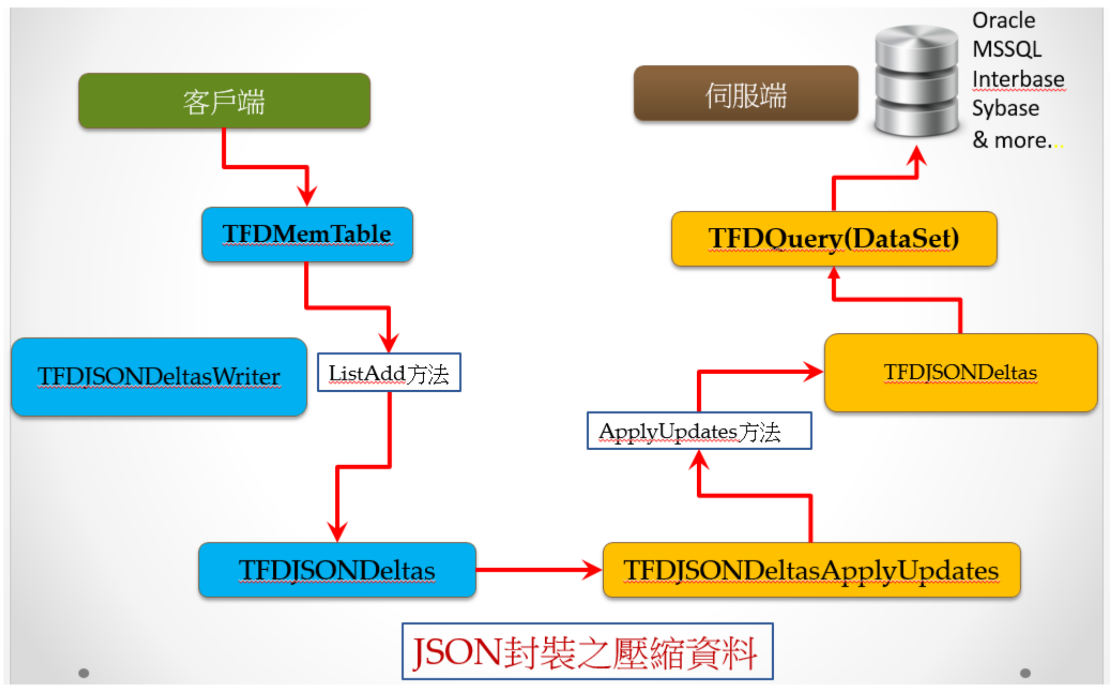
```
Name : Delta
```

因此 TFDJSONDeltasWriter 也可以把多個 Delta 寫入 TFDJSONDeltas 物件中。

版權所有 請勿翻印

使用 TFDJSONDeltas，TFDJSONDeltasWriter 和 TFDJSONDeltasApplyUpdates 這 3 個類別的方式如下圖所示。當客戶端要把異動的資料更新回資料庫時，就把 TFDMemTable 中的 Delta 藉由 TFDJSONDeltasWriter 類別的類別方法 ListAdd()寫入 TFDJSONDeltas 物件，再把 TFDJSONDeltas 物件傳遞給 DataSnap 伺服器。

到了 DataSnap 伺服器後就呼叫 TFDJSONDeltasApplyUpdates 物件的 ApplyUpdates()方法把 TFDJSONDeltas 物件中的 Delta 藉由 TFDQuery 元件更新回資料庫之中。




瞭解了上述的工作原理之後我們就可以回到範例 DataSnap 伺服器，在它的 ServerMethodsUnit 中加入一個 PostTheHotel()方法，PostTheHotel()接受從客戶端傳來的 TFDJSONDeltas 物件參數，005 行先建立 TFDJSONDeltasApplyUpdates 物件並傳入 TFDJSONDeltas 物件做為建構元參數，006 行呼叫它的 ApplyUpdates()方法同時傳入 2 個參數，第 1 個參數是 TFDJSONDeltas 物件中名為 sTheHotel 的 Delta，第 2 個參數則是資料模組中 fdqTaipeiHotels 物件的 Command 特性值：

版權所有 請勿翻印

```
001   function TServerMethods1.PostTheHotel(const ADeltaList: TFDJSONDeltas):
Integer;
002   var
003     LApply: IFDJSONDeltasApplyUpdates;
004   begin
005     LApply := TFDJSONDeltasApplyUpdates.Create(ADeltaList);
006     Result:= LApply.ApplyUpdates(sTheHotel, fdqTaipeiHotels.Command);
007     if LApply.Errors.Count > 0 then
008       raise Exception.Create(LApply.Errors.Strings.Text);
009   end;
```

上面 ApplyUpdates()方法的第 1 個參數 sTheHotel 是伺服端和客戶端共同使用的代表 TFDJSONDeltas 物件中的 Delta 名稱字串：

```
const
  sTheHotel = '旅館資料';
```

另外需要注意的是在上面的程式碼中 LApply 事實上是 IFDJSONDeltasApplyUpdates 介面的變數，這是因為 TFDJSONDeltasApplyUpdates 類別實作了 IFDJSONDeltasApplyUpdates 介面：

```
TFDJSONDeltasApplyUpdates = class(TFDJSONDeltasReader, IFDJSONDeltasApplyUpdates)
```

而 ApplyUpdates()方法是宣告在 IFDJSONDeltasApplyUpdates 介面中：

```
  IFDJSONDeltasApplyUpdates = interface(IFDJSONDeltasReader)
    ['{58213D3C-BFE8-4BE3-8197-4236FFC215C8}']
    function ApplyUpdates(const AKey: string; const ASelectCommand:
TFDCustomCommand): Integer; overload;
    function ApplyUpdates(const Index: Integer; const ASelectCommand:
TFDCustomCommand): Integer; overload;
    function ApplyUpdates(const AKey: string; const AAdapter: TFDTableAdapter):
Integer; overload;
    function ApplyUpdates(const Index: Integer; const AAdapter: TFDTableAdapter):
Integer; overload;
    function GetErrors: TFDJSONErrors;
    property Errors: TFDJSONErrors read GetErrors;
  end;
```

版權所有 請勿翻印

現在再次編譯和執行此範例 RESTful DataSnap 伺服器。

## 9-3-4 開發 CRUD 功能客戶端

回到範例客戶端，在下面更新旅館資料的確定按鈕中呼叫 PostTheHotel()方法把使用者在手機中修改的旅館資料更新回資料庫：

```
procedure TfmMainForm.Button4Click(Sender: TObject);
begin
  PostTheHotel;
end;
```

版權所有 請勿翻印

PostTheHotel()方法同樣在 015~020 行中把使用者輸入的資料寫入 mtTheHotel 這個 TFDMemTable 元件中，022 行是關鍵，它在 GetDeltas() 子方法中的 008 行先建立 TFDJSONDeltas 物件，再於 009 行呼叫類別方法 ListAdd()把 mtTheHotel 的 Delta 寫入 TFDJSONDeltas 物件並以 sTheHotel 命名此 Delta，當然在客戶端 sTheHotel 也是定義成相同的名稱：

```
const
  sTheHotel = '旅館資料';
```

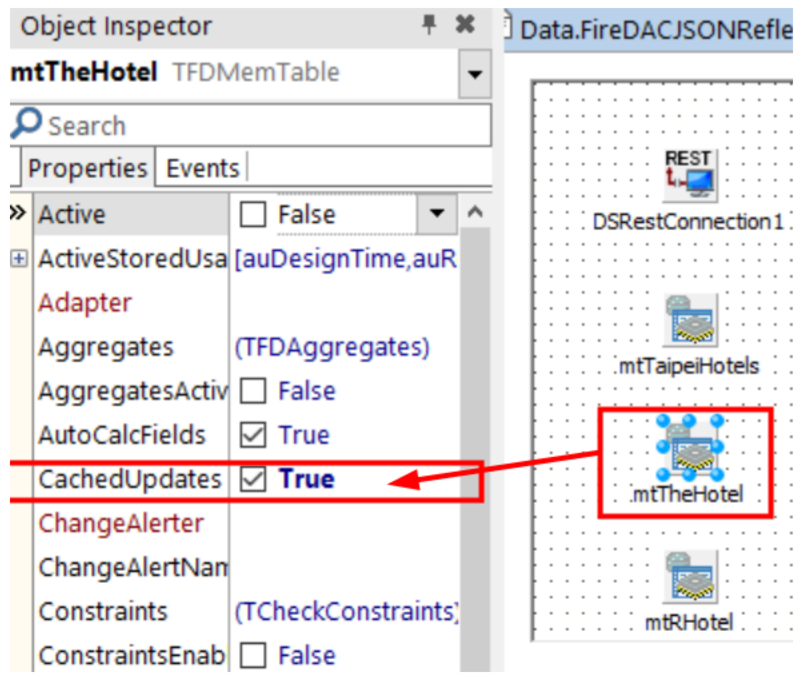
最後在 025 行呼叫 DataSnap 伺服器的 PostTheHotel()方法把 TFDJSONDeltas 物件傳遞給 DataSnap 伺服器以完成更新旅館資料的工作：

```
001   procedure TfmMainForm.PostTheHotel;
002   var
003     LDeltaList: TFDJSONDeltas;
004     aServer : TServerMethods1Client;
005
006     function GetDeltas: TFDJSONDeltas;
007     begin
008       Result := TFDJSONDeltas.Create;
```

```
009       TFDJSONDeltasWriter.ListAdd(Result, sTheHotel,
dmRESTfulClient.mtTheHotel);
010      end;
011
012    begin
013      lStart := Now;
014
015      dmRESTfulClient.mtTheHotel.Edit;
016      dmRESTfulClient.mtTheHotel.FieldByName('ROWNUMBER').Value :=
dmRESTfulClient.mtTheHotel.FieldByName('ROWNUMBER').Value;
017      dmRESTfulClient.mtTheHotel.FieldByName('STITLE').Value := edtTitle.Text;
018      dmRESTfulClient.mtTheHotel.FieldByName('MEMO_TEL').Value :=
edtPhone.Text;
019      dmRESTfulClient.mtTheHotel.FieldByName('MEMO_COST').AsString :=
edtPrice.Text;
020      dmRESTfulClient.mtTheHotel.Post;
021
022      LDeltaList := GetDeltas;
023      aServer :=
TServerMethods1Client.Create(dmRESTfulClient.DSRestConnection1);
024      try
025        aServer.PostTheHotel(LDeltaList);
026      finally
027        aServer.Free;
028      end;
029      lEnd := Now;
030    end;
```

版權所有 請勿翻印

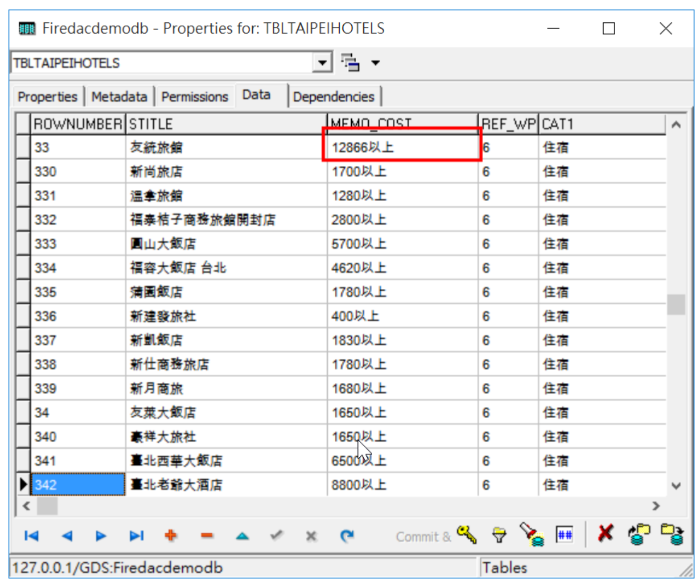
接著一個很重要的步驟就是必須開啟 **mtTheHotel** 元件使用 **CachedUpdates** 功能，否則資料無法成功更新回資料庫之中(因為沒有 **Delta**)：




編譯並執行範例手機 App，試著更新一個旅館的資料如下所示我們把價位改成 12866 然後點選確定按鈕更新資料：

版權
翻印

之後到後端的 InterBase 資料庫查看此筆旅館的資料，可以看到下圖在手機客戶端異動的資料果然成功藉由 RESTful DataSnap 伺服器更新回資料庫中了：




| ROWNUMBER | STITLE | MEMO_COST | REF_WP | CAT1 |
|---|---|---|---|---|
| 33 | 友統旅館 | 12866以上 | 6 | 住宿 |
| 330 | 新尚旅店 | 1700以上 | 6 | 住宿 |
| 331 | 溫拿旅館 | 1280以上 | 6 | 住宿 |
| 332 | 福泰桔子商務旅館開封店 | 2800以上 | 6 | 住宿 |
| 333 | 圓山大飯店 | 5700以上 | 6 | 住宿 |
| 334 | 福容大飯店 台北 | 4620以上 | 6 | 住宿 |
| 335 | 蒲園飯店 | 1780以上 | 6 | 住宿 |
| 336 | 新建發旅社 | 400以上 | 6 | 住宿 |
| 337 | 新凱飯店 | 1830以上 | 6 | 住宿 |
| 338 | 新仕商務旅店 | 1780以上 | 6 | 住宿 |
| 339 | 新月商旅 | 1680以上 | 6 | 住宿 |
| 34 | 友萊大飯店 | 1650以上 | 6 | 住宿 |
| 340 | 豪祥大旅社 | 1650以上 | 6 | 住宿 |
| 341 | 臺北西華大飯店 | 6500以上 | 6 | 住宿 |
| 342 | 臺北老爺大酒店 | 8800以上 | 6 | 住宿 |

瞭解了如何使用 TFDJSONDeltas，TFDJSONDeltasWriter 和 TFDJSONDeltasApplyUpdates 這 3 個類別進行對一個資料表的 RESTful 的 CRUD 工作後，就請讀者再挑戰一下修改此 RESTful 範例 DataSnap 伺服器和範例客戶端 App 來進行對多個資料表的 RESTful 的 CRUD 工作吧，Have Fun！

版權所有 請勿翻印

# 第10章 開發安全，高效率的DataSnap應用系統

DataSnap 的效率如何？可以支援多少客戶端？如何能夠開發安全,高效率的 DataSnap 應用系統呢？這應該是閱讀本書的讀者在學習了如何開發 DataSnap 應用系統之後最想詢問的問題。

然而要回答這些問題並不是簡單的事情，因為這牽涉到許多不同的內容，例如您如何設計您的 DataSnap 應用系統？您如何設計企業類別和物件？你使用的硬體能力和架構?你撰寫的程式碼的效率？以及最後本章要討論的主題，您如何開發您的 DataSnap 伺服器？

版權所有 請勿翻印

因此本章主要討論的內容將暫時排除硬體，設計和程式碼撰寫方式的因素，本章將為讀者說明如何開發安全，高效率的 DataSnap 伺服器，讓它可以支援合理數量的客戶端並提供良好的回應速度。

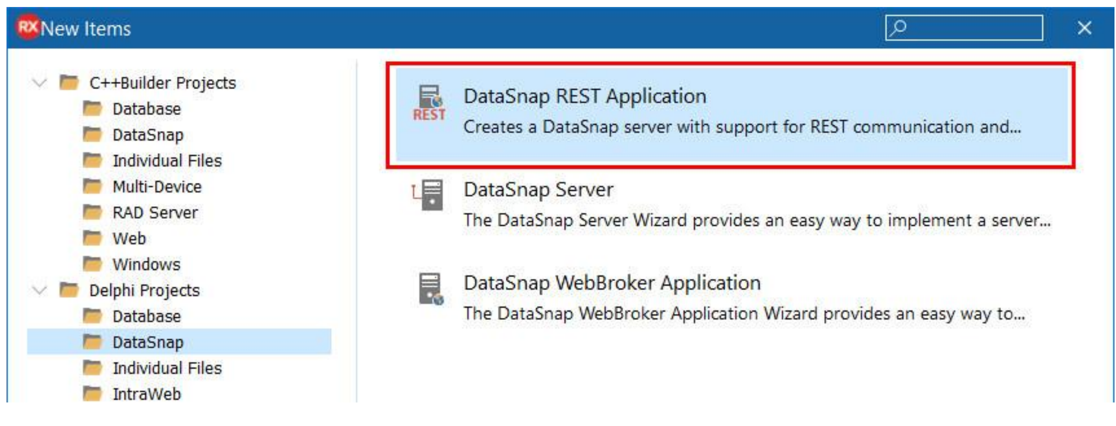
為了稍後內容的討論以及比較不同 DataSnap 伺服器，因此先讓我們討論如何開發 DLL 型態的 DataSnap 伺服器，這包含了 ISAPI 或 Apache DataSnap 伺服器，不過為了簡化說明的內容就讓我們先以如何開發 ISAPI 型態的 DataSnap 伺服器。那為什麼要使用 DLL 型態的 DataSnap 伺服器呢？簡單的說 DLL 型態的 DataSnap 伺服器其執行效率比前面章節說明的 EXE 型態的 DataSnap 伺服器來得好上許多，因此如果您是真的要在實際環境中使用 DataSnap 應用架構,那一定要使用 DLL 型態的 DataSnap 伺服器,因為 EXE 型態的 DataSnap 伺服器是使用來學習和易於除錯之用。

現在就讓我們開始探究執行效率這個重要又有趣的旅程吧。

開發 DLL 型態的 DataSnap 伺服器和前面討論 EXE 型態的 DataSnap 伺服器差不多，只是在選擇建立的 DataSnap 型態時要注意選擇 ISAPI 或是 Apache。

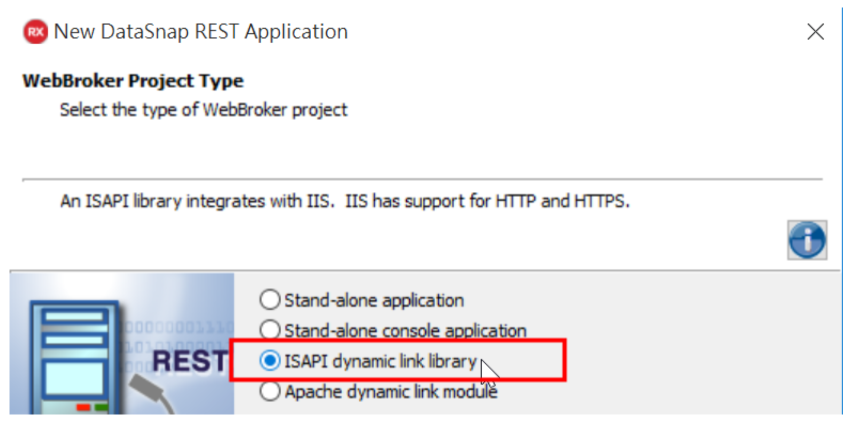
## 10-1-1 開發 ISAPI 型態的 DataSnap 伺服器

請在 IDE 中選擇建立 DataSnap REST Application 專案：

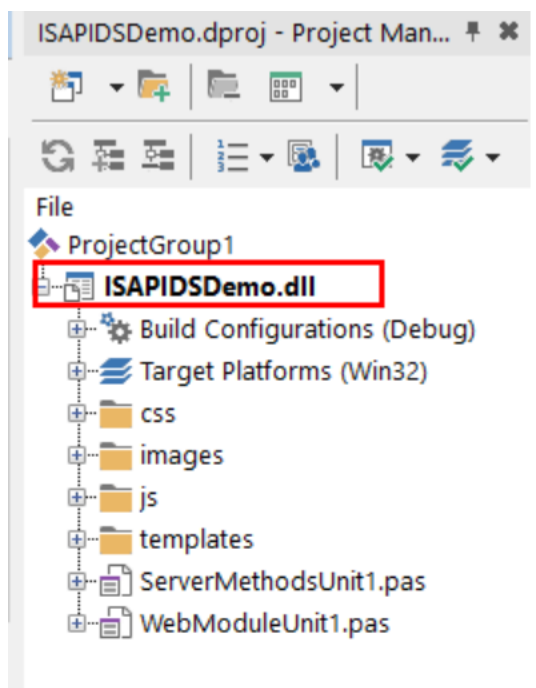



版權所有 請勿翻印

在按下來的選項中請選擇建立如下的 ISAPI 或是 Apache 選項：

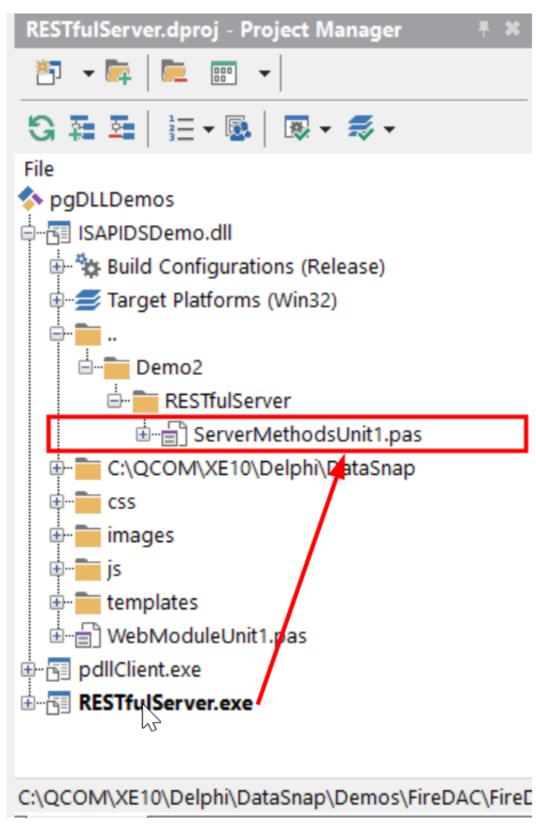



在執行完所有的建立過程後 Delphi 會建立一個 DLL 型態的專案,如下所示:




請讀者注意的是不管是 DLL 或是 EXE 型態的 DataSnap 伺服器都使用 ServerMethodUnit 程式單元，因此只要我們在 ServerMethodUnit 程式單元中小心撰寫程式碼，那麼 ServerMethodUnit 程式單元可以同時使用在 DLL 和 EXE 型態的 DataSnap 伺服器中，如此一來當我們在 EXE 型態的 DataSnap 伺服器除錯和單元測試完成之後就可以很放心的使用在 DLL 型態的 DataSnap 伺服器中了，例如在下圖筆者的 IDE 中即將討論的 ISAPI DataSnap 伺服器中的 ServerMethodUnit 程式單元就是重複使用前面使用在 EXE 型態的 DataSnap 伺服器中的 ServerMethodUnit 程式單元：

版權所有 請勿翻印




開發 ISAPI DataSnap 伺服器和前面開發 EXE 型態的 DataSnap 伺服器是一樣的，都是在 ServerMethodUnit 程式單元實作服務方法，差別只是當開發 ISAPI DataSnap 伺服器完成之後要部署到 IIS 之中，稍後會詳細說明如何部署 ISAPI DataSnap 伺服器，現在先讓我們在此專案的 ServerMethodUnit 程式單元中加入一個服務方法 GetTaipeiHotelsByJSONFromJSONFile()。

版權所有 請勿翻印

為了稍後測試執行效率，因此筆者把前面 TBLTAIPEIHOTELS 資料表中的資料以 JSON 格式都儲存在 TaipeiHotels.JSON 檔案中，因此 GetTaipeiHotelsByJSONFromJSONFile()方法先使用 TFDQuery2 元件把它載入成為資料集，再使用 TJSONArrayBuilder 物件把所有旅館名稱以 JSON 陣列的格式回傳：

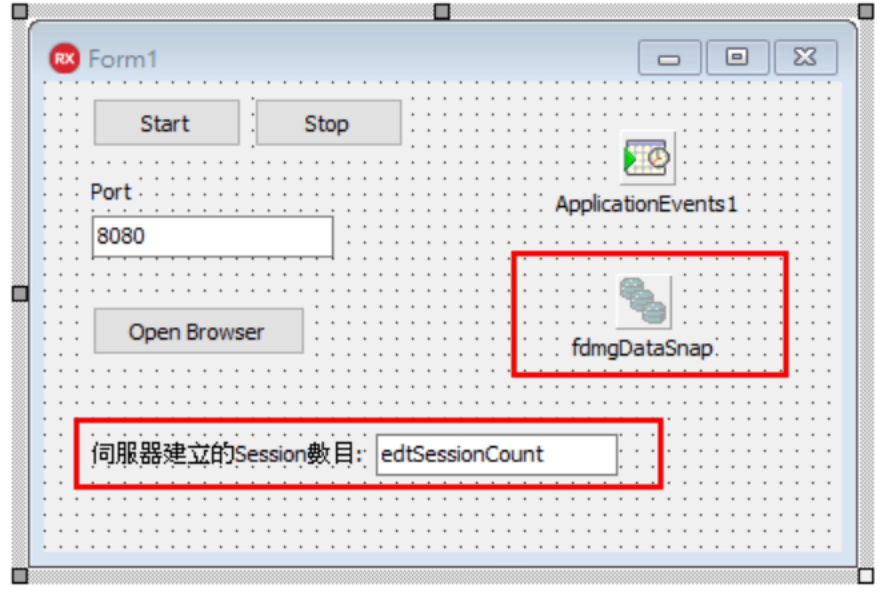
```
const
  TaipeiHotelsJSONFilePath =
'c:\QCOM\XE10\Delphi\DataSnap\DS_APPS\Data\TaipeiHotels.JSON';


function TServerMethods1.GetTaipeiHotelsByJSONFromJSONFile: String;
var
  sw : TStringWriter;
  jtw : TJsonTextWriter;
  jaBuilder : TJSONArrayBuilder;
```

```
  ele : TJSONCollectionBuilder.TElements;
begin
  Result := '';
  if (not FileExists(TaipeiHotelsJSONFilePath)) then
    Exit;

  sw := TStringWriter.Create;
  jtw := TJsonTextWriter.Create(sw);
  jaBuilder := TJSONArrayBuilder.Create(jtw);
  fdqGeneral.LoadFromFile(TaipeiHotelsJSONFilePath, sfJSON);
  fdqGeneral.Open();
  try
    ele := jaBuilder.BeginArray;
    while (not fdqGeneral.Eof) do
    begin
      ele := ele.Add(fdqGeneral.FieldByName('STITLE').AsString);
      fdqGeneral.Next;
    end;
    ele.EndAll;
    Result := sw.ToString;
  finally
    fdqGeneral.Close();
    jaBuilder.Free;
    jtw.Free;
    sw.Free;
  end;
end;
```

版權所有 請勿翻印

由於 ServerMethodUnit 程式單元同時使用於 2 種型態的 DataSnap 伺服器中，因此也順便讓我們同時修改 EXE 型的 DataSnap 伺服器加入 TFDManager 元件以便讓 2 種型態的 DataSnap 伺服器能夠開啟 FireDAC 多執行緒功能以便增加執行效率。

請在 IDE 中開啟前面章節討論的 RESTfulServer 專案，在它的主表單中加入一個 TFDManager 元件：fdmgDataSnap，並且加入可顯示客戶端連結時 EXE 型的 DataSnap 伺服器會建立的 Session 物件數目：




為了讓 FireDAC 順利執行在 IIS 中執行，我們需要正確的設定 FireDAC 使用的組態檔路徑，因此讓我們先在 EXE 型的 DataSnap 伺服器的主表單中定義路徑常數：

```
const
  sTheHotel = '旅館資料';
  DS_APPPATH = 'c:\QCOM\XE10\Delphi\DataSnap\DS_APPS\';
  DEFFILE = 'FDConnectionDefs.ini';
  DRIVERFILE = 'FDDrivers.ini';
```

版權所有 請勿翻印

接著在主表單的 OnCreate 事件呼叫 SetupFDManager()方法設定 TFDManager 元件：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FServer := TIdHTTPWebBrokerBridge.Create(Self);
  SetupFDManager;
end;
```

SetupFDManager()方法的工作就是設定要從什麼路徑載入 FireDAC 的 2 個 INI 組態檔：

```
procedure TForm1.SetupFDManager;
begin
  fdmgDataSnap.Active := False;
  fdmgDataSnap.ConnectionDefFileName := DS_APPPATH + PathDelim + DEFFILE;
```

```
  fdmgDataSnap.DriverDefFileName := DS_APPPATH + PathDelim + DRIVERFILE;
  fdmgDataSnap.Active := True;
end;
```

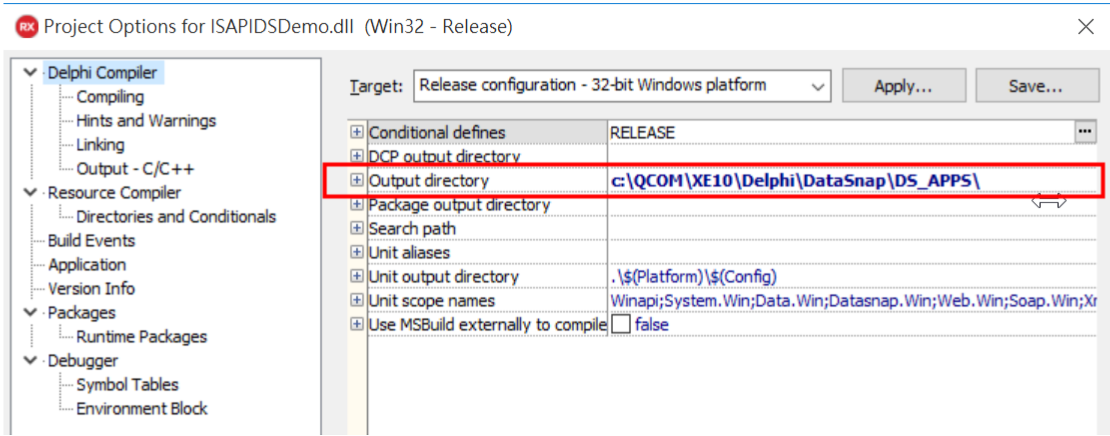
再切換到剛才建立的 DLL DataSnap 伺服器的主程式，同樣修改 DLL 主程式如下：

```
var
  fdmgDataSnap : TFDManager;

procedure SetupFDManager;
begin
  if (fdmgDataSnap = Nil) then
    fdmgDataSnap := TFDManager.Create(Nil);
  fdmgDataSnap.Active := False;
  fdmgDataSnap.ConnectionDefFileName := DS_APPPATH + PathDelim + DEFFILE;
  fdmgDataSnap.DriverDefFileName := DS_APPPATH + PathDelim + DRIVERFILE;
  fdmgDataSnap.Active := True;
end;

procedure TerminateThreads;
begin
  if (fdmgDataSnap <> Nil) then
  begin
    fdmgDataSnap.Close;
    FreeAndNil(fdmgDataSnap);
  end;
  TDSSessionManager.Instance.Free;
  Data.DBXCommon.TDBXScheduler.Instance.Free;
end;

begin
  CoInitFlags := COINIT_MULTITHREADED;
  Application.Initialize;
  Application.WebModuleClass := WebModuleClass;
  SetupFDManager;
  TISAPIApplication(Application).OnTerminate := TerminateThreads;
  Application.Run;
```

版權所有 請勿翻印

