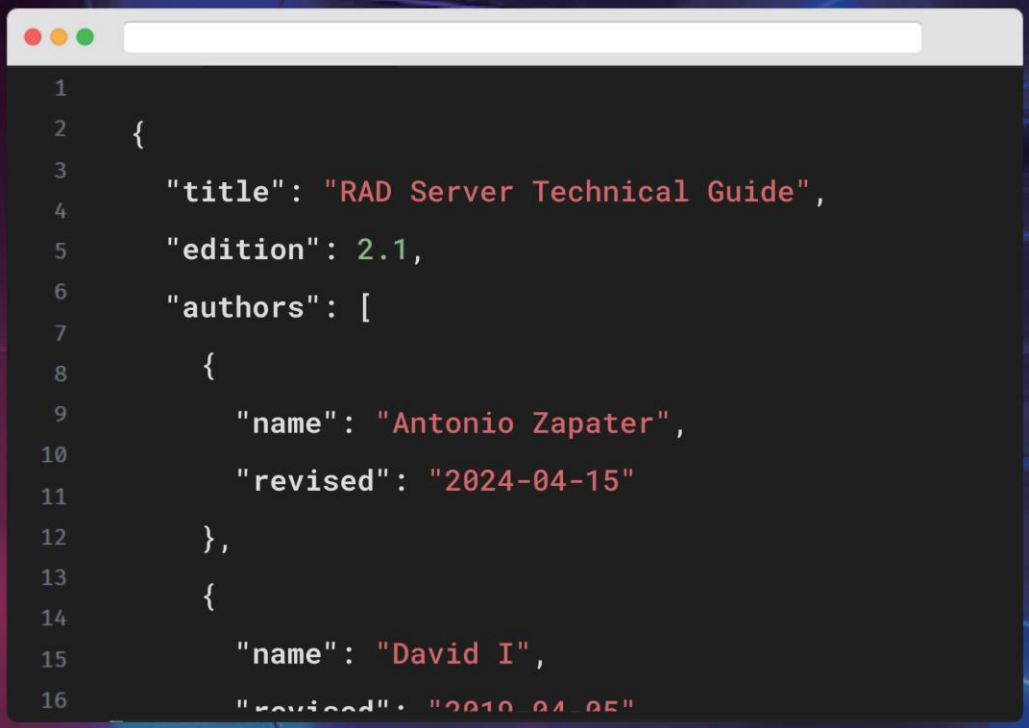
# RAD Server

# 技術手册



```
{
  "title": "RAD Server Technical Guide",
  "edition": 2.1,
  "authors": [
    {
      "name": "Antonio Zapater",
      "revised": "2024-04-15"
    },
    {
      "name": "David I",
      "revised": "2019-04-05"
```

embarcadero®

# 前言

RESTful 架構是現代 API 優先應用程式設計背後的關鍵驅動力. 本書重點介紹 RAD Studio (Delphi/C++Builder) 中所包含的 RAD 伺服器框架,以便用於開發此類平台.

RAD Server 是一個完整的後端 MEAP（行動企業應用程式平台）,支援以任何程式語言進行桌面,行動和 Web 前端開發,本書旨在作為開發人員的權威指南.

MEAP 的好處是,您擁有一個預先建置的雲端或本機伺服器,它具有許多核心功能(例如推播通知,用戶追蹤和分析),您可以快速插入這些功能以提供遠端資料庫和功能的服務.

本書 Embarcadero RAD 伺服器指南最初由 David I (2019) 撰寫,現已是第二版,由 Antonio Zapater (2023) 修訂,其中包括自 RAD 伺服器推出以來根據市場需求添加的許多附加功能. 第二版還配有支援每一章的 綜合影片系列, 以及 GitHub 上的原始碼範例. https://github.com/embarcadero/radserver-docs

videos

*您可以訪問 Youtube 上提供的與本文連結的所有影片系列. 另外,我們強烈建議從此 GitHub 儲存庫下載所有範例.*

## 內容

# 01

# 什麼是 RAD Server? 簡介



現今的運算環境不再侷限於桌面,裝置,伺服器或資料中心. 應用程式正在從桌面轉移到多個裝置,網路邊緣連接以及本地,公有和混合雲服務. 透過 RAD Server 和 RAD Studio,您可以建立涵蓋公司(和客戶)廣泛的運算需求和業務需求的解決方案.

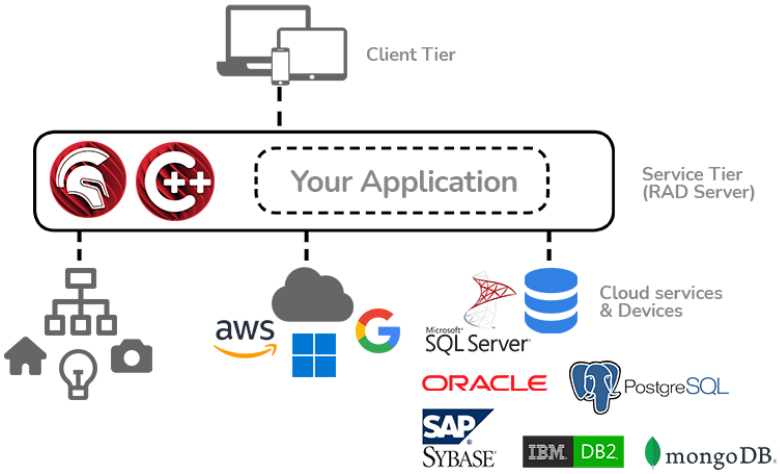
本書將向您展示如何使用 RAD Studio,Delphi 和 C++Builder 企業和架構師版本中提供的 RAD Server 基於 REST 的 API 託管引擎,元件和技術來快速設計,建置,偵錯和部署基於服務的多層應用程式.

note

*在整個 RAD 伺服器文件和原始碼中,您將看到對 EMS(企業行動服務)的引用. EMS 是現在的 RAD 伺服器產品的原始名稱.*

## RAD Server 概論

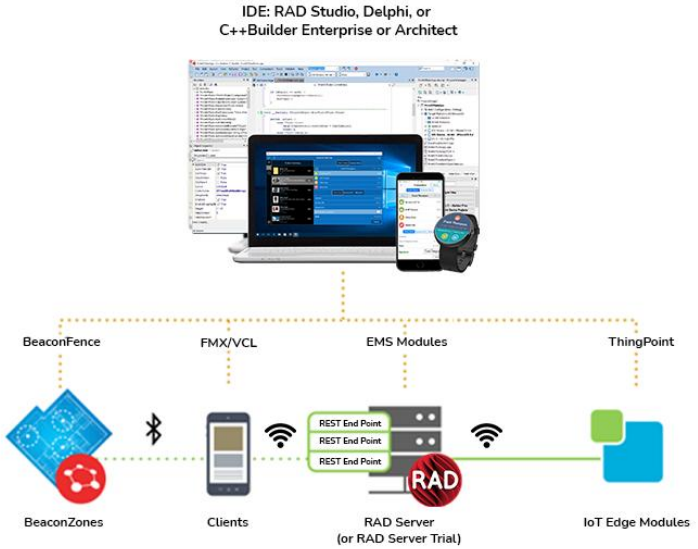
Embarcadero 的 RAD 伺服器是使用 Delphi 和 C++Builder 快速建置和部署基於服務的應用程式提供了一站式方案應用程式基礎. RAD Server 支援 REST(表述性狀態傳輸)協議,使用 JSON(或 XML)參數傳遞和傳回結果. 您可以發布 API,管理連接到 RAD 伺服器的使用者和裝置,擷取有關應用程式的使用和使用者的分析數據,使用 FireDAC 元件連接到本機和企業資料庫等等. RAD 伺服器也支援用戶身份驗證,推播通知,地理位置和資料存儲.




開發和測試 REST 端點和位置跟踪

借由 RAD Server 的精靈,元件和工具,您可以快速開發新的中間件和後端應用程式,或將現有的 Delphi 和 C++Builder 客戶端/伺服器應用程式遷移到基於 RAD Server 的應用程式,以便在伺服器或雲端運行. 您可以為來自桌面,行動裝置,控制台,Web 和其他類型應用程式的 REST 呼叫發布服務端點. RAD Server 附帶一整套工具,元件,資料庫連接和接口介面,您將在建立服務應用程式時依賴它們.

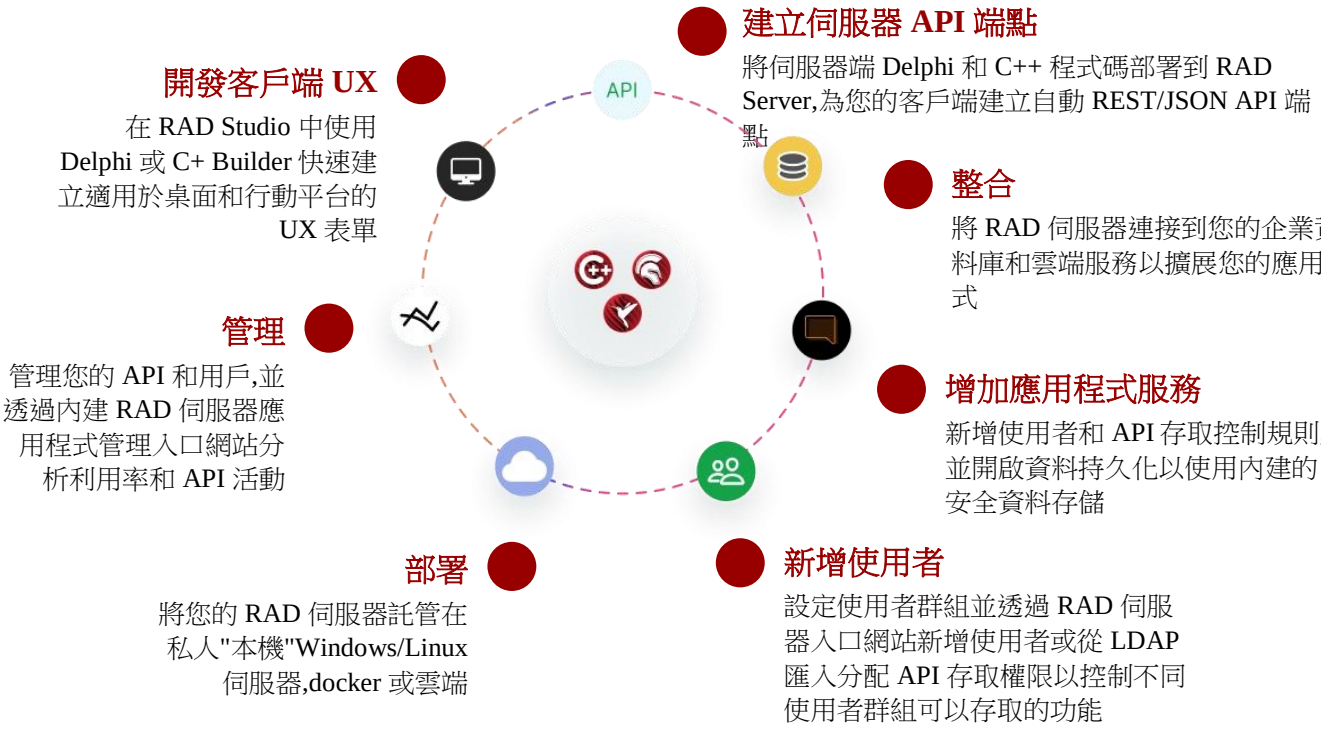
RAD 伺服器應用程式可以部署在 Microsoft Windows IIS 和 Apache Web 伺服器之上,您可以將基於 Delphi 的服務部署到 Linux Intel 64 位元伺服器. 有關 Linux 的 C++Builder 支持,請繼續關注 Embarcadero RAD Studio 部落格的更新.




開發和測試 REST 端點、位置追蹤和 IoT 邊緣軟體

## 開發基於 RAD 伺服器的應用程式 – 七個關鍵面

為了建立基於 RAD 伺服器的應用程式,下圖指導開發人員完成七個面向和開發階段.




讓多層開發變得簡單

首先,建立基於伺服器 REST/JSON API 的端點(如果需要的話,您也可以使用 XML 而不是 JSON). 接下來,您將透過整合廣泛的資料庫,雲端服務和其他技術來擴展端點.



您可以為使用者新增更多應用程式端點並建立 API 存取控制規則. 您可以編寫程式碼,利用 RAD Server 的內建安全資料儲存來追蹤持久性數據. 您可以透過控制台入口網站建立使用者群組並新增用戶,並透過基於 LDAP 的 API 服務匯入和驗證用戶.

開發和偵錯應用程式後,您可以在私人本地 Windows 和 Linux 伺服器上託管 RAD 伺服器應用程式. 您也可以將應用程式移轉到 Amazon AWS,Microsoft Azure,Google 和其他雲端供應商等雲端系統.

應用程式投入生產後,您可以使用內建應用程式管理介面管理對 API 的存取,控制使用者存取並分析端點 API 活動的使用率. 最後,您可以建立 RAD Studio 支援的桌面,行動,Web,控制台和其他應用程式類型. 您還可以使用 Sencha 的 Ext JS 元件集建立現代 Web 用戶端應用程式,並使用其他工具和程式語言建立支援 RAD 伺服器應用程式的 REST/JSON 功能的客戶端應用程式.

## 建立 RAD 伺服器應用程式的要求

以下部分包含建置,測試和部署 RAD 伺服器應用程式的產品和技術要求.除非另有說明,否則"RAD Studio"和 IDE 適用於 RAD Studio,Delphi 和 C++Builder 產品.

### 使用 RAD Studio IDE

建置 RAD 伺服器應用程式需要具有商業許可證的 RAD Studio 企業或架構師版本. RAD Studio 企業試用版可使用 30 天用於開發和測試.試用版不支援部署到生產環境伺服器.

## RAD Server 測試和部署授權

30 天 RAD Studio 免費試用版包括 RAD Server 5 個用戶開發試用版. RAD Server 部署授權包含在 RAD Studio 的 企業和架構師商業版本中. RAD Studio 企業版包括 RAD Server 的單一站點部署授權,而對於有效更新訂閱 RAD Studio 架構師版本的客戶則包含多站點部署授權. 自 RAD Studio Alexandria 版本起,企業和架構師版都可以選擇在多站點環境中部署 RAD Server Lite.

RAD Server 需要 InterBase 加密資料庫作為在生產環境中部署應用程式的一部分. 您需要使用有效的 RAD 伺服器授權才能安裝此版本的 InterBase.

tip

*如果您還想使用 InterBase 部署應用程式,那麼就需要執行 2 個 InterBase 實例: 一個用於您的應用程式,另一個用於 RAD 伺服器.*

# 核心 RAD 伺服器功能綜述

RAD Server 為開發人員提供了廣泛的功能來建立基於 REST 的服務應用程式. RAD Server(以前稱為 EMS)首次在 RAD Studio 版本 XE7 中引入. 自第一個版本以來,添加了增強功能和新功能,以滿足開發人員的需求並添加對新平台,架構和技術的支持.

## 核心功能



以下是您在建立基於服務的應用程式時需要利用的一些 RAD Server 核心功能的列表.

- **REST 端點 發布**–RAD Server 為您的應用程式後端 API 和服務實現一站式方案基礎. RAD Server 提供易於使用的 API 來發佈您的業務邏輯. Delphi 或 C++ 程式碼可以作為 API 託管,並作為 REST/JSON 端點自動發布,由 RAD 伺服器測量和管理.端點發布功能包括:
  - 存取控制 –您可以透過身份驗證設定對所有應用程式 API 的群組和使用者層級存取權限,並控制誰有權存取應用程式的 API 功能. 建立您自己的使用者和群組或從 LDAP 基礎架構自動匯入它們.
  - API 分析–所有 REST API 端點活動都會被記錄和測量,以進行可靠的統計追蹤和分析. 您可以每天,每月和每年分析使用者,API 和服務活動,以深入了解應用程式的使用情況. 您也可以篩選所有資源的活動或按特定群組,使用者,裝置安裝等篩選活動. 您也可以將分析結果匯出到 CSV 文件,以便使用其他工具進行額外分析.
  - 桌面,行動裝置和 Web 用戶端應用程式–RAD Server 上託管的所有 C++ 和 Delphi 程式碼均作為 REST/JSON 端點發布,可供多個平台上的任何類型的客戶端應用程式使用,以實現極高的靈活性和面向未來的需求.
- **整合中介軟體** –RAD Server 提供多種開箱即用的整合功能,可連接到外部伺服器,應用程式,資料庫,智慧型裝置,雲端服務和其他平台.整合能力包括:
  - 企業數據–RAD 伺服器為所有流行的企業 RDBMS 伺服器提供高效能的內建連接. 資料庫連接使用 FireDAC 元件和函式庫,可以輕鬆連接各種來源的數據.

  - 雲端服務–透過 RAD Server，您可以輕鬆整合來自各種雲端,社交和 BaaS 平台(例如 Google,Amazon,Facebook,Kinvey 等)的 REST 雲端服務.

- **應用程式服務** –RAD 伺服器包含一系列隨時可用的內建服務來為您的應用程式提供支援. RAD Server 包括使用者目錄服務和使用者管理、推播通知、使用者位置追蹤和內建資料儲存等核心功能. 其中一些應用程式和設備服務包括:

  - 推播通知–使用 RAD Server,您可以向應用程式使用者及其裝置發送程式設計或按需求通知. RAD Server 目前支援推播通知系統,包括 Apple 推播通知服務 (APN) 和 Google FireBase 雲端訊息傳遞 (FCM).您也可以編寫自訂程式碼來連接其他推播通知系統.

  - 內建安全資料存儲–借助 RAD Server 對保護 InterBase 伺服器加密資料儲存的支持,您可以使用內建 APIS 來儲存和檢索 JSON 數據,而無需單獨的資料庫伺服器.

  - 使用者/群組管理–使用 RAD 伺服器 API,您可以建立和管理使用者、使用者群組,並透過 RAD 伺服器控制台 (RSConsole.exe) 控制存取.整合您的 ActiveDirectory (LDAP) 或開發您自己的自訂身份驗證中間件.

  - 用戶位置/鄰近度–您的 RAD 伺服器應用程式可以利用 RAD Studio 對 GPS、信標和信標圍欄技術的支持. RAD 伺服器應用程式可以追蹤使用者在室內和室外的移動,並在使用者進入和退出自訂信標區域或接近指定信標點時回應接近事件.

  - 靜態文件提供者–將 URL 對應到資料夾並傳回 HTML、JS、CSS、圖像等檔案的內容.這在小型部署(便如：使用 RAD Server Lite)或開發環境中非常方便.

- **API 文件** –使用屬性和內建 Swagger OpenAPI 整合輕鬆建立 API 文件. 將 Swagger UI 嵌入 RAD Server 本身或透過自動產生的 YAML 和 JSON 檔案在遠端實例中配置它.

- **方便部署** –RAD 伺服器易於開發,部署和操作,非常適合 ISV 和 OEM 建置可重新部署的解決方案.部署在 Windows,Linux 或 Docker 上.



## 請參考

有關 RAD Studio 安裝和基於 RAD Server 的應用程式部署的最新更新信息,請參閱以下 Embarcadero 在線鏈接.

- RAD 伺服器產品概述
- RAD Studio 安裝說明
- RAD Studio 和 RAD Server 支援的目標平台
- 生產環境的 RAD 伺服器資料庫要求
- RAD Studio 的平台狀態頁面
- InterBase
- FireDAC
- FireDAC 支援的資料庫
- RAD Studio 企業行動服務
- RAD Studio 產品功能表(PDF - 查看 RAD 伺服器章節說明)
- Swagger 開放 API
- EMS 推播通知
- Apple 推播通知服務 (APN)
- Firebase 雲端訊息傳遞 (FCM)

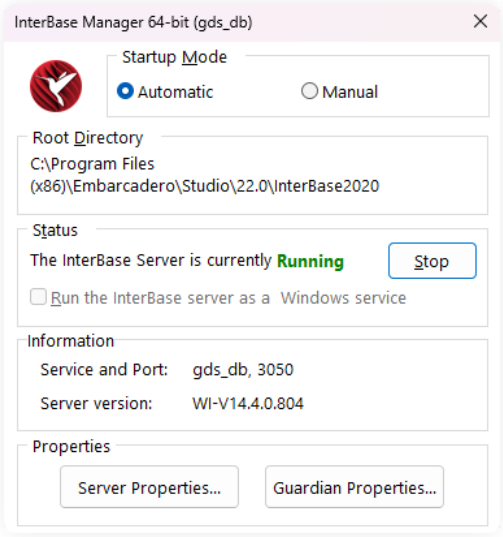
 

# 02

# 使用 RAD 精靈建立“Hello World”



是時候開始程式設計了.在本章中,您將學習如何使用 Delphi 和 C++Builder 建立第一個基於 RAD 伺服器的服務應用程式. 在開始之前,您需要確保您的 InterBase 資料庫伺服器正在執行. RAD 伺服器使用 InterBase 資料庫來儲存使用者資訊,使用者群組,分析,註冊設備,版本資訊,註冊邊緣模組,推播通知訊息等.

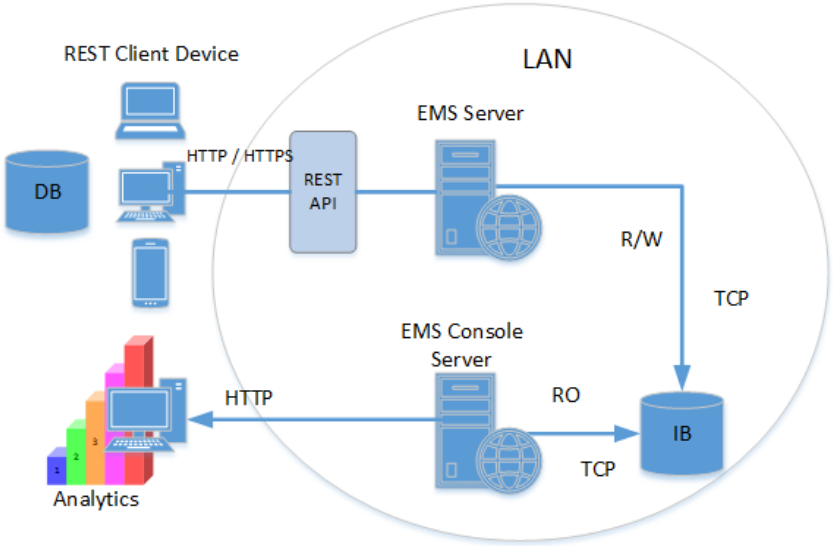


# 建構基於 REST 的服務

RAD Server 包括企業行動服務 (EMS) 並提供可在雲端或本地端託管的行動企業應用程式平台 (MEAP). 開發人員可以使用 RAD Server 公開自訂 REST API,並使用 FireDAC 資料存取庫和元件存取企業資料庫數據.

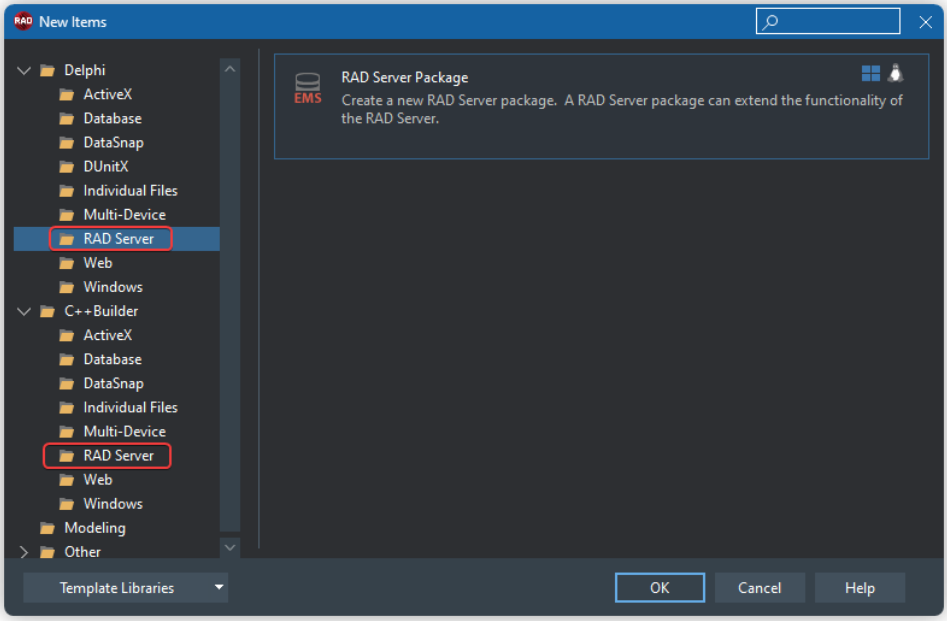
RAD Server 為開發人員提供了全面的解決方案,包括遠端資料庫存取,使用者追蹤,設備應用程式管理,使用分析等. 與其他解決方案相比,RAD Server 包含一個預先建置的應用程式伺服器,支援自訂套件的整合. 這些自訂套件可以輸出資料集,業務邏輯和其他基於 REST 的資源. 元件還可用於行動,Web 和桌面應用程式程式碼來存取 RAD 伺服器資源.




RAD 伺服器 REST API 架構

## 使用 RAD 伺服器專案精靈

最快的入門方法是使用"新建項目"選單(File | New | Other⋯)並選擇 Delphi 或 C++Builder 的 RAD Server | RAD Server Package 精靈.

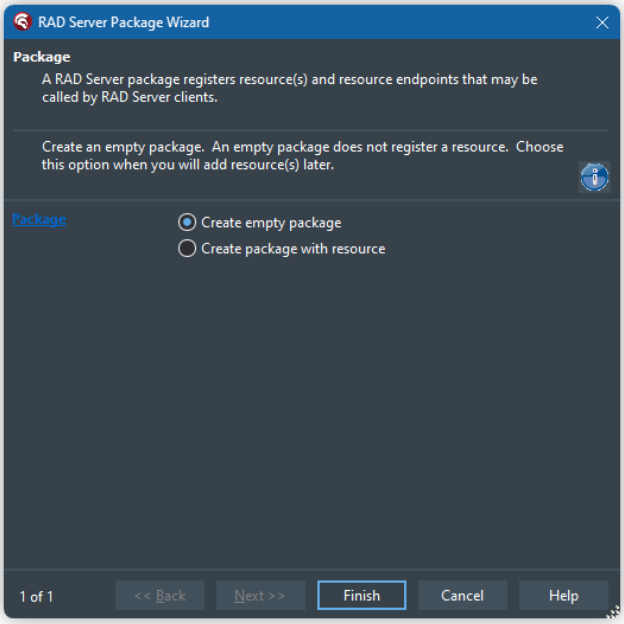



適用於 Delphi 和 C++ 的 RAD 伺服器專案精靈選項

選擇 RAD Server Package 專案. 將出現一個精靈來幫助建立起始項目. 在第一頁上,選擇精靈如何建立將出現在 RAD 伺服器應用程式中的資源和端點. TRAD 伺服器套件精靈提供了兩種繼續選擇.

選擇 1：建立一個不註冊資源的空白套件. 如果您稍後才要新增資源,使用此選擇,將建立起始主庫並建立一個套件項目.

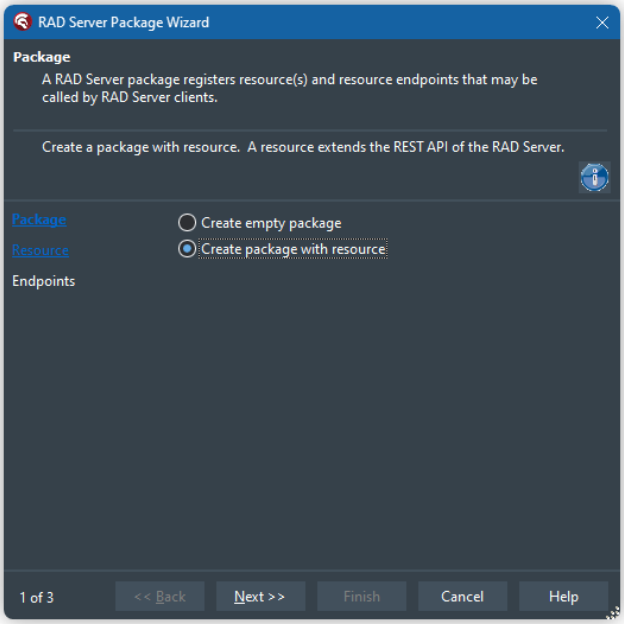





建立一個空的 RAD 伺服器套件

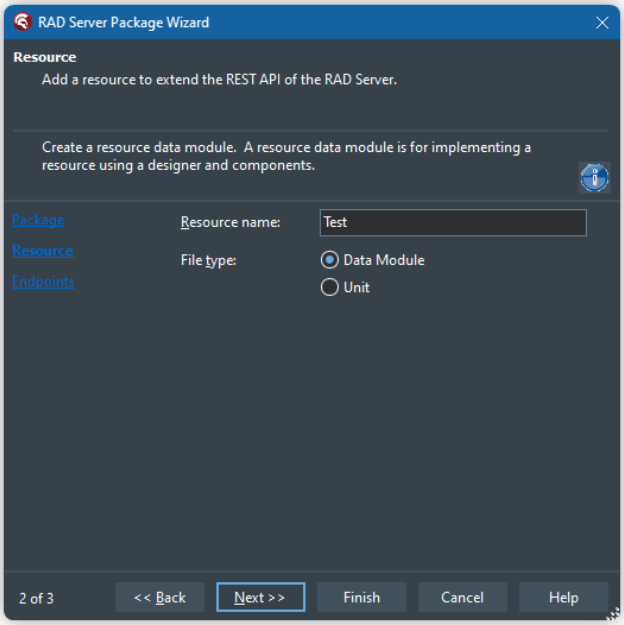
點擊「完成」按鈕將建立起始專案以進行額外的開發工作,以便建立最終的 RAD 伺服器應用程式.

選擇 2:使用擴充 RAD 伺服器 REST API 的資源建立套件封包. 按一下「下一步」按鈕,將出現兩個附加精靈步驟,以協助建立套件專案,資源和端點.要建立第一個 RAD 伺服器項目,請做此選擇.



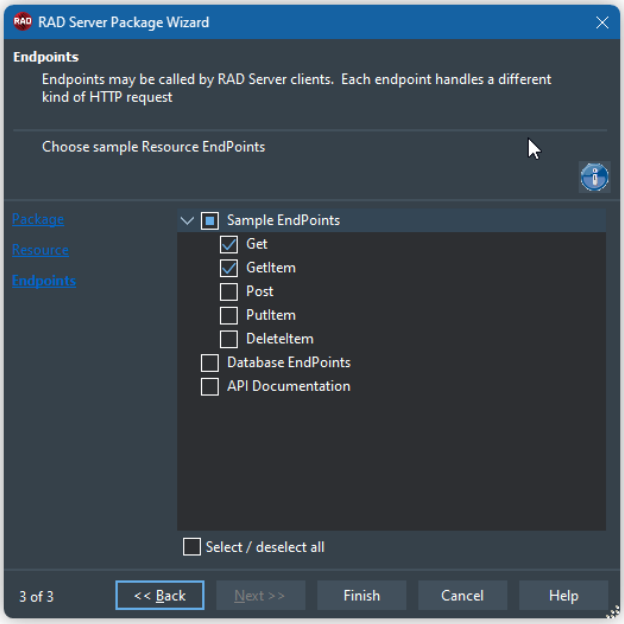

建立基於資源的 RAD 伺服器封包

在精靈的第二頁上,將資源名稱設定為"Test" 文件類型單選按鈕提供兩個選項：1）建立用於在程式碼中實現資源的程式單元,2）建立用於使用 IDE 設計器,元件和程式碼編輯器實現資源的資料模組.對於第一個 RAD 伺服器應用程式,請選擇使用資料模組.




RAD 伺服器套件精靈第 2 頁 - 設定資源名稱與檔案類型

點選下一步按鈕以建立一組起始端點.