```
end.
```

由於稍後我們會建立 IIS 的站台，因此我們需要把編譯的 DLL 部署到我們的 IIS 站台路徑之中，請在 DLL 專案的 Options 對話盒的 Delphi Compiler | Output Directory 中設定稍後 IIS 站台使用的路徑：




現在請同時編譯 DLL 和 EXE 型態的 DataSnap 伺服器，並把 FireDAC 使用的 2 個組態檔都部署到筆者使用的”c:\QCOM\XE10\Delphi\DataSnap\DS_APPS\”路徑中，下圖顯示了編譯並部署的 ISAPIDSDemo.dll 及 FireDAC 使用的 2 個組態檔：

版權所有 請勿翻印

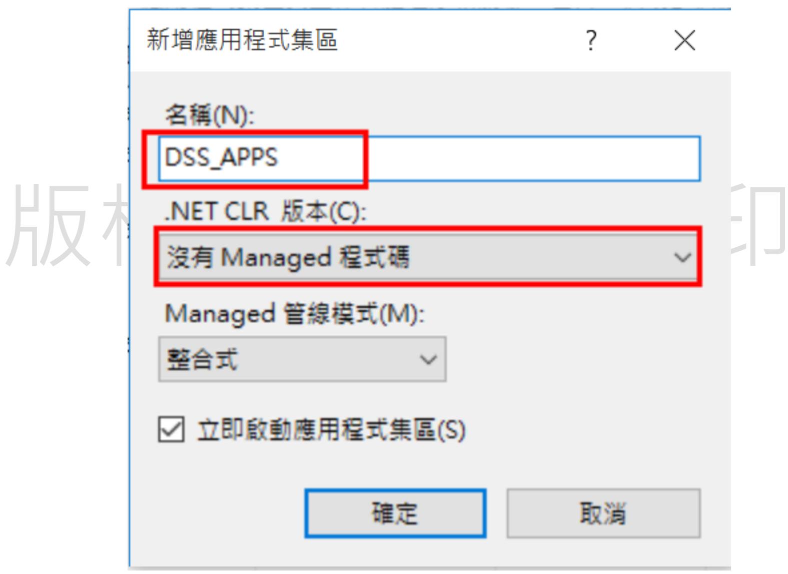
| Name | Ext | Size | Date | Attr |
|---|---|---|---|---|
| [..] | | <DIR> | 2015/10/28 14:09 | ---- |
| [css] | | <DIR> | 2015/10/28 14:09 | -a-- |
| [Data] | | <DIR> | 2015/10/23 14:33 | ---- |
| [images] | | <DIR> | 2015/10/28 14:09 | -a-- |
| [js] | | <DIR> | 2015/10/28 14:09 | -a-- |
| [templates] | | <DIR> | 2015/10/28 14:09 | -a-- |
| ISAPIDSDemo | dll | 7,886,848 | 2015/10/28 14:09 | -a-- |
| Test | html | 416 | 2015/10/21 16:53 | -a-- |
| web | config | 183 | 2015/10/21 16:44 | -a-- |
| FDConnectionDefs | ini | 2,323 | 2015/10/12 16:47 | -a-- |
| FDDrivers | ini | 35 | 2015/04/15 01:56 | -a-- |

接下來的重要工作就是部署此 ISAPI DataSnap 伺服器到 IIS 之中以便讓稍後的客戶端呼叫。

## 10-1-2 部署 ISAPI DataSnap 伺服器

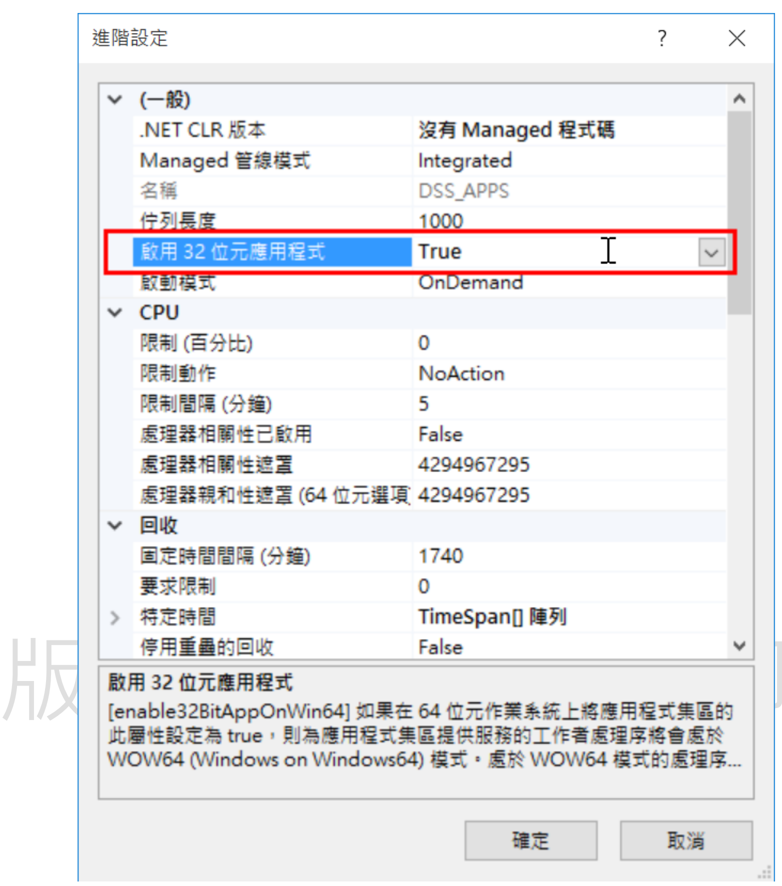
首先請啟動 IIS 管理員(如果您找不到 IIS 管理員，那應該是您尚未安裝 IIS，請到控制台 | 開啟或關閉 Windows 功能中安裝 IIS)，如下圖所示在”應用程式集區” 右擊滑鼠再選擇”新增應用程式集區…”選項：

在” 新增應用程式集區” 對話盒中為您的集區取一個名稱再選擇使用”沒有 Managed 程式碼”選項(這是當然的，因為 Delphi 編譯出來的 DLL 是原生的 DLL)：

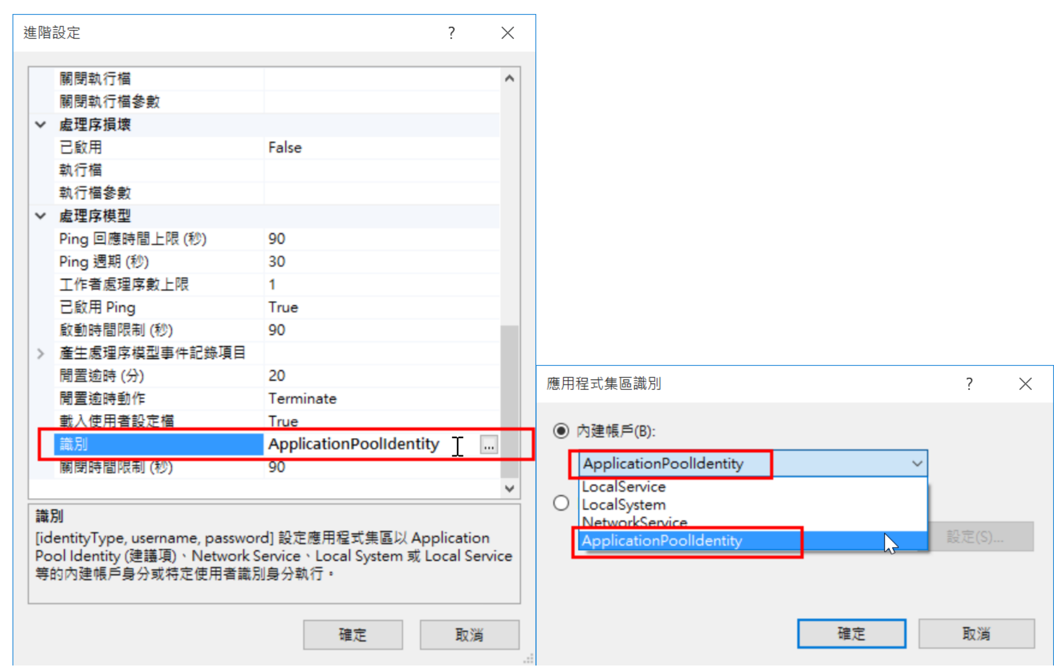



回選確定之後應該就可以看到我們建立了 DSS_APPS 集區，接下來請如下圖在右方點選”進階設定”：

由於我們編譯的 ISAPIDSDemo.dll 是 32 位元的 DLL 應用程式而筆者使用的 OS 是 64 位元，因此需要在”進階設定”中設定如下”啟用 32 位元應用程式”選項，當然如果讀者是把 ISAPIDSDemo.dll 編譯成 64 位元就不需要進行這個設定：




接著您需要設定 IIS 使用什麼身份識別來執行此 DLL 應用程式，在 IIS 7.5 之後新增了”ApplicationPoolIdentity”這個虛擬帳戶，在下圖中可以看到”進階設定”中就內定使用 ApplicationPoolIdentity 帳號權限來執行此應用程式集區，因此如果您的 ISAPI DataSnap 伺服器會存取到一些系統資源，例如檔案，資料庫等那麼您必須確定 ApplicationPoolIdentity 有適當的權限，例如我們的範例 ISAPI DataSnap 伺服器會存取到”c:\QCOM\XE10\Delphi\DataSnap\DS_APPS\”目錄下的 FireDAC 組態檔，那麼就必須設定 ApplicationPoolIdentity 擁有可以讀取此目錄的權限。

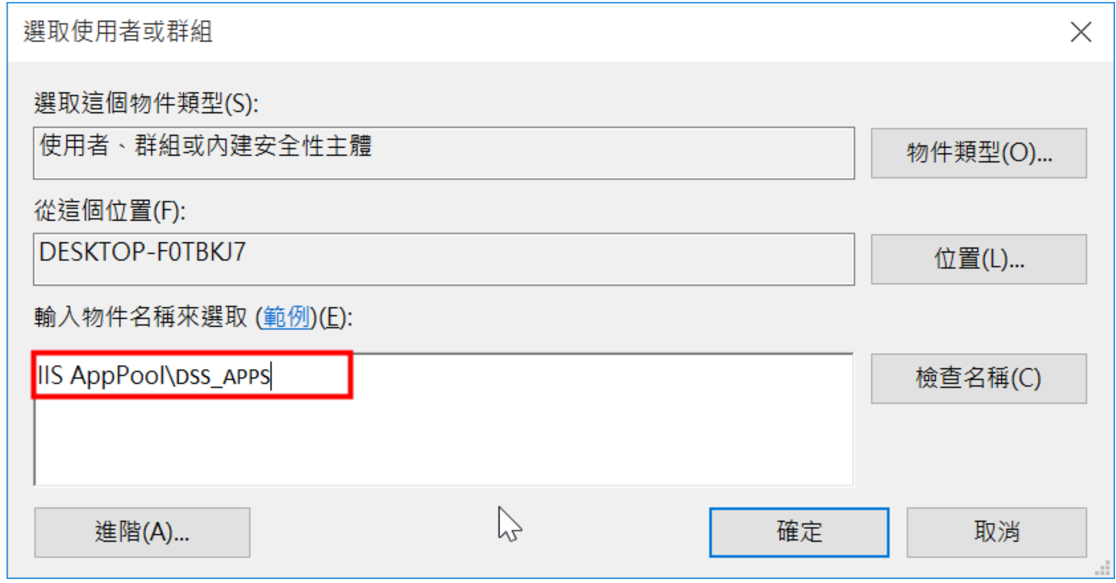



要指定 ApplicationPoolIdentity 可擁有存取”c:\QCOM\XE10\Delphi\DataSnap\DS_APPS\”目錄的權限，我們可以使用檔案管理員在此目錄的安全性設定中使用如下的格式來加入剛才建立的 DSS_APPS 集區有存取權限：

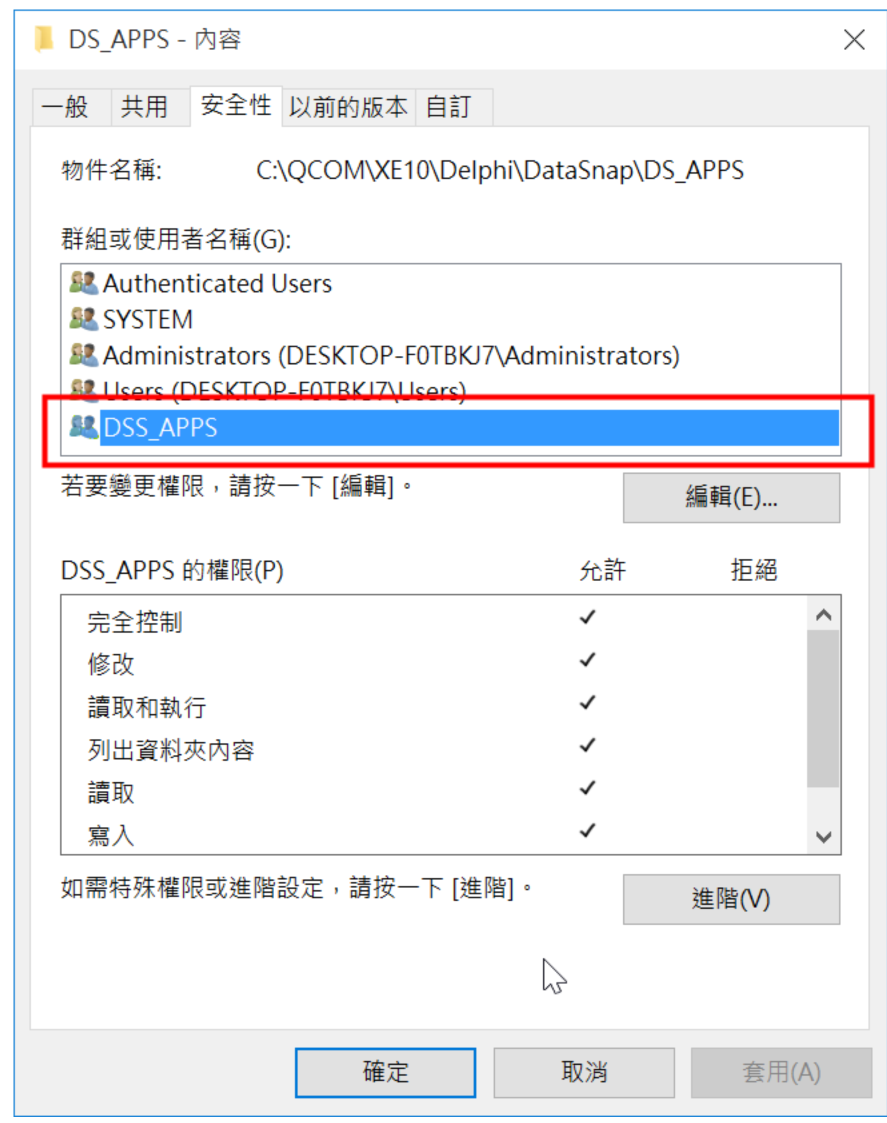
版權所有 請勿翻印

"IIS AppPool\應用程式集區名稱"

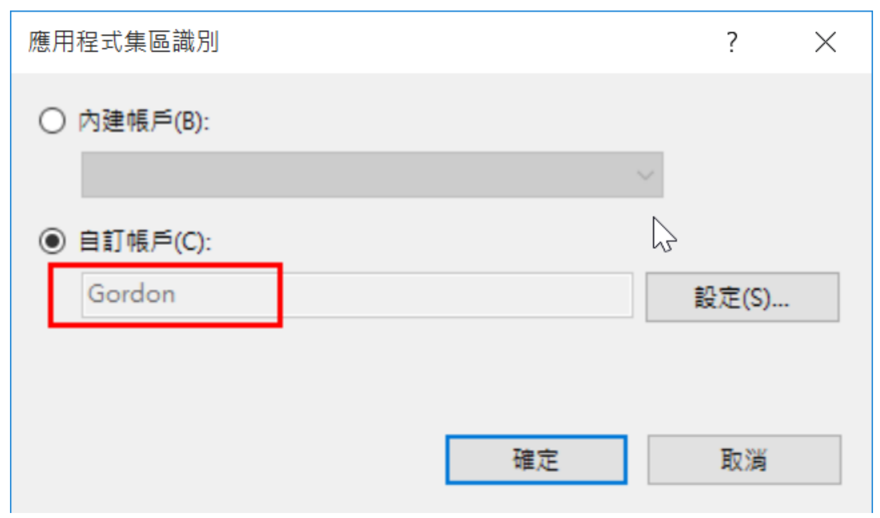
因此我們可以在” c:\QCOM\XE10\Delphi\DataSnap\DS_APPS\”目錄中加入 IIS AppPool\DSS_APPS 集區：



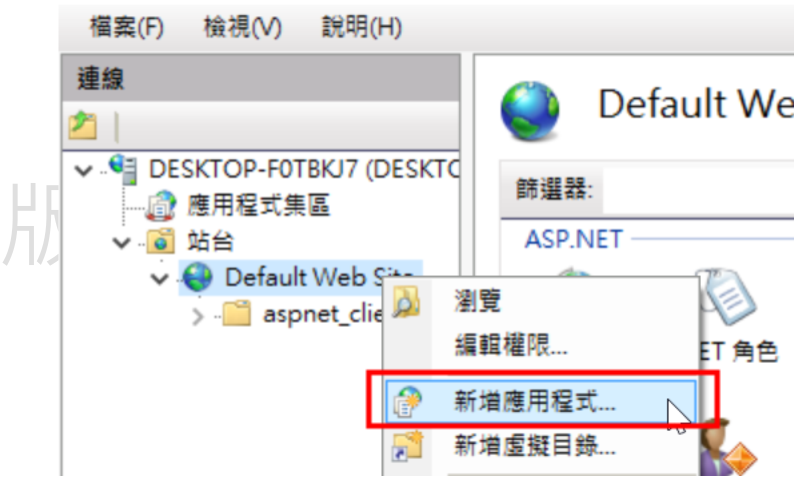
最後在此目錄可存取的群組或使用者中就可以看到 DSS_APPS 集區，接著讀者可以設定 DSS_APPS 集區的存取權限：



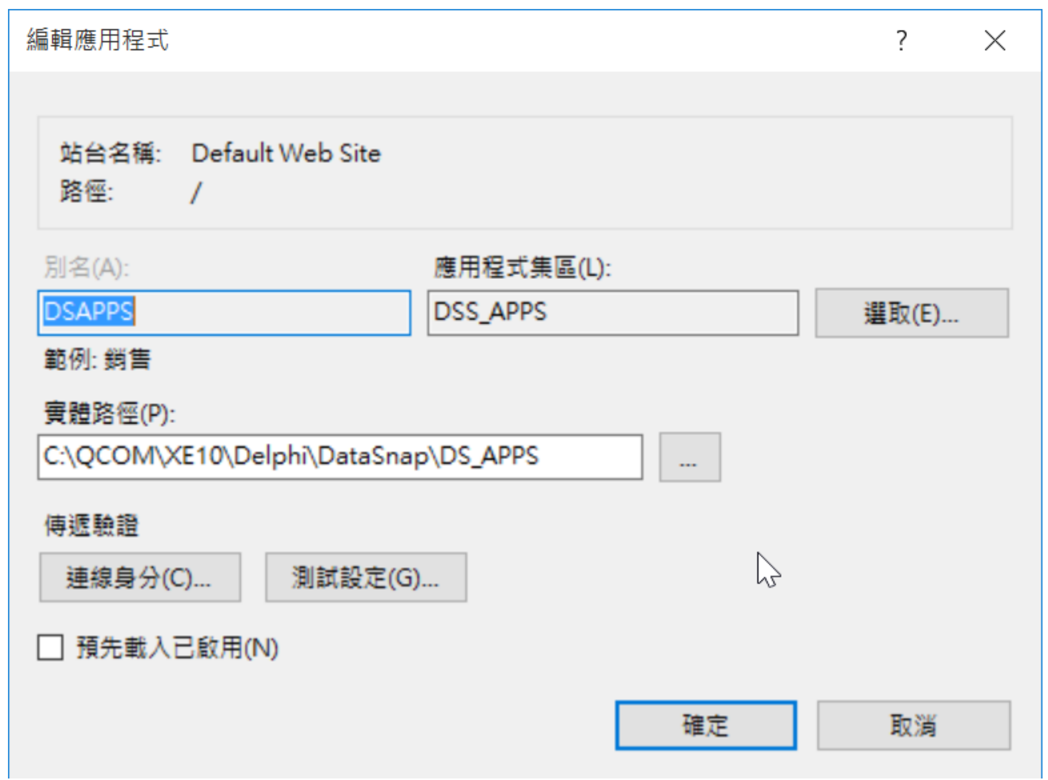

當然您可也可以決定使用特定的帳戶來執行：




設定好應用程式集區之後接下來就可以建立站台了，請如下圖點選 Default Web Site 右擊滑鼠再點選”新增祇用程式…”選項：

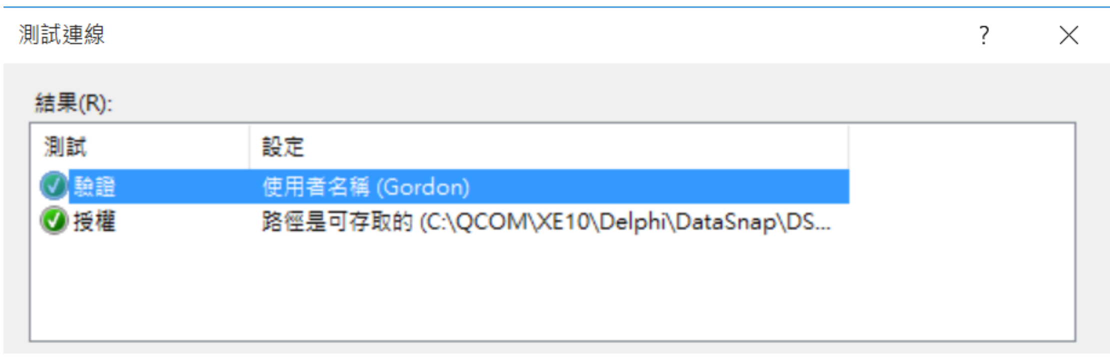



在編輯應用程式對話盒中取一個站台別名，選擇程式集區為 DSS_APPS，再設定此站台使用的實體路徑：



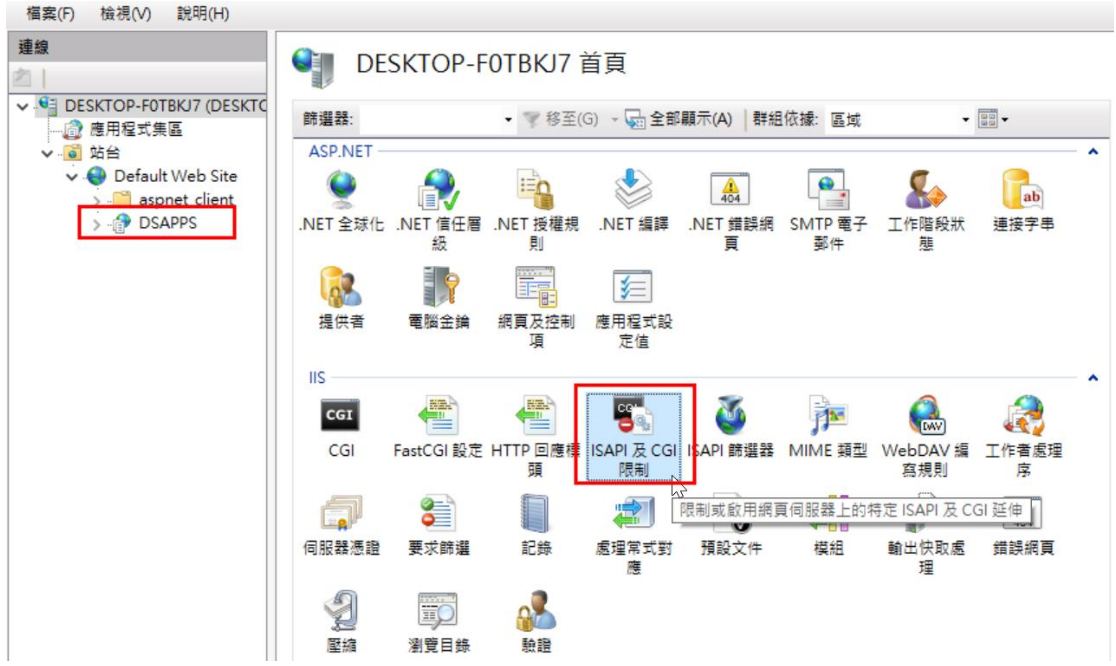

您可以點選"測試設定…"選項，如果您正確設定了使用帳號的存取權限的話，應該就可以看到如下圖顯示測試成功的畫面：

版權所有 請勿翻印

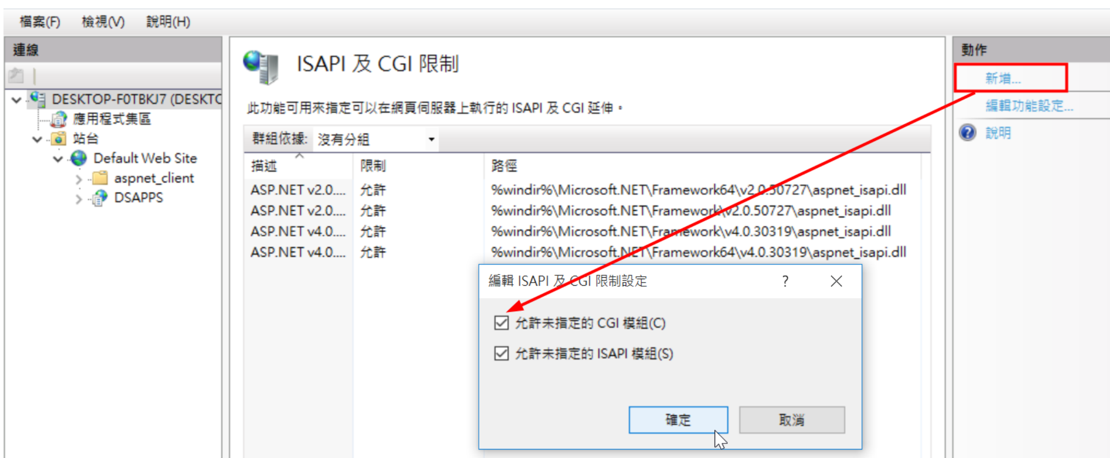



接著，由於我們部署的 DLL DataSnap 伺服器是 ISAPI 應用程式，因此我們需要設定 IIS 允許客戶端呼叫原生的 ISAPI DLL 應用程式。

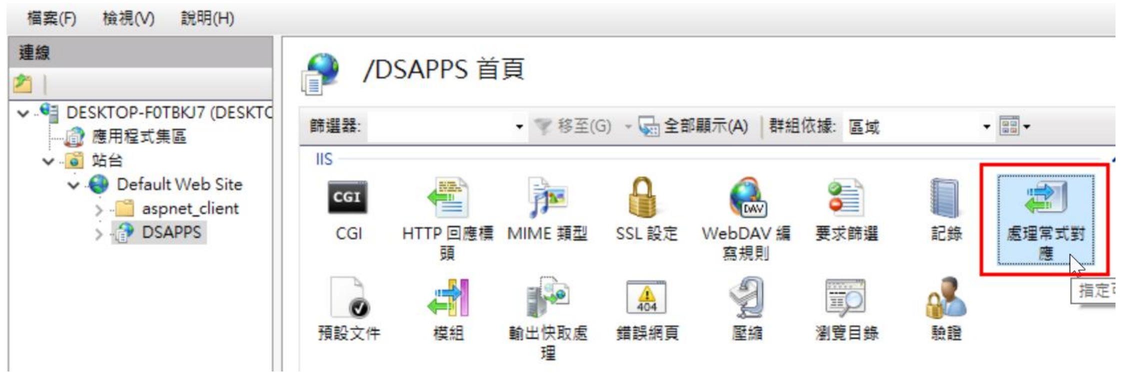
請在剛才建立的 DSAPPS 站台中點選 ISAPI 及 CGI 限制圖像：



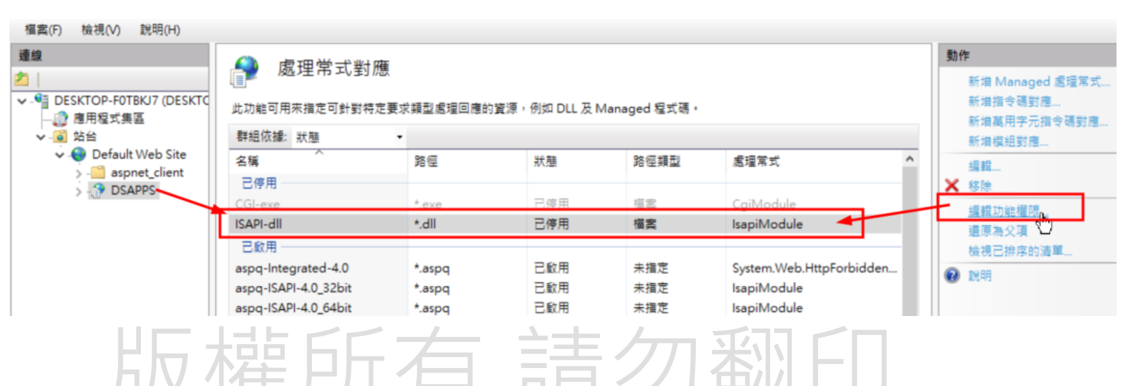

再點選右方的"新增…"選項，如下圖勾選允許 CGI 和 ISAPI：




由於我們的 DataSnap 伺服器是編譯成 DLL 形式，因當我們使用 RESTful URL 執行服務方法時需要設定 IIS 執行此 DataSnap 伺服器 DLL 而不是下載它。請點選下圖的"處理常式對應"圖像：

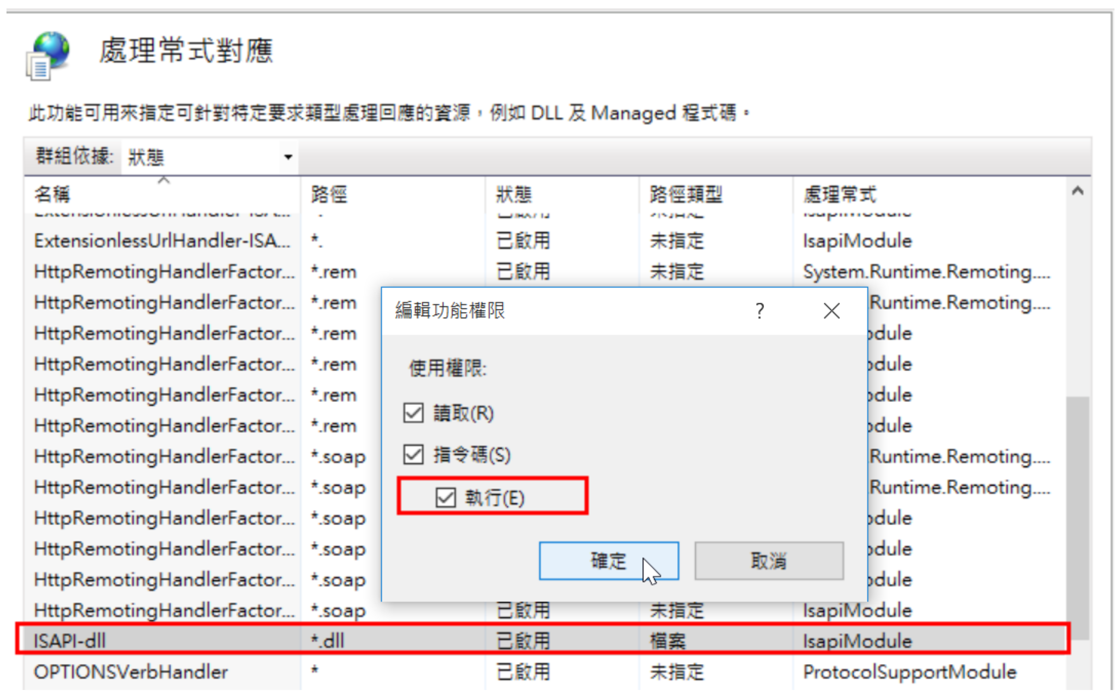



在下圖中可以看到在內定上 IIS 是停用 ISAPI-dll 的，所以我們需要開啟它。請點選右方的”編輯功能權限…”選項：



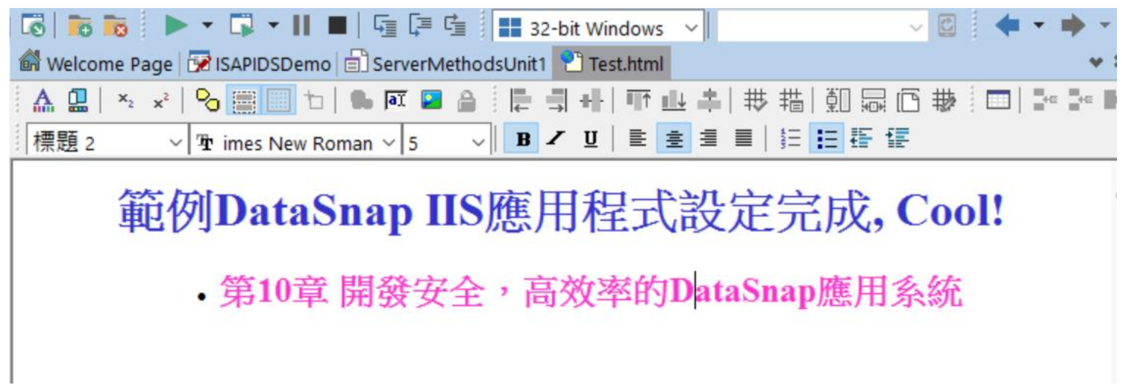

版權所有 請勿翻印

在出現的對話盒中勾選”執行”選項再點選確定按鈕：

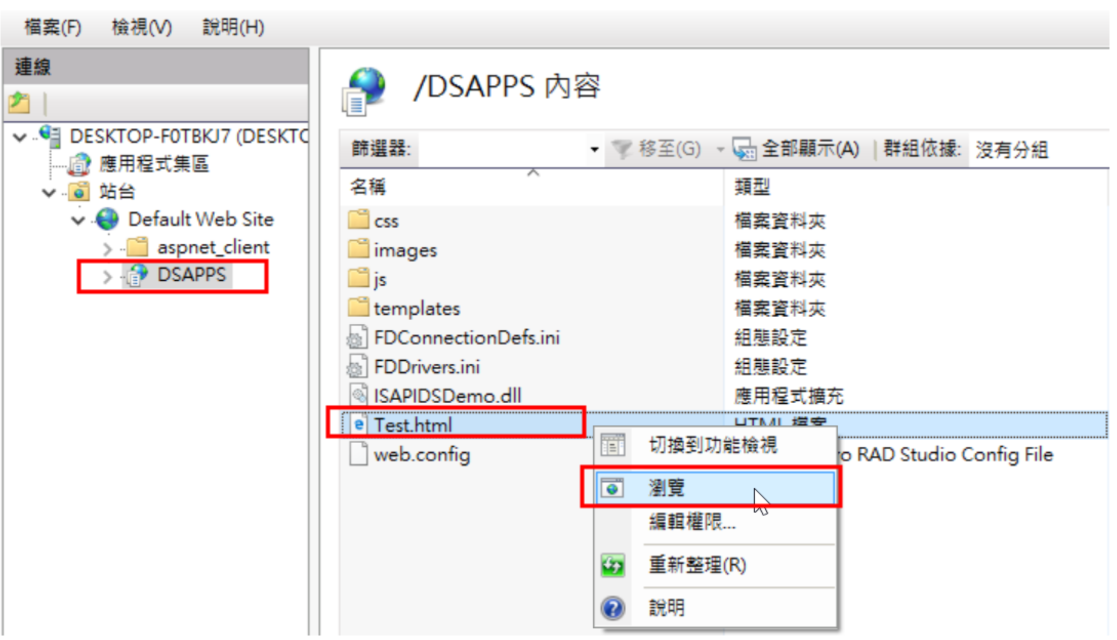



到了這裡所有的設定應該就完成了，現在我們可以測試一下在前面設定的站台是否可以正常工作。我們可以在 Delphi IDE 中建立一個如下的 Test.html

檔案(File | New | Other... | Web Document | HTML Page)並簡單的打入一些測試文字：




然後把它儲存到站台的實體目錄中，回到 IIS 管理員點選 DSAPPS 站台，再點選下方的”內容檢視頁面”就可以看到 Test.HTML，使用滑鼠右擊它再點選”瀏覽”選項：




如果能在瀏覽器中看到如下的畫面就代表我們建立的站台可以正常工作了：

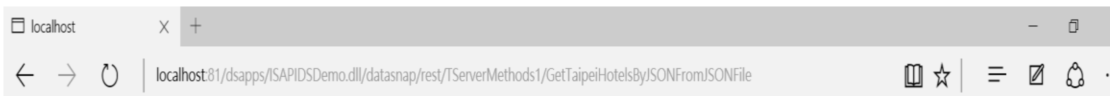
範例DataSnap IIS應用程式設定完成, Cool!

- 第10章 開發安全，高效率的DataSnap應用系統

由於這 2 個範例 DataSnap 伺服器都是 RESTful 型態的伺服器，因此我們現在就可以直接使用瀏覽器來呼叫其中的服務方法，例如使用如下的 URL 就可以呼叫 ISAPI 的 DataSnap 伺服器：

```
http://localhost:81/dsapps/ISAPIDSDemo.dll/datasnap/rest/TServerMethods1/GetTaip
eiHotelsByJSONFromJSONFile
```

因此啟動瀏覽器並使用如上的 URL 後如果看到如下的結果就代表成功的呼叫了 ISAPI 的 DataSnap 伺服器提供的 GetTaipeiHotelsByJSONFromJSONFile() 方法(別擔心顯示的結果，那只是 Unicode)：



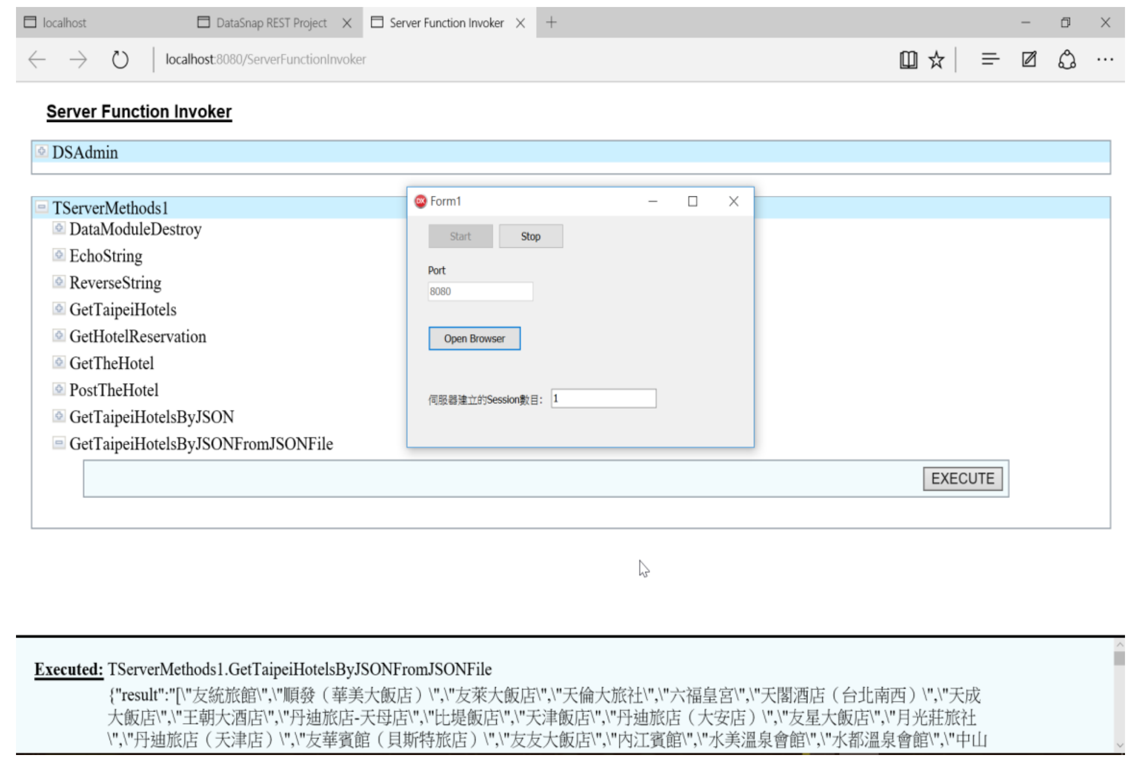

{"result":["[\"\u53CB\u7D71\u65C5\u9928\",\"\u9806\u767C\uFF08\u83EF\u7F8E\u5927\u98EF\u5E97\uFF09\",\"\u53CB\u840A\u5927\u98EF\u5E97\",\"\u5929
\u502B\u5927\u65C5\u793E\",\"\u516D\u798F\u7687\u5BAE\",\"\u5929\u95A3\u9152\u5E97\uFF08\u53F0\u5317\u5357\u897F\uFF09\",\"\u5929\u6210\u5927\u98EF\u5E97
\",\"\u738B\u671D\u5927\u9152\u5E97\",\"\u4E39\u8FEA\u65C5\u5E97-\u5929\u6BCD\u5E97\",\"\u6BD4\u5824\u98EF\u5E97\",\"\u5929\u6D25\u98EF\u5E97\",\"\u4E39
\u8FEA\u65C5\u5E97\uFF08\u5927\u5B89\u5E97\uFF09\",\"\u53CB\u661F\u5927\u98EF\u5E97\",\"\u6708\u5149\u838A\u65C5\u793E\",\"\u4E39\u8FEA\u65C5\u5E97
\uFF08\u5929\u6D25\u5E97\uFF09\",\"\u53CB\u83EF\u8CD3\u9928\uFF08\u8C9D\u65AF\u7279\u65C5\u5E97\uFF09\",\"\u53CB\u53CB\u5927\u98EF\u5E97\",\"\u5167

如果要呼叫 EXE 的 DataSnap 伺服器，那可以在瀏覽器中使用如下的 URL：

版權所有 請勿翻印

```
http://localhost:8080/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSO
NFile
```

或是直接藉由 EXE 的 DataSnap 伺服器啟動瀏覽器來呼叫 GetTaipeiHotelsByJSONFromJSONFile()方法，如果看到如下的結果畫面代表也成功的呼叫了 EXE 的 DataSnap 伺服器：

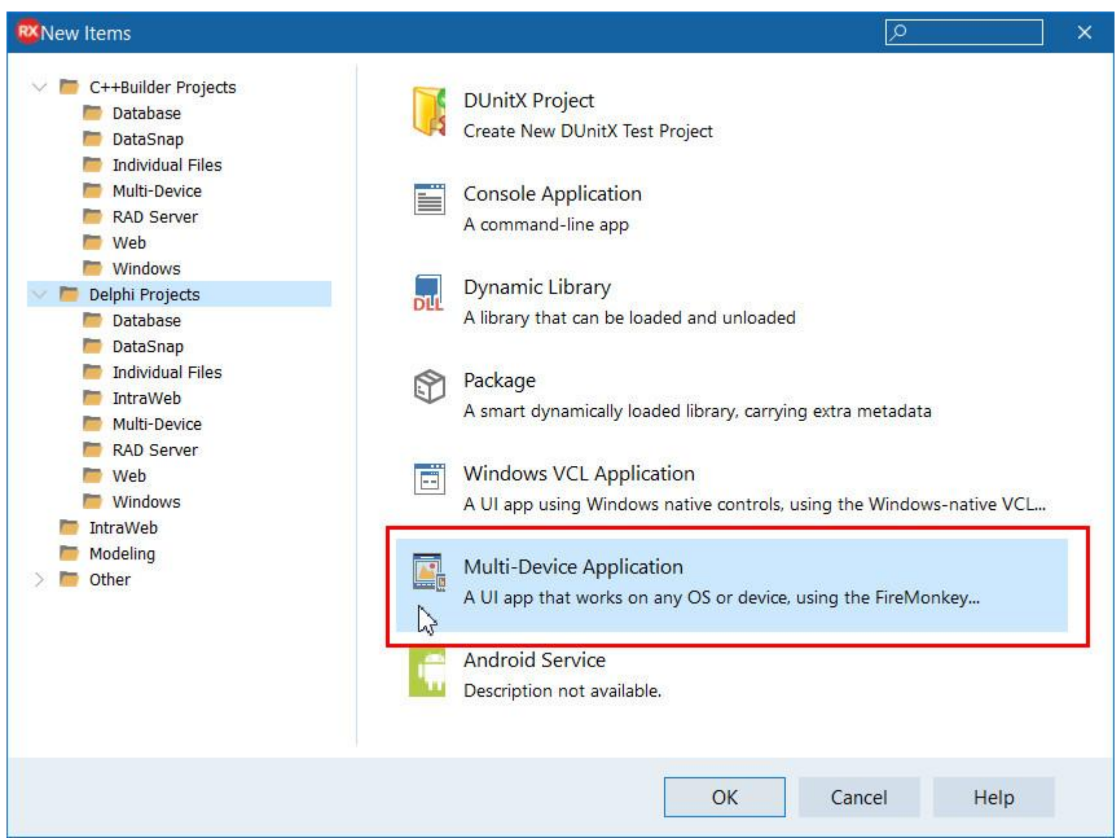



現在已經測試了不管是 EXE 或是 DLL 型態的 DataSnap 伺服器都可以正常的工作，接著讓我們開發一個簡單的 FireMonkey 客戶端來呼叫 GetTaipeiHotelsByJSONFromJSONFile() 方法並準備開始比較這 2 種 DataSnap 伺服器的執行效率，更重要的是開始調校它們的執行效率。

版權所有 請勿翻印

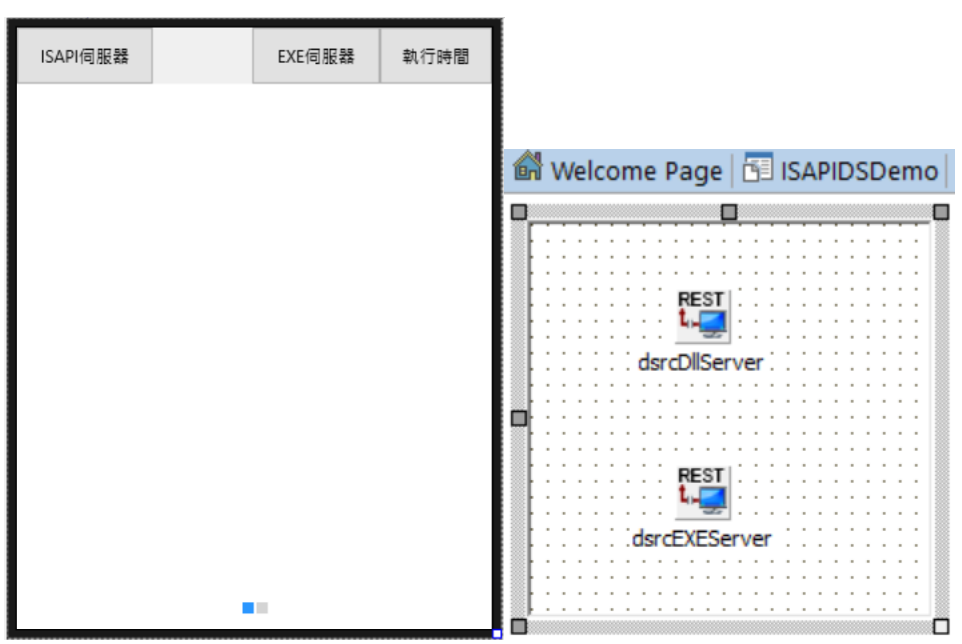
## 10-1-2 開發客戶端

先在專案群組中建立一個 Multi-Device 專案：




版權所有 請勿翻印

再於 Multi-Device 專案中建立一個資料模組，放入 2 個 TDSRestConnection 元件分別連結到 DLL 和 EXE 的 DataSnap 伺服器，再於主表格中使用 2 個 TButton 元件呼叫：




由於呼叫 DLL 和 EXE 的 DataSnap 伺服器幾乎是一樣的，因此讓我們只說明一個場景即可。下面是主表格中” EXE 伺服器”按鈕的實作程式碼，它呼叫

GetTaipeiHotelsFromEXEServer()方法來執行 EXE 的 DataSnap 伺服器中的服務方法：

```
procedure TfmMainForm.btnEXEGetHotelsClick(Sender: TObject);
begin
  GetTaipeiHotelsFromEXEServer;
  ShowRunTime('向 EXE 伺服器取得旅館時間');
end;
```

GetTaipeiHotelsFromEXEServer() 藉由建立客戶端的 TServerMethods1Client 物件後就可直接呼叫 EXE 的 DataSnap 伺服器中的 GetTaipeiHotelsByJSONFromJSONFile()方法：

```
procedure TfmMainForm.GetTaipeiHotelsFromEXEServer;
var
  aServer : TServerMethods1Client;
  sJSONData : String;
begin
  lStart := Now;
  aServer := TServerMethods1Client.Create(dmClient.dsrcEXEServer);
  try
    sJSONData := aServer. GetTaipeiHotelsByJSONFromJSONFile();
    ShowJSONHotels(lvTaipeiHotels, sJSONData);
  finally
    aServer.Free;
    lEnd := Now;
  end;
end;
```

版權所有 請勿翻印

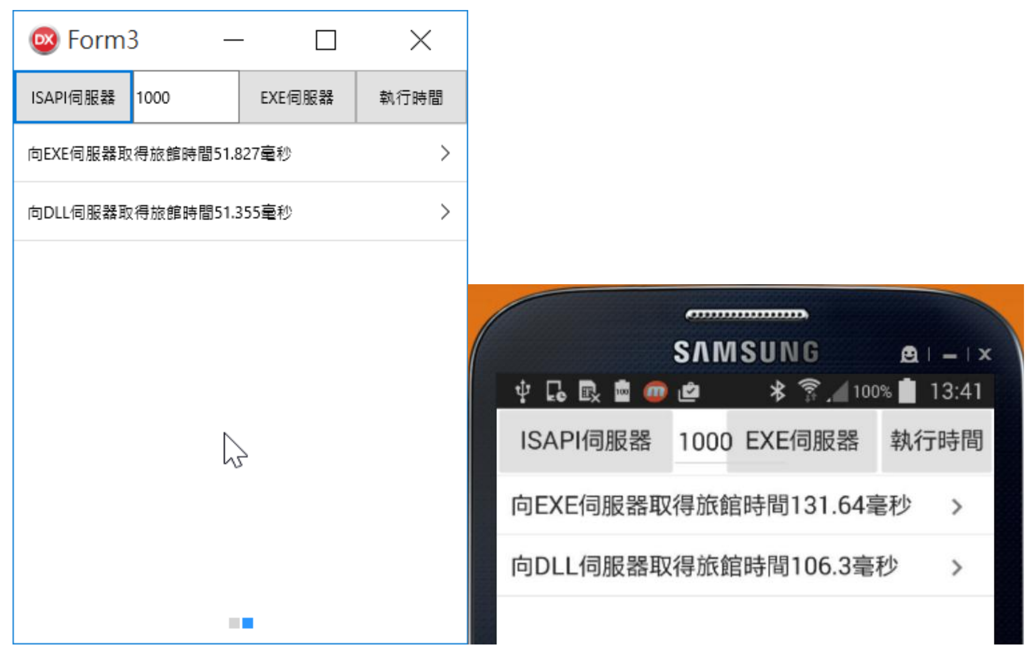
由於 GetTaipeiHotelsByJSONFromJSONFile()方法回傳 JSON 陣列的資料，因此 ShowJSONHotels()方法只是使用 TJSONTextReader 類別物件解析其中的資料並顯示在 TListView 元件中：

```
procedure TfmMainForm.ShowJSONHotels(aLV : TListView; sHotels: String);
var
  sr : TStringReader;
  jr : TJSONTextReader;
  iCount : Integer;
begin
  aLV.Items.Clear;
```

```
  sr := TStringReader.Create(sHotels);
  jr := TJSONTextReader.Create(sr);
  iCount := 0;
  try
    while (jr.Read) do
    begin
      case jr.TokenType of
        TJsonToken.String:
        begin
          aLV.items.Add.Text := jr.Value.ToString;
          Inc(iCount);
        end;
      end;
    end;
  finally
    aLV.items.Add.Text := '一共取得 ' + iCount.ToString() + ' 筆資料';
    sr.Free;
    jr.Free;
  end;
end;
```

版權所有 請勿翻印

下面的 2 個畫面是分別用 Windows 和 Android 手機呼叫 1000 次後平均每一次存取 396 筆台北旅館名稱的執行結果。



從上面的執行結果來看似乎 DLL 型態的 DataSnap 伺服器執行效率雖然比 EXE 型態的 DataSnap 伺服器來得好，但也沒有太大的差距，那麼值得花那麼多的功夫使用和部署 DLL 型態的 DataSnap 伺服器嗎？

其實上面的執行結果只是單一客戶端，多次呼叫的結果。然而在實際的應用中 DataSnap 伺服器會服務大量的客戶端，因此我們應該使用大量客戶端呼叫這 2 種 DataSnap 伺服器執再來比較，也才看得出來在真實世界中這 2 種 DataSnap 伺服器的差異。

筆者並沒有大量客戶端的硬體環境，但沒關係，我們可以藉由各種壓力測試工具來模擬各種狀況的客戶端來對這 2 種 DataSnap 伺服器進行壓力測試並根據壓力測試結果來進行效率調整，這正是下一節要討論的內容。

## 10-2 DataSnap 伺服器效能比較

現在我們需要使用實際的工具來找出上一小節 DLL 和 EXE 型態的 DataSnap 伺服器的差別，筆者將使用下面的 3 個開源工具來對不同型態 DataSnap 伺服器進行壓力測試：

| 測試工具 | URL |
|---|---|
| ApacheBench | http://httpd.apache.org/docs/2.2/programs/ab.html |
| WeigHttp | https://build.opensuse.org/package/show/home:stbuehler:lighttpd-utils/weighttp |
| Apache JMeter | http://jmeter.apache.org/ |

版權所有 請勿翻印

同時筆者也將使用下面的商用工具來對不同型態 DataSnap 伺服器進行壓力測試：

| 測試工具 | URL |
|---|---|
| WAPT | http://www.loadtestingtool.com/ |

一開始讓我們直接測試前面開發的 EXE 和 DLL 型態的 DataSnap 伺服器，看看這 2 者之間的真實差異。在下面的測試中，筆者是在 VMWare 的虛擬機中執行壓力測試，使用 4G Ram 以及 2.3G 的 Intel Core i7 CPU。

現在請執行前面的範例 EXE DataSnap 伺服器並啟動 IIS。

### ApacheBench

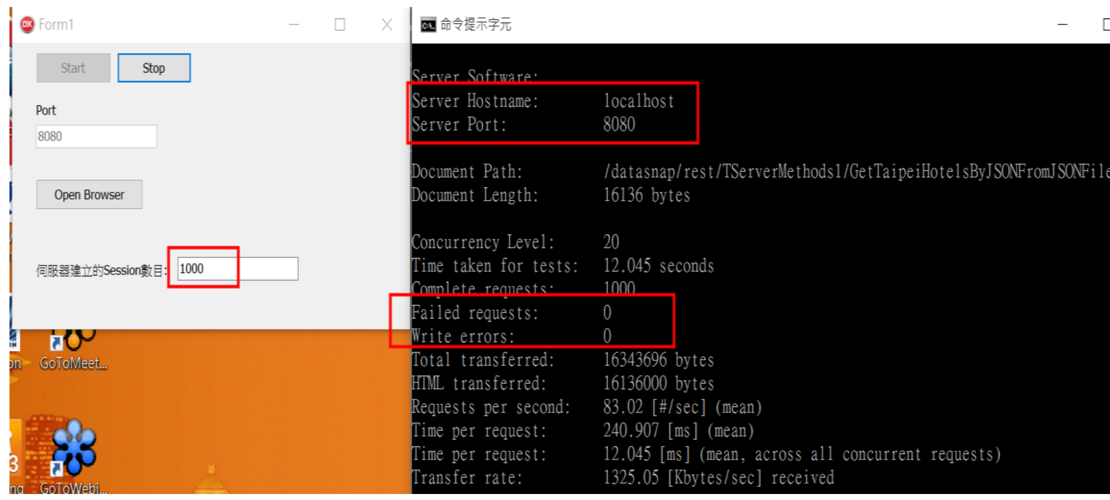
首先讓我們使用下面的指令來模擬 20 個客戶端提出 1000 個請求，分別呼叫 EXE 和 DLL 的 DataSnap 伺服器，下面是呼叫 EXE DataSnap 伺服器的指令：