RAD 伺服器套件精靈第 3 頁 - 選擇起始端點

在精靈第三頁上,保留建議的端點,如上圖所示:Get (REST GET) 和 GetItem(REST GET 在 URL 末尾帶有標識要取得的項目) 並且取消選取"API 文件" .若要建立您的起始項目,請按一下"完成"按鈕.

note

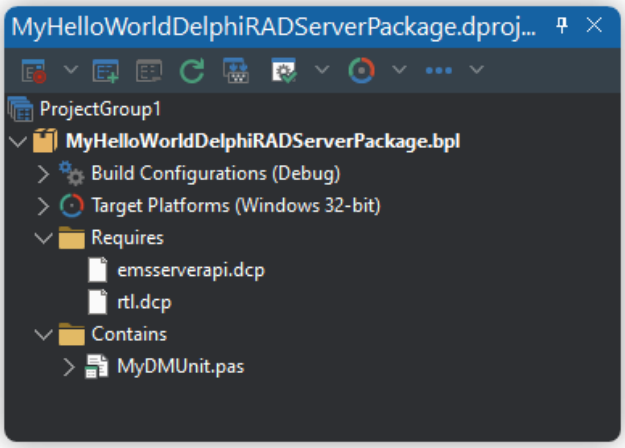
*精靈中有兩個額外選項: 資料庫端點(使用 FireDAC 將資料庫連結到端點)和 API 文件 (Swagger OpenAPI).我們將在接下來的章節中更詳細地討論這些內容.*

使用精靈完成後,您將返回 IDE.首先要做的是保存專案.對於 C++ 和 Delphi 資料模組,請使用名稱"MyDMUnit".對於 C++ 專案和套件,請使用名稱"MyHelloWorldCppRADServerPackage".對於 Delphi 專案和套件,使用名稱"MyHelloWorldDelphiRADServerPackage".

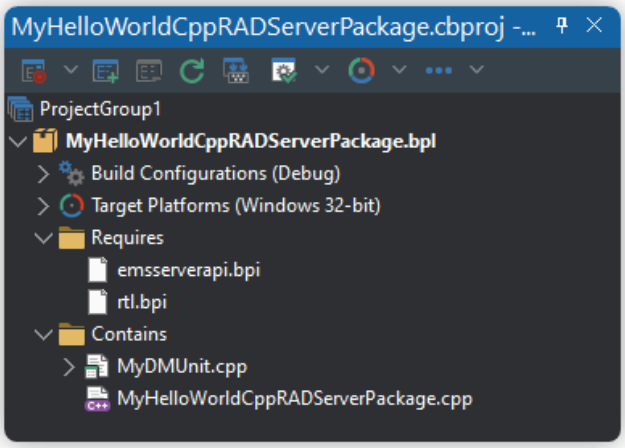
## 精靈 RAD 伺服器專案和原始碼

創建的專案程式碼非常少,並且只有幾個連結到每個端點的方法. RAD Studio 會自動填入一些預設程式碼,我們將對其進行一些調整,使其更加 Hello World 化.

Delphi 和 C++ 上的專案應如下所示.創建的 DataModule 將是空白的,其中沒有任何元件.在截圖之後您可以發現自動產生的範例程式碼中所做的修改.




產生的 Delphi 專案




產生的 C++專案

□ 備註

*本書中使用的所有原始程式碼和範例都託管在 GitHub 上並區分為章節.我們強烈建議您下載整個儲存庫,以便更好地遵循本書的進度.*



**MyDMUnit.pas:**

```
procedure TTestResource1.Get(const AContext: TEndpointContext; const ARequest:
TEndpointRequest; const AResponse: TEndpointResponse);
begin
  AResponse.Body.SetValue(TJSONString.Create('Hello World'), True)
end;

procedure TTestResource1.GetItem(const AContext: TEndpointContext; const ARequest:
TEndpointRequest; const AResponse: TEndpointResponse);
var
  LItem: string;
begin
  LItem := ARequest.Params.Values['item'];
  AResponse.Body.SetValue(TJSONString.Create('Hello World ' + LItem), True)
end;
```

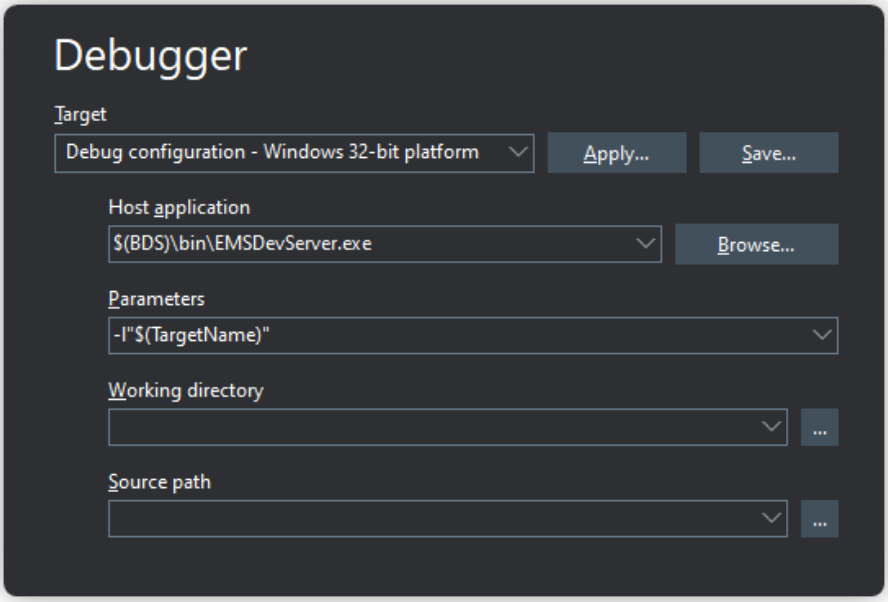
**MyDMUnit.cpp:**

```
void TTestResource1::Get(TEndpointContext* Acontext,
      TEndpointRequest* ARequest, TEndpointResponse* AResponse)
```

```
{
	AResponse->Body->SetValue(new TJSONString("Hello World"), True);}

void TTestResource1::GetItem(TEndpointContext* Acontext,
	TEndpointRequest* ARequest, TEndpointResponse* AResponse)
{
	String item;
	item = ARequest→Params→Values["item"];
	AResponse->Body->SetValue(new TJSONString("Hello World "+item), True);
}
```

## 為您的第一個應用程式設定 RAD 伺服器

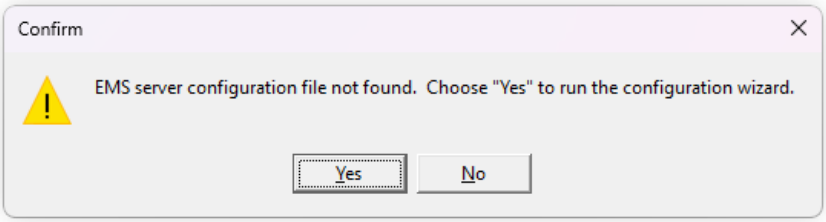
現在您已經使用精靈建立了第一個 RAD 伺服器應用程式,您可以使用 IDE 來編譯和測試該您的應用程式. IDE 使用 EMSDevServer 作為主機可執行檔(EMDDevServer.exe)開始執行,其參數為要載入的套件文件, 在此處也就是您剛才建立的套件. 用於 Win32 和 Win64 開發的 EMSDevServer 有兩個版本($(BDS)\bin\EMSDevServer.exe 和 $(BDS)\bin64\EMSDevServer.exe).這一切都是由 IDE 自動配置的.



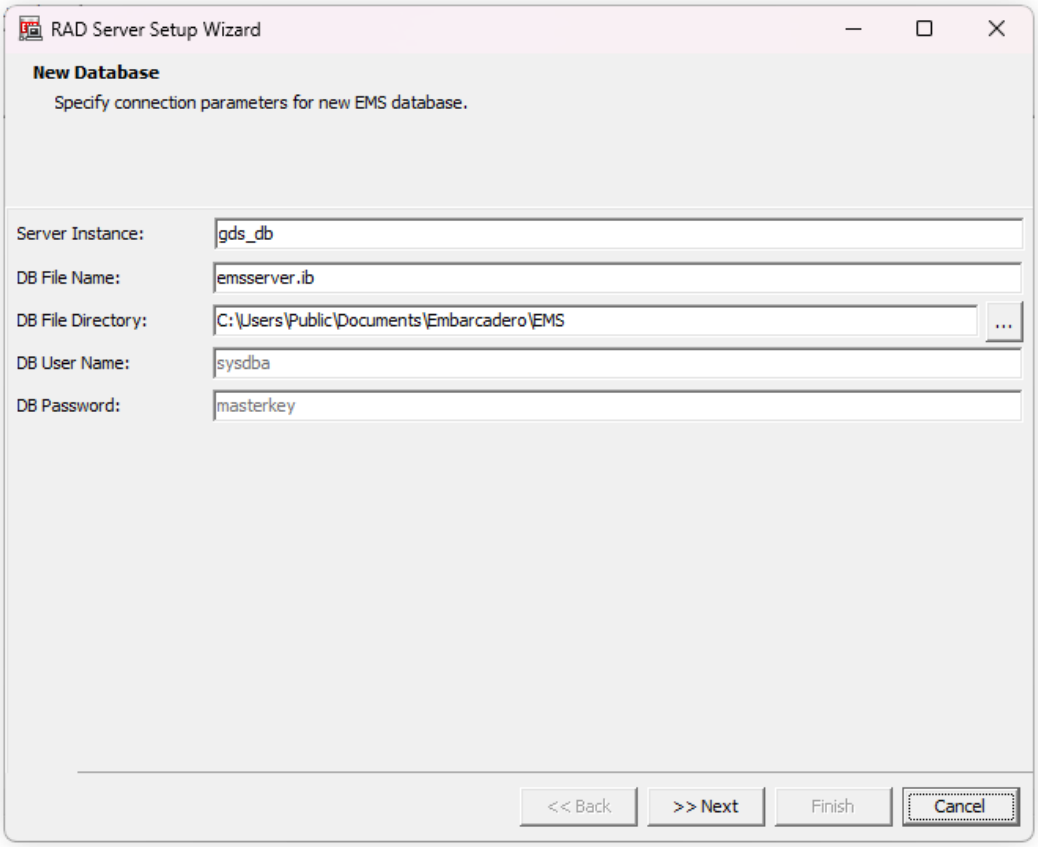

Run | Parameters...對話框顯示 EMSDevServer.exe 作為主機應用程式

RAD Studio 還包括 EMSDevConsole.exe,它將啟動 EMSDevConsole 伺服器並開啟 EMS 控制台伺服器視窗. EMS 控制台提供了一個 Web 應用程式,可顯示分析,提供使用者/群組管理以及 RAD 伺服器應用程式的更多功能.該控制台將在下一章中更詳細地介紹.

選擇 Run | Run 選單項目或按 F9 編譯,連結並執行啟動應用程式. 預設情況下,RAD 伺服器開發伺服器將使用 TCP 連接埠 8080 啟動.如果這是您第一次執行 RAD 伺服器應用程式,則會出現一個對話框,提示未找到 RAD 伺服器設定檔 emsserver.ini.當沒有 RAD 伺服器註冊表項或設定檔不存在時,就會出現這情況.




嘗試在沒有設定檔的情況下啟動 RAD Server 開發伺服器. 請點選"是"按鈕執行 RAD 伺服器設定精靈.




設定精靈第 1 頁 - 指定新的 EMS 資料庫連線參數

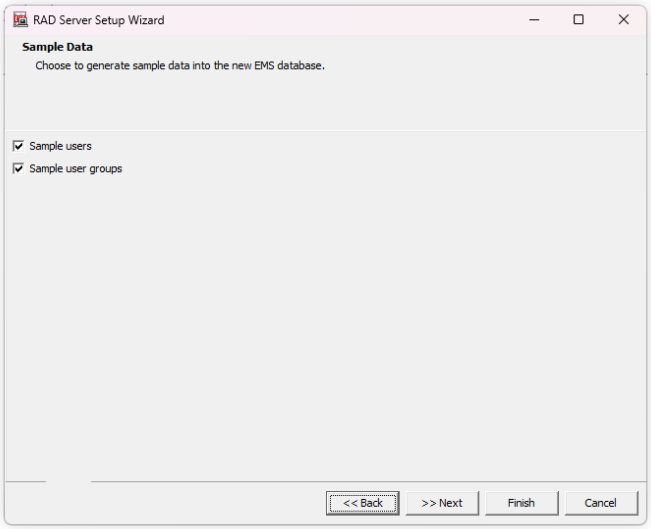
在第一個精靈步驟中,輸入互聯網伺服器實例名稱(預設情況下 RAD Studio 的 InterBase 伺服器的開發版本使用 gds_db).此精靈頁面還包含 RAD 伺服器資料庫的名稱 (emsserver.ib) 以及包含資料庫和設定檔的目錄.

□ 警告

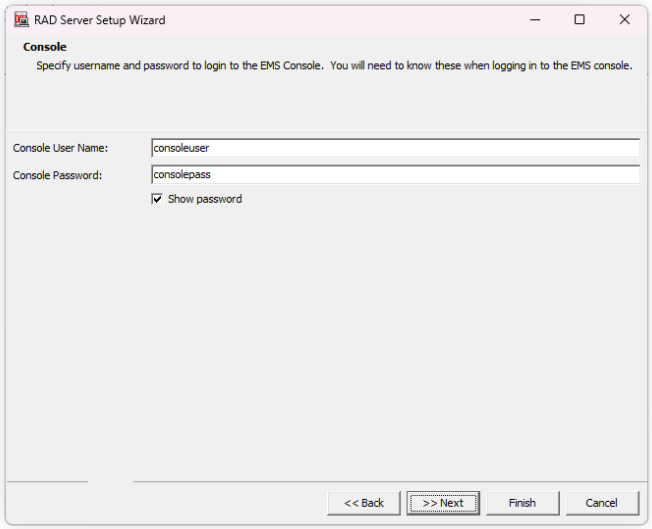
*如果您之前使用不同的伺服器實例名稱在電腦中安裝了 InterBase,請輸入該名稱字串.*

點選下一步按鈕告訴精靈是否為使用者和使用者群組建立範例 RAD 伺服器資料庫資料.對於開發和測試,我們將勾選此兩個選擇項目.

點選"下一步"按鈕設定用於登入 RAD 伺服器控制台的使用者名稱和密碼.

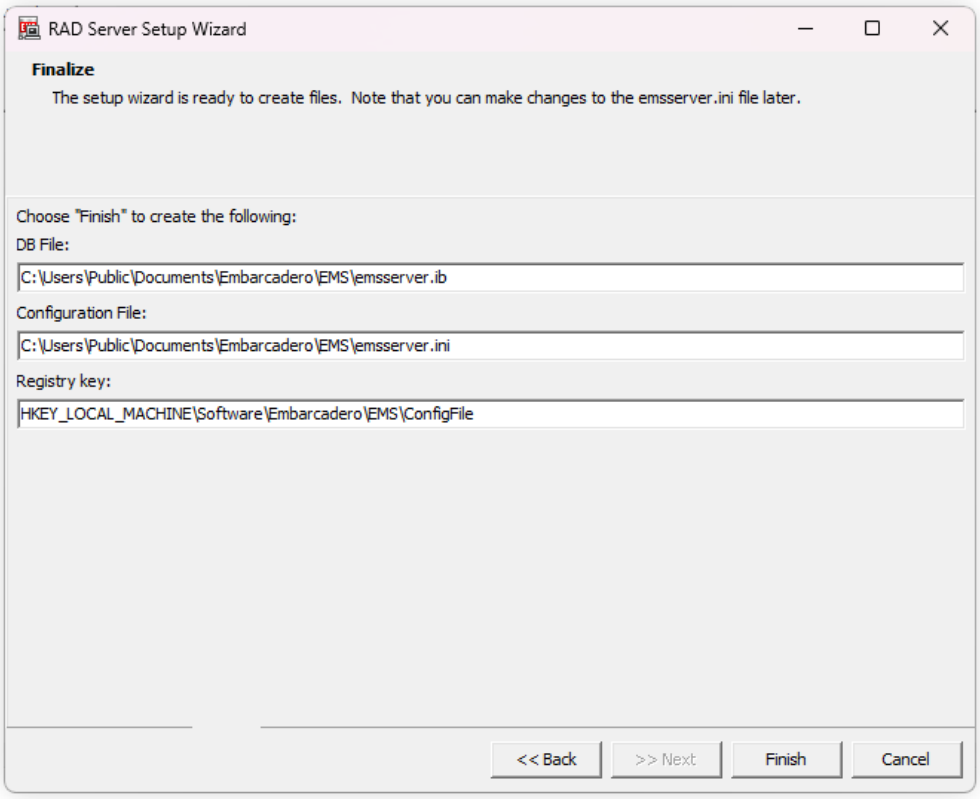



設定精靈第 2 頁 - 使用者和群組範例




設定精靈第 3 頁 - 選擇控制台使用者名稱和密碼

最後點選"下一步"按鈕前往精靈的最後一步.此精靈已準備好建立 RAD 伺服器資料庫檔案,設定文件,並為目前登入的使用者設定 Windows 註冊表項.

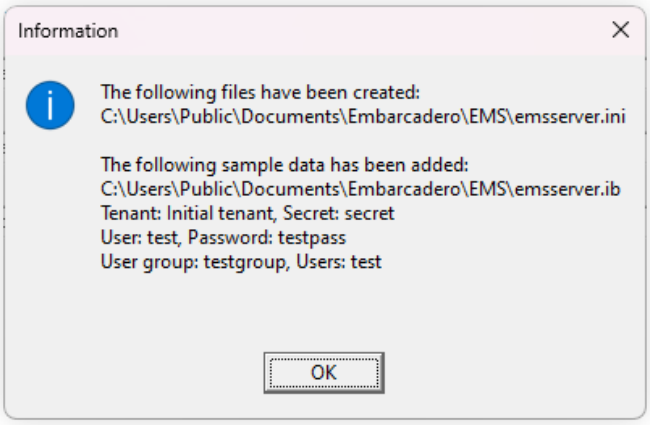



最終 RAD 伺服器設定精靈頁面

此頁面顯示資料庫檔案路徑和名稱,設定路徑和名稱以及 Windows 登錄項目. 您可以隨時變更 RAD 伺服器設定檔 (emsserver.ini). 點選完成按鈕. 此時將出現確認對話框,並提醒您設定將使用沒有 RAD 伺服器許可證

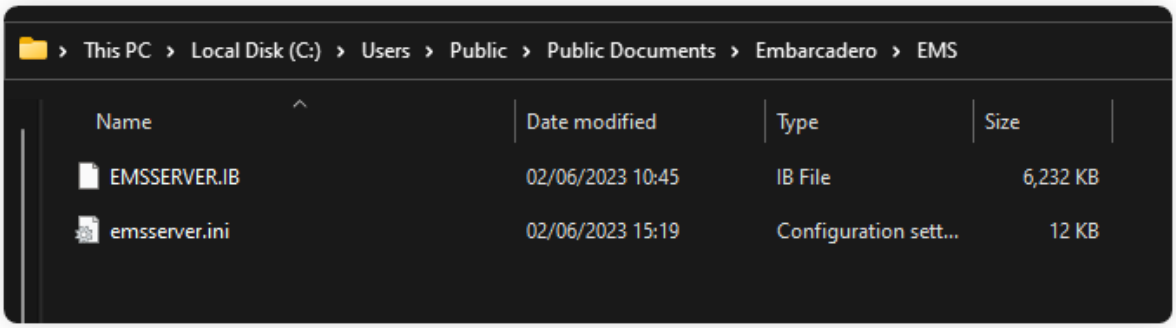
的 InterBase 實例. 開發授權將您的 RAD 伺服器應用程式限制為最多 5 個用戶. 當您準備好部署 RAD 伺服器應用程式時,您將能夠使用 RAD 伺服器和 InterBase 的部署授權.

點選"是"按鈕.此精靈將顯示 emsserver.ini 設定檔的位置.它還列出了已添加到資料庫的範例數據.




設定精靈建立的 RAD 伺服器檔案列表

按一下“確定”按鈕.現在,有兩個檔案出現在 C:\Users\Public\Documents\Embarcadero\EMS 目錄中.




RAD 伺服器設定精靈在磁碟上建立的文件檔案

備註

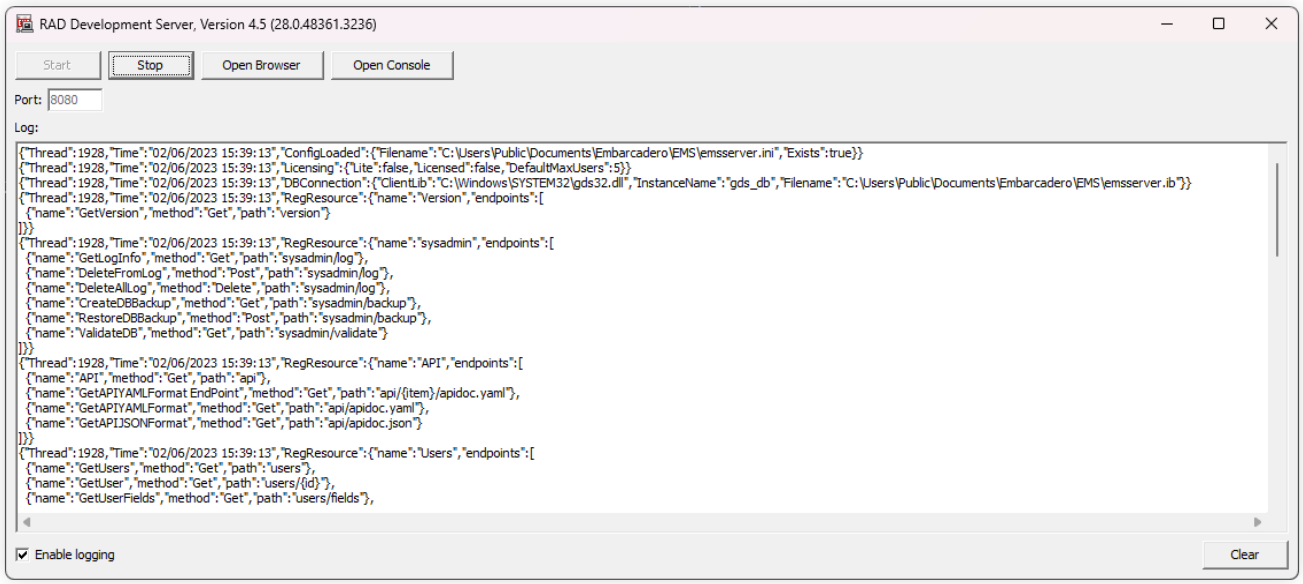
*emsserver.ini 檔案是定義 RAD 伺服器所有預設參數值的地方. 儘管它將在下一章中討論, 但請隨意檢查其內容以及文件本身中指定的擴展文檔.*
*一般來說,RAD Studio IDE 是在沒有管理員權限的情況下啟動. 因此,HKEY_LOCAL_MACHINE 的 Windows 登錄機碼在 Windows 64 位元作業系統上虛擬化為 HKEY_CURRENT_USER\Software\Classes\VirtualStore\MACHINE\SOFTWARE\WOW6432Node\Embarcadero\EMS.*

## 測試您的第一個 RAD 伺服器應用程式

當 RAD Server 設定伺服器建立完成後,RAD Server 開發伺服器將開始執行.

> □
> **警告**
>
> *EMSDevServer 運行的預設端口是 8080.如果您的電腦將該端口用於其他服務,您可以更改使用的預設端口,在"Port"欄位中更改它以滿足您的需求. 若要使此變更永久有效, 請修改 emsserver.ini 檔案中的參數*`[Server.Connection.Dev]`

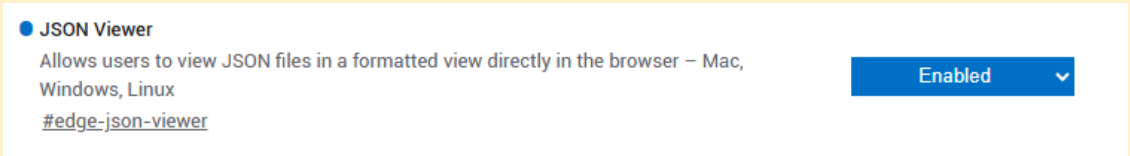
當您在 IDE 中點擊"執行"時,您的 RAD 伺服器開發伺服器將開始執行,並且日誌將顯示您的套件應用程式執行的流程. Delphi 和 C++ 上的開發伺服器完全相同.

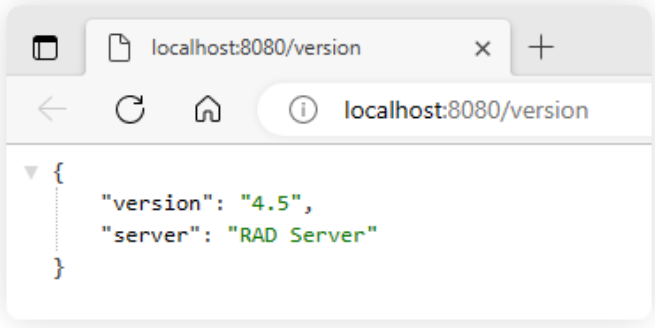



RAD Server 開發伺服器啟動第一個應用程式套件

RAD Server 發伺服器日誌將顯示設定、資料庫連線、授權資訊、已載入的應用程式套件、註冊的資源以及已建立的端點.

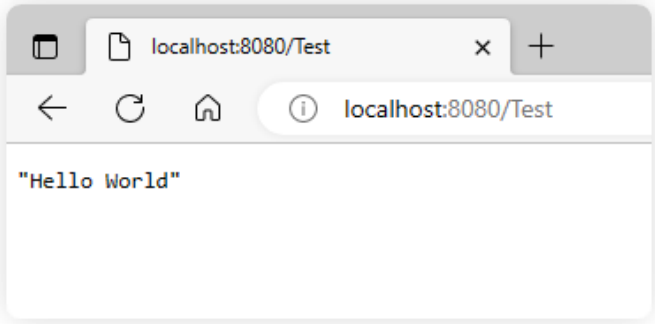
點擊"Open Browser"按鈕將啟動預設瀏覽器並顯示呼叫 GetVersion 內建端點的 JSON 結果.您現在已經使用了第一個 RAD 伺服器 REST 端點!

> □
> **竅門**
>
> *要在瀏覽器上獲得更易於閱讀的 JSON 回應,您可以安裝擴充功能"JSON Parser".它適用於所有主要瀏覽器.*
> *在 Microsoft Edge 上,"edge://flags" 下有一個名為"JSON Viewer"的設置. 啟用此功能可為您提供可讀的 JSON 回應,而無需安裝擴充*
>
> 
> 



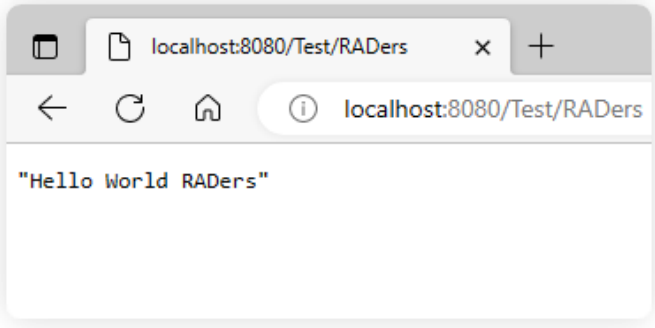

顯示呼叫版本端點的輸出的瀏覽器

在瀏覽器中,將 URL 變更為 localhost:8080/Test 並按 Enter 鍵.瀏覽器將從 Get 端點接收 JSON 回應.




瀏覽器顯示 Test 資源的 Get 方法的 JSON 結果

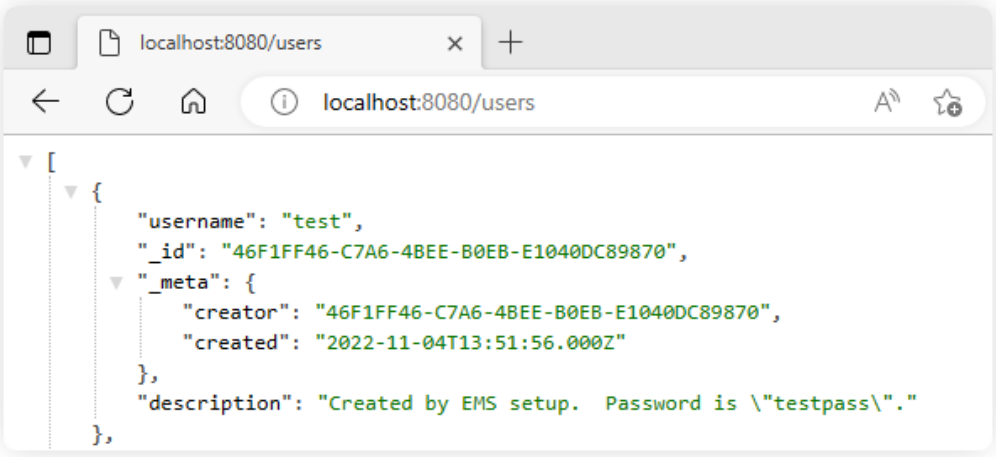
如果您在 URL 上傳遞附加項目,則會呼叫 GetItem 端點,並且後面的程式碼將傳回 JSON 字串,其中包含資源名稱以及您鍵入的項目.




瀏覽器顯示 Test 資源的 GetItem 方法的 JSON 結果

對於簡單的範例,可以傳回 JSON 字串,但對於更大,更複雜的資料結構,您可能不希望傳回大量的 JSON 字串.RAD Studio 提供了許多其他方法來產生 JSON 數據,包括使用 JSON 物件,JSON 流和 JSON 編寫器.

編輯 URL 以使用"users"資源,該資源將呼叫預設的 GetUsers 端點以顯示由 RAD 伺服器資料儲存中的 RAD 伺服器配置精靈產生的使用者的 JSON(只有一個可以啟動).




瀏覽器中呼叫 GetUsers 端點的 JSON 回應

現在您已經使用了 RAD 伺服器專案精靈產生的四個端點服務.

## 請參考

- 本章的程式碼範例
- RAD 伺服器引擎(EMS 伺服器)
- 設定您的 RAD Studio (EMS) 伺服器
- 在 Windows 上設定 EMS 伺服器或 EMS 控制台伺服器
- RAD 伺服器管理 API



# 03

# 建立您的第一個 CRUD 應用程式



RAD Studio 提供了多個隨時可用的元件,但在建立 CRUD API 時最有用的元件之一是 EMSDataSetResource. 該元件允許您將 FireDAC 查詢連結到它,不僅公開數據,還可以對其進行操作. 此元件會自動建立 CRUD 所需的所有端點,並提供額外的功能,如分頁、排序等.

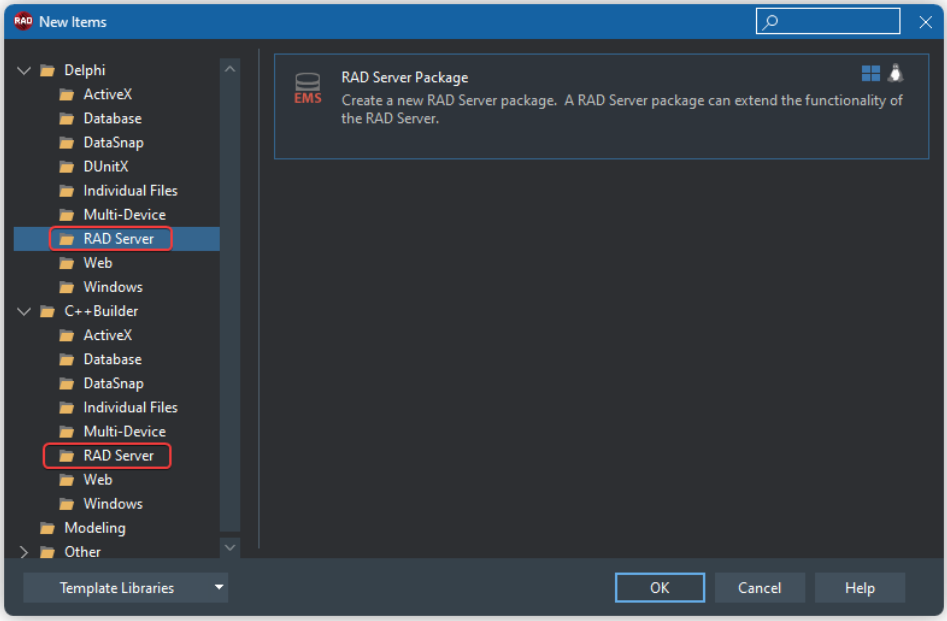
EMSDataSetResource 可以在您目前的任何單位中創建,或者甚至更容易一點,您可使用 RAD 伺服器精靈建立自動連結到 FDConnection 的所有必要組件.