```
ab -n 1000 -c 20
http://localhost:8080/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSO
NFile
```

下面是呼叫 DLL DataSnap 伺服器的指令：

```
ab -n 1000 -c 20
http://localhost:81/dsapps/ISAPIDSDemo.dll/datasnap/rest/TServerMethods1/GetTaip
eiHotelsByJSONFromJSONFile
```

從下圖可以看到當 ApacheBench 向 EXE 型態的 DataSnap 伺服器提出請求時 EXE 型態的 DataSnap 伺服器在伺服端建立了 1000 個 Session，而且沒有發生任何的錯誤：




但如果我們繼續增壓模擬 40 個客戶端：

```
ab -n 1000 -c 40
http://localhost:8080/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSO
NFile
```

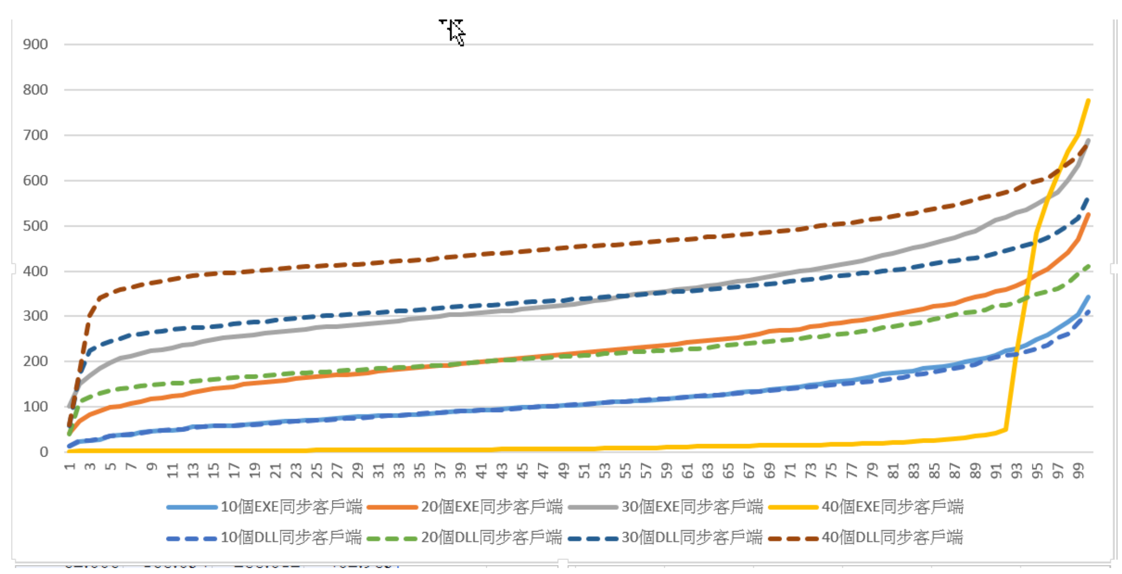
就會看到 EXE 型態的 DataSnap 伺服器開始出現錯誤：

```
Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
Document Length:        74 bytes

Concurrency Level:      40
Time taken for tests:   1.455 seconds
Complete requests:      1000
Failed requests:        80
   (Connect: 0, Receive: 0, Length: 80, Exceptions: 0)
Write errors:           0
```

但同樣模擬 40 個客戶端 DLL 型態的 DataSnap 伺服器仍然不會出現任何的錯誤：

```
Server Software:        Microsoft-IIS/10.0
Server Hostname:        localhost
Server Port:            81

Document Path:          /dsapps/ISAPIDSDemo.dll/datasnap/rest/TServerMethods1/GetTaipeiHotelsB
yJSONFromJSONFile
Document Length:        16136 bytes

Concurrency Level:      40
Time taken for tests:   11.600 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
```

筆者使用 ApacheBench 同時模擬 10，20，30 和 40 個客戶端繪出下面的比較圖，從這第 1 個比較圖中可以看到 2 個現象，第 1 是 DLL 型態的 DataSnap 伺服器隨著客戶端要求次數愈來愈後執行效率都比 EXE 型態的 DataSnap 伺服器好，另外就是下圖最下方的 40 個 EXE 同步客戶端很明顯的出了問題，無法回應客戶端的請求：




從上面的觀察可以瞭解 EXE 型態的 DataSnap 伺服器在客戶端超過 30 左右之後就會開始發生問題，但使用同一份程式的 DLL 型態的 DataSnap 伺服器從下圖可以看到 ApacheBench 在筆者的虛擬機中持續加壓到 500 個客戶端時仍然沒有發生任何的錯誤：

```
Server Software:        Microsoft-IIS/10.0
Server Hostname:        localhost
Server Port:            81

Document Path:          /dsapps/ISAPIDSDemo.dll/datasnap/rest/TServerMethods1/GetTaipeiHotelsB
yJSONFromJSONFile
Document Length:        16136 bytes

Concurrency Level:      500
Time taken for tests:   11.694 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      16374671 bytes
HTML transferred:       16136000 bytes
Requests per second:    85.51 [#/sec] (mean)
Time per request:       5847.245 [ms] (mean)
Time per request:       11.694 [ms] (mean, across all concurrent requests)
Transfer rate:          1367.39 [Kbytes/sec] received
```

EXE 型態的 DataSnap 伺服器只能服務 30 多個客戶端的應用是可能被接受的，如果真的如此那 EXE 型態的 DataSnap 伺服器只能使用來做上課學習之用，無法被使用在實際的應用中，但為什麼使用完全相同程式碼的 DLL 型態的 DataSnap 伺服器卻可以服務超過 500 個客戶端呢？因此這其中一定發生了什麼問題，但在找出問題並解決之前，讓我們再用其他的測試工具來加壓看看，是不是也會反映相同的現象。

## WeigHttp

WeigHttp 是筆者另外一個常使用的工具，它和 ApacheBench 使用的方法差不多，只是 WeigHttp 的好處是如果測試的機器如果有多核心的話，那麼 WeigHttp 可以同時使用多核心同時加壓測試讓測試過程更快速完成。

我們使用下面的命令列測試 2 種 DataSnap 伺服器：

```
weighttp -n 1000 -c 40
http://localhost:8080/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSO
NFile
```

```
weighttp -n 1000 -c 40
http://localhost:81/dsapps/ISAPIDSDemo.dll/datasnap/rest/TServerMethods1/GetTaip
eiHotelsByJSONFromJSONFile
```

從下面測試 EXE 型態的 DataSnap 伺服器的結果來看 WeigHttp 也明確的指出用 40 個客戶端測試時就會出現錯誤：

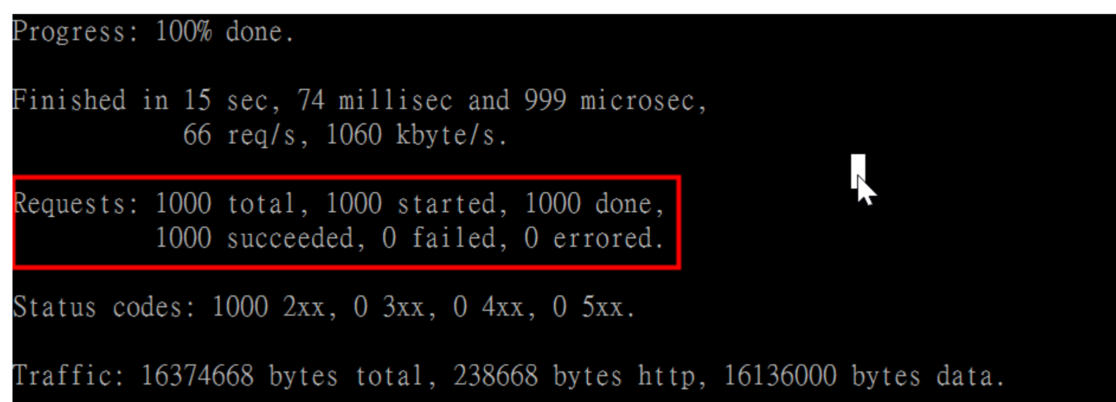
```
Spawning thread #1: 40 concurrent requests, 1000 total requests.
Progress:  10% done.
Progress:  20% done.
Progress:  30% done.
Progress:  40% done.
Progress:  50% done.
Progress:  60% done.
Progress:  70% done.
Progress:  80% done.
Progress:  90% done.
Progress: 100% done.

Finished in 1 sec, 627 millisec and 0 microsec,
            614 req/s, 990 kbyte/s.

Requests: 1000 total, 1000 started, 1000 done,
          88 succeeded, 912 failed, 0 errored.

Status codes: 88 2xx, 0 3xx, 0 4xx, 912 5xx.
```

但對於 DLL 型態的 DataSnap 伺服器即使使用 500 客戶端，使用 10 個執行緒加壓測試：

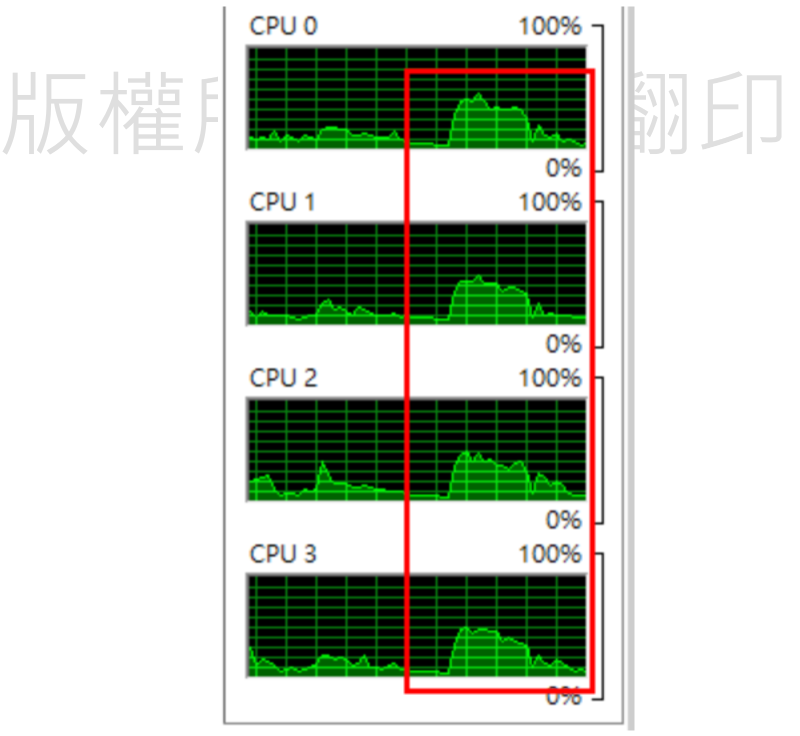
```
weighttp -n 1000 -c 500  -t 10
http://localhost:81/dsapps/ISAPIDSDemo.dll/datasnap/rest/TServerMethods1/GetTaip
```

```
eiHotelsByJSONFromJSONFile
```

我們仍然可以從下圖看到 DLL 型態的 DataSnap 伺服器完全可以應付請求，不會發生錯誤：




而且 OS 也可明確看到 WeigHttp 同時使用了筆者 VM 中的 4 個核心同時加壓測試：



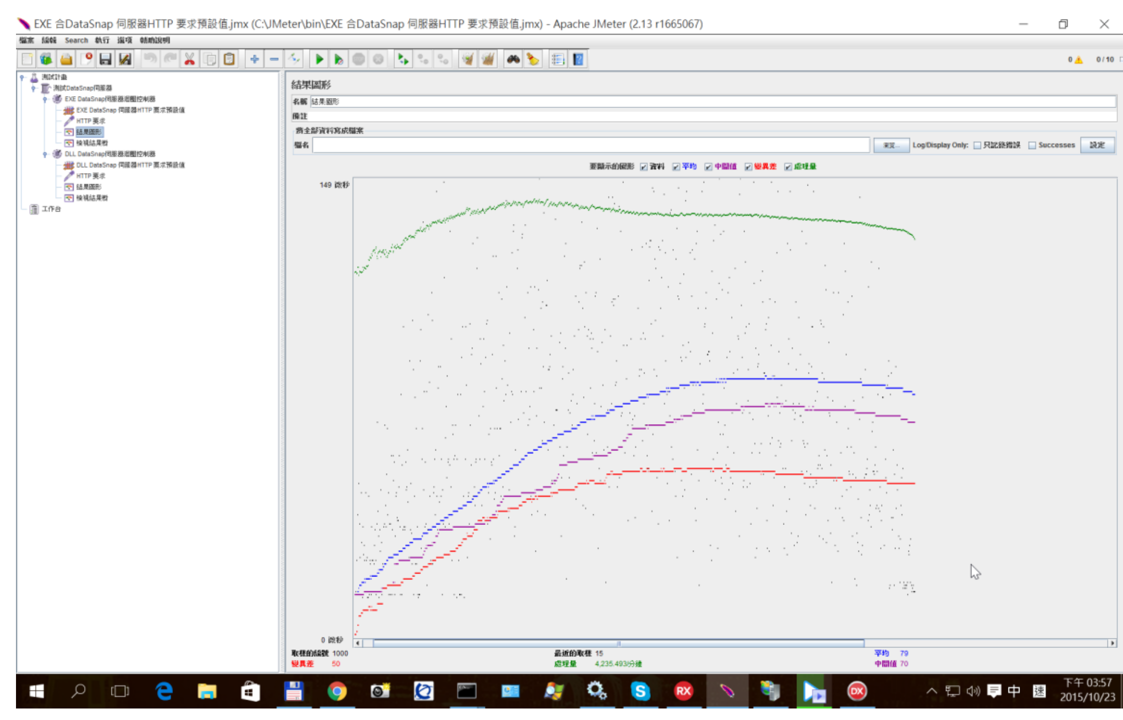

版權 翻印

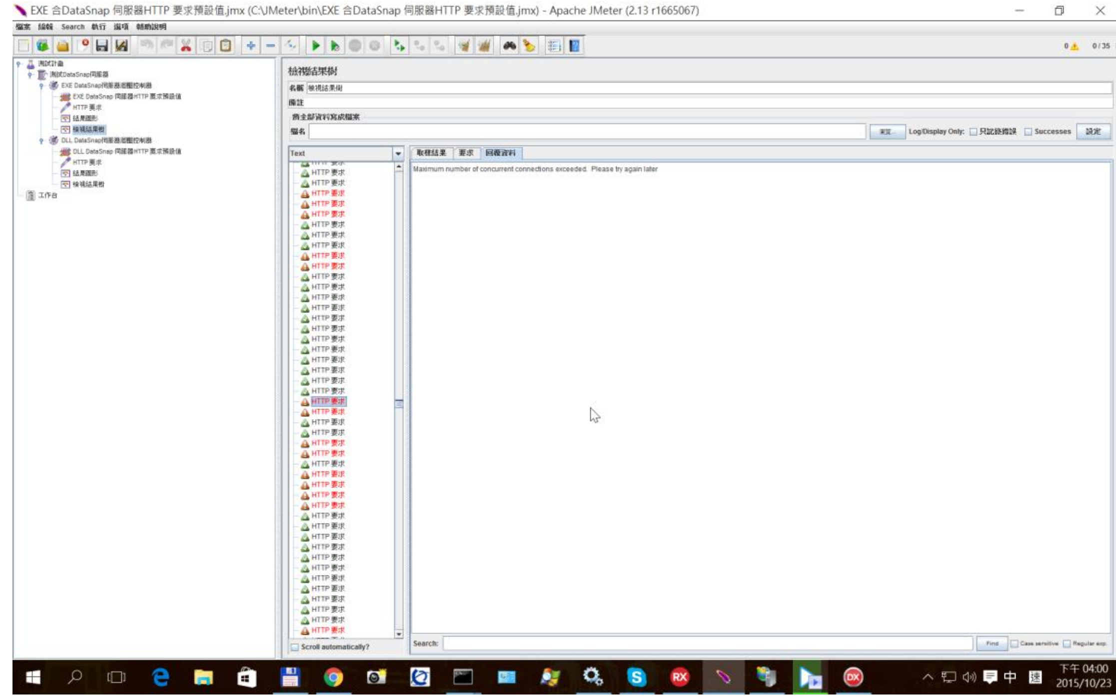
## Apache JMeter

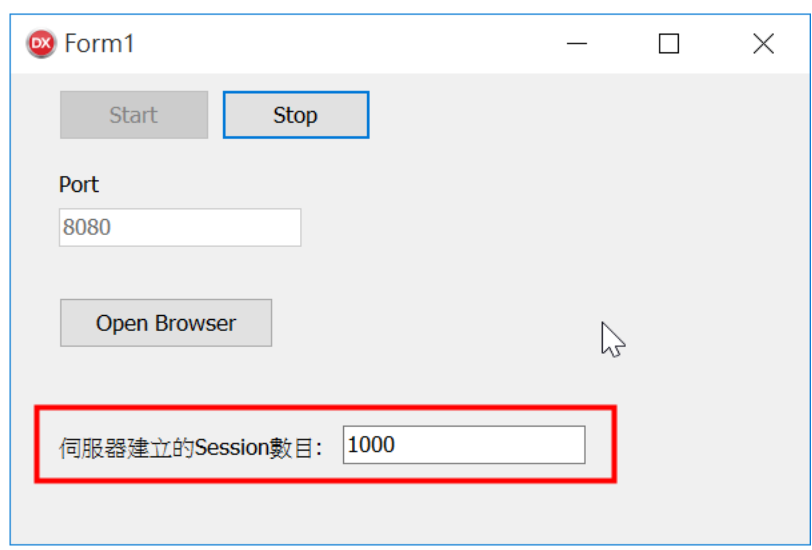
最後讓我們使用 Apache 的 JMeter 來測試這 2 種 DataSnap 伺服器，JMeter 應該是這 3 個壓力測試工具中最強大的，功能繁多且可提供詳細的結果

報告，例如 JMeter 就可以告訴我們當 EXE 型態的 DataSnap 伺服器在服務 40 個客戶端時發生了什麼錯誤。

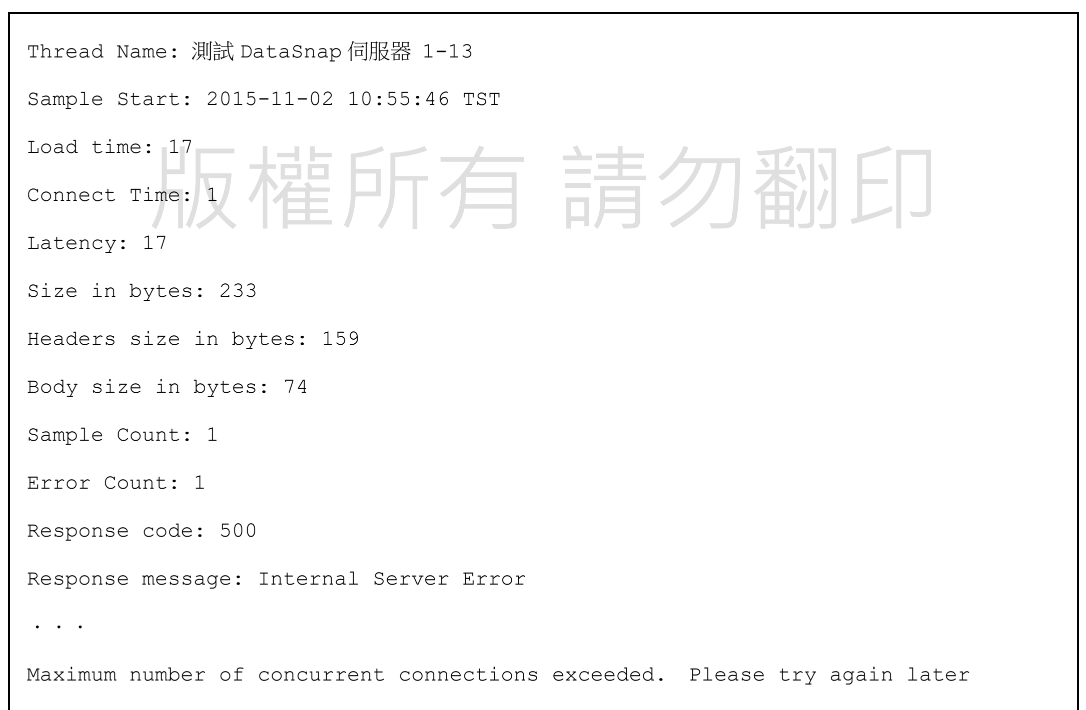
從下面測試 EXE 型態的 DataSnap 伺服器的結果來看 JMeter 也明確的指出用 40 個客戶端測試時就會出現錯誤：





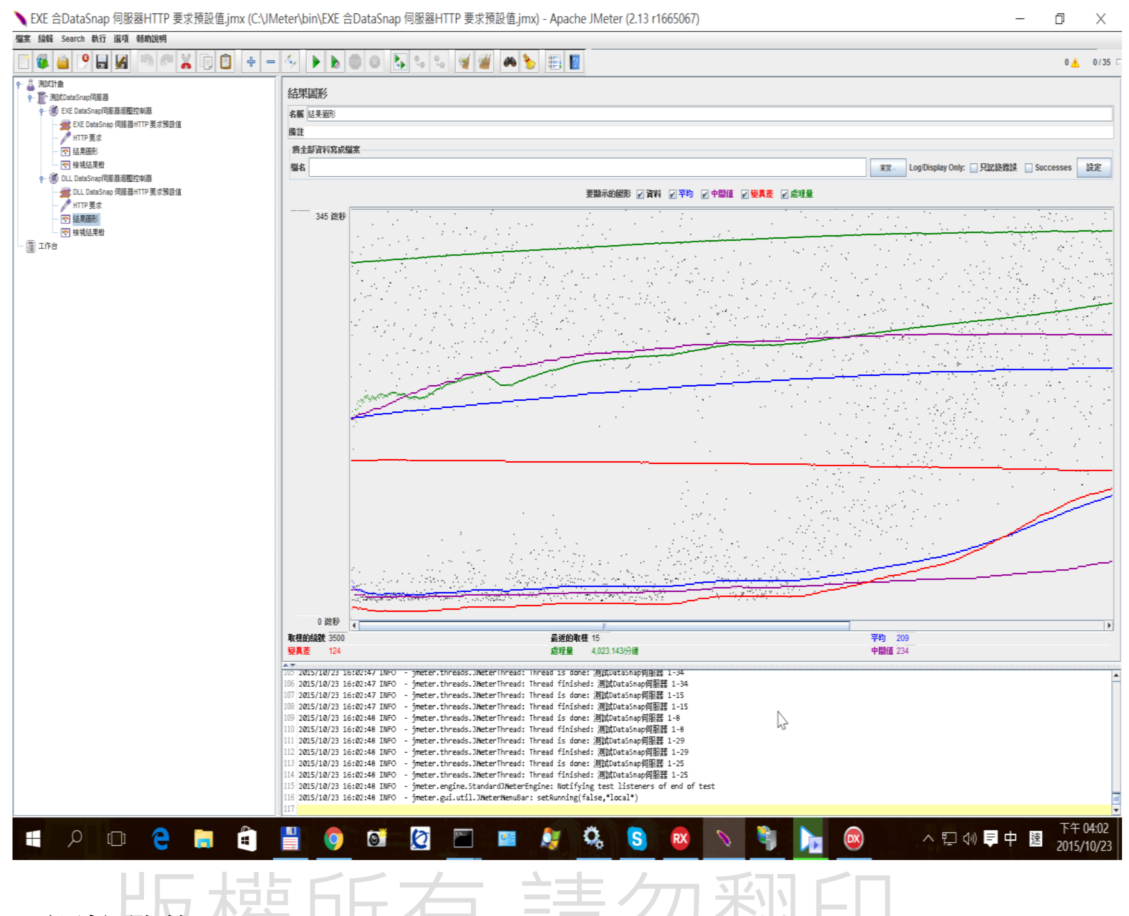

從 JMeter 出的報告可以明確看到 EXE 型態的 DataSnap 伺服器已經超過最大連結數目，因此無法再讓客戶端連結，自然也就無法提供客戶端服務了：



```
Thread Name: 測試 DataSnap 伺服器 1-13
Sample Start: 2015-11-02 10:55:46 TST
Load time: 17
Connect Time: 1
Latency: 17
Size in bytes: 233
Headers size in bytes: 159
Body size in bytes: 74
Sample Count: 1
Error Count: 1
Response code: 500
Response message: Internal Server Error
. . .
Maximum number of concurrent connections exceeded.  Please try again later
```

版權所有 請勿翻印

但使用 JMeter 測試 DLL 型態的 DataSnap 伺服器同樣可以看到完全沒有錯誤：




版權所有 請勿翻印

## 10-3 調校效能

在前面小節中使用壓力測試讓我們真實看到了 EXE 和 DLL 型態 DataSnap 伺服器的差異，EXE 型態的 DataSnap 伺服器在客戶端超過 32 個之後就會有錯誤發生，發生的錯誤種類是 DataSnap 伺服器已無法服務客戶端的請求，相反的 DLL 型態的 DataSnap 伺服器在卻可同時服務超過 500 個客戶端，為什麼這 2 種型態的 DataSnap 伺服器有如此大的差距？有沒有可能讓這 2 種型態的 DataSnap 伺服器有更好的執行效率呢？

這正是本小節要討論的重點。

### 調校 EXE 型態的 DataSnap 伺服器

調校 EXE 型態的 DataSnap 伺服器的第 1 步是解除在 30~40 個同步客戶端請求時發生” 最大連結數目”錯誤的狀況。由於 DataSnap 是使用 Indy 元件組做為核心通訊技術，因此這應該是 Indy 或是 DataSnap 框架的問題，因此我們可以在 IDE 中使用搜尋功能尋找 Indy 和 DataSnap 原始碼中有”MAXConnection”的字眼

很快的就可以在 TWebRequestHandler 類別中找到如下的程式碼，我們可以看到 TWebRequestHandler 設定最大的連結數目是 32，這也解釋了為什麼前面 EXE 型態的 DataSnap 伺服器在 30~40 個同步客戶端請求時便會發生錯誤而且顯示” 伺服器已經超過最大連結數目” 的錯誤：

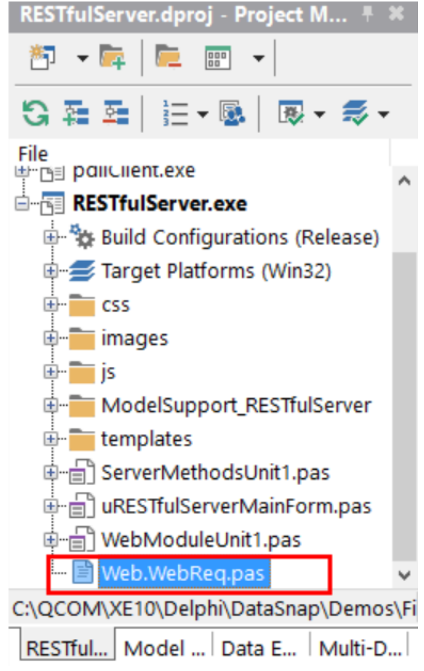
```
constructor TWebRequestHandler.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FCriticalSection := TCriticalSection.Create;
  FActiveWebModules := TList.Create;
  FInactiveWebModules := TList.Create;
  FMaxConnections := 32;
  FCacheConnections := True;
end;
```

因此讓我們先解除這個 DataSnap 框架限制，看看有什麼效果。

> 但使用同樣 DataSnap 框架原始碼的 DLL 型態的 DataSnap 伺服器不會有這種錯誤呢？讀者可以先想想嗎？

版權所有 請勿翻印

璋在請把 c:\Program Files (x86)\Embarcadero\Studio\17.0\source\data\dsnap\ 目錄下的 Web.WebReq.pas 拷貝到前面範例 RESTfulServer.dproj 專案的目錄中，再把 Web.WebReq.pas 加入到專案中：



開啟專案中的 Web.WebReq.pas 原始碼，修改 FMaxConnections 為 0，代表不限制客戶端的連結數目：

```
constructor TWebRequestHandler.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FCriticalSection := TCriticalSection.Create;
  FActiveWebModules := TList.Create;
  FInactiveWebModules := TList.Create;
//  FMaxConnections := 32;
  FMaxConnections := 0;
  FCacheConnections := True;
end;
```

再重新編譯此 EXE 型態的 DataSnap 伺服器並重新執行它，現在再讓我們使用壓力測試工具來測試這個修改過的 EXE 型態的 DataSnap 伺服器，看看它現在是否能處理超過 32 個同步客戶端的請求。

讓我們使用 40 個同步客戶端做為測試基準，測量它的反應時間做為隨後調整效率的起始值：

版權所有 請勿翻印

```
ab -n 1000 -c 40
http://localhost:8080/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
```

下面是 ApacheBench 的測試結果：

```
Server Port:            8080

Document Path:          /datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
Document Length:        16136 bytes

Concurrency Level:      40
Time taken for tests:   11.594 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      16343669 bytes
HTML transferred:       16136000 bytes
Requests per second:    86.25 [#/sec] (mean)
Time per request:       463.775 [ms] (mean)
Time per request:       11.594 [ms] (mean, across all concurrent requests)
Transfer rate:          1376.58 [Kbytes/sec] received
```

下面是 WeigHtpp 的測試結果：

```
Finished in 11 sec, 963 millisec and 999 microsec,
         83 req/s, 1334 kbyte/s.

Requests: 1000 total, 1000 started, 1000 done,
          1000 succeeded, 0 failed, 0 errored.

Status codes: 1000 2xx, 0 3xx, 0 4xx, 0 5xx.
```

我們可以看到修改了 FMaxConnections 之後 EXE 型態的 DataSnap 伺服器不再出現錯誤，最後讓我們使用 JMeter 加大同步客戶端到 500 個，看看它是否能像 DLL 型態的 DataSnap 伺服器一樣，完全可以應付客戶端的請求。

下圖是使用 JMeter 模擬 500 個同步客戶端，我們可以確定現在 EXE 型態的 DataSnap 伺服器也可以完全應付，不會發生錯誤了：

| Label | 取樣數 | 平均值 | 中間值 | 90% Line | 95% Line | 99% Line | 最小值 | 最大值 | 錯誤率 | 處理量 | 每秒仟位元組 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| HTTP 要求 | 500 | 3469 | 2860 | 7259 | 7693 | 7884 | 376 | 7987 | 0.00% | 59.7/sec | 952.8 |
| 總計 | 500 | 3469 | 2860 | 7259 | 7693 | 7884 | 376 | 7987 | 0.00% | 59.7/sec | 952.8 |

現在我們整理此基準 EXE 型態的 DataSnap 伺服器的執行特徵如下：

版權所有 請勿翻印

| 特性值 | 執行結果 |
|---|---|
| 同步客戶端 | 40 |
| 每次請求費時 | 11.594ms |
| 發生錯誤次數 | 0 |

## EXE 型態的 DataSnap 伺服器改良版 1

現在我們就可以開始調校 EXE 型態的 DataSnap 伺服器的執行效率了，第 1 步當然就是加快它接受客戶端請求的速度，這可以藉由增加它傾聽執行緒數目。這是因為在它啟動時它是建立了 TIdHTTPWebBrokerBridge 物件傾聽客戶端的請求：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FServer := TIdHTTPWebBrokerBridge.Create(Self);
  …
```

而 TIdHTTPWebBrokerBridge 是從 TIdCustomHTTPServer 類別繼承下來的，因此 TIdHTTPWebBrokerBridge 是一個 Indy 框架的類別。

```
  TIdHTTPWebBrokerBridge = class(TIdCustomHTTPServer)
```

TIdCustomHTTPServer 類別又是從 TIdCustomTCPServer 繼承下來的：

```
  TIdCustomHTTPServer = class(TIdCustomTCPServer)
```

而在 TIdCustomTCPServer 類別中有一個特性 ListenQueue：

```
    property ListenQueue: integer read FListenQueue write FListenQueue default
IdListenQueueDefault;
```

是 Indy 使用來傾聽客戶端請求的執行緒數目，它的內定值版設定成 15：

```
const
  IdListenQueueDefault = 15;
```

但這個值對現今的伺服器硬體來說實在太保守了，甚至對筆者的 VM 來說都太保守了，因此先讓我們加大這個設定值。

請開啟 EXE 型態的 DataSnap 伺服器的主表單，修改它的 OnCreate 事件處理函式如下：

版權所有 請勿翻印

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FServer := TIdHTTPWebBrokerBridge.Create(Self);
  FServer.ListenQueue := 150; //增加 Listener 執行緒數目
  SetupFDManager;
end;
```

增加 Indy 框架的傾聽執行緒數目可以加快伺服器回應的速度。

重新編譯此 EXE 型態的 DataSnap 伺服器並重新執行它，再使用上面相同的條件來測試此 EXE 型態的 DataSnap 伺服器，下面是 ApacheBench 的結果：

```
Concurrency Level:      40
Time taken for tests:   11.575 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      16343663 bytes
HTML transferred:       16136000 bytes
Requests per second:    86.39 [#/sec] (mean)
Time per request:       463.019 [ms] (mean)
Time per request:       11.575 [ms] (mean, across all concurrent requests)
Transfer rate:          1378.83 [Kbytes/sec] received
```

| 特性值 | 執行結果 |
| --- | --- |
| 同步客戶端 | 40 |
| 每次請求費時 | 11.575ms |
| 發生錯誤次數 | 0 |

ApacheBench 顯示 EXE 型態的 DataSnap 伺服器現在的確更快了一點，

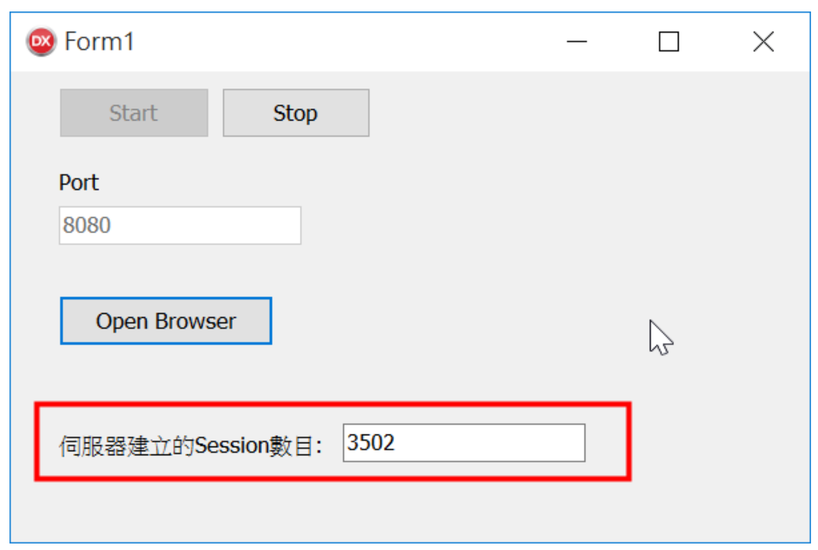
WeigHttp 也樣顯示現在的反應更快了：

```
Finished in 11 sec, 691 millisec and 999 microsec,
         85 req/s, 1365 kbyte/s.

Requests: 1000 total, 1000 started, 1000 done,
          1000 succeeded, 0 failed, 0 errored.
```

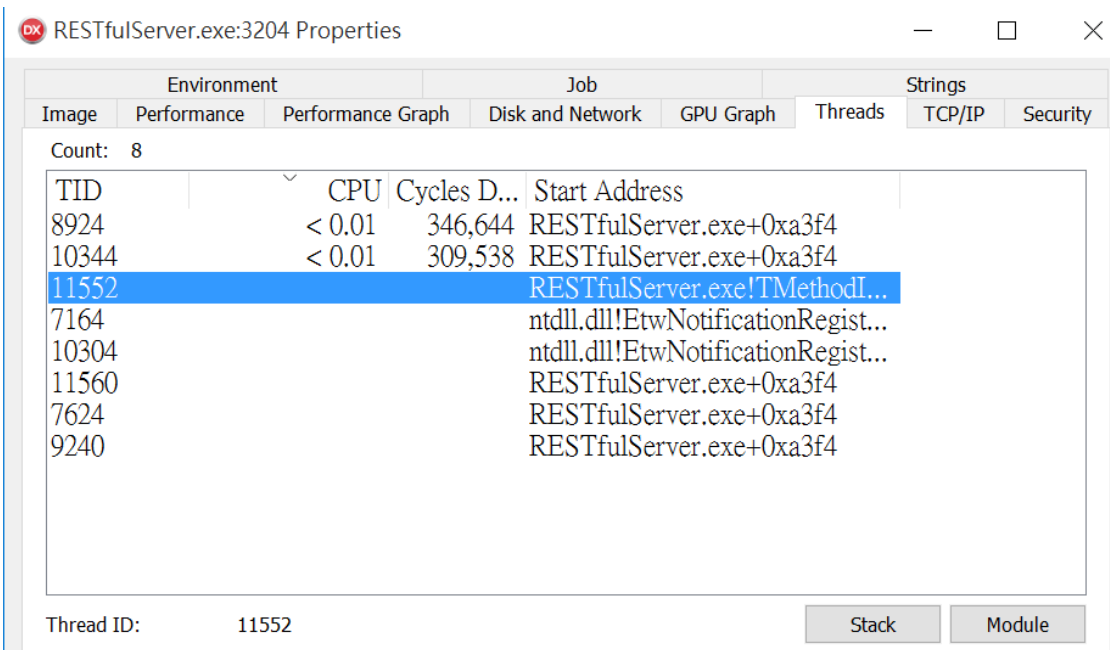
讀者也許覺得這只快了一點點，但別忘了在實際上線後當大量的客戶肯不斷的提出請求並且持續執行 24*7 的應用中，這將會有更明顯的差異。

## EXE 型態的 DataSnap 伺服器改良版 2

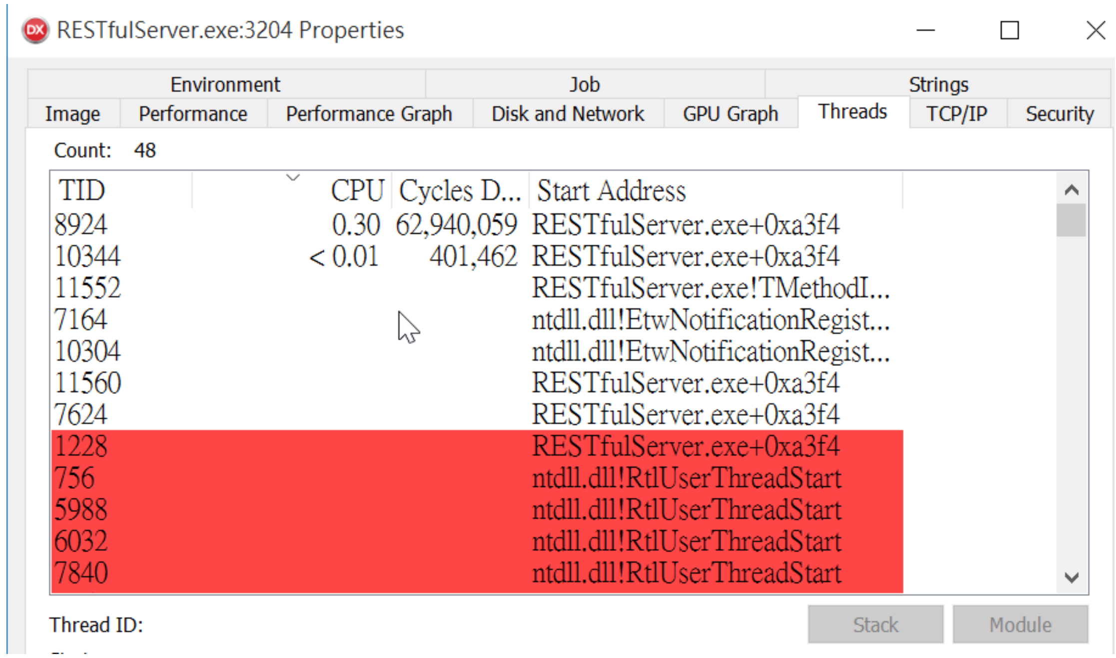
下一步改良版 DataSnap 伺服器的執行效率方法來自觀察當客戶端連結並提出服務請求時，從 EXE 型態的 DataSnap 伺服器的主表格可以看到每一個連結的客戶端都會在 DataSnap 伺服器建立一個 Session，如下所示：




如果我們啟動工作管理員進一步觀察 DataSnap 伺服器，那麼可以看到類似下面的初始狀態，DataSnap 伺服器在啟動後會建立數個執行緒同時執行:




但當客戶不斷增加並提出請求時在工作管理員中我們會看到 DataSnap 伺服器會不斷的建立新的執行緒服務客戶端並在服務完畢之後釋放執行緒，如此不斷的進行這種執行行為：




這個現象說明了 DataSnap 伺服器浪費了許多時間不斷的重覆建立和釋放執行緒，會造成這個現象的原因是 Indy 建立的執行緒池太小，因此當客戶端數目愈來愈多時就造成執行緒不夠使用，因此 DataSnap 伺服器需要不斷的建立和釋放執行緒。因此要進一步改善 DataSnap 伺服器執行效率，我們只需要啟動 Indy 的執行緒池功能並增加執行緒池的大小即可改善 DataSnap 伺服器不斷的重覆建立和釋放執行緒的負荷。

版權所有 請勿翻印

Indy 框架的執行緒池功能是實作在 TIdSchedulerOfThreadPool 類別中:

```
TIdSchedulerOfThreadPool = class(TIdSchedulerOfThread)
```

而 TIdSchedulerOfThreadPool 是定義在 IdSchedulerOfThreadPool 程式單元中，因此我們需要在 DataSnap 伺服器主表單中加入使用這個程式單元：

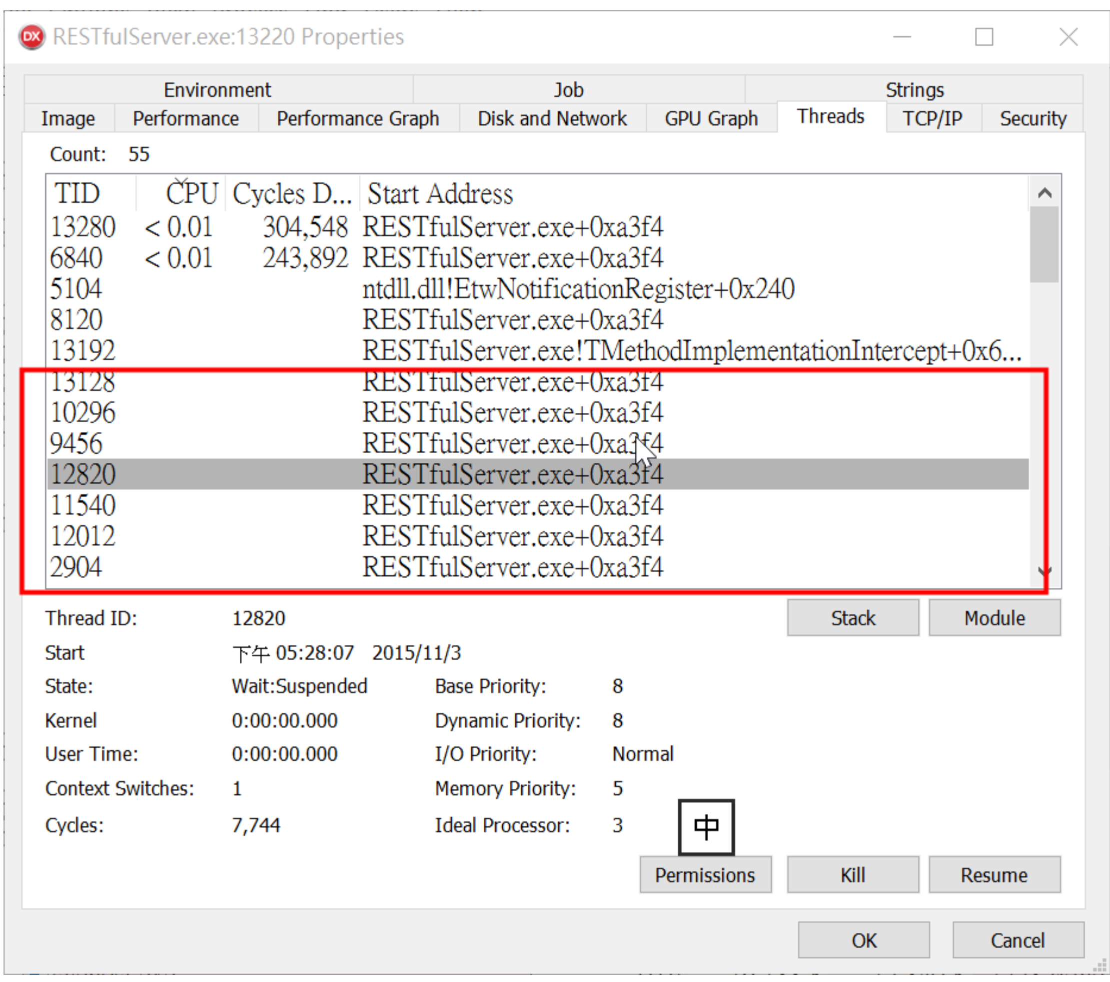
```
uses IdSchedulerOfThreadPool
```

並在主表單的 OnCreate 事件中建立 TIdSchedulerOfThreadPool 物件，設定它的 PoolSize 特性值，PoolSize 特性即控制了執行緒池的大小，例如筆者在下面的程式碼中設定執行緒池的大小為 50，當然如果讀者用的機器硬體配備很好，那麼可以再增加到 100 左右，執行緒池大小設定需要根據實際的環境來決定。對於筆者的VM來說 50 已經很多了。

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Scheduler: TIdSchedulerOfThreadPool;
```

```
begin
  FServer := TIdHTTPWebBrokerBridge.Create(Self);
  FServer.ListenQueue := 150; //增加 Listener 執行緒數目

  Scheduler := TIdSchedulerOfThreadPool.Create(FServer); //增加執行緒池
  Scheduler.PoolSize := 50; //建執行緒池中預先建立的執行緒
  FServer.Scheduler := Scheduler;
  SetupFDManager;
end;
```

重新編譯並執行此範例 DataSnap 伺服器，使用 ApacheBench 來測試，並使用工作管理員觀察範例 DataSnap 伺服器，從下面的畫面可以看到範例 DataSnap 伺服器啟動後就會建立 PoolSize 特性設定的執行緒數目，而且當客戶端不斷增加執行數目並提出請求時，範例 DataSnap 伺服器幾乎不會再不斷的建立和釋放執行緒，因此增加了 DataSnap 伺服器的執行效率：



下面是 ApacheBench 在筆者ＶＭ中加壓測試的結果，在客戶端增加到 1000 個時，服務每一個客戶端只需要 12.529 毫秒；

```
Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
Document Length:        16136 bytes

Concurrency Level:      1000
Time taken for tests:   12.529 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      16343684 bytes
HTML transferred:       16136000 bytes
Requests per second:    79.82 [#/sec] (mean)
Time per request:       12528.563 [ms] (mean)
Time per request:       12.529 [ms] (mean, across all concurrent requests)
Transfer rate:          1273.94 [Kbytes/sec] received
```

JMeter 加壓測試則顯示在筆者的 VM 中每一秒在 90%的請求都不超過 5435 毫秒：

| Label | 取樣數 | 平均值 | 中間值 | 90% Line | 95% Line | 99% Line | 最小值 | 最大值 | 錯誤率 | 處理量 | 每秒仟位元組 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| HTTP 要求 | 500 | 3313 | 3293 | 5435 | 5531 | 5758 | 113 | 5947 | 0.00% | 51.8/sec | 826.7 |
| 總計 | 500 | 3313 | 3293 | 5435 | 5531 | 5758 | 113 | 5947 | 0.00% | 51.8/sec | 826.7 |

範例 DataSnap 伺服器的確是愈來愈快而且可服務的客戶端已經從 32 個到現在超過 1000 個都沒問題了。

但，還能更快，服務更多的客戶端嗎？

## EXE 型態的 DataSnap 伺服器改良版 3

要再進一步調校範例 DataSnap 伺服器的執行效率之前我們需要再次想想基本的問題，那就是要使用 DataSnap 伺服器的原因為何？當然就程式碼開發而言使用 DataSnap 伺服器有其好處，但在這裡讓我們專注在 DataSnap 伺服器的應用原因。

應用 DataSnap 伺服器可能有：

1. 想服務大量客戶端

2. 想連結移動客戶端和其他智慧型設備

其實上面的 2 個原因互有牽連,因為就是要”結移動客戶端和其他智慧型設備”因此會有比以前的應用有更多的客戶端。

而連結更多的客戶端會對 DataSnap 伺服器造成沈重的負荷是因為如果每一個客戶端都需要對 DataSnap 伺服器使用過多的資源，那麼 DataSnap 伺服器當然就只能服務一定數量的客戶端。

因此在使用 DataSnap 伺服器的多層應用中我們應該盡可能的減少每一個客戶端在 DataSnap 伺服器中使用/佔用的資源：

1. 使用的資源是指服務程式碼要精簡快速

2. 佔用的資源是指使用資源之後立刻歸還，不要無義意的佔據

例如前面第 2 步的調校步驟,服務客戶端的執行緒使用完畢之後立刻歸還，重覆使用。另外一個例子就是在以前的 C/S 架構中客戶端對於資料庫的連結是一直佔據的，在 DataSnap 多層架構中應該是在客戶端取得需要的資料之後立刻歸還資料庫連結，重覆使用。

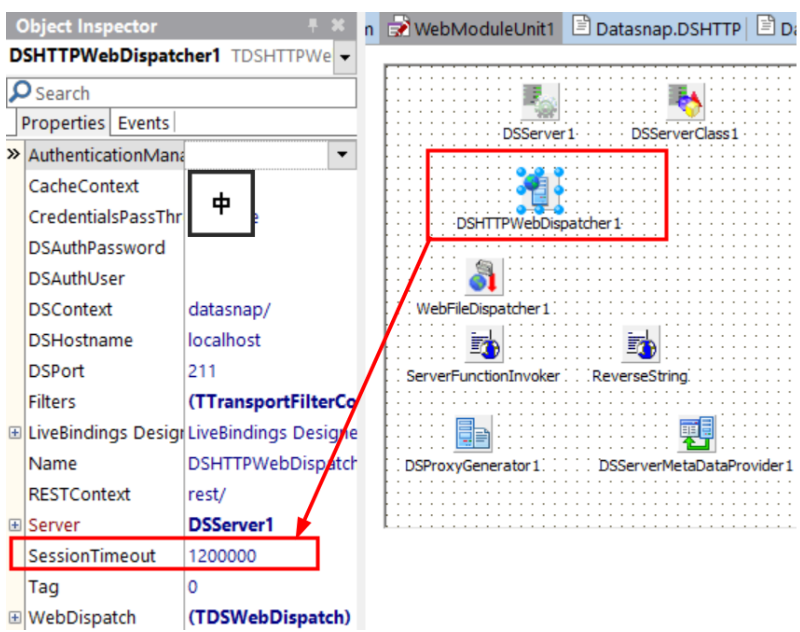
版權所有 請勿翻印

因此本步驟的調校就是再進一步主動釋放客戶端使用的資源，一旦 DataSnap 伺服器服務完畢客戶端的請求後就回收客戶端使用的資源，重覆使用。

現在回到範例 DataSnap 伺服器，由於它本身就是一個 RESTful 伺服器，因此客戶端在提出請求並取得結果之後本就應該離開並歸還資源,但請讀者開啟範例專案中的 WebModuleUnit1 程式單元，在其中可以看到 DataSnap 處理 HTTP 命令的元件 TDSHTTPWebDispatcher，它的 SessionTimeout 特性值是設定為 1200000 毫秒，這代表在範例 DataSnap 伺服器處理了客戶端的請求後仍然會在 DataSnap 伺服器中保留這個客戶端的 Session 物件最少 20 分鐘之後才會釋放，因此如果有大量的客戶端時這就對 DataSnap 伺服器造成了沈重的負荷。

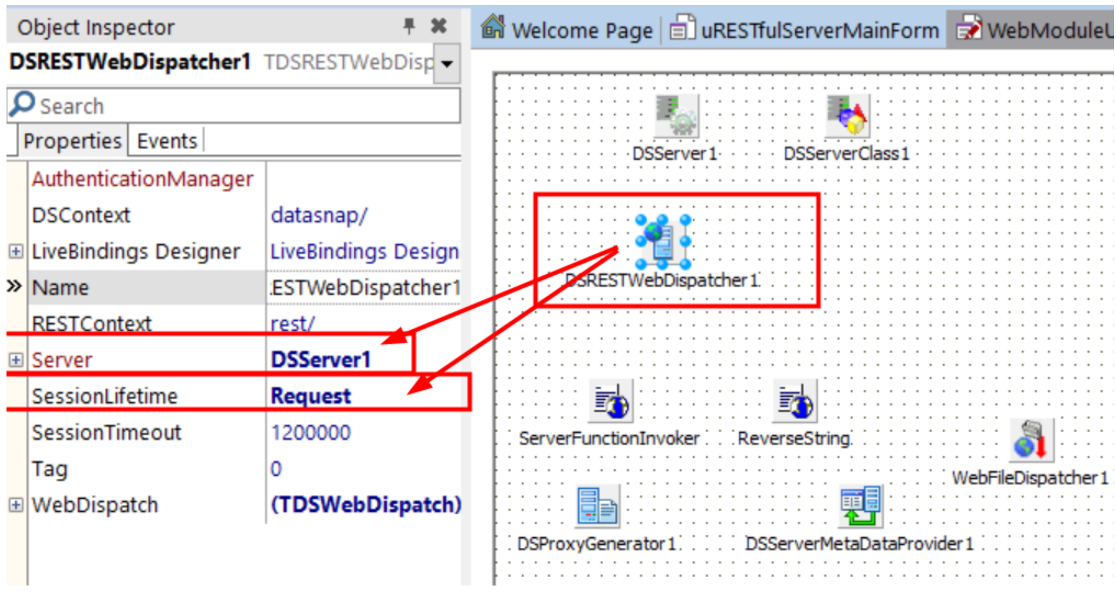
既然我們使用了 RESTful 的架構，因此就應該盡量在服務完客戶端請求後就釋放為客戶端建立的 Session 物件。也許讀者會立刻反應把 SessionTimeout 特性值設定為 0 不就好了? 噢這樣做有反效果，因為設定 SessionTimeout 特性值為 0 代表永不釋放 Session 物件。

因此我們需要的元件是服務完客戶端請求後能就自動釋放 Session 物件，在 DataSnap Server 元件組中正好有這麼一個元件：TDSRESTWebDispatcher。




我們可以使用 TDSRESTWebDispatcher 元件來取代原本的 TDSHTTPWebDispatcher 元件，因請在上圖的 WebModuleUnit1 程式單元中加入一個 TDSRESTWebDispatcher 元件，再刪除原來的 TDSHTTPWebDispatcher 元件，如下所示：

版權所有 請勿翻印



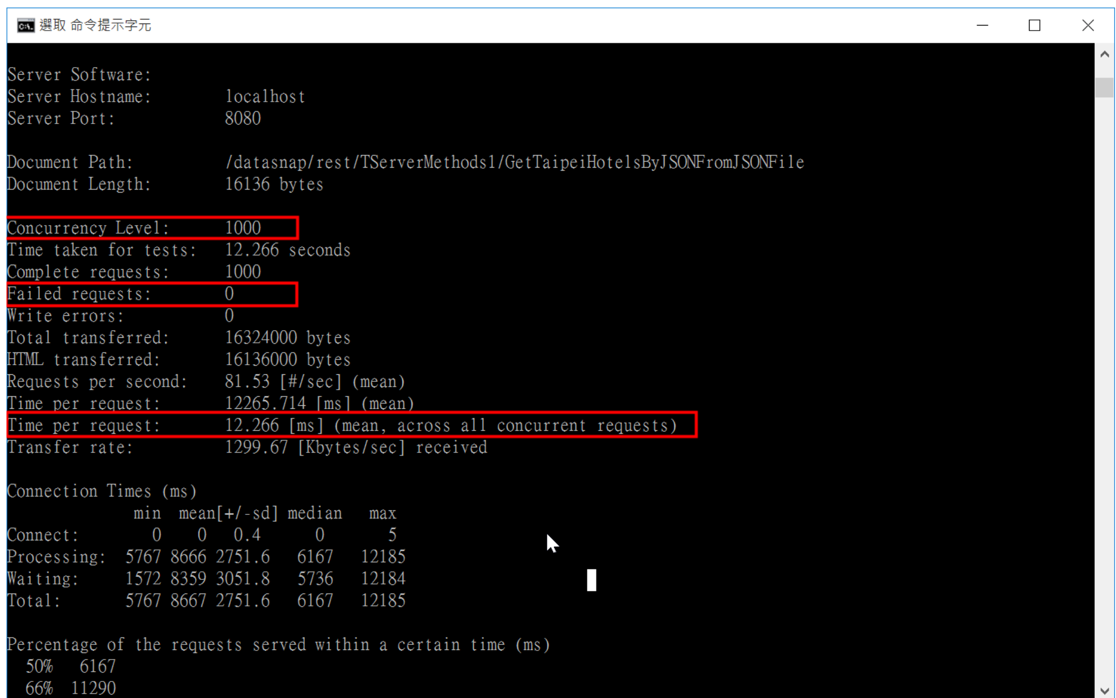

再於物件檢視器中設定 TDSRESTWebDispatcher 元件的特性如下：

| 特性 | 特性值 |
|---|---|
| Server | DSServer1 |
| SessionLifetime | Request |

重新編譯並執行此範例 DataSnap 伺服器，使用 ApacheBench 來測試：

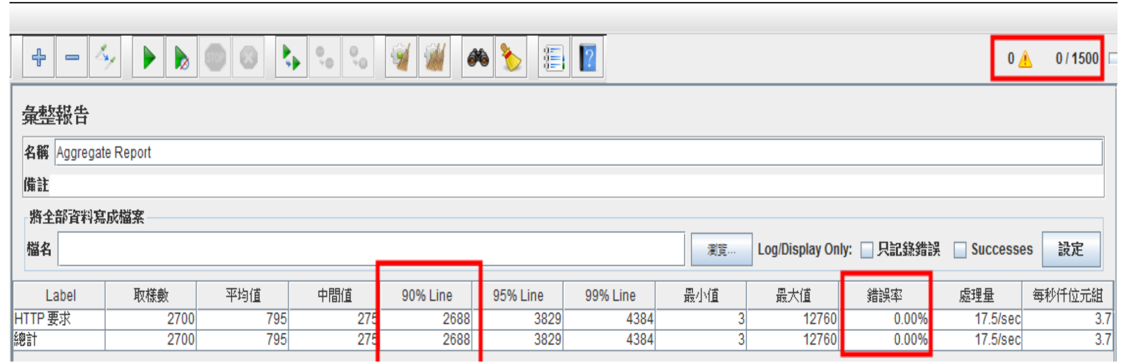
```
ab -n 1000 -c 1000
http://localhost:8080/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSO
NFile
```

從下面 ApacheBench 測試的結果來看使用 TDSRESTWebDispatcher 元件的範例 DataSnap 伺服器可以更快的速度服務更多的客戶端：




| 特性值 | 執行結果 |
|---|---|
| 同步客戶端 | 1000 |
| 每次請求費時 | 12.266ms |
| 發生錯誤次數 | 0 |

Jmeter 也顯示這個調校的範例 DataSnap 伺服器在 1000 個同步客戶端不斷請求服務時有 90%的客戶端可以在 2688 毫秒服務完畢並且沒有發生任何錯誤：




EXE 型態的 DataSnap 伺服器到這裡要再進一步的增加執行效率就需要討論到設計架構和程式師撰寫的程式碼品質了，在稍後我們會討論一些其他的技巧再次增進 EXE 型態 DataSnap 伺服器的執行效率。

還是不要忘記，在實際執行的應用環境中筆者還是建議使用 DLL 型態的 DataSnap 伺服器。

## 調校 DLL 型態的 DataSnap 伺服器

現在我們可以開始討論如何增加 DLL 型態的 DataSnap 伺服器的執行效率，上面對於增加 EXE 型態 DataSnap 伺服器的技巧都可以使用在 DLL 型態的 DataSnap 伺服器中，在下面的小節中討論的技巧是只針對 DLL 型態的 DataSnap 伺服器。

版權所有 請勿翻印

### DLL 型態的 DataSnap 伺服器改良版 4

對於 IIS 的 DLL 型態的 DataSnap 伺服器而言，在進行了和上面類似的調整步驟之後，第 1 個額外的調校就是設定 IIS 對於客戶端連結的數目限制，這可以在 IIS 主程式的地方設定它的 MaxConnections 特性值為 0，如下所示：

```
begin
  CoInitFlags := COINIT_MULTITHREADED;
  Application.Initialize;
  Application.WebModuleClass := WebModuleClass;
  SetupFDManager;
  Application.MaxConnections := 0;
  TISAPIApplication(Application).OnTerminate := TerminateThreads;
  Application.Run;
end.
```

進行了這個調整之後再使用 AppBench 對調整前和調整後比較我們可以看到調整後的執行速度再次有了明顯的增加，

調整前：

```
Server Software:        Microsoft-IIS/10.0
Server Hostname:        localhost
Server Port:            81

Document Path:          /dsapps/ISAPIDSDemo.dll/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
Document Length:        16136 bytes

Concurrency Level:      20
Time taken for tests:   14.124 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      16374682 bytes
HTML transferred:       16136000 bytes
Requests per second:    70.80 [#/sec] (mean)
Time per request:       282.472 [ms] (mean)
Time per request:       14.124 [ms] (mean, across all concurrent requests)
Transfer rate:          1132.21 [Kbytes/sec] received
```

調整後：

```
Server Software:        Microsoft-IIS/10.0
Server Hostname:        localhost
Server Port:            81

Document Path:          /dsapps/ISAPIDSDemoStep4.dll/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
Document Length:        16136 bytes

Concurrency Level:      20
Time taken for tests:   11.267 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      16355000 bytes
HTML transferred:       16136000 bytes
Requests per second:    88.75 [#/sec] (mean)
Time per request:       225.350 [ms] (mean)
Time per request:       11.267 [ms] (mean, across all concurrent requests)
Transfer rate:          1417.50 [Kbytes/sec] received
```

就一個簡單的設定就伯有這麼明顯的差別。

## DLL 型態的 DataSnap 伺服器改良版 5

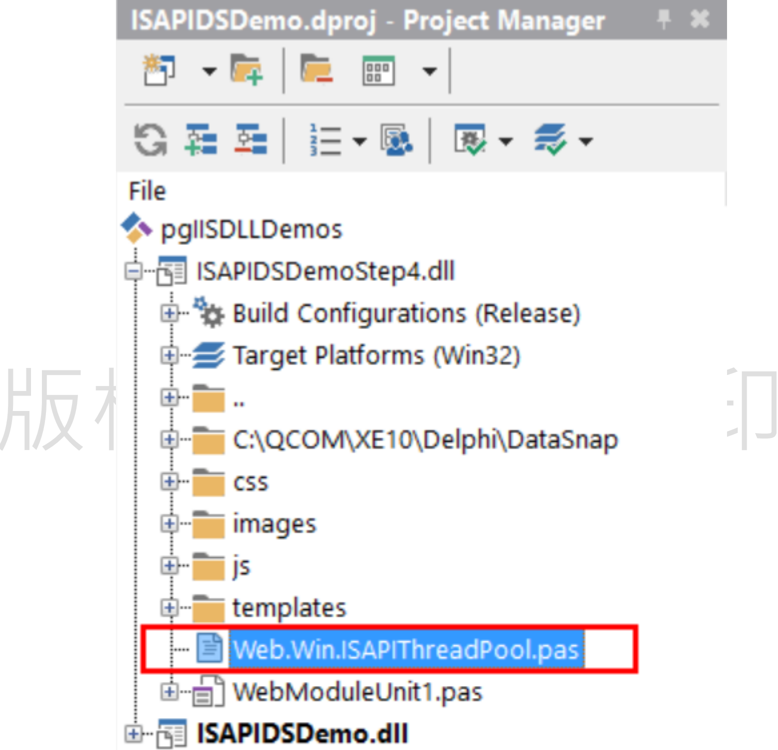
對於第 2 個 IIS DLL 型態的 DataSnap 伺服器調整步驟就是設定它的執行緒連結池數目的設定。