□ **備註**

*在此範例中,我們將使用 InterBase employee 資料庫,但也可以隨意使用與 FireDAC 相容的任何其他資料庫. 使用 RAD 伺服器精靈的唯一要求是在「Data Explorer」中預先配置資料庫連接,以便 RAD Studio 識別它.*

## 建立基於 REST 且具備 CRUD 功能的服務

正如我們在上一章中所做的那樣,最快的入門方法是使用 “New Projects” 選單(File|New|Other...}並選擇 Delphi 或 C++Builder 的 RAD Server|RAD Server Package wizard...選單選項.

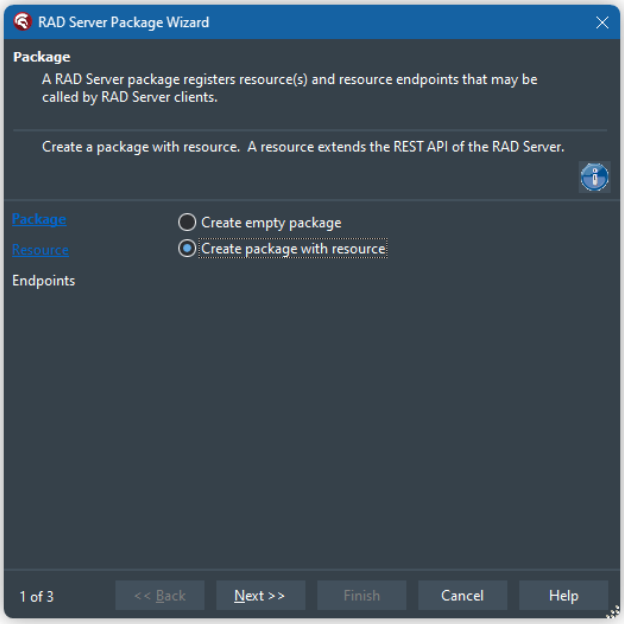



適用於 Delphi 和 C++ 的 RAD 伺服器專案精靈選項

選擇 RAD 伺服器套件項目.此時將出現一個精靈來幫助建立起始項目. 在第一頁上,選擇精靈如何建立將出現在 RAD 伺服器應用程式中的資源和端點. RAD 伺服器套件精靈提供了兩種繼續選項.

現在建立一個包含擴展 RAD 伺服器 REST API 資源的套件. 點選"下一步"按鈕,將出現兩個附加精靈步驟,以協助建立套件專案、資源和端點.要建立第一個 RAD 伺服器項目,請做出此選擇.

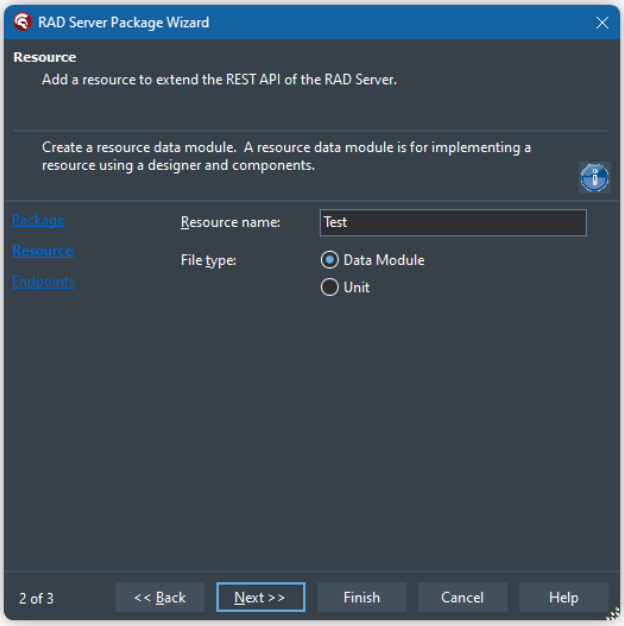





建立基於資源的 RAD 伺服器套件

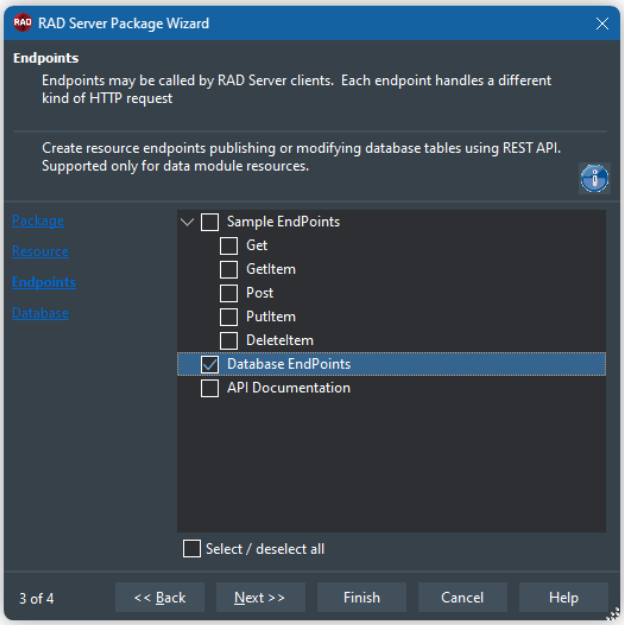
在精靈的第二頁上，將資源名稱設定為"Test". 文件類型單選按鈕提供兩個選項：1)建立用於在程式碼中實現資源的單元，2)建立用於使用 IDE 設計器、元件和程式碼編輯器實現資源的資料模組.對於第一個 RAD 伺服器應用程式,選擇使用資料模組.



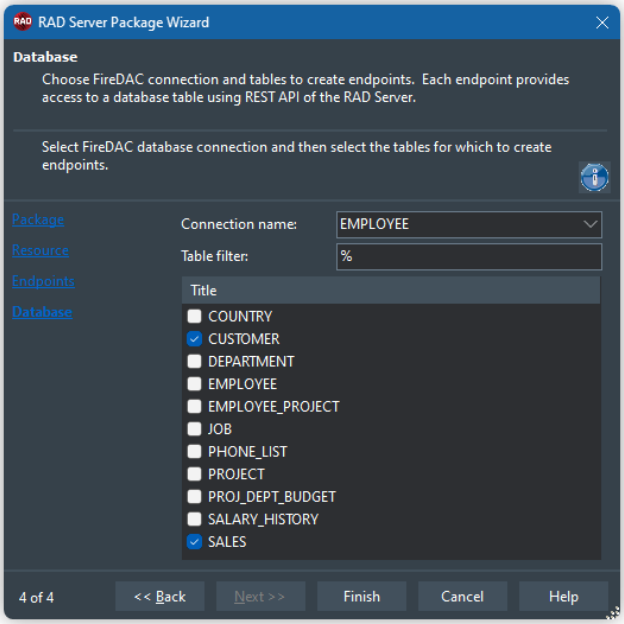

RAD 伺服器套件精靈第 2 頁 - 設定資源名稱與檔案類型

點選下一步按鈕以建立一組起始端點.



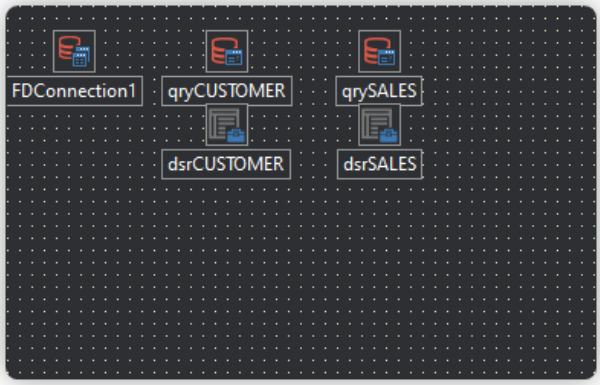

RAD 伺服器套件精靈第 3 頁 - 選擇起始端點

在精靈的第三頁上,我們將選擇與上一章相比不同的選項.在這種情況下,我們將取消選取"範例端點"並選取"資料庫端點". Now click “Next”.




RAD 伺服器套件精靈第 4 頁 - 選擇資料庫和表格

專案產生後,我們應該會看到一個 FDConnection、2 個 FDQueries 和 2 個 EMSDataSetResource.




精靈產生的資料模組

## 解釋產生的專案

這個範例的美妙之處在於我們已經可以構建它,並且我們將能夠訪問為我們自動生成的端點,但首先,讓我們修復一些問題: 存取 DataModule 的程式碼並更改端點的屬性. 這並不真正相關,但保持端點小寫和複數是常見的良好做法.

**Delphi:**

```
[ResourceName('test')]
TTestResource1 = class(TDataModule)
  FDConnection1: TFDConnection;
```

```
    qryCUSTOMER: TFDQuery;
    [ResourceSuffix('customers')]
    dsrCUSTOMER: TEMSDataSetResource;
    qrySALES: TFDQuery;
    [ResourceSuffix('sales')]
    dsrSALES: TEMSDataSetResource;
```

**C++:**

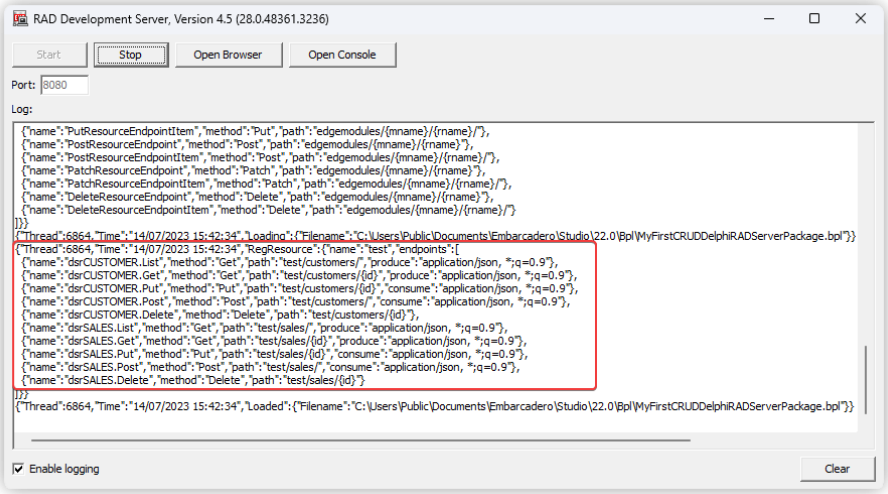
```
static void Register()
{
      std::unique_ptr<TEMSResourceAttributes> attributes(new
TEMSResourceAttributes());
      attributes->ResourceName = "test";
      attributes->ResourceSuffix["dsrCUSTOMER"] = "customers";
      attributes->ResourceSuffix["dsrSALES"] = "sales";
      RegisterResource(__typeinfo(TTestResource1), attributes.release());
}
```

正如我們所看到的,ResourceSuffix 屬性連結到 EMSDatasetResources.這意味著連結到該 DatasetResource 的查詢將在該端點下公開.



該項目再簡單不過了.到目前為止,我們沒有編寫任何邏輯程式碼,但我們已經擁有一個連結到 2 個表的功能齊全的 CRUD 系統. 讓我們建立該專案並進一步詳細分析它.

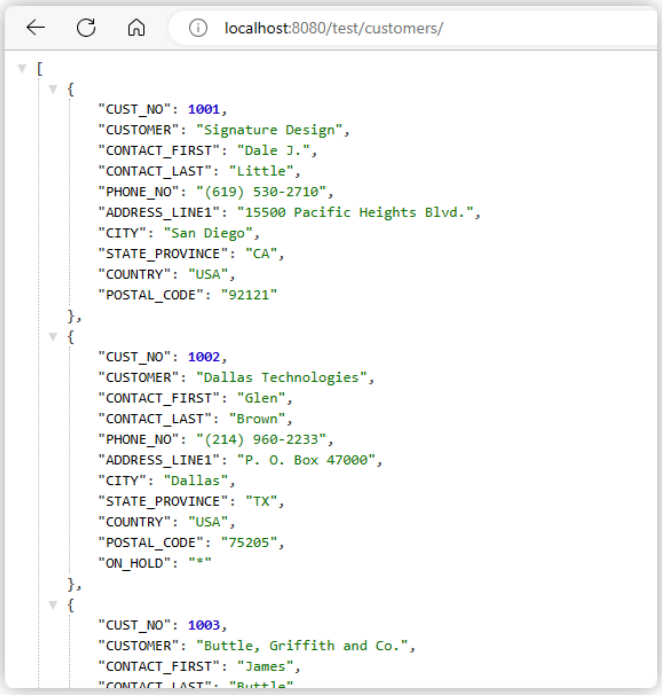
## 建置和測試項目




RAD 伺服器日誌顯示自動建立的所有端點

我們可以在 RAD 伺服器日誌中看到端點"customers"和"sales"已創建,而且還可以使用 URI 中的參數 {id} 來取得/儲存/刪除單一記錄或發布新記錄.

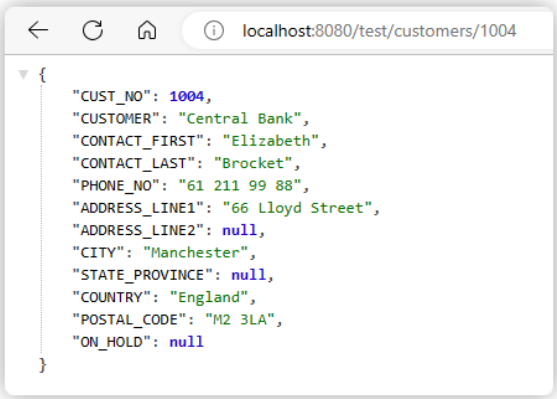
如果我們開啟瀏覽器並造訪 URL http://localhost:8080/test/customers/ 它會傳回一個資料集,其中包含 customer 表中的所有記錄.




RAD 伺服器傳回的客戶資料集



要使用 ID (Cust_No) 存取特定客戶,我們只需發送請求 http://localhost:8080/test/customers/1004. 正如您可能已經猜到的那樣,如果您想訪問銷售端點,您只需呼叫 http://localhost:8080/test/sales/ 等.



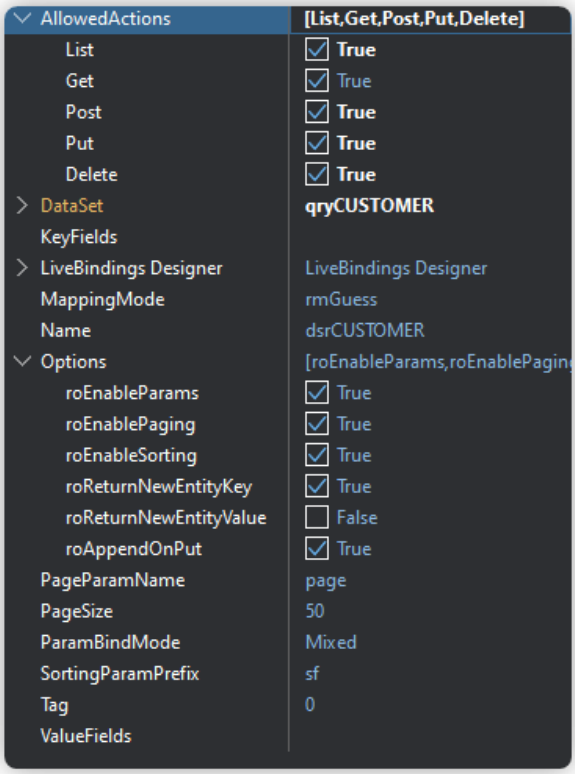

使用 ID 存取特定客戶

> **警告**
>
> *使用 TEMSDatasetResource 時,將斜線 / 保留在端點末端至關重要.在沒有使用 "/" 存取端點時 RAD 伺服器將拋出"未找到"異常.*

## TEMSDatasetResource 的附加功能

到目前為止我們所看到的已經非常令人印象深刻了.只需點擊幾下,我們就可以公開所需數量的資料集,並且非常快速地開發我們的 API,但 TEMSDatasetResource 提供了更多內建功能.讓我們來分析一下關鍵的功能:




**AllowedActions** –用於允許或阻止在端點上列出、存取、發布、放置和刪除的內建控制.

**DataSet** –連接到資料集：查詢、資料表等.

**KeyFields** –選擇進行查找時必須匹配的資料集欄位.

**PageParamName** –透過 URL 使用分頁的參數名稱.即：?page=1

**PageSize** –存取 LIST 作業時,定義有效負載的分頁大小.

**SortingParamPrefix** –將會新增到資料集 ValueFields 項目之前的文字字串.

**ValueFields** –選擇要在參數化查詢中使用並顯示在 JSON 回應中的字段欄位

**Options** –用於啟用/停用參數使用、行分頁、資料集欄位排序等的子屬性設置.



現在我們對這個元件有了更多的了解,讓我們嘗試在我們的 API 中使用其中一些功能如果我們造訪 http://localhost:8080/test/customers/?sfCONTACT_LAST=A&page=1,我們將透過欄位 CONTACT_LAST 按升序排列取得客戶的第一頁資料. 如果我們將值 =A 更改為 =D,則回應將按降序排列.

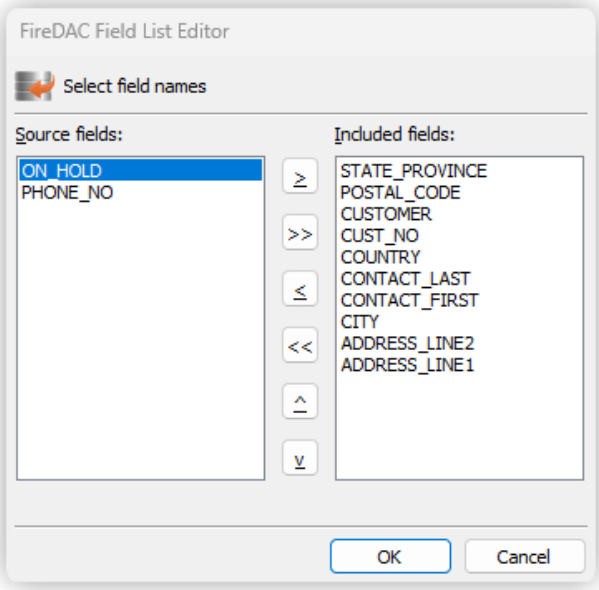
但是"order by"是如何注入到 SQL 中的呢?如果我們開啟 FDQuery 我們將看到下一語句:

```
select * from customer
{IF &SORT} order by &SORT {FI}
```

正如我們所看到的,SQL 語句非常簡單,但該巨集是其工作的關鍵. EMSDatasetResource 使用該巨集能夠在同一查詢中混合分頁和排序.

> □ **備註**
>
> *當使用分頁並且我們到達資料集的末尾時,RAD 伺服器將簡單地傳回一個空數組,讓我們知道該頁面中沒有其他內容.*

另一個非常有用的功能是從資料庫中獲取字段欄位以在我們的邏輯中使用的選項,但我們不想在 API 中公開這些字段欄位. 使用 ValueFields 屬性,我們可以輕鬆選擇要發佈的字段欄位.




要在我們的 API 端點中發布的選定字段欄位

# 04

# REST 除錯器



在第一章中,我們討論了 REST API 生態系統中的可用操作,但到目前為止,我們僅透過瀏覽器使用了 GET. 如果您想知道如何使用 POST、PUT 和 DELETE 操作,在本章中我們將看到 RAD Studio 附帶的一個名為 REST Debugger 的工具,它不僅可以簡化您的測試過程,還可以幫助您更快的開發應用程式.

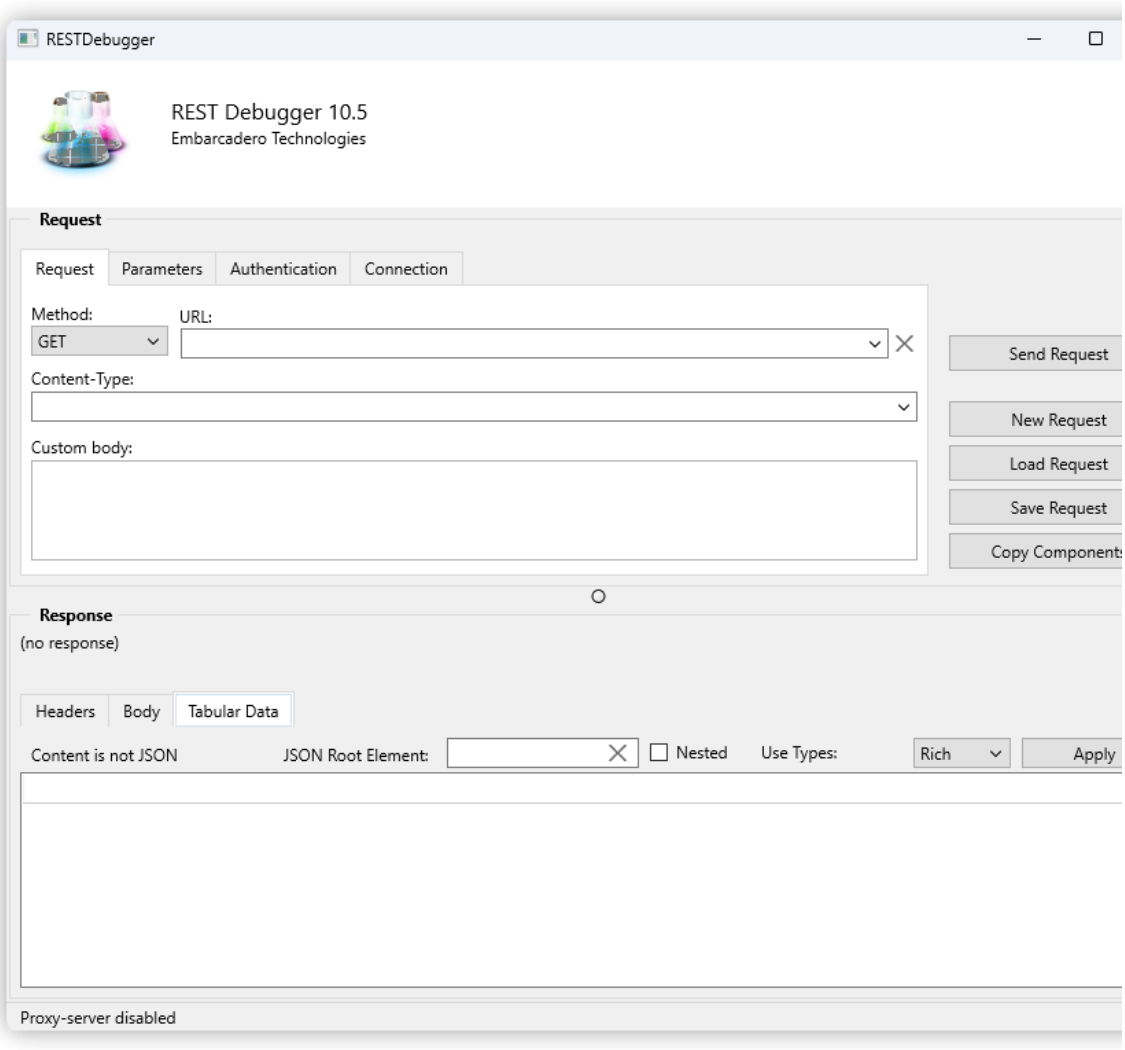
□
竅門

*REST Debugger 並不是只能與 RAD Server 一起使用的產品.您可以使用它來存取任何其他第三方 REST API 服務,並利用其與 RAD Studio 的整合來加快您的開發流程.*

## 什麼是 REST Debugger 以及在哪裡可以找到它

REST Debugger 是 Embarcadero 的免費解決方案,用於探索、理解 RESTful Web 服務並將其與 Delphi 和 C++Builder 應用程式整合. 它使開發人員能夠探索、測試並最終了解 RESTful Web 服務如何與可過濾的 JSON blob、簡化的 OAuth 1.0/2.0 身份驗證和可配置的請求/資源參數等功能一起使用. 不僅如此,您還可以透過幾次點擊將 REST 元件直接複製並貼上到您的專案中.

如果您想嘗試一下,可以在 RAD Studio 的"Tools/REST Debugger"功能表下找到它,或者您也可以免費下載獨立版本點選此連結.




REST Debugger 使用者介面

## 使用 REST Debugger 發送我們的第一個 PUT 請求

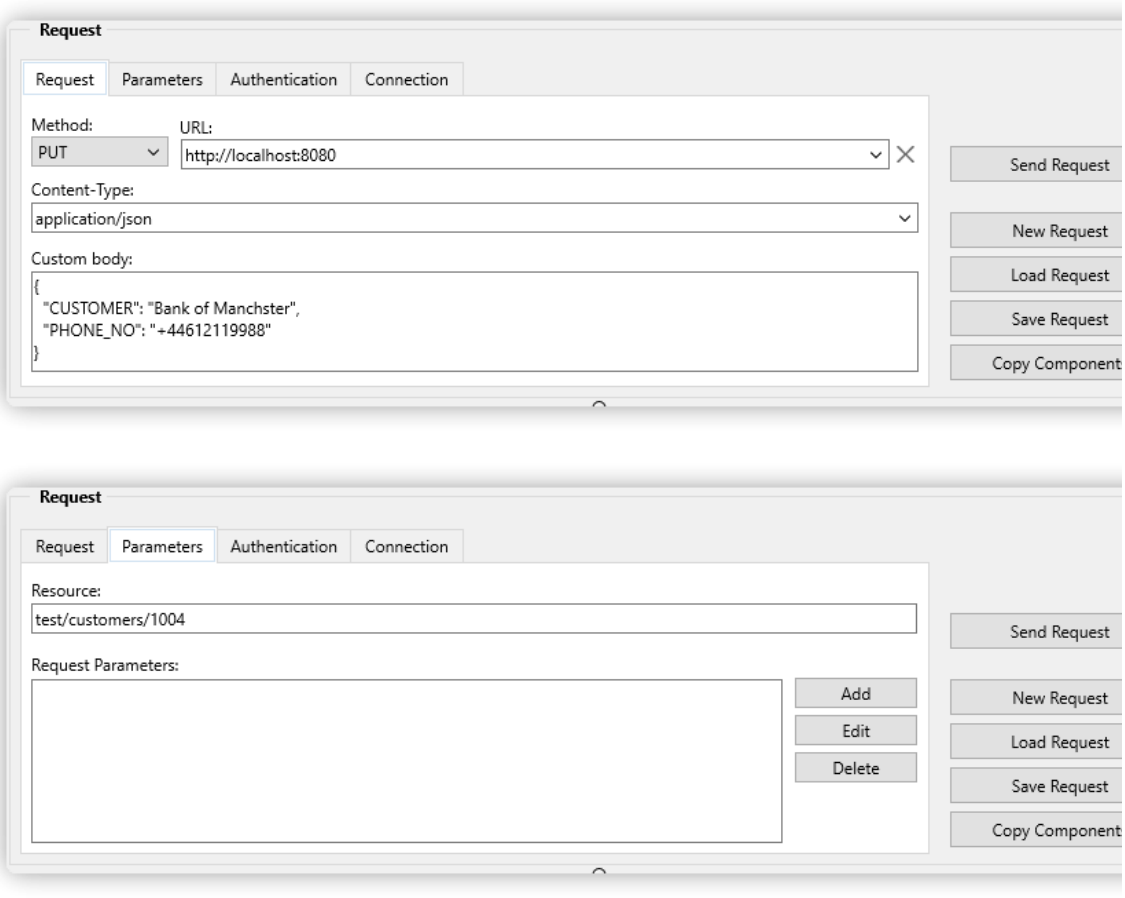
我們可以在左側的下拉清單中看到預設值是 GET,但我們現在可以選擇瀏覽器上無法選擇的其他值.

使用我們在第 3 章中建立的相同項目來修改客戶.

警告

*啟動並執行第 3 章中的 RAD 伺服器專案是必不可少的.否則我們將無法呼叫 API 端點.*

要修改客戶,我們只需使用要修改的 ID 呼叫客戶端點.此外.我們需要在請求正文中指定屬性的新數值.

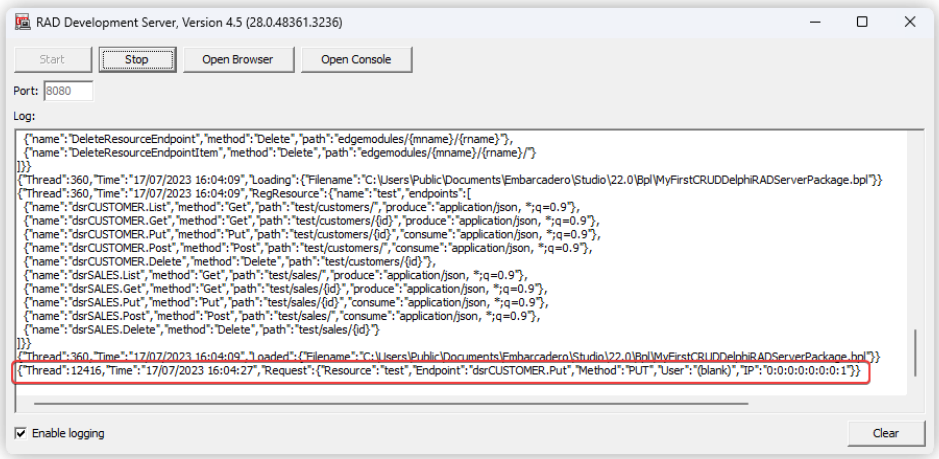



定義發送 PUT 請求所需的數值

為了配置請求,我們定義了 PUT 方法、URL、我們指向的資源(在本例中是 ID 為 1004 的客戶)以及包含我們要更新的屬性的 JSON 正文:客戶的姓名及其電話號碼.

最後一步是點選“Send Request”,如果我們收到 200 HTTP 回應,現在我們可以檢查請求經過的 RAD 伺服器日誌. 此外,如果我們向該特定客戶發送 GET 請求(我們可以在 REST 偵錯器或瀏覽器中執行此操作),我們將看到資料已成功更新.

如果我們想要建立新客戶,流程是相同的,儘管我們需要在正文中提供所有必需的資訊並將方法更改為 POST.





RAD 伺服器日誌，包含註冊的 PUT 請求和修改的數據

## REST Debugger 包含的其他功能

儘管它與 RAD Server 並不完全相關,但值得一提的是 REST Debugger 提供的一個非常強大的功能.定義 URL、參數等後,使用"Copy Components"按鈕在剪貼簿中產生所有必要的 RAD studio 元件,然後將它們貼上到您的任何專案項目中. 透過這種方式,您可以設計更快的 UI 原型來存取 RAD 伺服器或任何其他第三方 API.

在本節的 GitHub 儲存庫中,您將找到一個基本的 FMX 範例.使用"Copy Components"按鈕複製並貼上存取 API 所需的所有元件. 要測試它,您只需先執行 RAD 伺服器應用程式,然後執行 FMX 應用程式並按"Send Request".

REST API 的另一個重要主題是身分驗證. 如果您需要對 API 進行身份驗證，您可以使用"Authentication"標籤下的多種方法, 此外,您還可以使用"Parameters"標籤中的"Add"按鈕在請求中包含特定參數,例如標頭中增加 api-key 參數.

# 05

# 使用 FireDAC 批次移動和 JSONWriter

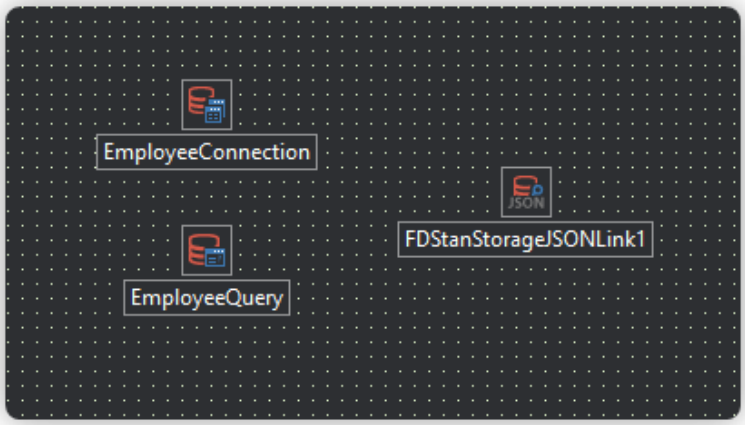


根據您的專案要求或您更熟悉的技術,RAD Studio 允許您使用更多工具來建立 REST API.

FireDAC 元件可用於產生和使用包含資料庫元資料和以 JSON 編碼的資料流,以回應來自 RAD 伺服器端點之一的回應. 如果您的客戶端應用程式將採用 VCL 或 FMX,則此方法非常有用. 您可以使用 MemoryTables 自動映射所有資料庫資訊和元資料. 使用 JavaScript 等語言的其他用戶端應用程式在處理回應中包含的資料庫資訊和資料時可能會出現問題,但 RAD Studio 提供了一種產生 JavaScript 或其他語言期望接收的乾淨 JSON 的方法.

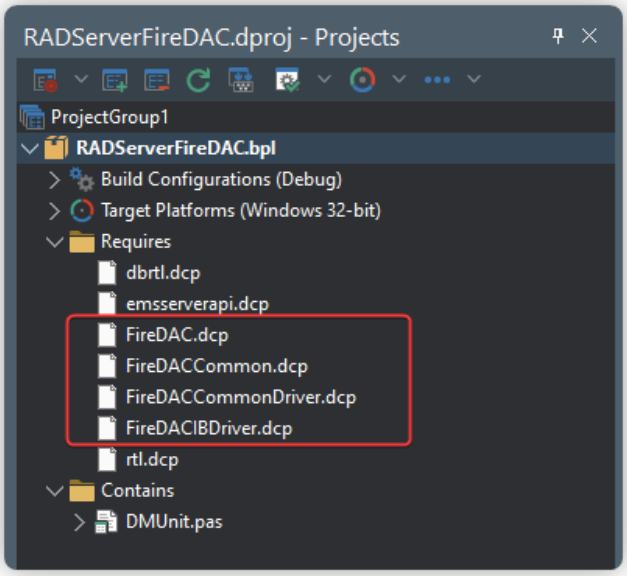
## 使用記憶體流返回 JSON 資料庫數據

FireDAC 包含用於存取資料庫表格資訊並產生 JSON 字串結果的元件.請使用資源模組建立 RAD 伺服器應用程式. 新增 FDConnection 元件並將其與 InterBase 範例 Employee.gdb 資料庫連接.新增 FDQuery 元件並設定 EmployeeQuery SQL 字串為 select * from employee.新增 FDStanStorageJSONLink 元件以方便建立 JSON.

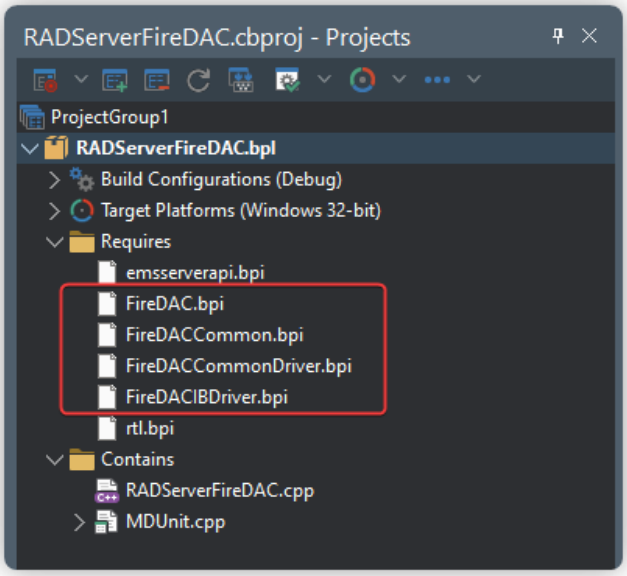



RAD Server 專案的資源模組

建置基於 RAD Server Delphi 的應用程式時,可能會出現一組警告,並會彈出一個對話框以允許應用程式套件與其他已安裝的套件相容. 按一下"OK"按鈕會將所需的套件檔案新增至專案中的"requires"部分.對於 C++Builder,可以手動新增套件(在專案管理器視窗中的 Requires 節點上按一下滑鼠右鍵,然後從彈出式功能表中選擇 Add Reference...).




Delphi RAD 伺服器 FireDAC 項目專案




C++ RAD 伺服器 FireDAC 項目專案

□
竅門

*您可以在每個目標平台的 C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\lib 中找到這些文件.*

以下是 RAD Server Get 方法實現,它使用記憶體流發送帶有員工表資料的 JSON 回應.

**Delphi:**

```
procedure TEmpfiredacResource1.Get(const AContext: TendpointContext;
      const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  mStream: TMemoryStream;
begin
  mStream := TMemoryStream.Create;
  AResponse.Body.SetStream(mStream,'application/json', True);
  EmployeeQuery.Open;
  EmployeeQuery.SaveToStream(mStream, sfJSON);
end;
```

**C++:**

```
void TFireDACResource1::Get(TEndpointContext* Acontext,
      TEndpointRequest* ARequest, TEndpointResponse* AResponse)
{
      TMemoryStream* mStream = new TMemoryStream;
      AResponse->Body->SetStream(mStream,"application/json", True);
      EmployeeQuery->Open();
      EmployeeQuery->SaveToStream(mStream, sfJSON);
}
```

使用瀏覽器和 URL http://localhost:8080/FireDAC 取得包含資料庫員工表的 JSON 資料的回應.JSON 所包含的資訊遠不止於資料.回應中還包括有關資料表、資料表欄位、型態等的元資料信息.

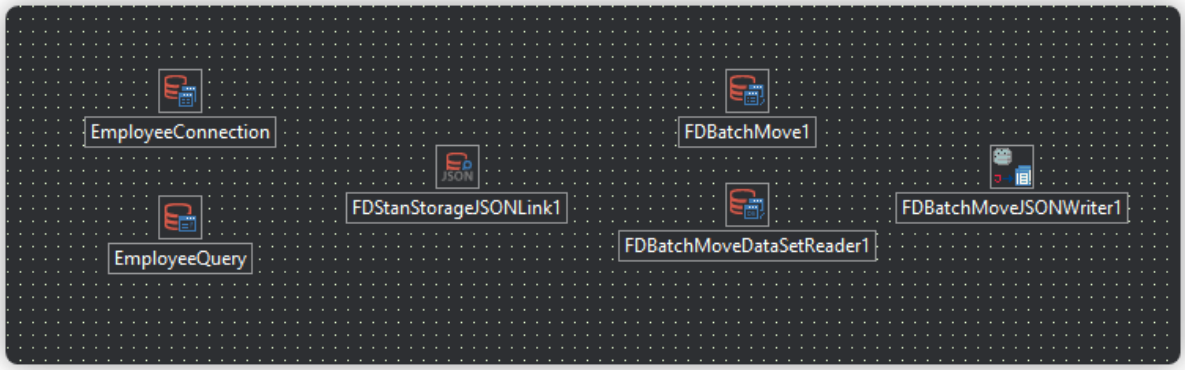

包含 JSON 回應的瀏覽器視窗

這絕對不是其他語言可以在不使用程式碼解析回應的情況下使用的簡單欄位和值的 JSON 數據,但如果您的客戶端要使用 RAD Studio 進行開發,它會變得非常方便.

## 使用 FireDAC 的 BatchMove、BatchMoveDataSetReader 和 BatchMoveJSONWriter

對於複雜的資料庫應用而言,使用上一章和上面提到的方法將需要編寫更多的程式碼. 利用 FireDAC 的 FDBatchMove、FDBatchMoveDataSetReader 和 FDBatchMoveJSONWriter 元件可大幅簡化 JSON 回應的創建.

我們將升級我們創建的同一個項目,並將元件 FDBatchMove、FDBatchMoveDataSetReader 和 FDBatchMoveJSONWriter 加入到資源模組中.




具有 FireDAC Query、BatchMove、DataSetReader 和 JSONWriter 的資源模組

S 將 FDBatchMoveDataSetReader 的 DataSet 屬性設為 EmployeeQuery.

我們將建立一個名為 GetBatchMove 的新端點.

**Delphi:**

```
procedure TEmployeeResource1.GetBatchMove(const AContext: TendpointContext;
      const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
begin
  FDBatchMoveJSONWriter1.JsonWriter := AResponse.Body.JSONWriter;
  FDBatchMove1.Execute;
end;
```

**C++:**

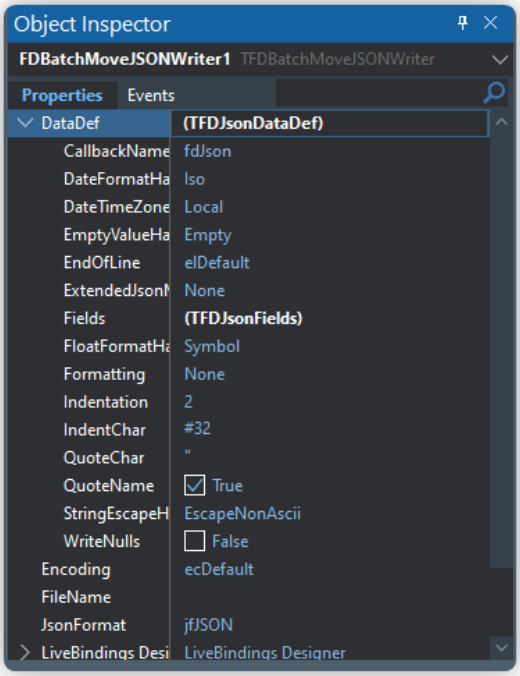
```
void TEmployeeResource1::GetBatchMove(TEndpointContext* Acontext,
      TEndpointRequest* ARequest, TEndpointResponse* AResponse)
{
   FDBatchMoveJSONWriter1->JsonWriter = AResponse->Body->JSONWriter;
   FDBatchMove1->Execute();
}
```

使用 URL http://localhost:8080/BatchMove 呼叫 GET 方法傳回 JSON 資料結果:


使用 BatchMove 提供 JSON 結果的瀏覽器

FDBatchMoveJSONWriter 提供了多個選項來格式化 JSON 結果,涉及 DateFormats、endlines、writeNulls 等.






物件檢視器中的 BatchMoveJSONWriter DataDef 子屬性

FDBatchMove 元件還允許您建立映射以設定來源和目標欄位映射並從目前來源記錄取得資料數值.

Mappings 屬性可以填為:

- 設計或執行時期手動設定映射.這允許指定自訂映射、轉換表達式等.
- 如果映射設定為空,則在執行 Execute 呼叫時自動執行. 這將透過執行來源列和目標匹配的欄位名稱來映射. 如果目標列沒有對應的來源欄位,則目標欄位將從對應中排除,並且在資料移動時不會被填入數據.

## 請參考

- Marco Cantu 部落格:資料集對應到 JSON - RAD 伺服器 Web 服務 Delphi 範例
- FireDAC.Comp.BatchMove.TFDBatchMove
- FireDAC.Comp.BatchMove.JSON.TFDBatchMoveJSONWriter
- Readers 和 Writers JSON 框架
- FireDAC.TFDBatchMove 範例
- RTL.JSONWriter



# 06

# JSONValue, JSONWriter 和 JSONBuilder



RAD Server 支援處理可由不同程式語言和工具使用的 JSON 資料.對於較小的資料量而言,建立 JSON 字串、傳輸字串作為回應,然後讓客戶端應用程式程式碼處理回傳結果是可行的. 但想像一下,對於整個資料庫或複雜的資料結構來說,JSON 數組回應有多大? RAD Studio 提供了三個用於處理 JSON 資料的主要框架.本章介紹 RAD 伺服器應用程式向呼叫應用程式傳回 JSON 的多種方法中的幾種.

## 處理 JSON 資料的框架

RAD Studio 提供了多個框架來處理 JSON 資料.最常見的三種是:

- JSON 物件框架 – 建立臨時物件來讀取和寫入 JSON 數據.
- Readers 和 Writers JSON 框架 – 允許您直接讀寫 JSON 數據.
- SONBuilder – 使用編寫器,以更易於維護的方式建立複雜的結構.

JSON 物件框架需要建立臨時物件來解析或產生 JSON 資料.要讀取或寫入 JSON 數據,您必須在讀取和寫入 JSON 之前建立一個中間記憶體對象,例如 TJSONObject、TJSONArray 或 TJSONString.

Readers 和 Writers JSON 框架允許應用程式直接讀取 JSON 資料並將其寫入串列流,而無需建立臨時物件. 無需建立臨時物件來讀取和寫入 JSON,可提供更好的效能並改善記憶體消耗.

JSON Builder 是前兩個的組合.它的創建是為了使您的程式碼更具可讀性和可維護性.它還遵循一種更現代的方法,您可以將方法一個接一個地連結起來.

在本章的範例專案中,您將發現 3 個不同的端點,它們產生完全相同的回應,但使用這三個可用的框架.您可以在您的專案中隨意使用您覺得更舒服的方式.


在每個端點上獲得相同的 JSON 回應

## 使用 JSONValue

使用 JSON 物件框架透過在程式碼中組裝 JSON 字串來建立它們. JSONValue 是用於定義 JSON 字串、物件、陣列、數字、布林值、true、false 和 null 值的所有 JSON 類別的祖先類別. RAD Studio JSON 實作中包含以下類別和方法:



TJSONObject –實作一個 JSON 物件。.TJSONObject 中的方法包括:

- Parse –解析 JSON 資料流並將遇到的 JSON 對儲存到 TJSONObject 物件的方法.
- ParseJSONValue –解析位元組陣列並從資料建立對應 JSON 值的方法.
- AddPair 方法 –將新的 JSON 對新增至 JSON 物件.
- GetPair 方法–傳回 JSON 物件對清單中具有指定 I 索引鍵值的對,如果指定 I 索引越界,則傳回 nil.
- GetPairByName 方法–從 JSON 物件傳回一個鍵值對,該物件的鍵部分與指定的 PairName 字串匹配,如果沒有與 PairName 匹配的鍵,則傳回 nil.
- SetPairs –定義此 JSON 物件包含的鍵值對列表.
- FindValue –尋找並傳回位於指定 JSON 路徑的 TJSONValue 實例.否則返回 nil.
- GetValue –傳回 JSON 物件中 Name 鍵指定的鍵值對中的值部分,如果沒有與 Name 相符的鍵,則傳回 nil.
- Pairs –存取 JSON 物件對清單中指定 Index 處的鍵值對,如果指定 Index 越界,則傳回 nil.
- GetCount –傳回 JSON 物件的鍵值對數量.

TJSONArray –實作 JSON 陣列. JSONArray 方法包括:

- Add –將透過 Element 參數給定的非空值新增至目前元素列表.

- Get –傳回 JSON 陣列數組中給定索引處的元素.
- Pop –從 JSON 陣列數組中刪除第一個元素.
- Size –傳回 JSON 陣列數組的大小.
- ToBytes –將目前 JSON 陣列數組內容序列化為位元組陣列數組.
- ToString –將目前 JSON 陣列數組序列化為字串並傳回結果字串.

其他 JSON 類別包括:

- TJSONString – 實作 1 個 JSON 字串.
- TJSONNumber –實作 1 個 JSON 數值.
- TJSONBool – JSON 布林值.
- TJSONTrue –實作 1 個 JSON true 數值.
- TJSONFalse –實作 1 個 JSON false 數值.
- TJSONNull –實作 1 個 JSON null 數值.

## 使用 JSON 類別的範例

以下的範例專案的 GetJSON 方法實作了一個 Get 端點,該端點使用多個 JSON 類別來建立、解析和顯示 JSONObjects 和 JSONArray 的結果.



**Delphi:**

```
procedure TTestResource1.GetJSON(const AContext: TEndpointContext; const ARequest:
TEndpointRequest; const AResponse: TEndpointResponse);
begin
  // create some JSON objects
  var JSONRed := TJSONObject.Create;
  JSONRed.AddPair('name', 'red');
  JSONRed.AddPair('hex', '#ff0000');
  JSONRed.AddPair('default', False);
  JSONRed.AddPair('customId', TJSONNull.Create);
  var JSONBlue := TJSONObject.Create;
  JSONBlue.AddPair('name', 'blue');
  JSONBlue.AddPair('hex', '#0000ff');
  JSONBlue.AddPair('default', True);
  JSONBlue.AddPair('customId', 653992);
  // create an array and assign the previous objects to it
  var JSONArray := TJSONArray.Create;
  JSONArray.Add(JSONRed);
  JSONarray.Add(JSONBlue);
  // create an extra object that will contain the array of colors
```

```
  var JSONObject := TJSONObject.Create;
  JSONObject.AddPair('colors', JSONArray);
  AResponse.Body.SetValue(JSONObject, True);
end;
```

**C++:**

```
void TTestResource1::GetJSON(TEndpointContext* AContext, TEndpointRequest* ARequest,
TEndpointResponse* AResponse)
{
  // create some JSON objects
  TJSONObject * JSONRed = new TJSONObject();
  JSONRed->AddPair("color", "red");
  JSONRed->AddPair("hex", "#ff0000");
  JSONRed->AddPair("default", True);
  JSONRed->AddPair("customId", new TJSONNull());
  TJSONObject* JSONBlue = new TJSONObject();
  JSONBlue->AddPair("color", "blue");
  JSONBlue->AddPair("hex", "#0000ff");
  JSONBlue->AddPair("default", False);
  JSONBlue->AddPair("customId", 653992);
  // create an array and assign the previous objects to it
  TJSONArray* JSONArray = new TJSONArray();
  JSONArray->Add(JSONRed);
  JSONArray->Add(JSONBlue);
  // create an extra object that will contain the array of colors
  TJSONObject* JSONObject = new TJSONObject();
  JSONObject->AddPair("colors", JSONArray);
  AResponse->Body->SetValue(JSONObject, True);
}
```

# 使用 JSONWriter

使用 JSONWriter 簡化了 RAD 伺服器應用程式開發,以製作自訂 JSON,為程式語言用戶端提供可供使用的數據. 使用 JSONWriter 啟動 JSON 對象,寫入屬性名稱和值,繼續寫入屬性和值,直到結束 JSON 對象.

## 使用 JSONWriter 的範例

以下是 Get 端點的實現,它使用 JSONWriter 的 WriteStartArray、WriteStartObject、WritePropertyName、WriteValue、WriteEndObject、WriteEndArray 方法傳回數據. AResponse 參數有一個內建的 JSONWriter,它可以非常方便地直接在回應上建立更複雜的結構.

**Delphi:**

```
procedure TTestResource1.GetJSONWriter(const AContext: TEndpointContext; const
ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
begin
  // to avoid typing AResponse.Body.JSONWriter on every line we store it in a variable
  var Writer := AResponse.Body.JSONWriter;
  // start the JSON object
  Writer.WriteStartObject;
  Writer.WritePropertyName('colors');
  // start the JSON Array
  Writer.WriteStartArray;
  Writer.WriteStartObject;
  Writer.WritePropertyName('name');
  Writer.WriteValue('red');
  // add WritePropertyName and WriteValue statements as often as needed
  Writer.WritePropertyName('hex');
  Writer.WriteValue('#ff0000');
  Writer.WritePropertyName('default');
  Writer.WriteValue(False);
  Writer.WritePropertyName('customId');
  Writer.WriteNull;
  Writer.WriteEndObject;
  // write as many additional JSON objects as you need
  Writer.WriteStartObject;
  Writer.WritePropertyName('name');
  Writer.WriteValue('blue');
  Writer.WritePropertyName('hex');
  Writer.WriteValue('#0000ff');
  Writer.WritePropertyName('default');
  Writer.WriteValue(True);
  Writer.WritePropertyName('customId');
  Writer.WriteValue(653992);
  // end the JSON object
  Writer.WriteEndObject;
  // end the JSON array
  Writer.WriteEndArray;
  Writer.WriteEndObject;
end;
```

**C++:**

```
void TTestResource1::GetJSONWriter(TEndpointContext* AContext, TEndpointRequest*
ARequest, TEndpointResponse* AResponse)
{
```

```
  // to avoid typing AResponse.Body.JSONWriter on every line we store it in a variable
  TJsonTextWriter* Writer = AResponse->Body->JSONWriter;
  // start the JSON object
  Writer->WriteStartObject();
  Writer->WritePropertyName("colors");
  // start the JSON Array
  Writer->WriteStartArray();
  Writer->WriteStartObject();
  Writer->WritePropertyName("name");
  Writer->WriteValue("red");
  // add WritePropertyName and WriteValue statements as often as needed
  Writer->WritePropertyName("hex");
  Writer->WriteValue("#ff0000");
  Writer->WritePropertyName("default");
  Writer->WriteValue(False);
  Writer->WritePropertyName("customId");
  Writer->WriteNull();
  Writer->WriteEndObject();
  // write as many additional JSON objects as you need
  Writer->WriteStartObject();
  Writer->WritePropertyName("name");
  Writer->WriteValue("blue");
  Writer->WritePropertyName("hex");
  Writer->WriteValue("#0000ff");
  Writer->WritePropertyName("default");
  Writer->WriteValue(True);
  Writer->WritePropertyName("customId");
  Writer->WriteValue(653992);
  // end the JSON object
  Writer->WriteEndObject();
  // end the JSON array
  Writer->WriteEndArray();
  Writer->WriteEndObject();
}
```

## 使用 JSONBuilder

該框架是一個 JSONWriter 包裝器,可讓您以更快、更易讀的方式建立 JSON. 它遵循流暢的介面(也稱為方法連結)方法,在非常複雜的 JSON 結構的情況下,可以簡化程式碼並使其更易於維護和閱讀.

在同一專案範例中,您將發現另一個使用 JSON 產生器來建立回應的端點.我們來看看程式碼:

**Delphi:**

```
procedure TTestResource1.GetJSONBuilder(const AContext: TEndpointContext; const
```

```
ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
begin
  var Writer := Aresponse.Body.JSONWriter;
  // link the JSONWriter from the response to the builder
  var Builder := TJSONObjectBuilder.Create(Writer);
  try
    Builder
      .BeginObject
        .BeginArray('colors')
          .BeginObject
            .Add('name', 'red')
            .Add('hex', '#ff0000')
            .Add('default', False)
            .AddNull('customId')
          .EndObject
          .BeginObject
            .Add('name', 'blue')
            .Add('hex', '#0000ff')
            .Add('default', True)
            .Add('customId', 653992)
          .EndObject
      .EndArray
    .EndObject;
  finally
    Builder.Free;
  end;
end;
```

**C++:**

```
void TTestResource1::GetJSONBuilder(TEndpointContext* AContext, TEndpointRequest*
ARequest, TEndpointResponse* AResponse)
{
  TJsonWriter* Writer = AResponse->Body->JSONWriter;
  // link the JSONWriter from the response to the builder
  TJSONObjectBuilder* Builder = new TJSONObjectBuilder(Writer);
  try {
  Builder
       ->BeginObject()
         ->BeginArray("colors")
              ->BeginObject()
                ->Add("name", "red")
                ->Add("hex", "#ff0000")
                ->Add("default", False)
                ->AddNull("customId")
```

```
            ->EndObject()
            ->BeginObject()
              ->Add("name", "blue")
              ->Add("hex", "#0000ff")
              ->Add("default", True)
              ->Add("customId", 653992)
            ->EndObject()
        ->EndArray()
      ->EndObject();
  } __finally {
      delete Builder;
  }
}
```

您可以找到 RAD Studio 提供的一個非常有用的範例專案項目,稱為 fmWorkBench(儘管它僅適用於 Delphi).您可以在下面路徑中找到它：

C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\Samples\Object Pascal\RTL\Json

## 請參考

- JSON
- Readers 和 Writers JSON 框架
- JSONBuilder
- WorkBench 範例專案
- 教學：使用 REST 用戶端函式庫存取基於 REST 的 Web 服務



# 07

# 建立您自己的自訂端點



在本章之前,我們已經了解了基本的 JSON 結構:陣列數組、物件…還有相當簡單的 URI：/customers、/sales…但在 REST API 最佳實踐中查找子資源 URI 和嵌套數組/是很常見的. 在本章中,我們將討論如何使用 RAD Server 完成這些類型的結構以及建立您自己的 GET、POST、PUT 或 DELETE 方法.

## 良好做法的範例

儘管您可以按照自己想要的方式建立 API,但仍有數千篇文章討論標準化或 REST API 的最佳實踐. A 歸根結底,這取決於您想要如何建立 API,但了解這些標準的基礎知識,並盡量不要重新發明輪子, 是值得閱讀和思考的.

例如:當我們存取特定客戶數據時,我們了解到我們使用 URI /customers/{id} 但如果我們想存取該特定客戶的銷售情況該怎麼辦? 一個非常常見的選項是定義另一個端點,例如：/customers/{id}/sales.該端點將傳回由其 ID 定義的特定客戶的銷售訂單.

這被認為是良好實踐,通常稱為嵌套資源或子資源. 這定義了端點之間的層次關係,可以幫助第三方或您自己的開發團隊以更簡單的方式理解您的 API.

## 避免 API 過於囉嗦

開發應用程式時,訪問表單/網頁並發現自己需要來自多個端點的資料是很常見的. 假設您存取客戶的銷售訂單:您可能需要客戶的詳細資料、訂單資訊、送貨地址、訂單行,可能還需要付款、發票…請求清單可能會很長,當涉及 REST API 時,會出現一個問題:請求的成本很高. 我的意思並不是說伺服器處理所有這些請求的成本很高,而且互聯網給這個方程式帶來的延遲也很昂貴. 每個請求都需要幾毫秒來回,如果我們需要

發送 10 個甚至更多請求來訪問一個頁面,那麼這裡還有很大的改進空間.這就是嵌套 JSON 回應更有意義的時候.

想像一下,您可以在一次請求中向 RAD 伺服器請求一個客戶及其所有銷售額的摘要.這可能相當於一種經典的主從關係,但全部在一個請求中返回.我們也將了解如何實現這一目標.

## 新增子資源

對於子資源,我們仍然可以使用我們在前面的章節中看到的相同的 TEMSDatasetAdapter.我們只需要調整屬性中的一些內容,RAD Studio 和 FireDAC 將為我們完成剩下的工作.

使用我們迄今為止使用過的相同項目(有 2 個查詢：qryCUSOTMER、qrySALES),讓我們像這樣修改 qrySALES 的 SQL:

```
select * from SALES
where CUST_NO = :CUST_NO
{if !SORT}order by !SORT{fi}
```

還有 drsSALES EMSDatasetAdapter 之上的屬性,讓我們改變這些屬性:

**Delphi**

```
[ResourceSuffix('customers/{CUST_NO}/sales')]
[ResourceSuffix('List', './')]
[ResourceSuffix('Get', './{PO_NUMBER}')]
[ResourceSuffix('Post', './')]
[ResourceSuffix('Put', './{PO_NUMBER}')]
[ResourceSuffix('Delete', './{PO_NUMBER}')]
dsrSALES: TEMSDataSetResource;
```

**C++:**

```
attributes->ResourceSuffix["dsrSALES"] = "customers/{CUST_NO}/sales";
attributes->ResourceSuffix["dsrSALES.List"] = "./";
attributes->ResourceSuffix["dsrSALES.Get"] = "./{PO_NUMBER}";
attributes->ResourceSuffix["dsrSALES.Post"] = "./";
attributes->ResourceSuffix["dsrSALES.Put"] = "./{PO_NUMBER}";
attributes->ResourceSuffix["dsrSALES.Delete"] = "./{PO_NUMBER}";
```

在 SQL 語句中,我們剛剛新增了一個 WHERE 子句,其中包含一個過濾特定客戶銷售的參數.

如果我們檢查屬性,就會發生更有趣的事情. RAD 伺服器會自動將 {CUST_NO} 中的值注入 FireDAC 查詢中,並過濾該客戶的銷售額. 另外,我們需要指定其餘的方法(List、Get、Post 等),因為現在相同的端點涉及 2 個鍵值,並且必須指定它們的名稱才能使其工作. 好消息是,我們仍然可以使用這些端點來建立、修改或刪除特定銷售,就像我們對使用 EMSDatasetAdapter 建立的任何其他端點所做的那樣.

## 在回應中新增嵌套資料(主/從詳細資料)

現在我們有了第一個子資源端點,讓我們建立一個具有多個值的巢狀回應.為此,我們最後需要寫一些程式碼.

讓我們在 TTestResource1 資料模組類別中建立一個已發佈的方法.

**Delphi:**

```
  published
    [ResourceSuffix('./customers-details/{CUST_NO}')]
    procedure GetCustomerDetails(const AContext: TEndpointContext; const ARequest:
TEndpointRequest; const AResponse: TEndpointResponse);

// and it's implementation

procedure TTestResource1.GetCustomerDetails(const AContext: TEndpointContext; const
ARequest: TEndpointRequest;
  const AResponse: TEndpointResponse);
begin
  var lCustomerNo := ARequest.Params.Values['CUST_NO'].ToInteger;
  // We use a parameter instead of concatenating the CustomerNo to avoid SQL injection
  qryCUSTOMER.MacroByName('MacroWhere').AsRaw := 'WHERE CUST_NO = :CUST_NO';
  qryCUSTOMER.ParamByName('CUST_NO').AsInteger := lCustomerNo;
  qryCUSTOMER.Open;
  try
    if qryCUSTOMER.RecordCount = 0 then
      AResponse.RaiseNotFound('Not found', 'Customer ID not found');

    qrySALES.ParamByName('CUST_NO').asInteger := lCustomerNo;
    qrySALES.Open;
    var lFields := ExcludeMasterFieldFromFields(qrySALES);
    try
      AResponse.Body.SetValue(
        SerializeMasterDetail(qryCUSTOMER, qrySALES, 'SALES', lFields)
        , True);
      qrySALES.Close;
    finally
      lFields.Free;
    end;
  finally
    qryCUSTOMER.Close;
    qryCUSTOMER.MacroByName('MacroWhere').Clear;
  end;
end;
```

**C++:**

```
attributes->ResourceSuffix["GetCustomerDetails"] = "./customers-details/{CUST_NO}";

// and it's implementation

void TTestResource1::GetCustomerDetails(TEndpointContext* AContext, TEndpointRequest*
ARequest, TEndpointResponse* AResponse)
{
	int lCustomerNo = ARequest->Params->Values["CUST_NO"].ToInt();
    // We use a parameter instead of concatenating the CustomerNo to avoid SQL
injection
	qryCUSTOMER->MacroByName("MacroWhere")->AsRaw = "WHERE CUST_NO = :CUST_NO";
	qryCUSTOMER->ParamByName("CUST_NO")->AsInteger = lCustomerNo;
	qryCUSTOMER->Open();
	try {
		if (qryCUSTOMER->RecordCount == 0) {
			AResponse->RaiseNotFound("Not found", "Customer ID not found");
		}
		qrySALES->ParamByName("CUST_NO")->AsInteger = lCustomerNo;
		qrySALES->Open();

		TStringList* lFields = ExcludeMasterFieldFromFields(qrySALES);
		try {
			AResponse->Body->SetValue(
				SerializeMasterDetail(qryCUSTOMER, qrySALES, "SALES",
lFields),
				true
			);
		} __finally {
			lFields->Free();
		}
	} __finally {
		qryCUSTOMER->Close();
		qryCUSTOMER->MacroByName("MacroWhere")->Clear();
    }
}
```

我們可以在這段簡單的程式碼中看到,我們只是使用巨集檢索從 URL 獲取的特定客戶 ID 的詳細信息,並將其作為參數傳遞給我們已經使用的相同 qrySALES. 如此做就無需額外的查詢.

儘管使用像 EMSDatasetAdapter 這樣的元件對低程式碼方法有很大幫助,但有時我們需要特定的要求,並且需要編寫自己的實作程式碼. 正如我們在前面的章節中所看到的,我們可以使用 JSONWritters 來自訂我們的回應.

如果我們檢查此範例的程式碼,對於每個傳入請求,我們都可以存取 ARequest 和 AResponse 參數來存取所有必需的資訊並建立我們自己的回應. 在此範例中,我們使用"WriteStartObject"方法建立一個新物件,然後使用 TQuerySerializer 類別中的兩個方法(稍後將詳細介紹),然後結束該物件.

我們可以將多種有用的方法和屬性與 JSONWriters 和 JSONReaders t 一起使用,這將使您的編碼體驗變得更加輕鬆. 我強烈建議您查看文件以了解所有可用功能.

現在你可能會問自己: 但這兩種方法是什麼: **ExcludeMasterFieldFromFields** 和 **SerializeMasterDetail**? 這兩種方法已經針對這個特定的演示範例進行了編碼,但是您可以在 GitHub 儲存庫演示中取得它們,當然,也可以在您的專案中隨意使用這些程式碼. 他們的任務在方法中有很好的說明,但總而言之,他們只是將主/詳細關係轉換為 JSON 對象,並將詳細查詢作為 JSON 陣列數組插入到主 JSON 對像中. **ExcludeMasterFieldFromFields** 可能不是必需的,但為了避免冗餘數據,我們從詳細資訊中排除了 MasterField.