在 Delphi 的 Web.Win.ISAPIThreadPool 程式單元中定義了 Delphi IIS 型態的應用程式使用的執行緒連結池數目，在內定上 Delphi 是設定使用 25 個執行緒：

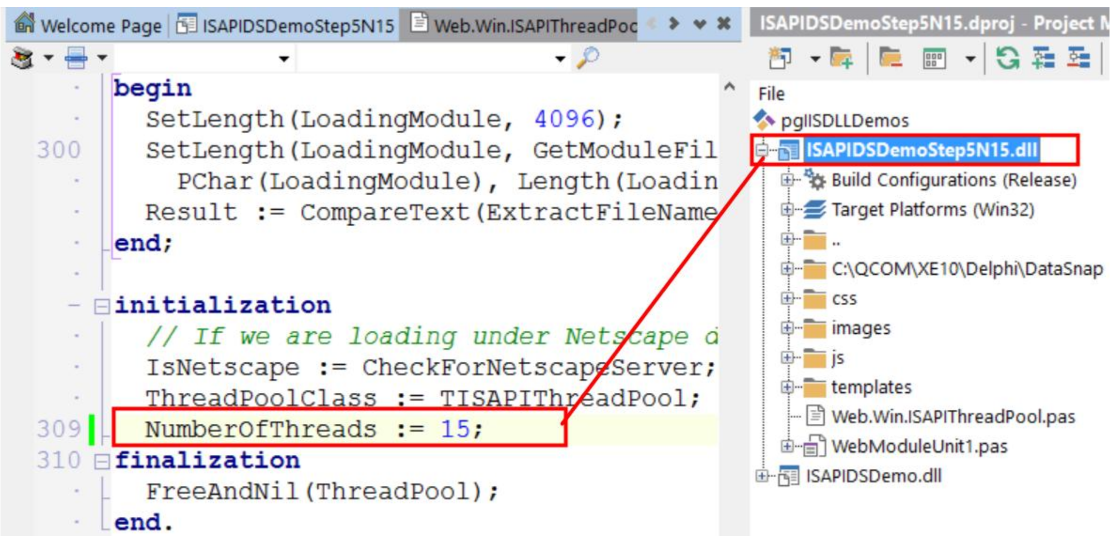
```
initialization
  // If we are loading under Netscape do not use the thread pool
```

```
IsNetscape := CheckForNetscapeServer;
ThreadPoolClass := TISAPIThreadPool;
NumberOfThreads := 25;
```

但這個內定值不一定是最好的，開發人員同樣應該根據實際的環境來設定執行緒連結池的大小，因為這會影響執行效率。為了接下來比較不同 NumberOfThreads 設定值對於執行效率的影響，下面的圖形顯示到目前為止 2 個 DLL DataSnap 伺服器專案，而 ISAPIDSDemoStep4 專案就是剛才 MaxConnections 特性值為的專案，現在讓我們在此專案中加入 Web.Win.ISAPIThreadPool 程式單元，準備設定 NumberOfThreads 數值：




然後我們可以在 IDE 中設定不同的 NumberOfThreads 數值並產生不同的輸出檔：

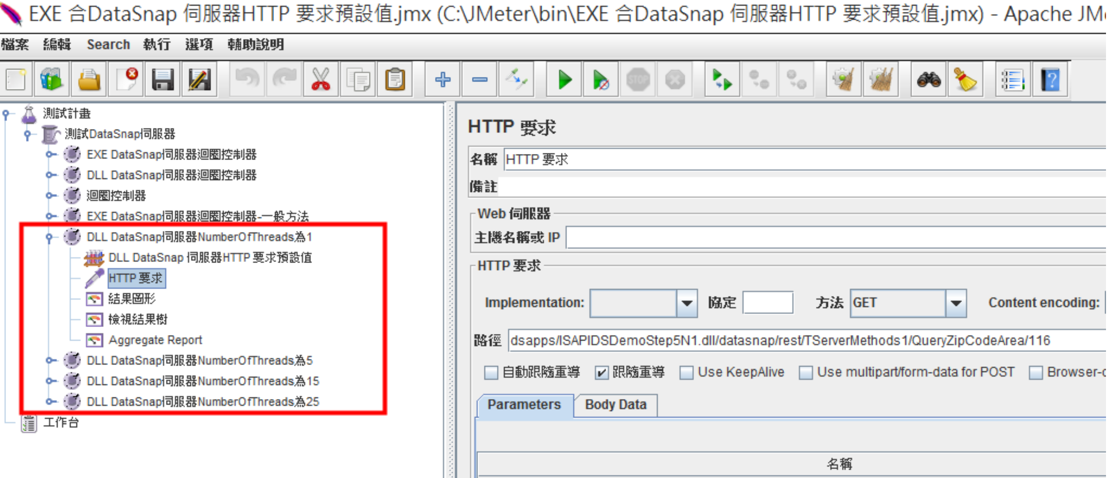



例如下面是分別設定 NumberOfThreads 數值為 1，5，15，25 編譯出的如輸出檔：

| | | | | |
|---|---|---|---|---|
| ISAPIDSDemo | dll | 7,887,360 | 2015/11/02 10:48 | -a-- |
| ISAPIDSDemoStep4 | dll | 7,885,824 | 2015/12/03 14:23 | -a-- |
| ISAPIDSDemoStep5N1 | dll | 7,885,824 | 2015/12/04 18:02 | -a-- |
| ISAPIDSDemoStep5N15 | dll | 7,885,824 | 2015/12/04 17:59 | -a-- |
| ISAPIDSDemoStep5N25 | dll | 7,885,824 | 2015/12/04 17:58 | -a-- |
| ISAPIDSDemoStep5N5 | dll | 7,885,824 | 2015/12/04 18:02 | -a-- |

使用 ApacheBench 和 JMeter 測試上面不同 NumberOfThreads 數值設定的結果顯示同時在 1000 個客戶端存取服務時顯示似乎 NumberOfThreads 設定在 10~25 之間的數值是擁有比較好的效率，不過當客戶端超過 1000 個時，增加 NumberOfThreads 數值設定則似乎有比較少的無法服務錯誤發生。

```
ab -n 1000 -c 1000
http://localhost:81/dsapps/ISAPIDSDemoStep5N100.dll/datasnap/rest/TServerMethods
1/QueryZipCodeArea/116
```




由於筆者個人使用的 VM 是 Windows 7 專業版，如果把調校到這裡的範例 DLL 型態的 DataSnap 伺服器部署到 Windows Server 2008 上執行的話應該可以應付大於 1500 個同步客戶的請求了。

## 10-4 改善程式碼調校效能

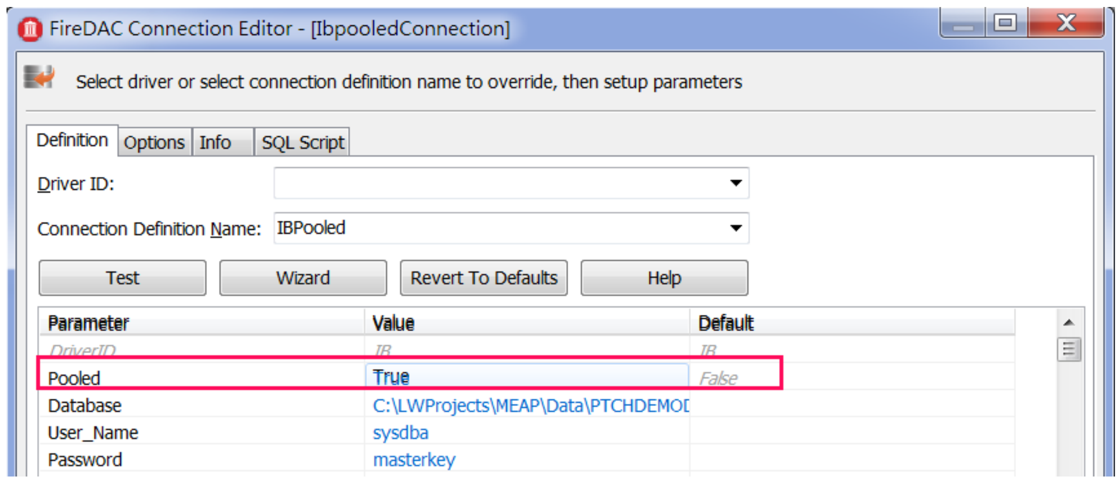
完成了上述的調校步驟之後我們仍然有方法可以簡單的再次增加 DataSnap 伺服器的執行效率。在前面的調校步驟中我們增加了執行緒池的功能，降低了 Session 的負荷，使用 RESTful Broker 增加執行效率，接下來我們可以再開啟資料庫的連接池功能讓 DataSnap 伺服器更有效率的存取資料庫。

版權所有 請勿翻印

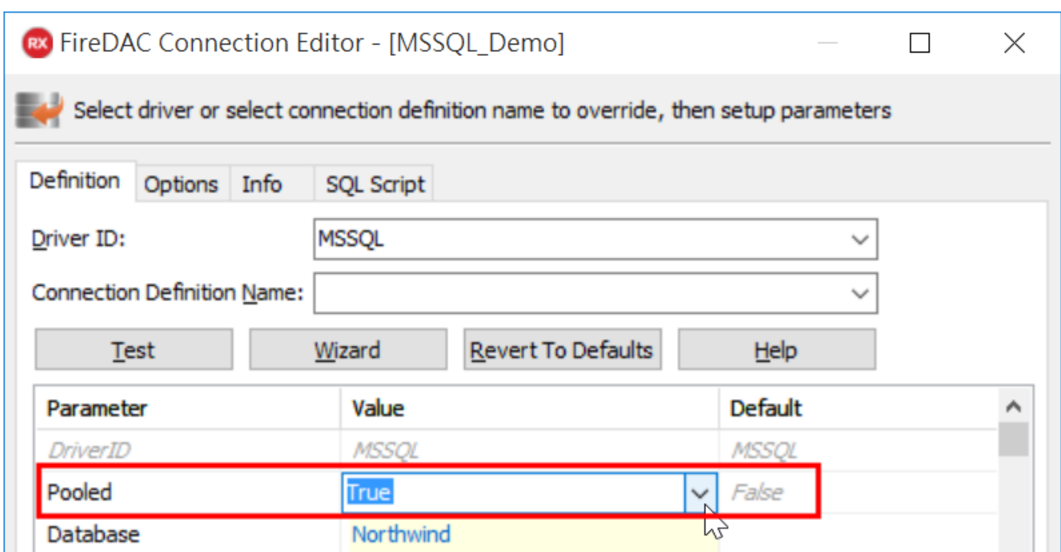
### 10-4-1 開啟資料庫連結池

在 FireDAC 中要開啟資料庫連結池功能非常的簡單，只要啟動 TFDConnection 的元件編譯器就可以如下圖看其中包含了一個”Pooled”選項，在內定上 FireDAC 會關閉資料庫連結池功能，因此下圖中的”Pooled”選項內定值是 False，但讀者可以把它設定為 True，以開啟資料庫連結池功能。

例如下圖是筆者使用的 InterBase XE7 版在 FireDAC 中開啟資料庫連結池功能：




同樣對於 MS SQL Serevr 2012 也可以在 FireDAC 中開啟資料庫連結池功能：




一旦設定好了開啟資料庫連結池功能之後，我們就可以在 FireDAC 的 FDConnectionDefs.ini 組態檔中看到 Pooled=True 設定：

```
[MSSQL_Demo]
Server=DESKTOP-XXX\SQLEXPRESS
Database=Northwind
MetaDefSchema=dbo
MetaDefCatalog=Northwind
ExtendedMetadata=True
Pooled=True
```

```
DriverID=MSSQL
```

讀者在部署 DataSnap 伺服器時一定也要部署此開啟資料庫連結池功能的組態檔，否則在 DataSnap 伺服器實際執行時仍然不會使用資料庫連結池功能。

一旦使用料庫連結池功能之後從下面的測試程式結果可知開啟料庫連結池功能比不開啟料庫連結池功能快了 8 倍的執行速度：

一個簡單的設定就可以增加支援的客戶端數目又可大幅增加執行效率，何樂而不為呢？

## 10-4-2 結合 ArrayDML

版權所有 請勿翻印

到了現在我們又加入了資料庫連結池功能，那麼還可以讓 DataSnap 伺服器更快嗎？當然，還有架構設計和程式碼風格，但那些不在本書討論內容之後，不過在結束本章內容之前讓筆者再分享一個小秘訣，那就是結合 FireDAC 超快速的 ArrayDML 功能。

FireDAC 的 ArrayDML 功能可以數倍的速度進行大量資料的新增資料的工作，而且由於 ArrayDML 的原理是在客戶端維持一個記憶體新增表格並一次把所有資料利用後端資料庫 Bulk Insert 的功能一次把所有的資料寫入資料庫中，因此和 DataSnap 的多層架構非常的契合。當結合這 2 者的功能時效果異常的良好快速。

讓我們使用一個簡單的範例來說明。

## DLL 型態的 DataSnap 伺服器改良版 5-結合 ArrayDML

首先我們可以在 DataSnap 伺服器中定義 2 個方法，一個是 GetData()取得資料，一個是 BulkInsert()方法可以藉由資料庫的 Bulk Insert 功能一次把所有資料寫入資料庫：

```
function GetData : TStream;
function BulkInsert(dataStream : TStream) : Boolean;
```

GetData()方法把後端資料以 TStream 格式送到客戶端顯示：

```
function TServerMethods1.GetData: TStream;
begin
  Result := TMemoryStream.Create;
  try
    TbltestdataTable.Close;
    TbltestdataTable.Open;
    TbltestdataTable.SaveToStream(Result, TFDStorageFormat.sfBinary);
    Result.Position := 0;
  except
    raise;
  end;
end;
```

BulkInsert()方法使用了數個巧妙的技巧，首先 007~008 行巧妙的把 FireDAC 的 Stream 格式回轉到 TMemoryStream 物件中，接著 010 行就能把資料還完成 FireDAC 的 DataSet，之後的程式碼就可以使用一般的 ArrayDML 功能把資料寫入資料庫中(請參考本系列"FireDAC 資料庫程式設計"一書)：

版權所有 請勿翻印

```
001   function TServerMethods1.BulkInsert(dataStream: TStream): Boolean;
002   var
003     iIndex: Integer;
004     LMemStream : TMemoryStream;
005   begin
006     Result := True;
007     LMemStream := CopyStream(dataStream);
008     LMemStream.Position := 0;
009     try
010       fdqBulkInsert.LoadFromStream(LMemStream, TFDStorageFormat.sfBinary);
011
012       fdqryInsert.Params.ArraySize := fdqBulkInsert.RecordCount;
013       for iIndex := 0 to fdqBulkInsert.RecordCount - 1 do
014       begin
015         fdqryInsert.Params.ParamByName('NEW_ADDDT').AsDates[iIndex] :=
fdqBulkInsert.FieldByName('ADDDT').Value;
016
fdqryInsert.Params.ParamByName('NEW_ADDDATETIME').AsStrings[iIndex] :=
```

```
fdqBulkInsert.FieldByName('ADDDATETIME').Value;
017         fdqryInsert.Params.ParamByName('NEW_RNAME').AsStrings[iIndex] :=
fdqBulkInsert.FieldByName('RNAME').Value;
018         fdqryInsert.Params.ParamByName('NEW_FAKEPHONE').AsStrings[iIndex] :=
fdqBulkInsert.FieldByName('FAKEPHONE').Value;
019         fdqBulkInsert.Next;
020       end;
021       fdqryInsert.Execute(fdqryInsert.Params.ArraySize);
022     except
023       Result := False;
024       LMemStream.Free;
025     end;
026   end;
```

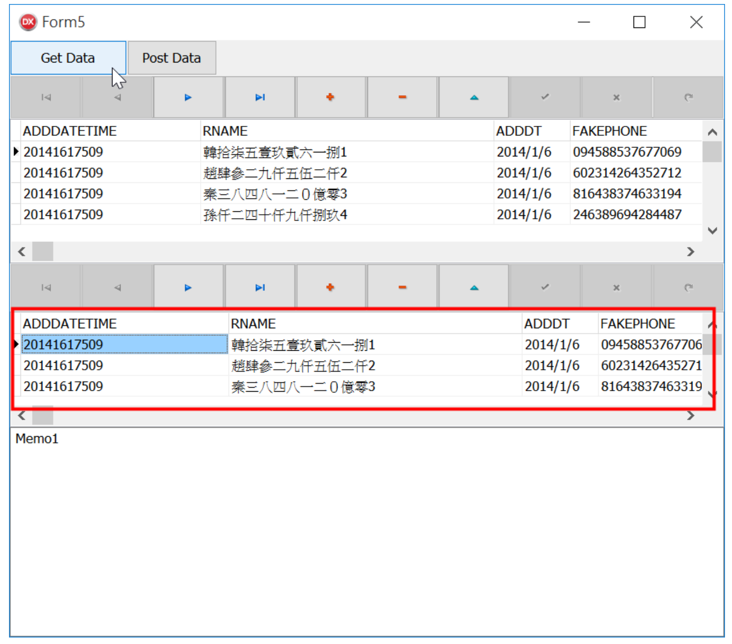
之後客戶端就可以使用如下的程式碼一之把大量的資料新增到資料庫中了：

```
001   procedure TForm5.PostData;
002   var
003     ms : TMemoryStream;
004     dss : TStringStream;
005   begin
006     ms := TMemoryStream.Create;
007     dss := TStringStream.Create;
008     try
009       fdmtDelta.ResourceOptions.StoreItems := [siData,  siDelta, siMeta];
010       fdmtDelta.SaveToStream(ms, TFDStorageFormat.sfBinary);
011       fdmtDelta.SaveToStream(dss, TFDStorageFormat.sfJSON);
012       Memo1.Lines.Text := dss.DataString;
013       ms.Position := 0;
014       fdspBulkInsert.Params.ParamByName('dataStream').AsStream := ms;
015       fdspBulkInsert.ExecProc;
016
017       fdmtDelta.Close;
018       dss.Position := 0;
019       fdmtDelta.LoadFromStream(dss, TFDStorageFormat.sfJSON);
020     finally
021       ms.Free;
022       dss.Free;
023     end;
```

版權所有 請勿翻印

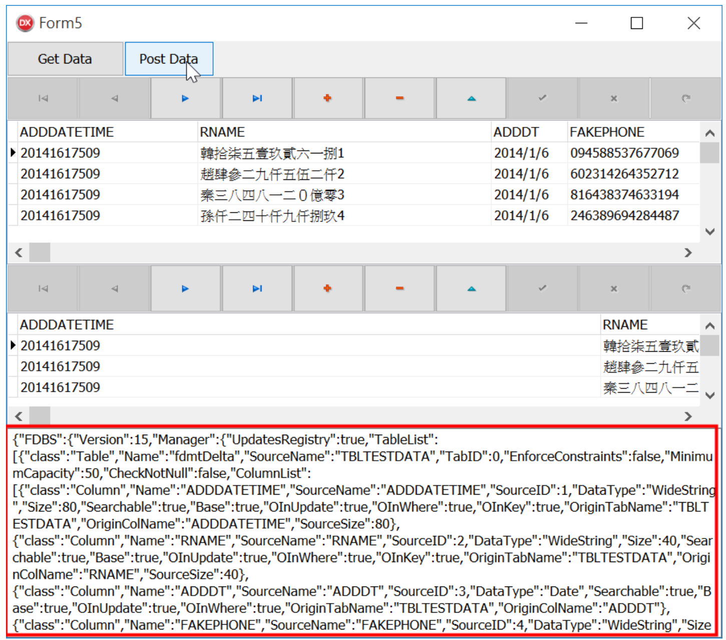
```
024    end;
```

下圖是客戶端呼叫 GetData()方法從 DataSnap 伺服器取得資料:




版權所有 請勿翻印

下圖是在客戶端新增大量資料之後呼叫 DataSnap 伺服器的 BulkInsert()方法一次把資料更新回資料庫，速度超快：



結合上面的調校技術和 ArrayDML 功能之後，在要處理大量新增資料的 DataSnap 應用中執行速度又提昇了 3~5 倍的速度，真是令人印象深刻。

## 10-5 結論

本章討論了各種技術讓讀者能夠學習開發並調校 DataSnap 伺服器的執行效率，是非常實用的內容。DataSnap 伺服器在採用這些調校技術後的確可以使用在實際的應用中處理大量，複雜的應用。如果讀者再搭配 DataSnap 的過濾器，安全功能和 HTTPS 等功能，那麼仍果就一定是一個”安全，高效率的 DataSnap 應用系統” 了。

Have Fun!

版權所有 請勿翻印

embarcadero
授權代理
捷康科技股份有限公司
電話: 02-23650238
傳真: 02-23650196
信箱： sales@qcomgroup.com.tw
http://embarcadero.qcomgroup.com.tw
版 權 所 有 · 請 勿 翻 印