□
竅門

*您可查看 Data.DBJson 單元,它包含多個類別來幫助您將資料集轉換為 JSON,反之亦然. 在此範例中,我們使用了 TDataSetToJSONBridge 類別,它允許我們更快,更精細地序列化這些內容.*

**Delphi:**

```
// given 2 queries with a master/detail relationship, returns 1 JSON object with a
nested array with the detail query
function TTestResource1.SerializeMasterDetail(AMasterDataset: TFDQuery;
ADetailDataset: TFDQuery; APropertyName: string; AFields: TStringList = nil):
TJSONObject;
begin
  var lBridge := TDataSetToJSONBridge.Create;
  try
    // takes the current record of the master query and converts it to a JSON object
    lBridge.Dataset := AMasterDataset;
    lBridge.IncludeNulls := True;
    // specifies that the we only require to process the current record
    lBridge.Area := TJSONDataSetArea.Current;
    // adds the master record as an object in the JSON result
    Result := TJSONObject(lBridge.Produce);

    // in case we passed a list of fields we want to export we assign them to the
bridge, otherwise the default behaviour is exporting all fields in the query
    if Assigned(AFields) then
      lBridge.FieldNames.Assign(AFields);
    // the same bridge is being reused, but now the detail dataset is being assigned
    lBridge.Dataset := ADetailDataset;
    // in this case all the records from the query will be processed
    lBridge.Area := TJSONDataSetArea.All;
    // stores the detail array in a temp array to add it afterwards in the main object
    var lJSONarray := TJSONArray(lBridge.Produce);
    // the array is being added to the main object as an array with the propertyname
passed in the argument
    Result.AddPair(APropertyName, lJSONarray);
  finally
```

```
    lBridge.Free;
  end;
end;

// if a query has a masterfield assigned, it retuns a stringlist with all the fields
but that masterfield
function TTestResource1.ExcludeMasterFieldFromFields(ADataset: TFDQuery): TStringList;
begin
  var lMasterField := ADataset.MasterFields;
  Result := TStringList.Create;
  Result.Assign(ADataset.FieldList);
  var i := Result.IndexOf(lMasterField);
  if i > -1 then
    Result.Delete(i);
end;
```

**C++:**

```
// given 2 queries with a master/detail relationship, returns 1 JSON object with a
nested array with the detail query
TJSONObject* TTestResource1::SerializeMasterDetail(TFDQuery* AMasterDataset, TFDQuery*
ADetailDataset, System::UnicodeString APropertyName, TStringList* AFields)
{
	TDataSetToJSONBridge *lBridge = new TDataSetToJSONBridge;
	try {
		// takes the current record of the master query and converts it to a JSON
object
		lBridge->Dataset = AMasterDataset;
		lBridge->IncludeNulls = True;
		// specifies that the we only require to process the current record
		lBridge->Area = TJSONDataSetArea::Current;
		TJSONObject* lJSONObject = new TJSONObject;
		// adds the master record as an object in the JSON result
		lJSONObject = (TJSONObject*) lBridge->Produce();

		// in case we passed a list of fields we want to export we assign them to
the bridge, otherwise the default behaviour is exporting all fields in the query
		if (AFields != NULL) {
			lBridge->FieldNames->Assign(AFields);
		}
	  // the same bridge is being reused, but now the detail dataset is being
assigned
		lBridge->Dataset = ADetailDataset;
	  // in this case all the records from the query will be processed
		lBridge->Area = TJSONDataSetArea::All;
```

```
            TJSONArray* lJSONArray = new TJSONArray;
            // stores the detail array in a temp array to add it afterwards in the
main object
            lJSONArray = (TJSONArray*) lBridge->Produce();
            // the array is being added to the main object as an array with the
propertyname passed in the argument
            lJSONObject->AddPair(APropertyName, lJSONArray);
            return lJSONObject;
      } __finally {
            lBridge->Free();
      }
}

// if a query has a masterfield assigned, it retuns a stringlist with all the fields
but that masterfield
TStringList* TTestResource1::ExcludeMasterFieldFromFields(TFDQuery* ADataset)
{
      System::UnicodeString lMasterField = ADataset->MasterFields;
      TStringList* fields = new TStringList;
      fields->Assign(ADataset->FieldList);
      int i = fields->IndexOf(lMasterField);
      if (i > -1) {
            fields->Delete(i);
      }
      return fields;
}
```

□ 備註

*在實際專案中,將這些方法抽像到另一個類別/單元中會更有意義,因為它們可以輕鬆重複使用,但為了簡單起見,我們將它們保留在同一個 DataModule 中.*

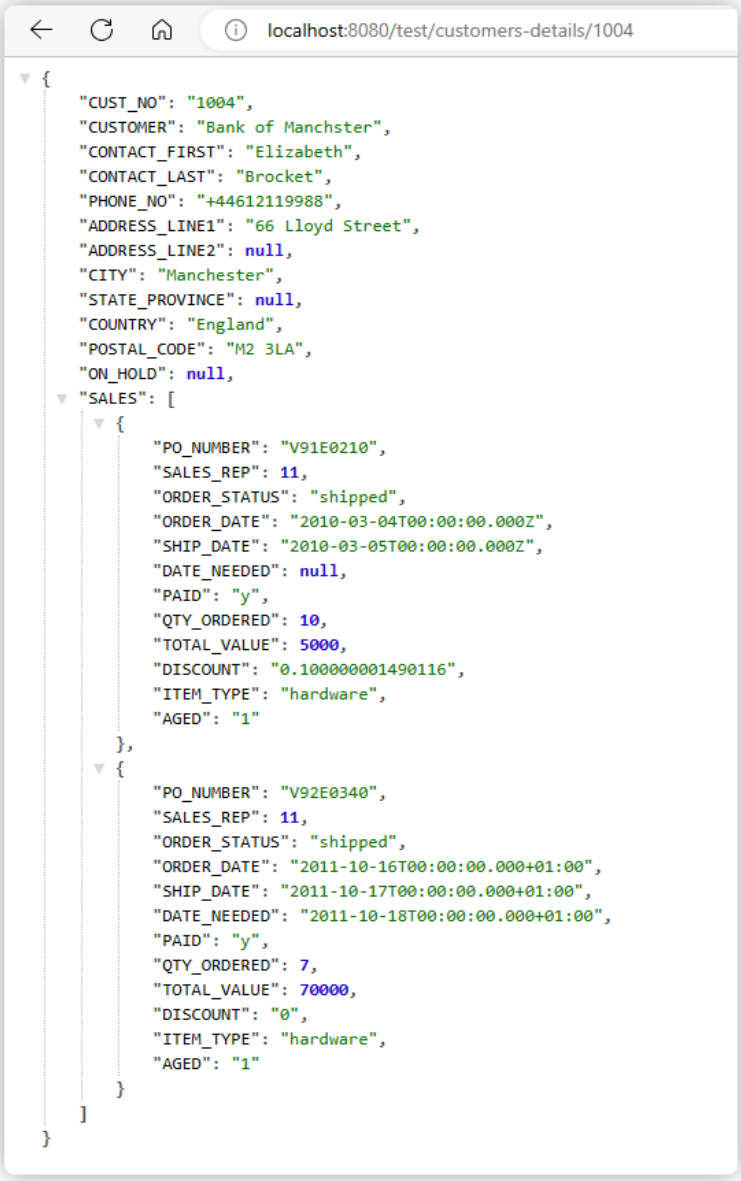
## 測試新的實作

讓我們執行此範例專案項目並存取 URL http://localhost:8080/test/customers/1004/sales/


使用子資源方法存取特定客戶的銷售情況

我們現在正在過濾客戶 1004 的銷售,但是如果我們想要存取/修改/刪除特定的一項銷售,我們也可以透過這個相同的 URI 進行存取.我們只需要在末尾添加訂單 Id 即可,例如: http://localhost:8080/test/customers/1004/sales/V91E0210

現在讓我們存取我們定義的另一個端點: http://localhost:8080/test/customers-details/1004




使用子資源(客戶及其銷售)存取端點

在同一個請求中,我們獲得了特定客戶的所有銷售額,這意味著我們只需一次即可獲得所需的所有信息,而不是兩次呼叫 RAD 伺服器. 顯然,透過多個嵌套層級等,此類請求可能會變得更加複雜.

□ 備註

*在與本章相關的 github 儲存庫示範專案中,您將找到一個額外的端點來存取客戶清單及其相關銷售. 我們不會過濾特定的一個,而是取得所有的資料. 請注意,大部分程式碼已被重複使用,這使得在進一步的開發中讓實作非常簡單.*

## 建立自訂 GET、POST、PUT、DELETE 方法

到目前為止,我們已經了解如何創建 GET 自訂方法,但在某些情況下,我們需要使用其他動詞,例如 POST、PUT 和 DELETE. 要編寫這種實作程式碼,我們只需要用我們想要使用的動詞來命名方法名稱, 例如: **“procedure PutMethodName(..”**. 正如您可能在前面的範例中看到的那樣,所有自訂的方法都以"Get"開頭:如果我們將其更改為"Post", 我們將定義一個 POST 方法.讓我們來看一個例子:

**Delphi:**

```
  published
    [ResourceSuffix('./custom/{ID}')]
    procedure PostCustomEndPoint(const AContext: TEndpointContext; const ARequest:
TEndpointRequest;
      const AResponse: TEndpointResponse);

// and it's implementation

procedure TTestResource1.PostCustomEndPoint(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  lId: integer;
  lName: string;
  lJSON: TJSONObject;
begin
  if not(ARequest.Body.TryGetObject(lJSON) and lJSON.TryGetValue<string>('name',
lName)) then
    AResponse.RaiseBadRequest('Bad request', 'Missing data');
  lID := ARequest.Params.Values['ID'].ToInteger;
  // Add your extra business logic
  lName := 'The name is ' + lName;
  AResponse.Body.JSONWriter.WriteStartObject;
  AResponse.Body.JSONWriter.WritePropertyName('id');
  AResponse.Body.JSONWriter.WriteValue(lId);
  AResponse.Body.JSONWriter.WritePropertyName('name');
  AResponse.Body.JSONWriter.WriteValue(lName);
  AResponse.Body.JSONWriter.WriteEndObject;
end;
```

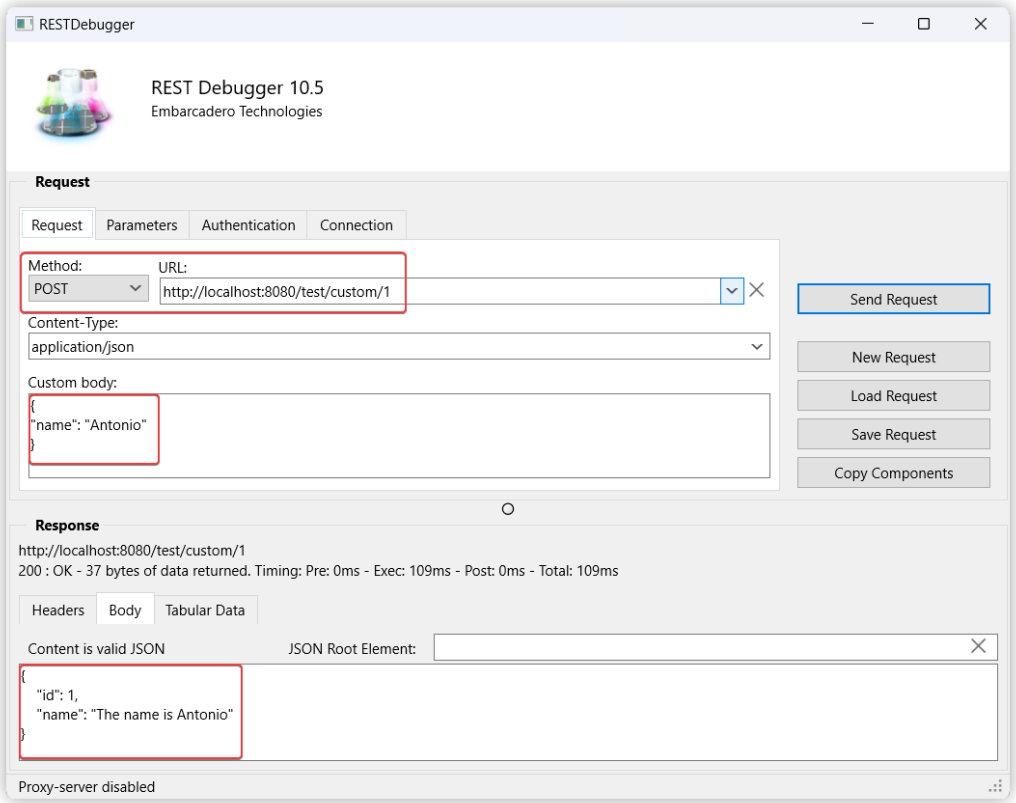
**C++:**

```
attributes->ResourceSuffix["PostCustomEndPoint"] = "./custom/{ID}";

// and it's implementation

void TTestResource1::PostCustomEndPoint(TEndpointContext* AContext, TEndpointRequest*
```

```
ARequest, TEndpointResponse* AResponse)
{
       TJSONObject *lJSON;
       System::UnicodeString lName;
       if (!ARequest->Body->TryGetObject(lJSON) && lJSON->TryGetValue("name",lName)) {
         AResponse->RaiseBadRequest("Bad Request", "Missing Data");
    }
       int lID = ARequest->Params->Values["ID"].ToInt();
       // Add your extra business logic
       lName = "The name is " & lName;
       AResponse->Body->JSONWriter->WriteStartObject();
       AResponse->Body->JSONWriter->WritePropertyName("id");
       AResponse->Body->JSONWriter->WriteValue(lID);
       AResponse->Body->JSONWriter->WritePropertyName("name");
       AResponse->Body->JSONWriter->WriteValue(lName);
       AResponse->Body->JSONWriter->WriteEndObject();
}
```

現在我們有了一個新的額外端點 ./custom/{id}.如果我們從 REST 偵錯器發送 POST 請求, 並在正文中添加預期的"name"屬性,我們將得到如下的結果.



來自自訂 POST 方法的回應

## 處理回應錯誤

正如我們在前面的自訂 POST 端點範例中所看到的,如果我們沒有獲得預期的所有數據,我們會觸發錯誤. RAD 伺服器提供了一個內建解決方案, 可以直接從 AResponse 物件觸發並傳回最常見的錯誤. 您可以在以下位置找到有關可用錯誤的更多詳細資訊此連結.

## 請參考

- JSON Writers 和 Readers
- REST API 最佳實踐



# 08

# 存取內建的分析功能



RAD 伺服器控制台是一項提供預先配置 Web 應用程式的服務,該應用程式顯示來自 RAD 伺服器引擎的多個資料以及分析數據. 它允許您更深入地了解 RAD 伺服器實例上的活動,並根據真實資料做出決策.分析使用者、API 和服務活動,深入了解應用程式的使用情況.

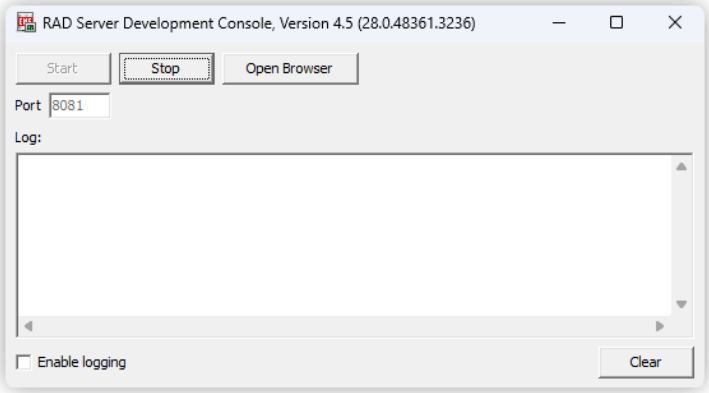
## 主要特點

RAD 伺服器控制台以唯讀模式存取資料庫伺服器.

- 它透過 RAD 伺服器引擎資源的統計資料提供 API 呼叫的回饋:使用者、群組、安裝、模組及其資源.
- 您可以將控制台作為獨立應用程式用於測試目的,或在 Microsoft IIS 伺服器上設定控制台以用於生產環境.
- 注意:Microsoft IIS Server 在 Linux 上不可用.您可以於 Linux 的生產環境中使用 Apache
- RAD 伺服器控制台透過擴充伺服器功能提供新資源分析.
- RAD 伺服器控制台為註冊的 RAD 伺服器使用者提供分析數據.
- 您可以將分析資料匯出並儲存到系統中的.csv 檔案中.

## 存取 RAD 伺服器控制台

返回 RAD Server 開發伺服器並點選 Open Console 按鈕. 這將啟動在連接埠 8081 上自動執行的 RAD 伺服器開發控制台伺服器,並且還將開啟具有分析控制台登入視窗的瀏覽器.

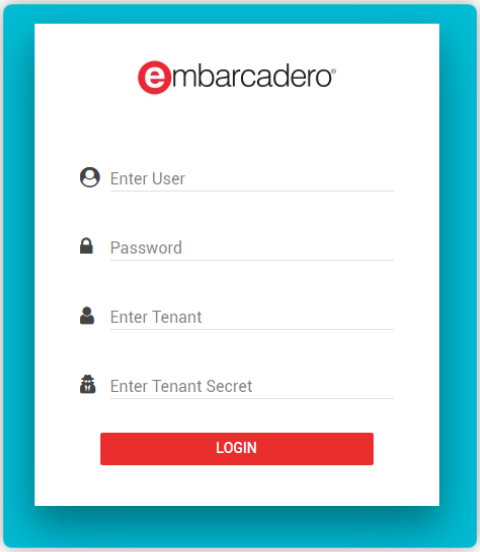



RAD 伺服器開發控制台伺服器 UI

備註

*如果您的電腦已經使用了預設端口 8081,您只需將 8081 變更為電腦上可用的任何端口,按"開始"按鈕,然後"開啟瀏覽器".*



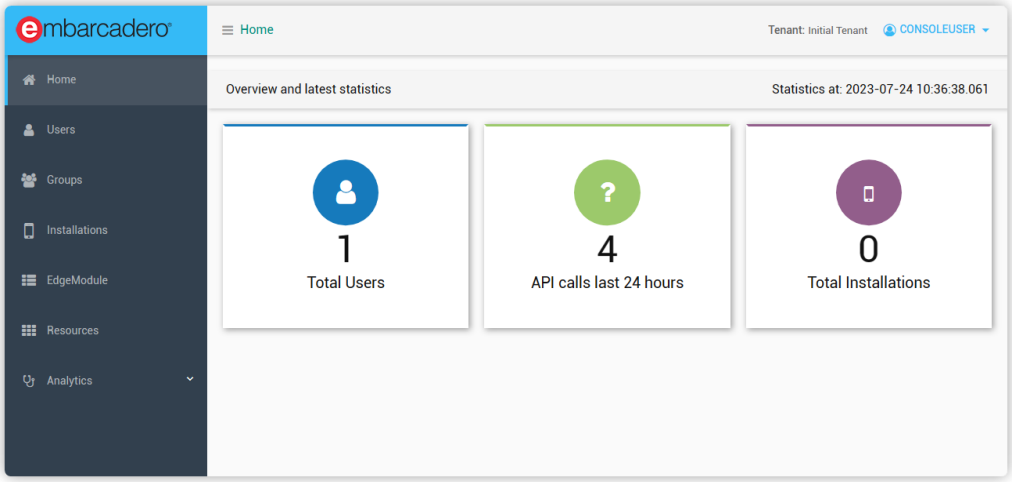



RAD 伺服器控制台使用者登入畫面

若要存取控制台,RAD 伺服器附帶了預先設定的預設憑證(將 tenant 資訊留空):
user: **consoleuser**
password: **consolepass**

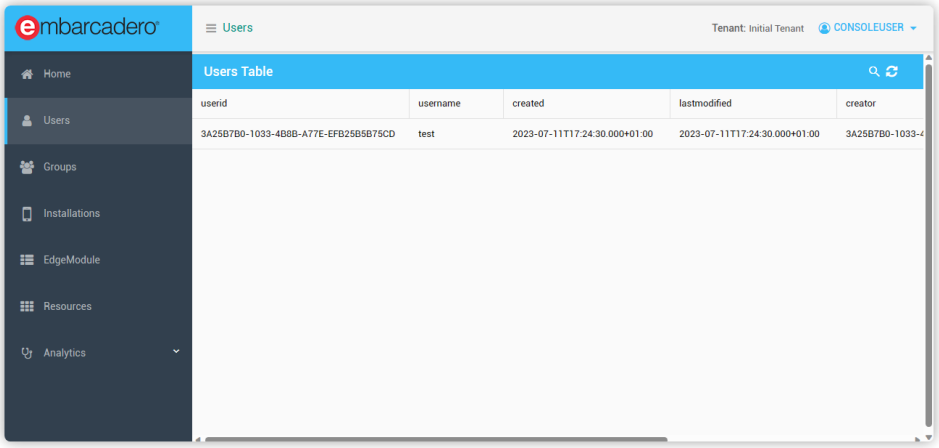
警告

*RAD 伺服器提供預設使用者和密碼來存取控制台. R 請記住在 emsserver.ini 設定檔中更改這些憑證(查看有關此設定檔的章節以取得更多詳細資訊).*




RAD 伺服器控制台首頁

登入後,您將看到 RAD 伺服器控制台單頁 JavaScript 應用程式的圖形視圖,左側是選單,右側是內容. 選單提供了如下凡資訊: users, groups, device installations, EdgeModules, Resource Modules 和 Analytics. 這是顯示使用者清單及其資訊的螢幕,包括使用者的建立時間和上次修改使用者資訊的時間.



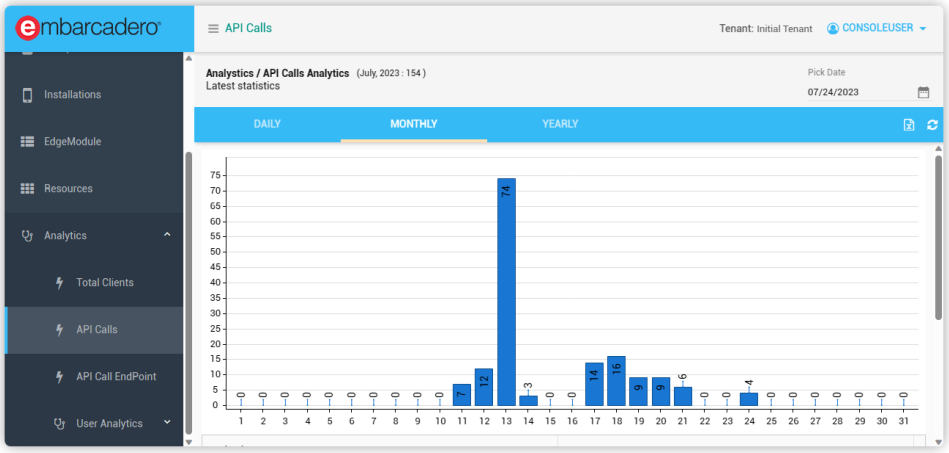

RAD 伺服器控制台使用者列表

點選"Analytics"選單項目會開啟一個選單,可從一系列分析中進行選擇,包括客戶端總數、API 呼叫、呼叫的 API 端點等. 分析數據可依日、月、年選擇分析. 分析還可以按使用者,群組等和特定端點進行過濾.分析結果還可以保存到 .CSV 檔案中,以便外部應用程式進行額外處理.

□
竅門

*這些分析數據為決策過程和審計提供了重要訊息. 查看您的服務的使用量可提供您用於規劃更新,或查看哪些端點很少使用,這些都是非常有價值的見解的範例.*

下圖顯示了選定月份的 API 呼叫圖表.




RAD 伺服器控制台 Ext JS API 呼叫分析頁面

# 09

# 部署 RAD 伺服器



在此之前,第一個 RAD 伺服器應用程式是使用 RAD 伺服器 (EMSDevServer.exe) 和控制台 (EMSDevConsole.exe) 應用程式的開發版本進行測試的. 本章介紹了您可以在生產環境中部署 RAD 伺服器的多個平台.如果您對 RAD Server Lite 有興趣,請跳至下一章.

警告

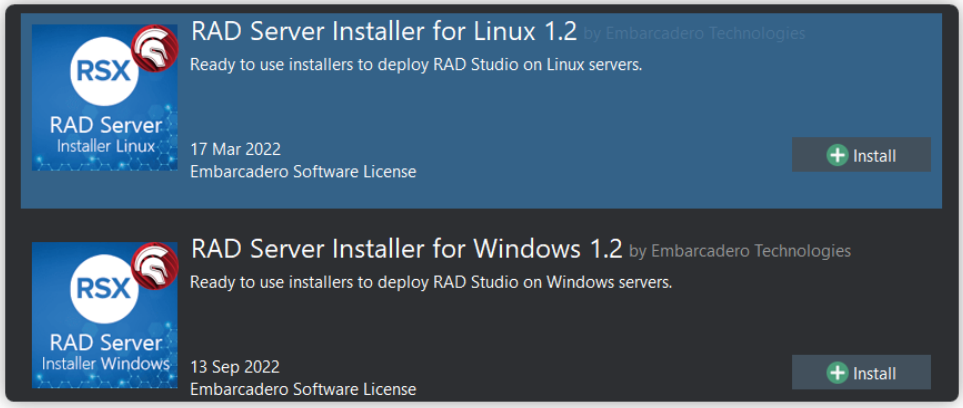
*編譯新的 bpl 或 dcp 資源時,它不會包含在專案的"export"資料夾中(二進位檔案通常所在的位置).這些資源將預設建立在您的 Embarcadero Studio 安裝路徑中: C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\"Bpl 或 Dcp" Bpl 或 Dcp 資料夾內有特定於平台的專屬資料夾.*

## RAD 伺服器可以部署在那些平台

RAD Server 與 **Windows**、**Linux** 和 **Docker** 平台相容. 雖然從概念上講,每個平台所需的服務是相同的,但我們將在本章中看到一些差異,但首先,讓我們討論一下相似之處以及 RAD Server 在幕後如何運作.

## 使用 GetIt 中的安裝程式

如果您要在 Windows 或 Linux 上部署 RAD 伺服器應用程式, 最快的安裝方法是使用可從 GetIt 下載的安裝程式. 你只需要搜尋"RAD Server",你就會找到這兩個安裝程式:




來自 GetIt 的 RAD 伺服器安裝程式

一旦"安裝"完成後(其實這只是一個下載的動作),您可以在下列路徑中找到安裝程式:C:\Users\<YourUserName>\Documents\Embarcadero\Studio\<XX>.0\CatalogRepository 並將整個資料夾複製到您的生產環境電腦,此資料夾包含完整安裝 RAD Server 所需的所有檔案(包括 InterBase 安裝程式).

在生產環境中執行安裝程式之前,您必須安裝 IIS 或 Apache,以便安裝程式可以相應地配置所有要求.

安裝程式將引導您完成安裝所需的不同選項.

> 備註
>
> *在安裝過程中,系統會要求您提供有效的 InterBase 授權.請使用您的 EDN 帳戶和 RAD 伺服器序號註冊 InterBase 實例.*

## 手動部署 RAD 伺服器的先決條件

本章重點在於討論部署 RAD 伺服器所需安裝或設定的所有部分. 即使您使用安裝程式,為了更好地除錯和解決問題,了解所有要求也很重要. 另外,如果您需要將 RAD 伺服器更新到較新的版本,則無需重新安裝整個應用程式,只需更新一些 dll 和 bpls/so 檔案就足夠了.

下面是在生產環境中安裝 RAD 伺服器的強制性要求:

- InterBase 伺服器引擎
- RAD 伺服器授權
- RAD 伺服器安裝
- Web 伺服器(IIS 7+ 或 Apache 2.4+)
- 使用 RAD Studio 編譯的資源文件
- 配置 EMSServer.ini 文件

無論您選擇部署哪個平台,您都需要安裝/配置所有這些步驟. 例如,在 Windows 上,您需要為 Windows 設定 Microsoft 的 Web Server IIS 或 Apache,而在 Linux 上，,將需要設定 Apache. 重要的是要了解 RAD Server 本身並不是可執行檔(除了 Lite 版本,稍後會詳細介紹). 資源以 Windows 的 BPL 或 Linux 的 SO 函式庫的形式編譯.這就是為什麼我們需要一個網頁伺服器來存取這些資源.

對於 InterBase,RAD Server 內部需要使用一個資料庫實例. 保存了大量資訊需要被儲存(統計資料、使用者、角色等),這就是為什麼它需要自己的資料庫來儲存所有這些資訊.

RAD Server 在內部使用 InterBase 的事實並不意味著您必須為自己的資料使用此資料庫引擎。.FireDAC 連接到廣泛的資料庫,您可以選擇適合您需求的資料庫.

> □ 備註
>
> *如果您選擇使用 InterBase 資料庫並且將其部署在同一台電腦上,則您將需要在不同連接埠上執行兩個實例. 通常的做法是為您自己的資料庫執行個體保留連接埠 3050,並在另一個連接埠 3051 安裝 RAD Server InterBase 執行個體.. 無法使用相同實例,因為 RAD Server 使用自己的加密系統.*

# 在 Windows 上手動部署

## InterBase Server 引擎

從 https://my.embarcadero.com 下載適用於 Windows 的最新 InterBase 安裝程式並將其安裝在您的生產電腦上. 如果您以前從未安裝過,可以按照 本教學 進行操作. 您也可以在此處找到 Windows 上生產環境的 RAD 伺服器資料庫要求.

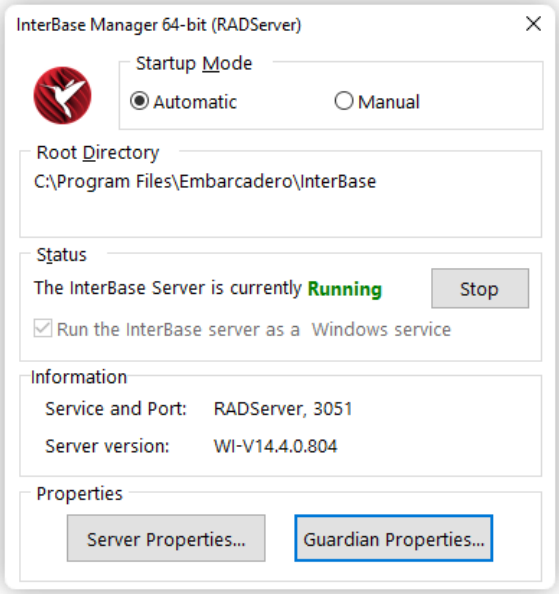
安裝的具體細節:

- 選擇“Server 和 Client”
- 允許在同一台電腦上執行 InterBase 的多個實例
- 建議將預設連接埠變更為 3051
- 將實例命名為 RADServer(而不是預設的 gds_db)
- 若要註冊 InterBase,請使用為您提供的相同 RAD 伺服器序號和您的 EDN 帳戶.



安裝完成後,您需要啟動 InterBase 伺服器的 RADServer 實例. 選擇 Start | Programs | Embarcadero InterBase | 64-bit instance = RADServer | InterBase Server Manager. 如果您希望 InterBase 作為服務執行(預設值),請選取該方塊.如果您希望 InterBase 在電腦啟動時運行,請按一下"Automatic"單選按鈕.然後點擊開始按鈕.




適用於 RADServer 的 64 位元 InterBase 管理員

竅門

*您可以在程式清單中的"Embarcadero InterBase" 下或安裝它的路徑中的資料夾 ".\bin\IBMgr.exe" 下找到 InterBase Manager,並指定實例名稱 ，例如:".\IBMgr.exe RADServer".另一個選擇是簡單地使用 Windows 搜尋並輸入"InterBase Manager".*

## RAD Server 安裝

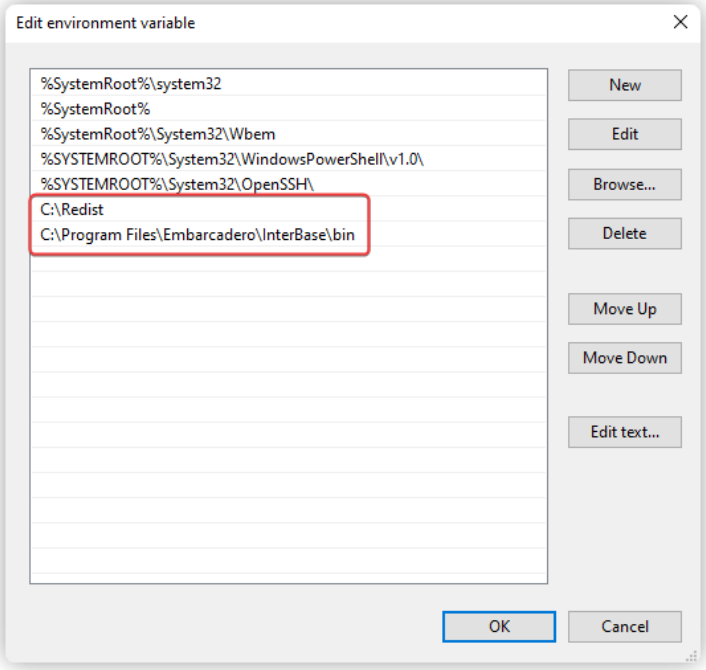
要在 Windows 電腦上安裝 RAD 伺服器,我們需要遵循與在開發電腦中設定它時非常相似的步驟. 此過程所需的大部分文件都可以在這些資料夾中找到:

- C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\bin64
- C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\redist\win64

在 docwiki 中有一個關於如何在 Windows 上安裝 RAD Server 的非常詳細的教學課程. 您可以 在這裡找到它 儘管如此,我們將在這裡解釋基本步驟:

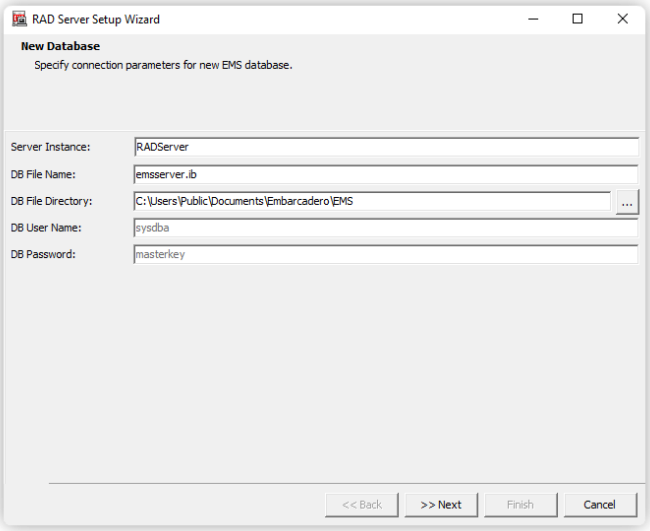
請依照下列步驟準備生產伺服器以測試並使用 InterBase RAD Server 實例和 EMSDevServer.EXE 以便建立 RAD Server 資料庫和設定檔.

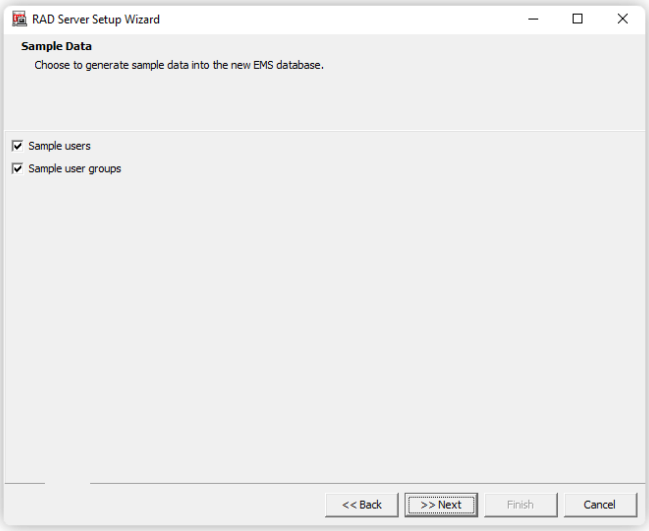
1. 將 64 位元 EMSDevServer.exe 複製到生產伺服器的 c:\installs\EMS 資料夾中
2. 將所需檔案從 RAD Studio Redist/win64 資料夾複製到生產伺服器上名為 c:\Redist 的資料夾中
3. 編輯生產伺服器上的系統路徑環境變數,新增 c:\Redist 和 c:\Program Files\InterBase\bin 資料夾.



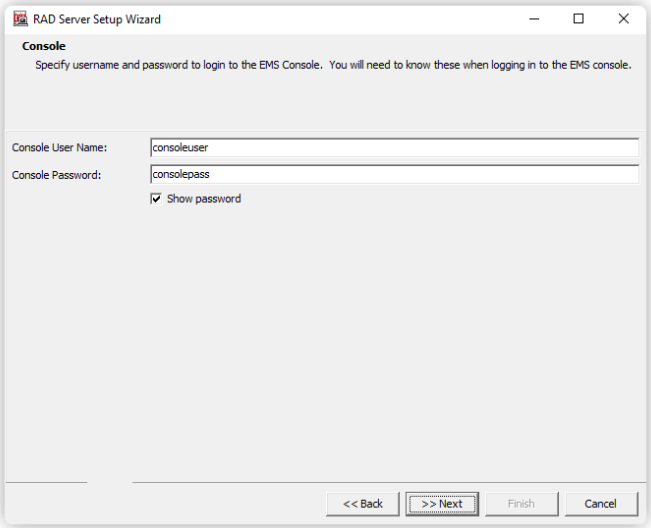

將兩個資料夾新增至您的系統路徑

4. 將開發電腦上的 EMS 範本和 Web 資源檔案從 C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\ObjRepos\en\EMS 複製到名為 c:\installs\ObjRepos\EMS 的生產伺服器資料夾中 (EMSDevServer.EXE 會在放置 EMSDevServer 資料夾的相同父資料夾下的 ObjRepos\EMS 子資料夾中尋找範本和 Web 資源檔案)
5. 確定在生產伺服器上啟動了具有 RAD Server 許可證的 InterBase 伺服器.
6. 執行 EMSDevServer.exe(如同第一個 RAD 伺服器開發設定中所做的)以設定生產 RAD 伺服器設定檔和 InterBase RAD 伺服器資料庫.以下的畫面顯示這些步驟.

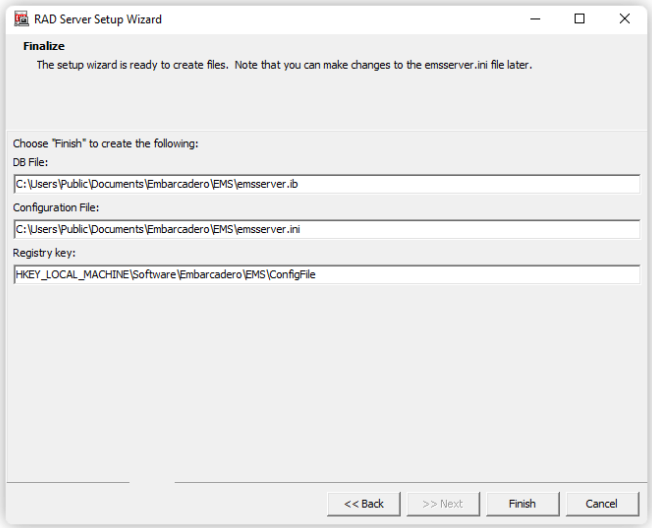


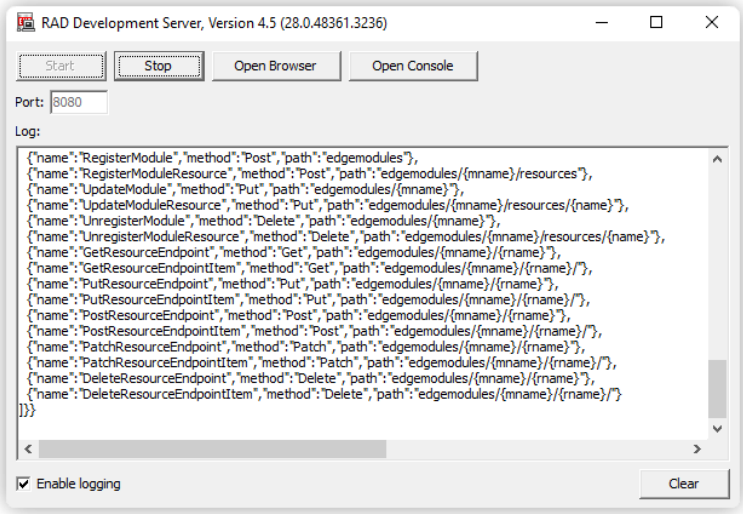
RAD 伺服器設定精靈 - 設定 RAD 伺服器的連線參數




RAD 伺服器設定精靈 - 選擇產生範例數據

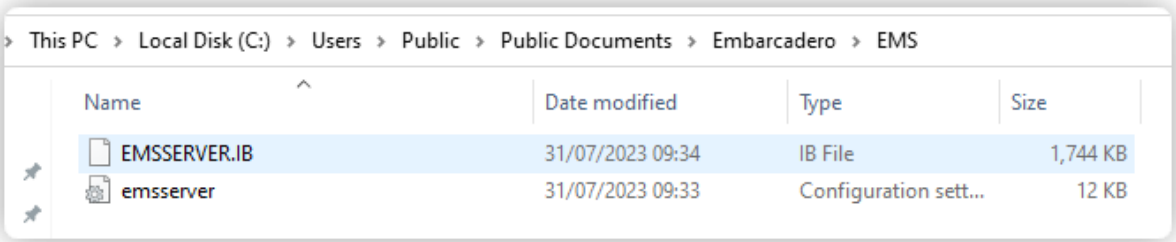



RAD 伺服器設定精靈 - 設定 RAD 伺服器的連線參數 RAD 伺服器安裝精靈 - 檢視將建立的檔案和登錄項




RAD Server 開發伺服器正在執行

RAD 伺服器精靈將在 Public Documents 資料夾下的預設資料夾中建立兩個文件.



RAD 伺服器安裝精靈 - 在公用文件資料夾中建立的兩個文件

### Web 伺服器(IIS 或 Apache)

如果您在 Windows 上部署,則可以使用 Microsoft 隨 Windows 提供的 Web 伺服器(又稱為 IIS),也可以使用 Apache for Windows. 在 本連結您可以查看詳細指南,它不僅詳細說明需要複製到生產環境的文件,還說明如何在 Windows 電腦上設定 IIS 或 Apache.

I 如果您選擇 IIS,Microsoft 有不同的版本,且服務的安裝過程可能會有所不同. 如果您以前沒有使用此服務的經驗,您可以在以下位置找到資訊: 連結.

最後一步是複製我們編譯的資源.到本章結尾查看如何操作.

竅門

*如果您在伺服器管理員上使用"新增角色或功能"選項來新增 Web 伺服器 (IIS),則必須在"應用程式開發"部分下勾選安裝"ISAPI 擴充功能"和"ISAPI 過濾器"*

## 在 Linux 上手動部署

將 RAD 伺服器和應用程式部署到 Linux 伺服器提供了以下選項:

- 若要建立獨立 RAD 伺服器,請參閱 RAD 伺服器安裝章節
- 若要建立適用於 Apache 的 RAD 伺服器,請參閱設定適用於 Apache 的 RAD 伺服器章節



### 相容的 Distros

RAD Server 正式支援 Ubuntu 18+ 和 RHEL 7+.這並不意味著它不能安裝在 RockyLinux、Debian 等其他發行版中,但內部測試始終使用官方支援的發行版進行測試.

### 安裝 InterBase Server 引擎

從 https://my.embarcadero.com 下載適用於 Linux 的最新 InterBase 安裝程式. 在 zip 檔案內,您將找到安裝程式. 在這裡您還可以找到 Linux 上生產環境的 RAD 伺服器資料庫要求.

解壓縮下載的檔案後,為安裝程式指派執行權限並執行它:

```
chmod +x install_linux_x86_64.sh
sudo ./install_linux_x86_64.sh
```

安裝的具體細節:

- 選擇 “Server 和 Client”
- 允許在同一台電腦上執行 InterBase 的多個實例
- 建議將預設連接埠變更為 3051

- 將實例命名為 RADServer(而不是預設的 gds_db)
- 安裝路徑: /opt/interbase

> 竅門
>
> *InterBase 安裝程式將自動偵測您的Linux 安裝是否有桌面環境. 如果您想強制控制台模式執行安裝程式, 請使用此參數:sudo ./install_linux_x86_64.sh* ***-i Console***

> 備註
>
> *您可以使用您喜歡的名稱定義實例和路徑的名稱.如果這樣做,請記住在配置過程中相應地參考這些內容.*

## 註冊並啟動 InterBase Server

若要啟動註冊精靈,請執行指令:

```
sudo /opt/interbase/bin/LicenseManagerLauncher -i Console
```

這將啟動授權精靈. 對於控制台模式,我們建議選擇選項 2“Direct register”,您可以在其中指定您的 RAD 伺服器序號以及您的 EDN 帳戶. 助理將完成其餘的工作,並驗證您連接到 Embarcadero 伺服器的授權.

如果您現在想驗證授權是否已正確加載,您可以使用上一個選單"List license"中的選項 1 來確認一切按預期進行.

InterBase 實例已安裝並獲得授權,但需要啟動它.為此,我們需要進入 InterBase 控制台執行此命令

```
sudo /opt/interbase/bin/ibmgr -start
```

為了簡化其他應用程式和服務與 InterBase 資料庫的連接,最簡單的方法是建立一個到 InterBase 函式庫的符號連結並將其指向 /usr/lib. 這將避免您需要將函式庫複製到需要 InterBase 連接的每個服務的位置.

```
sudo ln -s /opt/interbase/lib/libgds.so.0 /usr/lib/libgds.so
```

## 將 InterBase 作為服務執行

InterBase 也可以設定為服務,以便在 Linux 啟動時運作.在終端機視窗中使用以下命令.

存取您安裝 interbase 的路徑中的"examples」"資料夾,並將 ibserverd 腳本檔案複製到您安裝的伺服器實例版本的位置:

```
sudo cp ibserverd ibserverd_RADServer
```

透過以'sudo'或'root'身分執行上述腳本來設定自動服務啟動.

```
sudo ./ibservice.sh -s /opt/interbase RADServer
```

第二個參數是安裝資料夾,第三個參數是實例名稱.現在,當您重新啟動系統時,只要獲得正確許授權, 服務就會自動啟動.

檢查以確保 InterBase 設定為下次重新啟動時作為服務啟動.

```
ps -ef | grep ibserver
```

當將 InterBase 作為服務運行時,只要電腦在多用戶模式下執行,InterBase 伺服器就會自動啟動.

如果您喜歡手動建立服務(或者您的 Linux 發行版使用稍微不同的方法),您可以在以下位置找到有關此設定的詳細信息連結



備註

*若要將 InterBase 作為服務刪除,請執行:*
*sudo /opt/interbase/examples/ibservice.sh -r[emove]*

## 安裝 RAD Server

在安裝了 RAD Studio 的電腦上,您可以在下列路徑中找到 RAD Server Linux 安裝程式：C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\EMSServer

將這些檔案複製到您的 Linux 電腦並執行安裝程式.您可能需要授予它執行權限.

警告

*確保已安裝 libcurl.如要安裝它,請使用您的發行版套件管理器.例如基於 Debian 的可執行：**apt install libcurl4***

安裝 shell 腳本將建立一個目錄 /usr/lib/ems,其中包含 EMSDevServerCommand、EMSDevConsoleCommand 以及執行命令檔案所需的幾個執行時期函式庫 (.so) 文件. 您也可以在 此連結 中的 docwiki 中找到包含詳細資訊的教學課程.

安裝完成後,在設定模式下執行 EMSDevConsole:

```
/usr/lib/ems/EMSDevServerCommand -setup
```

輸入 **start** a 並且按下 enter.

透過輸入以下值指定連線參數:

- 伺服器實例：鍵入下列預設實例名稱 **RADServer**
- DB 檔案名稱: 內定名稱是 **emsserver.ib**
- DB 檔案目錄: **/usr/lib/ems**
- DB 使用者名稱: 內定參數是 **sysdba**
- DB 密碼: 內定參數是 **masterkey**
- Console 使用者名稱: 內定值是 **consoleuser**
- Console 密碼: 內定值是 **consolepass**

輸入**"n"** 如果配置選項是正確的. emsserver.ini 和 emsserver.ib 檔案已創建,RAD 伺服器在連接埠 8080 上開始執行..



配置過程完成後,您可以在 **/usr/lib/ems** 中找到 RADServer 資料庫,在 **/etc/ems** 中找到設定檔**.**

保持 RADServer 執行,現在讓我們測試一下我們是否可以正確存取它並得到回應.存取: *http://<LinuxMachineIP>:8080/version*


顯示呼叫版本端點輸出的瀏覽器

EMSDevServerCommand 和 EMSDevConsoleCommand 可用於開發和測試 Linux RAD 伺服器應用程式, 而無需使用 Apache. 下一步是設定和測試 RAD 伺服器和 Delphi/C++ 編譯的應用程式模組,以便在 Linux 和 Apache 上以生產模式執行.

竅門

*如果您想在 Linux 上部署 RAD 伺服器,同時也使用 InterBase 作為您的資料選擇資料庫,您可以參照本教學.*

## 為 Apache 設定 RAD 伺服器

使用 InterBase iSQL 指令（在 /opt/interbase/bin 目錄中）確保 RAD 伺服器能夠連接到 emsserver.ib 資料庫文件.

```
sudo ./isql -user sysdba -pass masterkey localhost/RADServer:/usr/lib/ems/emsserver.ib
ISQL> SHOW VERSION;
ISQL> SHOW DATABASE;
ISQL> exit;
```

設定 Apache HTTP Server 以載入 Apache RAD 伺服器 (libmod_emsserver.so) 和 Apache RAD 伺服器控制台 (libmod_emsconsole.so) 模組. 儘管無論您使用哪種 Linux 發行版,Apache 的配置都非常相似,但請記住 RHEL 和基於 Debian 的發行版之間存在一些差異.

> 備註 檢查 Linux 發行版的文檔,以驗證載入模組和定義位置標籤的建議方法.



新增下面的設定以載入 RAD Server Apache 伺服器模組 (libmod_emsserver.so) 和 RAD Server Apache 控制台模組 (libmod_emsconsole.so).

```
LoadModule emsserver_module /usr/lib/ems/libmod_emsserver.so
LoadModule emsconsole_module /usr/lib/ems/libmod_emsconsole.so
```

新增位置標籤以建立容器,您可以在其中指定給定 URL 的存取控制規則.

```
<Location /radserver>
   SetHandler libmod_emsserver-handler
</Location>
<Location /radconsole>
   SetHandler libmod_emsconsole-handler
</Location>
```

若要測試您的 RAD 伺服器是否正確執行,請使用瀏覽器透過存取來顯示 RAD 伺服器版本號: http://<LinuxMachineIP>/radserver/version

最後一步是複製我們編譯的資源,請到本章末尾查看如何操作.

# 在 Docker 中部署

在 Docker 中部署 RAD Server 比使用 Windows 和 Linux 簡單得多。 Embarcadero 在 dockerhub 中有多種可用於此平台的映像.

您會發現 2 個與 RAD Server 相關的鏡像：這兩個鏡像之間的唯一區別是，一個在容器內運行 InterBase 伺服器引擎，另一個假設運行 RAD Server 所需的 InterBase 伺服器將託管在其他地方.

□
竅門

*InterBase 伺服器也與 Docker 相容，而 Embarcadero 提供了一個映像來進行容器化。這是 DockerHub 的連結.*

這些 Docker 映像的建置方式是完全開源的，並可在 GitHub 上公開取得. 這只是一種方法，但如果您對 Docker 足夠熟悉，請隨意使用這些作為模板並根據您的特定需求進行調整.

下面的 DockerHub 和 GitHub 連結中有大量有關如何部署和自訂這些映像的信息.

## 選項 1: PA-RADServer-IB

這個鏡像就是我們所說的"包含所有電池".容易，但請記住，第一次執行此容器時，您無法在分離模式下執行此操作. 需要執行第一個精靈來設定 RAD 伺服器授權和一些額外的詳細資訊。一切設定完畢後，您可以獨立運行它.



另一件需要記住的事情是，如果您沒有計劃擴展應用程式並且希望將所有內容放在一個地方，那麼這個容器非常方便, 另一件需要記住的事情是，如果您沒有計劃擴展應用程式並且希望將所有內容放在一個地方，那麼這個容器非常方便

但如果您想在未來擴展應用程式，也許最好的方法是將 RAD 伺服器與 InterBase 伺服器分開，並將它們放在單獨的容器/機器中.

DockerHub 連結

GitHub 連結

本鏡像包含:

- InterBase Server
- PAServer
- RADServer 所需文件
- 預先配置的 Apache

## 選項 2: PA-RADServer

該容器需要連接到安裝了有效 RAD 伺服器授權的 InterBase 伺服器，否則它將無法運作. 如果您想要擴展應用程式並部署連接到相同 InterBase Server 的多個實例，它是一個理想的容器.

DockerHub 連結

GitHub 連結

本鏡像包含:

- PAServer
- RADServer 所需文件
- 預先配置的 Apache

> □ 竅門 *請記住，對於簡單的環境，您可以使用 PAServer 將資源更新直接上傳到容器，而無需重新產生它.*

存取 本連結以獲取有關在 Docker 上部署 RAD 伺服器的更多信息.

## 複製使用 RAD Studio 編譯的 RAD 伺服器模組

無論您選擇哪種作業系統，將模組或所需的附加函式庫部署到生產電腦的過程都幾乎相同. 對於您自己的資源，您只需將.bpl/.so 檔案複製到您的生產機器上.

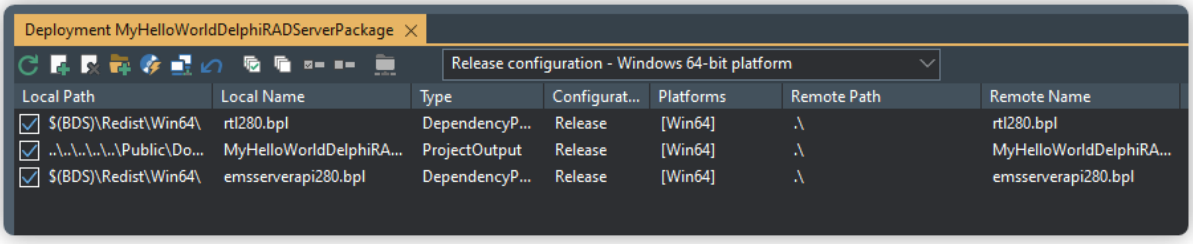
RAD 伺服器應用程式套件檔案根據專案設定編譯到相對應的資料夾。預設的 Delphi 套件輸出和 C++ 最終輸出目錄是:



- 對於 Delphi:
  - 32 位元 Windows - C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\Bpl
  - 64 位元 Windows - C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\Bpl\Win64
  - Linux - C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\Bpl\Linux64
- 
- 對於 C++，所有 RAD 伺服器應用程式包檔案都編譯到 .\$(Platform)\$(Config) 資料夾.

有多種方法可以將所需的 RAD 伺服器應用程式和執行時期 DLL 檔案部署到生產伺服器。三種常見的傳輸方法是:

- 將套件檔案複製到 RADServer 安裝的生產伺服器路徑下
- 透過 FTP 將檔案傳輸到生產伺服器
- 使用平台助理(PAServer) 和"Project| Deployment"選單項目讓 IDE 將檔案移到生產伺服器。此螢幕截圖顯示了 Windows 64 的範例.



Project | Deployment PAServer 可以傳輸的文件

將編譯好的 RAD Server 擴充套件檔案（例如，第 1 章 MyHelloWorldDelphiRADServerPackage 中的專案）複製到生產 RADServer 資料夾.

竅門

*使用 PAServer 時，可以透過在生產電腦中編輯檔案 paserver.config 來變更部署檔案的預設路徑. 執行 PAServer 時可能需要提升權限，這取決於寫入檔案所需的路徑.*

## 配置 EMSServer.ini 文件

現在我們已將新資源新增至生產資料夾中，我們需要在 EMSServer.ini 檔案中指定有可用的新資源.

編輯 emsserver.ini 檔案以便在 [Server.Packages] 部分下新增每個 RAD 伺服器擴充包.

**Windows**

```
[Server.Packages]
;# This section is for extension packages.
;# Extension packages are used to register custom resource endpoints
;c:\mypackages\basicextensions.bpl=mypackage description
c:\inetpub\wwwroot\RADServer\MyFirstDelphiRADServerPackage.bpl=First Windows Test Demo
```

**Linux**

```
[Server.Packages]
;# This section is for extension packages.
;# Extension packages are used to register custom resource endpoints
;c:\mypackages\basicextensions.bpl=mypackage description
/usr/lib/ems/bplMyFirstDelphiRADServerPackage.so=First Linux Test Demo
```

**Docker**

若要設定已執行實例的 emsserver.ini 文件，請執行 ./config.sh 腳本。該腳本將自動重新啟動 Apache.

# 10

# RAD Server 精簡版(Lite)



## 什麼是精簡版?

RAD Server 需要一個基於 InterBase 的後端資料庫,並且通常部署為 IIS 或 Apache 的 Web 伺服器 DLL 模組. 因此,標準部署需要:

- RAD 伺服器模組的 Web 伺服器及其配置
- RAD 伺服器部署與配置
- 使用特殊用途 RAD 伺服器授權安裝的 InterBase(使用者需要在目標裝置上註冊才能啟動的授權)

對於開發來說,我們長期以來提供了基於 Indy HTTP 伺服器的獨立版本的 RAD Server,它提供有限的效能,但更容易部署並且能夠在除錯器下執行(因此您可以除錯 RAD Server 模組程式碼). 開發版本並不能使用來部署.它對您可以建立的使用者數量有限制,並且可以與本機 InterBase 開發者版本一起使用(它的授權是 RAD Studio 授權的一部分).

RAD Server 精簡版 (**RSLite**) 為不需要大量吞吐量的測試伺服器和場景提供更簡單的部署模型,它透過使用 InterBase 嵌入式資料庫引擎 IBToGo(而不是成熟的伺服器)來實現這一點,並將其與簡化的授權模型相結合.

RSLite 使用與開發版本相同的二進位檔案(隨 RAD Studio 一起提供)以及 IBToGo 二進位檔案和授權文件,您可以連同您的解決方案一起部署(無需在部署到的電腦上註冊). 由於它使用嵌入式資料庫並且使用 Indy HTTP Server 元件,因此它無法提供與常規完整 RAD Server 安裝相同數量的每秒請求服務數量,並且無法透過多個 RAD Server 前端進行擴展.

RSLite 使用的底層架構只有有限的可擴展性,但我們希望它足以滿足許多簡單的部署場景 - 請記住,服務吞吐量也取決於 RAD 伺服器模組執行的特定程式碼.

□
竅門

*對於公共系統上的部署,我們建議避免直接公開 RSLite HTTP 伺服器,而是透過代理配置對其進行訪問,以便您仍然擁有一個 Web 伺服器(如 Apache 或 IIS),為傳入的 HTTPS 呼叫提供安全上下文並轉發這些呼叫到 RSLite.*

## 如何取得 RAD Server Lite 授權

您可以使用 RAD Studio 11(包括 Delphi 11 和 C++Builder 11)的任何企業版或 架構師授權來兌換授權.. 請造訪此頁面 並按照提供的說明進行操作.

□
備註

*您需要註冊金鑰和 EDN 帳戶.*

這裡的流程不僅僅是接收 RSLite 的授權金鑰,而是一個可以在安裝時部署的 slip 檔案(儲存在 .TXT 檔案中的授權). 此授權對安裝數量沒有限制,但您不能在同一台電腦上執行兩個實例.

□
備註

*授權文件需要放置在特定的子資料夾中,這與兌換網站上的一般資訊似乎暗示的不同.*

## 部署 RAD Server 精簡版項目專案

在部署專案之前獲得授權後,有兩個不同的注意事項:

- 首先,您需要使用 RSLite、所需的執行時間套件和 IBToGo 部署配置 (以下是取得它的步驟)
- 其次,您需要產生一個適合生產的資料庫文件,與 IBToGo 授權相容 - 由 RAD Server 開發者版本建立的本機資料庫將不相容

# 要部署的文件檔案

## 手動部署

實際上,這些是部署 RSLite 解決方案所需的檔案(除了您的應用程式套件及其相依性之外):

1. RSLite 執行文件,與開發人員版本相同:RAD Studio bin 資料夾中提供的 EMSDevServer.exe(或類似的 64 位元版本)
2. 所需的 RAD Studio 執行時間套件包,其中包括最小安裝所需的套件包(此處列出並在 RAD Studio win32 或 win64 redist 資料夾中提供)以及 RAD 伺服器模組中的程式碼所需的任何其他運行時套件包:
   - bindengine<XX>0.bpl
   - dbrtl<XX>0.bpl
   - emsclientfiredac<XX>0.bpl
   - emsserverapi<XX>0.bpl
   - FireDAC<XX>0.bpl
   - FireDACCommon<XX>0.bpl
   - FireDACCommonDriver<XX>0.bpl
   - FireDACIBDriver<XX>0.bpl
   - rtl<XX>0.bpl
   - vcl<XX>0.bpl
   - vcldb<XX>0.bpl
   - vclFireDAC<XX>0.bpl
   - vclimg<XX>0.bpl
   - vclwinx<XX>0.bpl
   - vclx<XX>0.bpl
   - Xmlrtl<XX>0.bpl
3. 在公用文件 InterBase redist 資料夾(例如,C:\Users\Public Documents\Embarcadero\Interbase redist\InterBase2020)下的子資料夾 win32_togo 或 win64_togo 中找到 InterBase ToGo 部署檔案— 對於 Linux,您可以找到 libibtogo. so 檔案位於正確的 InterBase redist 資料夾中
4. 將上面獲得的授權檔案加入到 interbase/license 資料夾中(IBToGo redist 配置的一部分)



## 使用部署精靈

請依照下列步驟使用部署精靈中的 RSLite 功能部署文件:

1. 新增 RSLite 功能.
2. 接下來新增 IBToGo 功能.
3. 取消 iblite 註冊文件的勾選
4. 在如何取得 RAD Server Lite 授權部分中,新增要部署的檔案：產生 rslite 啟動檔案並將其目標設定為 "interbase/license".
5. 接下來,新增從建立生產資料庫部分獲得的文件,以將我的 **emsserver.ini** 部署到 ./.
6. 最後,新增從建立生產資料庫部分獲得的文件,以將我的 **emsserver.ib** 部署到 ./.

# MSVC 執行時期檔案

要在目標 Windows 電腦上執行 IBToGo(以及使用 IBToGo 的 RSLite),需要安裝 Visual C++ 2013 執行時期程式庫. 在裝有 RAD Studio 的開發電腦上,您很可能已經安裝了它.但是,在通用目標部署電腦上,您可能必須安裝它, 您可以從這裡下載 Microsoft.

## 建立生產資料庫

使用此配置,您可以透過執行 EMSDevServer.exe 應用程式來啟動 RSLite. 請注意,如果目標電腦有 InterBase 用戶端,它將作為更高優先級的選擇,並且如果 InterBase 用戶端是 RAD Studio 附帶的開發者版本,則一切都將正常工作,但是在標準 RAD Server 開發者模式配置中.

您可以透過查看 RAD 伺服器啟動時日誌中的前幾行來弄清楚這一點.如果是"RSLite"配置,前幾行將如下所示:

```
{"Thread":19124,"ConfigLoaded":{"Filename":"[folder]emsserver.ini","Exists":true}}
{"Thread":19124,"Licensing":{"Lite":true,"Licensed":true,"LicensedMaxUsers":2}}
{"Thread":19124,"DBConnection":{"InstanceName":"","Filename":"[folder]emsserver.ib"}}
```

如果程式碼顯示"Lite"設定為 false,您可能需要手動停用 gds32.dll(或其 64 位元版本)InterBase 用戶端程式庫的加載,該程式庫通常位於 C:\Windows\SysWOW64 中(如果 InterBase 用戶端找不到庫它, 它就會載入本地 **ibtogo.dll**).

現在,如果您啟動 RSLite(使用正確的配置)並且沒有 emsserver.ini 文件和 emsserver.ib 資料庫文件,它將提示您建立一個. 為此,RSLite 必須在 RAD Studio 的物件儲存庫資料夾(產品資料夾下的 ObjRepos)尋找配置. 更簡單的方法是將 Program Files (x86)\Embarcadero\Studio\<XX>.0\ObjRepos\en\ems 下的檔案複製到具有 **emsdevserver.exe** 相對路徑的資料夾中："../ObjRepos /EMS".換句話說,您需要一個與包含 RSLite 安裝的資料夾(專案部署目錄)處於相同等級目錄的 ObjRepos 資料夾.



> □ 備註
>
> *每個 RSLite 部署都不需要重覆這樣做,只需產生一次生產資料庫,您可以稍後按原樣複製到目標電腦上. 事實上,在開發環境中建立的資料庫與 RSLite 部署不相容.*

我們建議您指定與 RSLite 部署相同的目標資料夾,以便精靈將在您的部署資料夾中建立 **emsserver.ini** 檔案和 **emsserver.ib** 資料庫文件. 現在,取得 RSLite、這些設定檔、執行時間套件和 IBToGo(包括授權)和整個資料夾,您就擁有了在目標 Windows 電腦上部署所需的所有內容.

## Proxy 配置

由於 RSLite 在保護和加密方面的局限性,不建議直接將 RSLite 公開為公共 Web 應用程式. 我們建議使用代理層和專用服務或使用流行的 Web 服務之一作為前端. 例如,在 Apache 中,您設定虛擬主機,啟用 HTTPS,並使用下列設定將流量重新導向至 RSLite 實例:

```
ProxyPass / http://localhost:8088
ProxyPassReverse / http://localhost:8088
ProxyPreserveHost On
```

## 對於 Linux

對於 Linux,您可以按照與上面類似的步驟操作,一切都應該按預期工作.作為替代方案,您也可以考慮安裝完整的 RAD 伺服器,然後將 IBToGo 新增至安裝中:

- 使用 RAD 安裝資料夾中提供的 ems_install.sh 安裝 RAD 伺服器
- 將 IBToGo 檔案從 InterBase"redist"資料夾複製到 Linux 上的 EMS 資料夾 (/usr/lib/ems)
- 執行 EMSDevServerCommand,依照精靈建立 EMS 資料庫和設定檔

□
備註

*您可能需要透過 sudo 運行應用程式以獲得適當的權限*

# 11

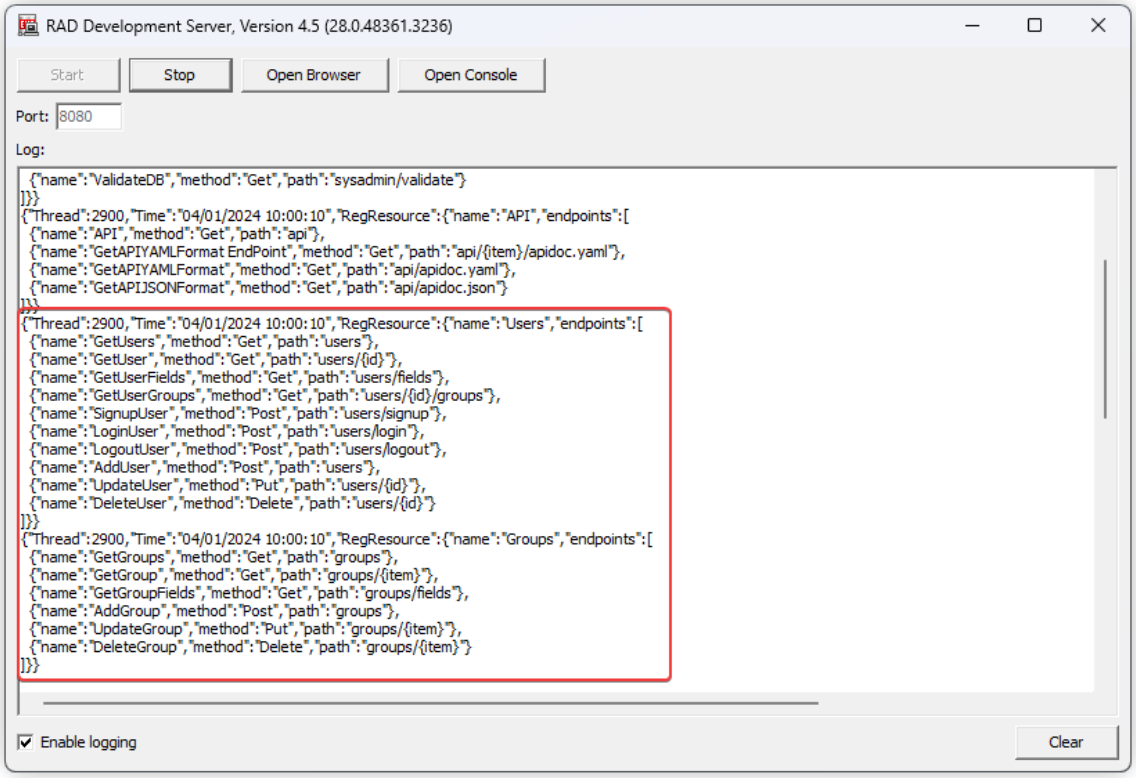
# 認證與授權



RAD Server 為使用者驗證和授權提供開箱即用的解決方案. 多個端點可用於處理所有這些事項,並且不僅可以透過請求存取它們,還可以從使用 RAD Studio 開發的其他基於 RAD 伺服器的應用程式或以程式設計方式從 RAD 伺服器資源存取它們,如果您想整合自己的身份驗證系統並且覆蓋 RAD 伺服器提供的服務的話, ,這樣的開放功能非常的有用..

## 整合性的身份驗證:管理使用者和群組

在我們的開發機器中執行 RAD 伺服器實例後,如果我們分析日誌,我們可以看到一些重要的服務端點:

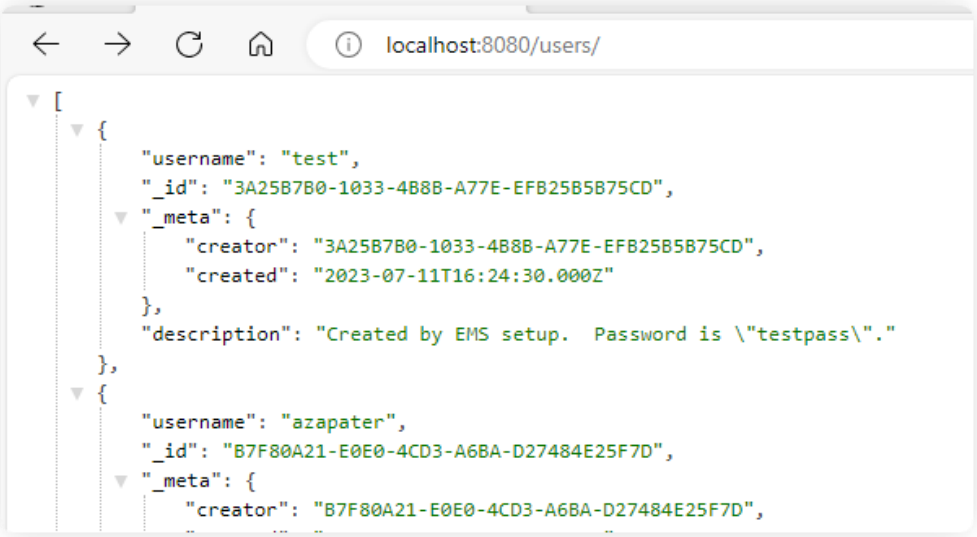





RAD 伺服器為使用者和群組管理所建立的預設端點

有 2 個與身分驗證相關的資源: 使用者 和 群組. RAD 伺服器不僅允許我們定義用戶,還可以將這些用戶分配到特定群組,以便可以以更精細的方式定義角色並授予或拒絕對特定端點和/或資源的使用權限.

讓我們訪問端點 users/ 看看我們能得到什麼.



存取 RAD 伺服器中建立的使用者列表

RAD 伺服器傳回一個數組陣列,其中包含資料庫中建立的所有使用者.正如我們在螢幕截圖中看到的,預設情況下,RAD 伺服器會建立一個測試用戶,密碼為"testpass"(在安裝精靈期間定義).

> □ 備註
>
> 如果您想查看所有資源的更多詳細信息,您可以在 OpenAPI json/yaml apidoc 文件規範中定義的文檔中找到它們.

使用者和群組資源遵循標準 CRUD 約定,因此如果您想要建立新記錄,則需要向此端點發送 POST 請求.要修改一個記錄,則需要 PUT 請求(在正文請求中包含新值)等等.

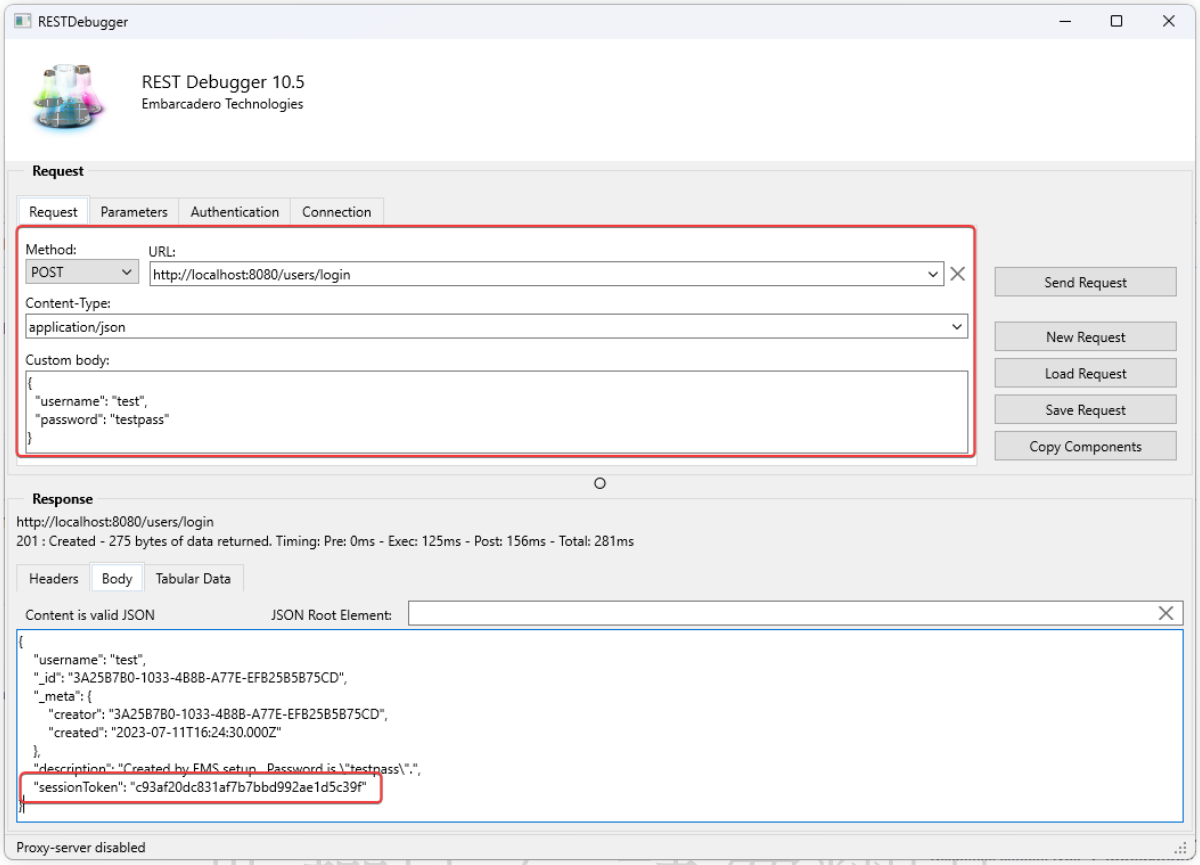
> □ 竅門
>
> 可以向使用者和群組新增自訂欄位.只需在請求正文中添加任意字段欄位,它將儲存為額外的字段欄位.

## 登錄



創建的用戶將能夠登入 RAD 伺服器並獲取標誌,該標誌將用於在將來的請求中識別他們.

正如我們在自動建立的端點中看到的,要登錄 RAD Server,我們只需透過 POST 請求存取 users/login 端點,並在正文中包含使用者名稱/密碼.






存取 POST users/login 端點以取得 sessionToken

一旦我們有了 sessionToken,我們必須將該值插入到未來的請求標頭中,以便 RAD 伺服器可以正確識別用戶.

```
X-Embarcadero-Session-Token=<value>
```

## 登出

預設情況下,會話標誌的到期日期是無限的.然而,出於安全原因,這可能不是最好的設定.可以使用 EMSServer.ini 中的下列設定來微調標誌到期日期/時間.有一些參數可以配置這個:

```
SessionInactivityTimeout=
;# Set SessionInactivityTimeout=60 to limit maximum time of a session inactivity in
seconds.
;# This operates on a session token.  Default is 0 (no timeout).
SessionLiveTimeout=
;# Set SessionLiveTimeout=60 to limit maximum time of a session live in seconds.
;# This operates on a session token.  Default is 0 (no timeout).
```

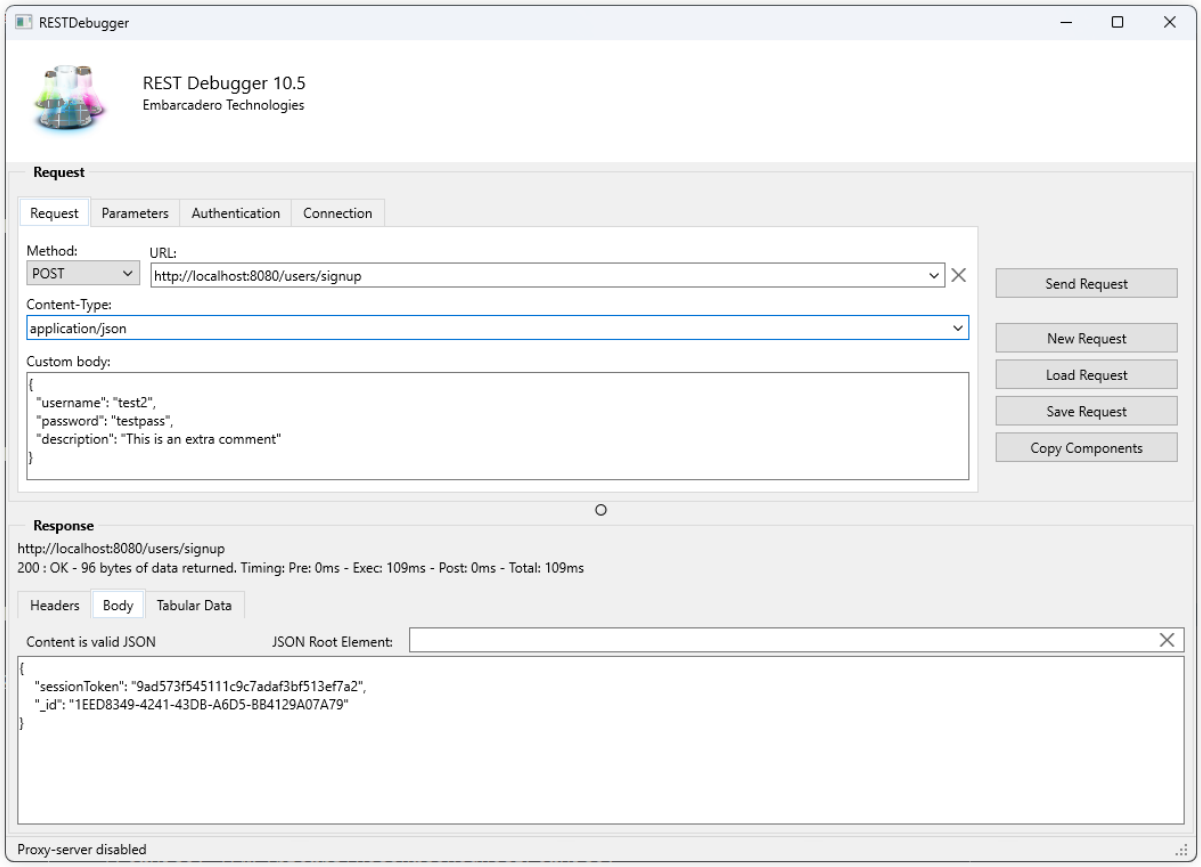
在這兩種情況下,超時都必須以秒為單位指定.一旦超過其中一個超時時間,標誌將被自動停用.

如果我們想要強制登出並手動停用活動標誌,我們可以向端點 users/logout 發送請求(在標頭中包含使用者的標誌).

## 報名

如果我們向 /users/signup 發送請求,我們將在 RAD Server 中註冊用戶,同時我們將取得 ID 以及會話標誌.




存取 POST users/singup 端點以建立新用戶

在上面的範例中,需要注意的是,可以在請求正文中發送任意數量的自訂欄位(例如，我們使用了'description'自訂欄位).

使用者名稱是唯一的,因此如果發送使用現有使用者名稱的註冊請求,RAD 伺服器將回應 409 錯誤.

> □ 備註
>
> 不想要公開註冊程序?不要忘記透過將其限制為角色部分中的授權使用者和群組來限制對 users.signup 端點的存取.欲了解更多信息,請參閱授權章節.

### 管理群組

也可以使用整合端點來管理群組.此資源也遵循標準 CRUD 約定.

一個用戶可以屬於多個群組,一個群組當然可以包含多個用戶.

重要提示:要將使用者指派到群組,必須將 POST 請求設定為 /users/groups/{id},其中包含該群組的所有成員的陣列數據.

```
{
  "fieldName": "string",
  "users": [
    "string"
  ]
}
```


竅門

即使您不打算在群組層級定義角色,我們也鼓勵您將使用者指派到群組. 這在分析中非常有用,因為可以根據這些群組顯示特定的詳細信息.

## 整合性授權

### 全域憑證

可以定義多個安全等級的全域憑證. 在 RAD 伺服器引擎設定檔(emsserver.ini 檔案)中,[Server.Keys] 群組下有 3 個與此相關的參數:

**MasterSecret**
它授權您完全存取儲存在 RAD 伺服器資料庫中的所有 RAD 伺服器數據.

來執行管理任務.您可以使用它來存取 RAD 伺服器引擎(EMS 伺服器)中的所有 RAD 伺服器資源.

**AppSecret**
它授權您從 RAD 伺服器客戶端應用程式存取授權端點.

**ApplicationID**
應用程式 ID 用於識別來自基於 RAD 伺服器的用戶端的請求,因此僅處理具有有效應用程式 ID 的請求,而拒絕具有無效應用程式 ID 的請求. 此識別碼可用於區分不同的 RAD 伺服器實例.

您可以在此處找到更詳細的信息 docwiki link.

## 使用者和群組授權

保護 RAD 伺服器端點的最簡單方法是透過 EMSServer.ini 檔案.所有規則必須在[Server.Authorization]部分下定義.讓我們看看 ini 檔案中預設的範例:

```
[Server.Authorization]
;# This section is for setting authorization requirements for resources and endpoints.
;# Authorization can be set on built-in resource (e.g.; Users) and on custom
resources.
;# Note that when MasterSecret authentication is used, these requirements are ignored.
;# Resource settings apply to all endpoints in the resource.
;# Endpoint settings override the settings for the resource.
;# By default, all resource are public.
;# Settings are specified in JSON.
;# JSON attributes
;#  {"public": true} - any client is authorized
;#  {"public": false} - a client may be authorized depending on user or group.  user
credentials (sessionid) must passed in the request
;#  {"users": ["username1", "username2"]} - authorize a user by username.
;#  {"users": ["userid1", "userid2"]} - authorize a user by userid.
;#  {"users": ["*"]} - authorize any user.
;#  {"groups": ["groupname1", "groupname2"]} - authorize a user in a user group.
;#  {"groups": ["*"]} - authorize a user in any user group
;#
;# Examples
;#
;# Make all methods in the resource "Users" private except for LoginUser and
SignupUser endpoints
;Users={"public": false}
;Users.LoginUser={"public": true}
;Users.SignupUser={"public": true}
;#
;# Make all methods in the custom resource "Resource1" available to users in group1
;Resource1={"groups": ["group1"]}
;#
;# Make all methods in the custom resource "Resource2" available only with
MasterSecret authentication
;Resource2={"public": false}
;#
;# Special rules for user and group creators.
;# The creator of user is automatically authorized for the following endpoints:
;#   Users.GetUser, Users.UpdateUser, Users.DeleteUser
;# The creator of a group is automatically authorized for the following endpoints:
;#   Groups.GetGroup, Groups.UpdateGroup, Groups.DeleteGroup
```

儘管該文件已經提供了非常不言自明的註釋,讓我們看幾個範例.

```
Orders={"groups": ["sales"]}
```

銷售組的所有成員都可以存取資源 Sales 中的所有端點.

```
Users={"public": false}
Users.LoginUser={"public": true}
Users.SignupUser={"public": true}
```

在上面的範例中,我們限制了對名為'Users'的整個資源的公共訪問. 但是,我們特別允許公共存取端點'LoginUser'和'SignupUser'. 這是限制對整個資源進行存取, 但一些必需的例外的訪問的非常有用的方法.金鑰'public'意味著將可以在沒有任何身份驗證標的情況下處理請求.

```
Customers={"users": ["*"]}
```

在上面的範例中,所有具有有效證標的請求都可以存取資源 Customers,無論其角色如何,但它們必須註冊.

## 自訂認證



如果您已經實現了身份驗證服務,例如 ActiveDirectory/LDAP 或任何其他第三方服務,則可以將其與 RAD Server 整合. 在範例目錄中,您可以找到 2 個自訂登入範例以及與 AD 基本整合的範例(只要 RAD 伺服器在 Windows 電腦上執行).

要考慮的主要前提是我們需要驗證我們的身份驗證服務所提供的憑證是否正確. 驗證後,我們必須在 RAD 伺服器內建驗證中對使用者進行身份驗證(如果是第一次連接,則建立使用者),然後返回 RAD 伺服器標誌.

為了從我們的程式碼存取 RAD 伺服器的內部 API,我們可以建立它的一個新實例,指定應使用的上下文.讓我們來看一個例子:

**Delphi**

```
// Custom EMS login
procedure TCustomLogonResource.PostLogin(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  lEMSAPI: TEMSInternalAPI;
  lResponse: IEMSResourceResponseContent;
  lValue: TJSONValue;
  lUserName: string;
  lPassword: string;
begin
```

```
  // Create in-process EMS API
  lEMSAPI := TEMSInternalAPI.Create(AContext);
  try
    // Extract credentials from request
    if not (ARequest.Body.TryGetValue(LValue) and
      lValue.TryGetValue<string>(TEMSInternalAPI.TJSONNames.UserName, lUserName) and
      lValue.TryGetValue<string>(TEMSInternalAPI.TJSONNames.Password, lPassword)) then
      AResponse.RaiseBadRequest('', 'Missing credentials');

    var lExternalUserGUID := ValidateExternalCredentials(lUserName, lPassword);

    if not LEMSAPI.QueryUserName(lUserName) then
    begin
      // Add user when there is no user for these credentials
      // in-process call to actual Users/Signup endpoint
      var lUserFields := TJSONObject.Create;
      lUserFields.AddPair('ExternalUserGUID', lExternalUserGUID);
      lUserFields.AddPair('comment', 'This user added by RAD Server CustomLoginUser');
      lResponse := lEMSAPI.SignupUser(LUserName, GenerateHashedPassword(lUserName),
lUserFields);
    end
    else
      // in-process call to actual Users/Login endpoint
      lResponse := lEMSAPI.LoginUser(lUserName, GenerateHashedPassword(lUserName));
    if lResponse.TryGetValue(lValue) then
      AResponse.Body.SetValue(lValue, False);
  finally
    LEMSAPI.Free;
  end;
end;
```

**C++**

```
void TCustomLoginResource::PostLogin(TEndpointContext* AContext, TEndpointRequest*
ARequest, TEndpointResponse* AResponse)
{
      // Create in-process EMS API
      std::unique_ptr<TEMSInternalAPI> lEMSAPI(new TEMSInternalAPI(AContext));
      // Extract credentials from request
      TJSONObject * lValue;
      String lUserName;
      String lPassword;

      if(!(ARequest->Body->TryGetObject(lValue) &&
            (lValue->GetValue(TEMSInternalAPI_TJSONNames_UserName) != NULL) &&
            (lValue->GetValue(TEMSInternalAPI_TJSONNames_Password) != NULL)))
```

```
            AResponse->RaiseBadRequest("", "Missing credentials");

      lUserName = lValue->Get(TEMSInternalAPI_TJSONNames_UserName)->JsonValue-
>Value();
      lPassword = lValue->Get(TEMSInternalAPI_TJSONNames_Password)->JsonValue-
>Value();

      String lExternalUserGUID = ValidateExternalCredentials(lUserName, lPassword);

      _di_IEMSResourceResponseContent lResponse;
      if (!lEMSAPI->QueryUserName(lUserName)) {
            // Add user when there is no user for these credentials
            // in-process call to actual Users/Signup endpoint
            std::unique_ptr<TJSONObject> lUserFields;
            lUserFields->AddPair("ExternalUserGUID", lExternalUserGUID);
            lUserFields->AddPair("comment", "This user added by
CustomResource.CustomLoginUser");
            lResponse = lEMSAPI->SignupUser(lUserName,
GenerateHashedPassword(lUserName), lUserFields.get());
      } else
            // in-process call to actual Users/Login endpoint
            lResponse = lEMSAPI->LoginUser(lUserName,
GenerateHashedPassword(lUserName));
      if(lResponse->TryGetObject(lValue)) {
            AResponse->Body->SetValue(lValue, false);
      }
}
```

為了讓 RAD 伺服器知道該請求來自授權用戶,如果外部身分驗證服務驗證了請求中包含的憑證,我們只需註冊或登入該用戶即可.

您可能已經注意到,我們沒有使用相同的密碼來建立 RAD 伺服器使用者.這是為什麼?好吧,如果用戶在外部身份驗證服務中更改密碼,我們也需要更新 RAD 伺服器密碼,這會增加額外的複雜性以保持所有內容為最新. 對使用者密碼進行雜湊處理將使我們能夠完全依賴身分驗證外部服務,同時仍在 RAD 伺服器實例中保留安全機制.您可以檢查範例項目專案以了解更詳細的實施情況.

> □ 備註
>
> customLogin 和 AD 整合的範例可以在 RAD Studio 的範例資料夾中找到:
> C:\Users\Public\Documents\Embarcadero\Studio\XX.0\Samples\Object Pascal\Database\EMS
> C:\Users\Public\Documents\Embarcadero\Studio\23.0\Samples\CPP\Database\EMS

是否必須在 RAD 伺服器中單獨建立每個用戶? 從技術上來說是不用的，但強烈推薦. 主要原因是為了分析和日誌記錄. 如果我們在 RAD Server 中單獨建立每個用戶,我們將能夠利用嵌入式分析並單獨獲取每個用戶的精細數據. 從技術上講,可以為所有登入重複使用相同的 RAD 伺服器"常規"用戶,並為每個登入建立一個標誌,但需要明確瞭解的是,分析功能將變得不太有用.

在前面的範例中,我們了解如何使用內部 RAD 伺服器 API 以程式設計方式註冊和登入用戶,但這只是此 API 的多種可能性中的 2 種. 建立群組,將使用者指派給群組…透過我們在前面幾節中討論的 API REST 實現的幾乎所有功能都可以透過程式設計方式實現.


竅門

所有 RAD 伺服器內部 API 均位於 EMS.Services 程式單元中. 請查看程式碼或這份 文件 以便查看所有可用的方法.

## 客製化授權

另一種選擇是以程式方式保護端點.您可以在每個端點中定義特定規則以允許或禁止存取使用者和/或群組.

連結到請求的每個方法都有一個 AContext 參數,我們可以使用它來檢查分配給請求的使用者或群組.

**Delphi**

```
//Returns the userName associated with the request
AContext.User.UserName
//Returns the userID associated with the request
AContext.User.UserID
//Returns the groups which the user belongs to
AContext.User.Groups
```

**C++**

```
//Returns the userName associated with the request
AContext->User->UserName
//Returns the userId associated with the request
AContext->User->UserId
//Returns the groups which the user belongs to
AContext->User->Groups
```

經過所需的驗證後,如果我們不想授予對該方法的存取權限,我們可以簡單地使用參數 AResponse 傳回未經授權的錯誤.

**Delphi**

```
AResponse.RaiseUnauthorized('Unauthorized access', '');
```

**C++**

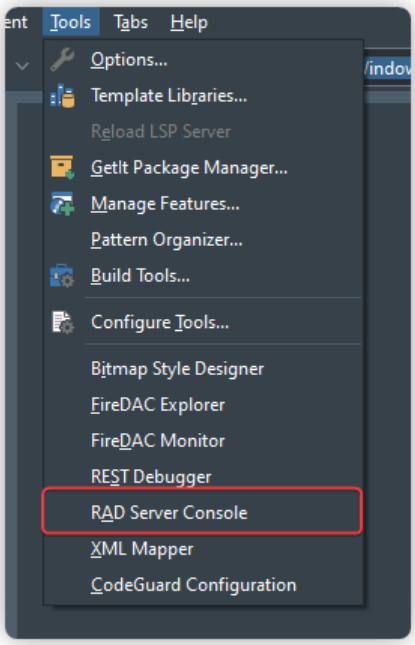
```
AResponse->RaiseUnauthorized("Unauthorized access", "");
```

## RAD 伺服器管理控制台

RAD 伺服器控制台又名 RSConsole 已經存在很多年了,但它有點隱藏在路徑中:

32 bits: C:\Program Files (x86)\Embarcadero\Studio\22.0\bin
64 bits: C:\Program Files (x86)\Embarcadero\Studio\22.0\bin64

從 RAD Studio 12 Athens 開始，可以在"Tools"功能表下找到它.






用於存取 RAD 伺服器控制台的選單項

### 建立新的設定檔

此控制台可用於連接到本機開發環境或生產中的遠端伺服器.它允許您定義多個設定檔並按需連接到它們.

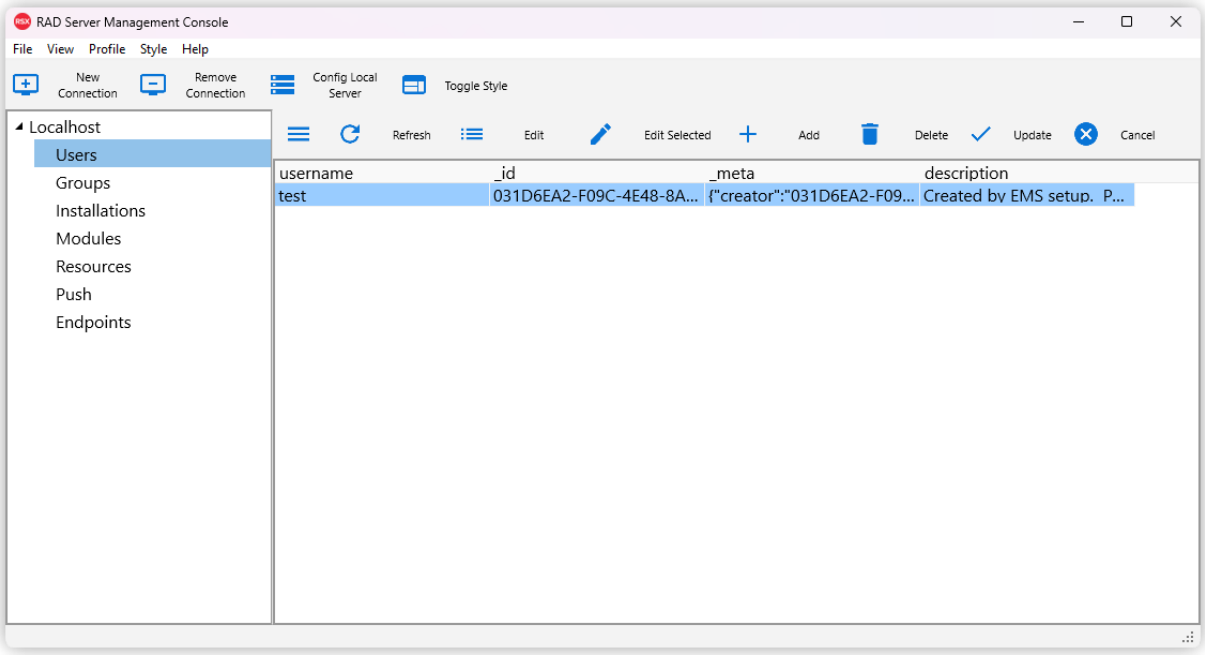
To create a new profile/connection simply press the button “New Connection” or access the menu “要建立新的設定檔/連接,只需按下 “New Connection” 按鈕或存取功能表 “Profile/New Profile…” . 配置非常簡單，只需要連接到主機的基本細節.

> □ 警告
>
> 如果 RAD 伺服器開發未執行或未在生產環境中,則無法建立與 RAD 伺服器實例的連線（Windows 上的 EMSDevServer.exe 和 Linux 上的 EMSDevServerCommand）.

## 管理使用者和群組

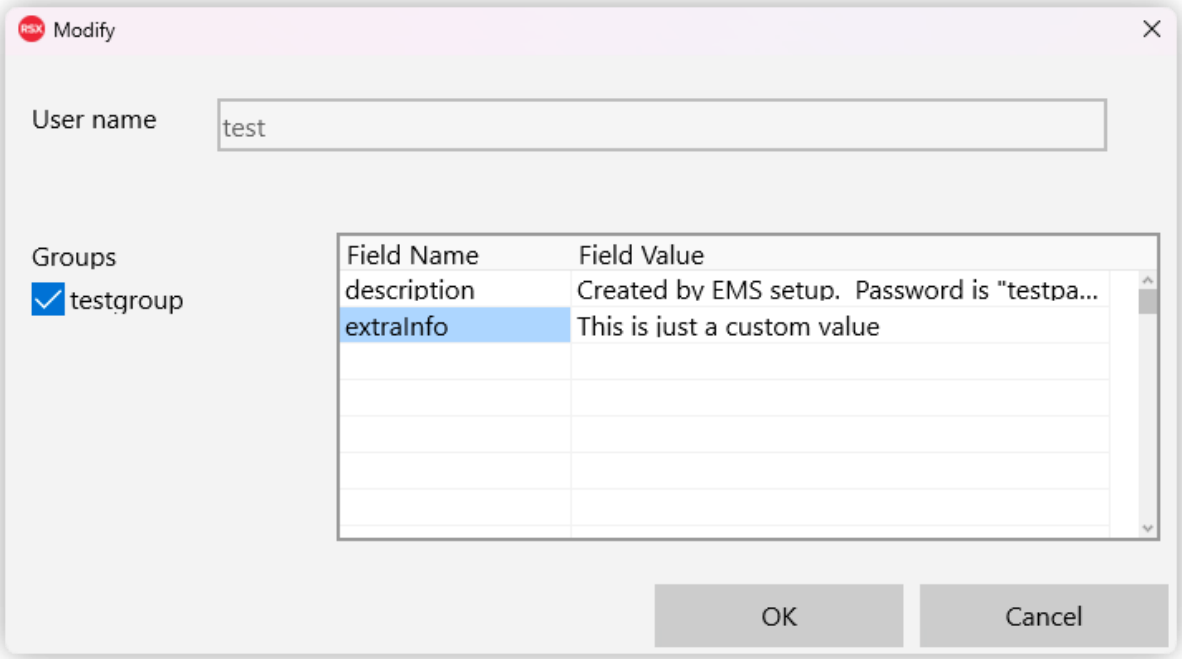
連接到所需實例後，左側將顯示所有可用資源的清單。在這種情況下，我們將討論“Users”和“Groups”.




使用 RAD 伺服器控制台建立的使用者列表

若要建立、編輯或刪除使用者或群組，只需點擊左側的特定部分，您將看到一個帶有多個選項的工具列: Edit, Add, Delete 等等.

作為範例，讓我們看一下"modify user"視窗（儘管"create user"功能幾乎相同）:




“Modify User” 視窗

竅門

建立使用者時，密碼欄位可見，但編輯使用者時無法存取. 若要修改使用者的密碼，只需在自訂欄位部分新增"password"欄位並將新密碼作為值.

建立新群組的過程也非常簡單。按一下“groups”左側部分，然後按一下“Add”。在此視窗中，可以指定哪些使用者將成為群組的成員.

## 深入了解 RSConsole

如果您有興趣了解 RAD Server 控制台在幕後如何運作，該 FMX 應用程式的所有原始程式碼都包含在 RAD Studio 中。你可以在這條路徑上找到它:

C:\Program Files (x86)\Embarcadero\Studio\XX.0\source\data\ems\rsconsole

儘管它是一個相當複雜的應用程式，但它是查看集成 RAD 伺服器 API 提供遠端訪問的所有可能性以及每個操作發送的請求類型的完美場所.

# 12

# 使用 OpenAPI 記錄和測試您的端點 (Swagger)



## 什麼是 OpenAPI/Swagger 以及為什麼要使用它?

P 以前稱為 Swagger 規範, OpenAPI 是一個很好的規範，可以以 JSON/YAML 格式動態建立 API 文檔，使用它，開發人員可以建立其 API 的 Swagger 實例. 憑藉最大的 API 工具生態系統之一，支援 Swagger 的 API 提供互動式文件、客戶端 SDK 生成和可發現性.

RAD Studio 讓開發人員直接使用屬性（包括摘要、參數和詳細資訊/回應）記錄利用這些規範的 RAD 伺服器.

□ 備註

如果您使用 C++ Builder，您可能知道 C++ 本身不像 Delphi 那樣支援屬性，但如前面的章節所示，有一個解決方法可以在 C++ Builder 中提供此功能。此方法也用於以該語言記錄您的 API.

## 將 Swagger UI 嵌入 RAD 伺服器

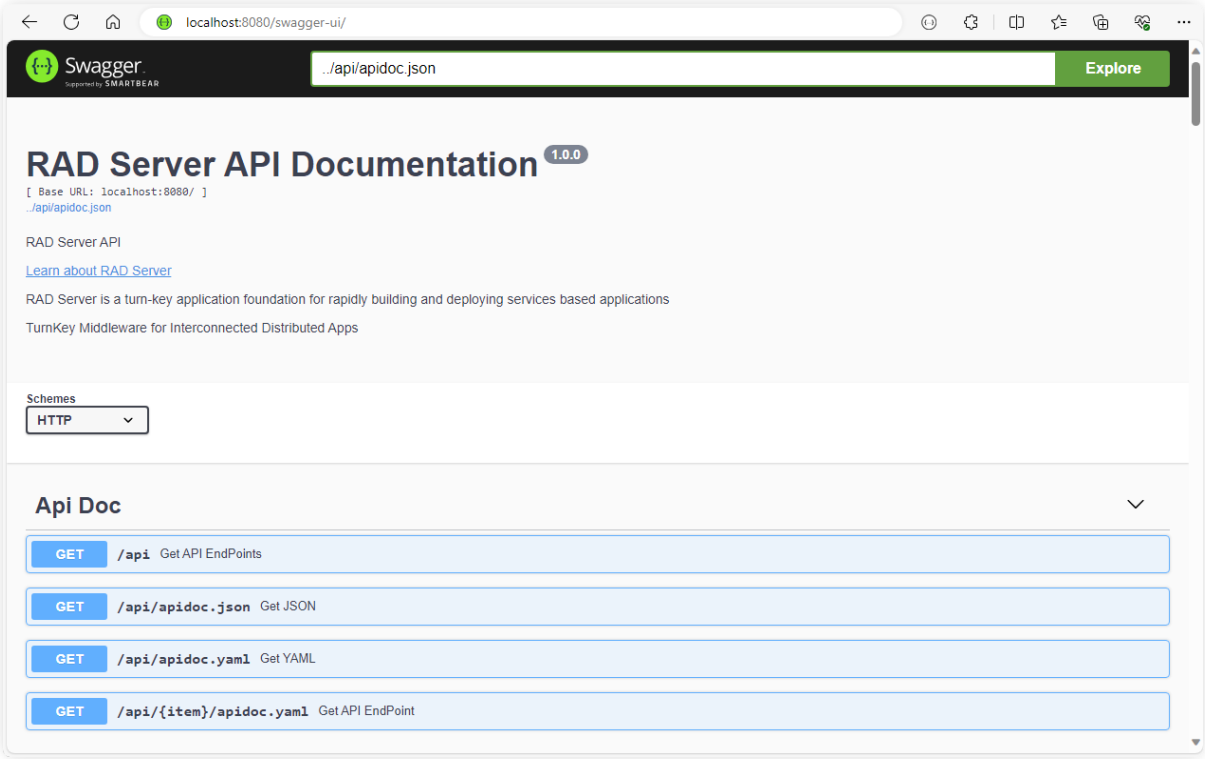
RAD Server 可以提供靜態檔案（例如：託管網站的前端），但此功能也可用於直接在 RAD Server 中託管 Swagger UI.

為此，我們只需訪問 EMSServer.ini 檔案並搜尋鍵值 [Server.PublicPaths]。在此之下，您將找到引用 Swagger UI 路徑位置的特定行。取消註解該行並指示正確的路徑.

正如前面章節中提到的，該文件可以在您的開發機器的路徑中找到：C:\Users\Public\Documents\Embarcadero\EMS\emsserver.ini

```
[Server.PublicPaths]
...
;# The following entry specifies the root path for swagger-ui
Path3={"path": "swagger-ui", "directory": "C:\\swagger-ui\\", "default": "index.html",
"extensions": ["css", "html", "js", "map", "png"], "charset": "utf-8"}
```

定義後，啟動 RAD 伺服器並存取 URL/swagger-ui/。範例: http://localhost:8080/swagger-ui/




使用 apidoc.json 定義的 Swagger 首頁

如我們在螢幕截圖中看到的，swagger UI 會自動引用 JSON 規格。我們可以將其更改為 /api/apidoc.yaml ，並且將顯示相同的規範.

此外，RAD 伺服器中嵌入的所有可用端點均已完整記錄在產生的規格文件中.

□
竅門

Swagger UI 檔案隨 RAD Studio 提供在路徑中："C:\Program Files (x86)\Embarcadero\Studio\XX.0\ObjRepos\en\EMS\swagger-ui".它們也可以從官方儲存庫下載.

# 建立自訂文檔

## 範例

RAD Studio 提供了一個很好的範例，可以詳細了解所有可能性以及每個可用屬性.

此範例適用於 Delphi 和 C++，可以在路徑中找到:

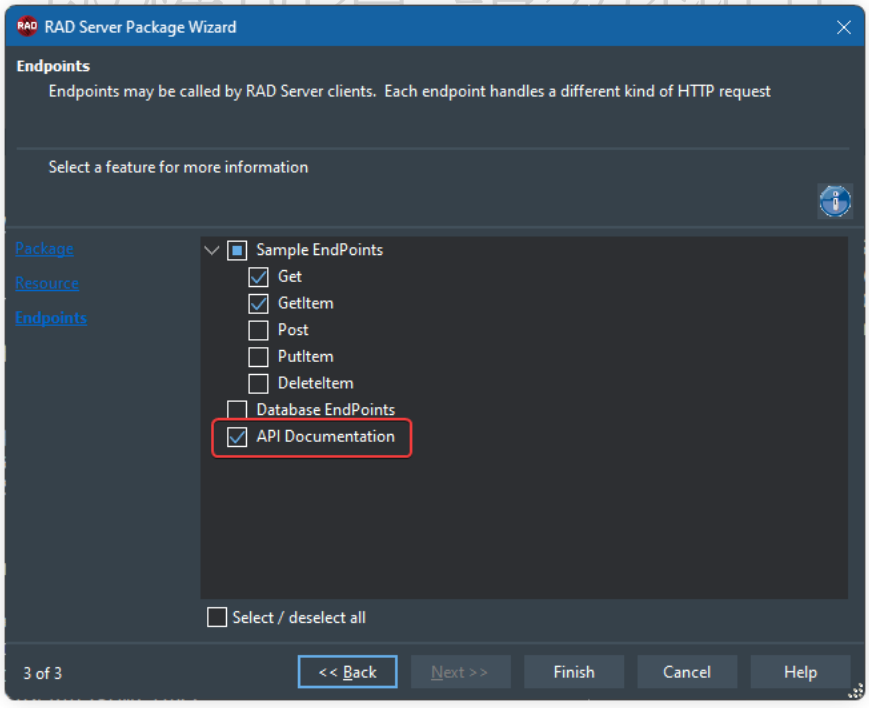
**Delphi:**
C:\Users\Public\Documents\Embarcadero\Studio\XX.0\Samples\Object Pascal\Database\EMS\APIDocAttributes

**C++:** C:\Users\Public\Documents\Embarcadero\Studio\XX.0\Samples\CPP\Database\EMS\APIDocAttributes

> □ 備註
>
> 本章的範例專案是 RAD Studio 提供的範例專案的更新版本. 由於正在使用新的多字串文字功能，因此該項目將僅與 12 Athens 或更高版本相容.

另一種選擇是使用精靈建立新的 RAD 伺服器並選擇 API 文件複選框。在這種情況下，基本屬性將自動填充.





自動建立 API 文件範例的精靈選項

## EndPointRequestSummary

方法描述:

```
[EndPointRequestSummary('ATags', 'ASummary', 'ADescription', 'AProduces', 'AConsume')]
```

- **Tags**: 定義一個標籤.
- **Summary**: 方法標題.
- **Description**: 方法說明.
- **Produces**: API 可以產生的 MIME 類型。這對於所有 API 都是全域的，但可以在特定 API 呼叫上被覆寫。該值必須如 Mime 類型中所述.
- **Consume**: API 可以使用的 MIME 類型。這對於所有 API 都是全域的，但可以在特定 API 呼叫上被覆寫。該值必須如 Mime 類型中所述.

EndPoint 的 GET 方法的描述聲明範例:

**Delphi**



```
[EndPointRequestSummary('Sample Tag', 'Summary Title', 'Get Method Description',
'application/json', '')]
procedure Get(const AContext: TEndpointContext; const ARequest: TEndpointRequest;
const AResponse: TEndpointResponse);
```

**C++**

```
std::unique_ptr<EndPointRequestSummaryAttribute> RequestSummary(new
EndPointRequestSummaryAttribute("Sample Tag", "Summary Title", "Get Method
Description", "application/json", ""));
attributes->RequestSummary["Get"] = RequestSummary.get();
```

## EndPointRequestParameter

請求中使用的參數的描述.

唯一參數由名稱和位置的組合定義.

有五種可能的參數類型: Path, Query, Header, Body, 和 Form.

```
[EndPointResponseDetails('ACode', 'ADescription', 'AType', 'AFormat', 'ASchema',
'AReference')]
```

- **ParamIn**: 參數的位置: Path, Query, Header, Body 或 Form.
- **Name**: 參數的名稱。參數名稱區分大小寫.
  - 當參數位置為 ‘Body’ 時，名稱必須為'body'.
  - 當參數位置為'Path'時，名稱必須對應於 Paths 物件中路徑欄位中關聯的路徑段.
  - 其餘情況，名稱與 ParamIn 對應.
- **Description**: 參數的簡要說明。這可能包含使用範例。 GFM 語法可用於豐富文本表示.
- **Required**: 確定該參數是否為強制參數。如果參數在 ‘Path’ 中，則該屬性為必填項，其值必須為 True，否則如果是可能包含該屬性，則其預設值為 False.
- **ParamType**: 參數的型別。對於'Body'以外的其他值，該值必須是以下值之一: ‘spArray’, ‘spBoolean’, ‘spInteger’, ‘spNumber’, ‘spNull’, ‘spObject’, ‘spString’, ‘spFile’. 如果 ParamType 為'spFile'，則使用的 MIME 類型必須為 “multipart/form-data” 或 “application/x-www-form-urlencoded” ，且參數必須在 “form-data” 中。對於值 “Body” ，需要 JSONSchema 和 Reference.
- **ItemFormat**: ParamType 的擴展格式: ‘None’, ‘Int32’, ‘Int64’, ‘Float’, ‘Double’, ‘Byte’, ‘Date’, ‘DateTime’, ‘Password’.
- **ItemType**: 如果 ParamType 為'array'，則為必要。它描述了數組陣列中項目的類型.
- **Schema**: 發送到伺服器的原語的架構定義。主體請求結構的定義。如果 ParamType 是’Array’ 或’Object’，則可以以 JSON 和/或 YAML 格式定義模式.
- **Reference**: 發送到伺服器的原語的架構定義。主體請求結構的定義。如果類型是 “'Ayyay'或'Object'，則可以定義模式。例如：'#/definitions/pet'

參數定義範例:

**Delphi**

```
[EndPointRequestParameter(TAPIDocParameter.TParameterIn.Path, 'item', 'Path Parameter
item Description', true, TAPIDoc.TPrimitiveType.spString,
TAPIDoc.TPrimitiveFormat.None, TAPIDoc.TPrimitiveType.spString, '', '')]
```



**C++**

```
ResponseParameter.reset(new
EndPointRequestParameterAttribute(TAPIDocParameter::TParameterIn::Path, "item", "Path
Parameter item Description", true, TAPIDoc::TPrimitiveType::spString,
TAPIDoc::TPrimitiveFormat::None, TAPIDoc::TPrimitiveType::spString, "", ""));
attributes->AddRequestParameter("GetItem", ResponseParameter.get());
```

## EndPointResponseDetails

請求回應的描述.

```
[EndPointResponseDetails('ACode', 'ADescription', 'AType', 'AFormat', 'ASchema',
'AReference')]
```

- **Code**: 回應代碼.
- **Description**: 回應代碼說明.
- **PrimitiveType**: 回傳的 原始資料類型 型態. Swagger 規範中的原始資料類型是基於 JSON-Schema Draft 4 支援的類型. JSON Schema 為 JSON 值定義了七種基本型別: ‘spArray’,‘spBoolean’, ‘spInteger’, ‘spNumber’, ‘spNull’, ‘spObject’, ‘spString’. 請參考 JSON 模式原始類型. 附加的原始資料型態, ‘spFile’, 由參數物件和回應物件使用來將參數類型或回應設定為文件.
- **PrimitiveFormat**: 傳回的原始資料的格式，例如: ‘None’, ‘Int32’, ‘Int64’, ‘Float’, ‘Double’, ‘Byte’, ‘Date’, ‘DateTime’, ‘Password’.
- **Schema**: 發送到伺服器的原語的架構定義。主體請求結構的定義。如果 ParamType 是'Array'或'Object'，則可以以 JSON 和/或 YAML 格式定義模式.
- **Reference**: 發送到伺服器的原語的架構定義。主體請求結構的定義。如果類型是'Array'或'Object'，則可以定義模式。例如： “#/definitions/pet”

回應定義範例:

**Delphi:**

```
[EndPointResponseDetails(200, 'Ok', TAPIDoc.TPrimitiveType.spObject,
TAPIDoc.TPrimitiveFormat.None, '', '#/definitions/EmployeeTable')]
[EndPointResponseDetails(404, 'Not Found', TAPIDoc.TPrimitiveType.spNull,
TAPIDoc.TPrimitiveFormat.None, '', '')]
```

**C++:**

```
ResponseDetail.reset(new EndPointResponseDetailsAttribute(200, "OK",
TAPIDoc::TPrimitiveType::spObject, TAPIDoc::TPrimitiveFormat::None, "",
"#/definitions/EmployeeTable"));
attributes->AddResponseDetail("GetItem", ResponseDetail.get());
ResponseDetail.reset(new EndPointResponseDetailsAttribute(404, "Not Found",
TAPIDoc::TPrimitiveType::spNull, TAPIDoc::TPrimitiveFormat::None, "", ""));
attributes->AddResponseDetail("GetItem", ResponseDetail.get());
```

## EndPointObjectsDefinitions

可以以 JSON 和/或 JSON 格式定義物件定義。讓我們來看一個例子:

**Delphi**

```
[EndPointObjectsJSONDefinitions(cJSONDefinitions)]
[EndPointObjectsYAMLDefinitions(cYamlDefinitions)]
```

**C++**

```
attributes->YAMLDefinitions["SampleAttributesCpp"] = initYamlDefinitions();
attributes->JSONDefinitions["SampleAttributesCpp"] = initJSONDefinitions();
```

但這些定義是如何指定的呢？由於篇幅較長，我們在這裡只會看到一個簡短的 Delphi 範例。對於完整的範例，您可以查看 RAD Studio 提供的範例項目.

**Delphi**

```
cYamlDefinitions = '''
#
 PutObject:
   properties:
     EMP_NO:
       type: integer
     FIRST_NAME:
       type: string
     LAST_NAME:
       type: string
#
''';
```

□
竅門

*請記住，定義 YAML 檔案對於注意縮排至關重要。這種格式沒有像 JSON 那樣使用大括號的物件結構. 在 YAML 中，縮排定義了結構的層次結構，因此在使用多行字串的情況下，請確保在何處定義結束三引號.*

## 定義 EMSDatasetResource 的屬性

EMSDatasetResource 是一個很棒的元件，可以幫助非常快速地創建標準 CRUD 端點，但是由於該元件在幕後完成了大部分繁重的工作，因此最初我們無法存取每個端點來定義我們想要的屬性.

預設情況下，僅定義通用 EndPointRequestSummary 屬性，RAD 伺服器將使用 {id} 作為主鍵建立通用 CRUD 定義，且端點將無法測試。為了解決這個問題，我們不僅可以定義主鍵名稱，還可以定義每個操作及其自訂詳細信息.

**Delphi**

```
    [EndPointRequestParameter(
       'Get',
       TAPIDocParameter.TParameterIn.Path,
       'CUST_NO', // Param name
       'Customer number', //desc
       true, // required
       TAPIDoc.TPrimitiveType.spInteger,
       TAPIDoc.TPrimitiveFormat.Int64,
       TAPIDoc.TPrimitiveType.spInteger,
       '', // Schema
       '')] // Reference
```

**C++**

```
std::unique_ptr<EndPointRequestParameterAttribute>
        ResponseParameter(new EndPointRequestParameterAttribute(
       TAPIDocParameter::TParameterIn::Path,
       "CUST_NO", // Param name
       "Customer number", // desc
       true, // required
       TAPIDoc::TPrimitiveType::spInteger,
       TAPIDoc::TPrimitiveFormat::Int64,
       TAPIDoc::TPrimitiveType::spInteger,
       "", // Schema
```

```
        "")); // Reference
attributes->AddRequestParameter("dsrCUSTOMER.Get", ResponseParameter.get());
```

在連結到本章的 GitHub 儲存庫中，您可以找到我們在第 3 章中使用的相同專案。. 您可以在 README.md 檔案以及單元的註釋中找到有關所遵循約定的更多詳細信息.

□
竅門

要提供 YAML 和 JSON 規範，需要單獨定義這兩個規範。一個簡單的解決方法是在 YAML 中建立所有定義（更容易閱讀），然後使用線上轉換器或任何 GPT 機器人為您產生等效的 JSON 定義.



# 13

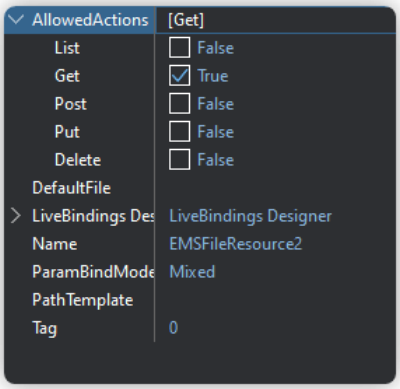
# 文件管理和儲存



在本章中，我們將分析 RAD Server 提供的多種管理文件和存取基礎架構的方式. 此外，我們還將了解如何指定每個端點可以在全域範圍內產生或使用的檔案類型，或者如何在每個端點上以精細的方式定義它.

## TEMSFileResource

TEMSFileResource 組件的工作方式與 TEMSDatasetResource 非常相似，它可讓我們的生活更加輕鬆。它抽象化了文件管理的大部分業務邏輯，所有需要的程式碼都由屬性來完成.




自動建立 API 文件範例的精靈選項

**AllowedActions**: 在預設情況下，只有 GET 處於活動狀態，但可以自訂元件以允許任何操作，以便可以取得檔案清單並上傳、更新或刪除文件.

**DefaultFile**: 如果 GET 檔案請求不包含指定檔案的參數，則傳回定義為 DefaultFile 的文件.

**PathTemplate**: 這是一個簡單而強大的屬性。最基本的範例可能是像 *c:\temp\{id}*這樣的值. 這定義了儲存該元件處理的檔案的絕對路徑, {id} 是一個通配符，我們將在屬性中使用它作為包含檔案名稱的參數. 在此屬性中使用大括號允許選擇建立更複雜的路徑，例如為子資料夾甚至擴充檔案定義通配符: *c:\uploads\{folder}\{file}.{extension}*.在這種情況下，需要在屬性中定義 3 個參數：資料夾、檔案和副檔名。我們稍後會看到一個例子.

> □ 備註
>
> *通常，PathTemplate 將根據環境(除錯或發布)具有不同的值。建議使用編譯指令更改此屬性.*

## 範例

對於 Delphi，在資料模組中的 EMSFileResource 定義之前新增三個 ResourceSuffix 定義項目.

**Delphi**

```
TFilesResource1 = class(TDataModule)
[ResourceSuffix('list', './')]
[ResourceSuffix('get', './{id}')]
[ResourceSuffix('post', './')]
EMSFileResource1: TEMSFileResource;
```

**C++**

```
static void Register()
{
	std::unique_ptr<TEMSResourceAttributes> attributes(new
TEMSResourceAttributes());
	attributes->ResourceName = "test";
	attributes->ResourceSuffix["PostUpload"] = "./upload";
	attributes->ResourceSuffix["EMSFileResource1"] = "./fileResource";
	attributes->ResourceSuffix["EMSFileResource1.List"] = "./";
	attributes->ResourceSuffix["EMSFileResource1.Get"] = "./{id}";
	attributes->ResourceSuffix["EMSFileResource1.Post"] = "./";
	RegisterResource(__typeinfo(TDataResource1), attributes.release());
}
```

在此範例中，我們定義了 3 個操作：list、get 和 post，但 put 和 delete 也可以按照相同的模式實現. 端點 test/fileResource/ 現在允許列出"PathTemplate"屬性中的所有字段欄位，使用 {id} 通配符存取一個特定文件

或上傳新文件. 重要的是要知道要使用此元件上傳文件，正文必須包含文件本身的二進位輸出. 不能使用多部分表單上傳多個檔案（我們將在稍後看到如何完成此操作的範例）.

> □ 警告
>
> *要在 C++ 上使用 TEMSFileResource ，您必須在"require"部分中新增函式庫 emsserverresource.bpi。你可以在路徑中找到這個文件: C:\Program Files (x86)\Embarcadero\Studio\XX.0\lib\{Platform}\release\emsserverresource.bpi*

## 從程式碼管理文件

對於稍微複雜一點的應用、TEMSFileResource 可能不敷使用，那麼您也可以透過程式碼來處理存取文件. 每個請求上的 ARequest 參數允許存取正文中的檔案。這些必須以多部分表格形式發送。讓我們來看看一些程式碼:

**Delphi**

```
const UPLOAD_PATH = 'c:\uploads';
var lFileName := ARequest.Body.Parts[0].FileName;
var lFile := TFile.Create(TPath.Combine(UPLOAD_PATH, lFileName));
lFile.CopyFrom(ARequest.Body.Parts[0].GetStream, 0);
```



**C++**

```
const System::UnicodeString UPLOAD_PATH = "c:\\uploads";
System::UnicodeString lFileName = ARequest->Body->Parts[0]->FileName;
TStream* lFile = new TStream;
lFile = TFile::Create(TPath::Combine(UPLOAD_PATH, lFileName));
lFile->CopyFrom(ARequest->Body->Parts[0]->GetStream(), 0);
```

在這個基本範例中，我們存取正文的第 0 部分（當然它可以包含多個部分/檔案），並透過將正文流寫入 TFile 變數本身，使用 TFile 類別將其儲存在給定路徑中. 真的就是這麼簡單。在儲存庫範例中，您可以找到一個稍微複雜的情況，其中我們循環遍歷主體的所有可能部分以允許將多個檔案上傳到端點.

## Content-Type HTTP 表頭

RAD Server 對 Content-Type 和基於 Accept 的映射的 EndPoint 屬性的新支援為資源映射提供了更好的支持，而不僅僅依賴 URL. 這意味著您可以將兩種不同的方法對應到相同的 URL 和 HTTP 動詞，同時根據請求傳回不同類型的數據 t.

新增了兩個新的 EndPoint 屬性:

- EndpointProduce: 指定此端點可以產生的 MIME 類型/檔案副檔名作為對 GET 方法的回應。端點選擇將基於 Accept HTTP 請求標頭.
- EndpointConsume: 指定此端點可為 PUT、POST、PATCH 方法使用的 MIME 類型/檔案副檔名。端點選擇將基於 Content-Type HTTP 請求標頭.

本章將展示 RAD 伺服器應用程式資源如何使用 EndpointProduce 屬性進行基於 image/jpeg 和 application/xml MIME 類型的多個 REST Get 請求.

## 一個簡單的範例

使用專案包精靈啟動 RAD 伺服器專案。選擇資料模組檔案類型並將資源名稱設定為 AcceptTypes. 取消選擇所有標準端點並點擊“Finish”按鈕.

在路徑 c:\temp\page 中建立一個資料夾，並將名為 content.jpeg 的任何 jpeg 檔案和名為 content.txt 的文字檔案複製到其中.

**Delphi**

在資源類別的已發佈部分中新增兩個以 ResourceSuffix 和 EndpointProduce 屬性修飾的方法聲明.

```
[ResourceName('AcceptTypes')]
TAcceptTypesResource1 = class(TDataModule)
published
  [ResourceSuffix ('*')]
  [EndpointProduce ('image/jpeg')]
  procedure GetImage(const AContext: TEndpointContext;
    const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
  [ResourceSuffix ('*')]
  [EndpointProduce ('application/xml')]
  procedure GetText(const AContext: TEndpointContext;
    const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
end;
```

在實作部分為 GetImage 和 GetText 端點新增以下程式碼.

```
procedure TAcceptTypesResource1.GetImage(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  fs: TFileStream;
begin
  fs := TFileStream.Create('c:\temp\page\content.jpeg', fmOpenRead);
  AResponse.Body.SetStream(fs, 'image/jpeg', True);
end;

procedure TAcceptTypesResource1.GetText(const AContext: TEndpointContext;
```

```
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  fs: TFileStream;
begin
  fs := TFileStream.Create('c:\temp\page\content.txt', fmOpenRead);
  AResponse.Body.SetStream(fs, 'text/plain', True);
end;
```

**C++**

**ServerUnit.h**

```
// EMS Resource Modules
//---------------------------------------------------------------------------

#ifndef ServerUnitH
#define ServerUnitH
//---------------------------------------------------------------------------
#include <System.Classes.hpp>
#include <System.SysUtils.hpp>
#include <EMS.Services.hpp>
#include <EMS.ResourceAPI.hpp>
#include <EMS.ResourceTypes.hpp>
//---------------------------------------------------------------------------
#pragma explicit_rtti methods (public)
class TAcceptTypesResource1 : public TDataModule
{
__published:
private:
public:
      __fastcall TAcceptTypesResource1(TComponent* Owner);
      void GetImage(TEndpointContext* Acontext,
            TEndpointRequest* ARequest, TEndpointResponse* AResponse);
      void GetText(TEndpointContext* Acontext,
            TEndpointRequest* ARequest, TEndpointResponse* AResponse);
};
#endif
```

**ServerUnit.cpp**

為 GetImage 和 GetText 端點新增以下程式碼。在 Register 函數中，為 GetImage 和 GetText 端點新增 ResourceSuffix(s) 和 EndpointProduce 屬性.

```
//---------------------------------------------------------------------------
#pragma hdrstop

#include "ServerUnit.h"
#include <memory>
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma classgroup "System.Classes.TPersistent"
#pragma resource "*.dfm"
//---------------------------------------------------------------------------
__fastcall TAcceptTypesResource1::TAcceptTypesResource1(TComponent* Owner)
      : TDataModule(Owner)
{
}

void TAcceptTypesResource1::GetImage(TEndpointContext* Acontext, TEndpointRequest*
ARequest, TEndpointResponse* AResponse)
{
      TFileStream * fs =  new TFileStream("c:\\temp\\page\\content.jpeg", fmOpenRead);
      AResponse->Body->SetStream(fs, "image/jpeg", True);
}

void TAcceptTypesResource1::GetText(TEndpointContext* Acontext,   TEndpointRequest*
ARequest, TEndpointResponse* AResponse)
{
      TFileStream * fs =  new TFileStream("c:\\temp\\page\\content.txt", fmOpenRead);
      AResponse->Body->SetStream(fs, "text/plain", True);

}

static void Register()
{
  std::auto_ptr<TEMSResourceAttributes> attributes(new TEMSResourceAttributes());
      attributes->ResourceName = "AcceptTypes";
      attributes->ResourceSuffix["GetImage"] = "*";
      attributes->EndPointProduce["GetImage"] = "image/jpeg";
      attributes->ResourceSuffix["GetText"] = "*";
      attributes->EndPointProduce["GetText"] = "application/xml";
      RegisterResource(__typeinfo(TAcceptTypesResource1), attributes.release());
}

#pragma startup Register 32
```

儲存並建置 RAD 伺服器專案。現在，您可以根據發送的 Content-Type 標頭存取 content.jpeg 和 content.txt.