RAD Server
技术手册

```
{
  "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 架构是现代 API 优先应用程序设计背后的关键驱动力. 本书重点介绍 RAD Studio (Delphi/C++Builder) 中所包含的 RAD 服务器框架,以便用于开发此类平台.

RAD Server 是一个完整的后端 MEAP（行动企业应用程序平台）,支持以任何程序语言进行桌面,行动和 Web 前端开发,本书旨在作为开发人员的权威指南.

MEAP 的好处是,您拥有一个预先建置的云端或本地服务器,它具有许多核心功能(例如推播通知,用户追踪和分析),您可以快速插入这些功能以提供远程数据库和功能的服务.

本书 Embarcadero RAD 服务器指南最初由 David I (2019) 撰写,现已是第二版,由 Antonio Zapater (2023) 修订,其中包括自 RAD 服务器推出以来根据市场需求添加的许多附加功能. 第二版还配有支援每一章的 综合影片系列, 以及 GitHub 上的原始码范例. https://github.com/embarcadero/radserver-docs

videos

您可以访问 Youtube 上提供的与本文链接的所有影片系列. 另外,我们强烈建议从此 GitHub 储存库下载所有范例.

内容

01

什么是 RAD Server? 简介

现今的运算环境不再局限于桌面,装置,服务器或数据中心. 应用程序正在从桌面转移到多个装置,网络边缘连接以及本地,公有和混合云服务. 透过 RAD Server 和 RAD Studio,您可以建立涵盖公司(和客户)广泛的运算需求和业务需求的解决方案.

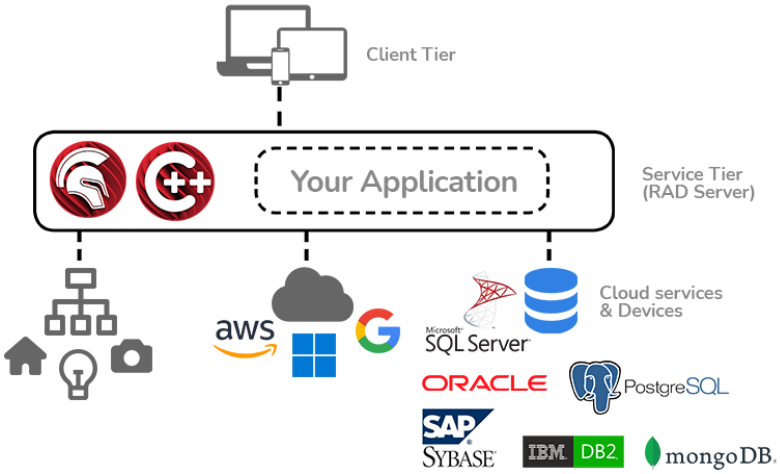
本书将向您展示如何使用 RAD Studio,Delphi 和 C++Builder 企业和架构师版本中提供的 RAD Server 基于 REST 的 API 托管引擎,组件和技术来快速设计,建置,侦错和部署基于服务的多层应用程序.


note

在整个 RAD 服务器文件和原始码中,您将看到对 EMS(企业行动服务)的引用. EMS 是现在的 RAD 服务器产品的原始名称.

RAD Server 概论

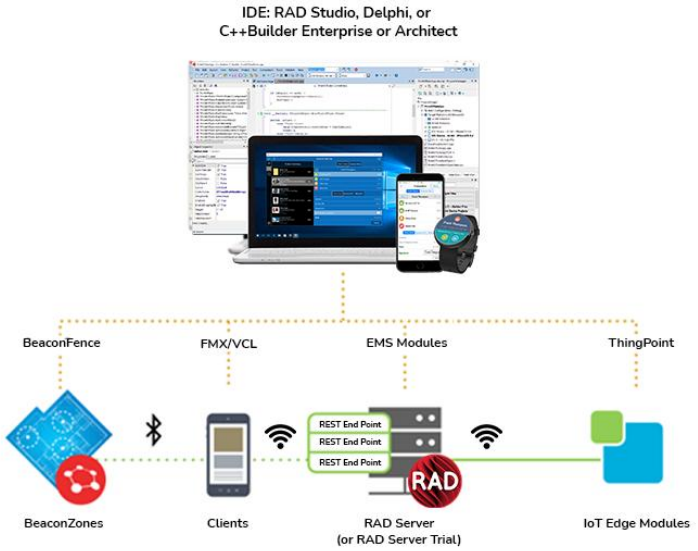
Embarcadero 的 RAD 服务器是使用 Delphi 和 C++Builder 快速建置和部署基于服务的应用程序提供了一站式方案应用程序基础. RAD Server 支持 REST(表述性状态传输)协议,使用 JSON(或 XML)参数传递和传回结果. 您可以发布 API,管理连接到 RAD 服务器的用户和装置,撷取有关应用程序的使用和用户的分析数据,使用 FireDAC 组件连接到本机和企业数据库等等. RAD 服务器也支持用户身份验证,推播通知,地理位置和数据存储.



开发和测试 REST 端点和位置跟踪

借由 RAD Server 的精灵,组件和工具,您可以快速开发新的中间件和后端应用程序,或将现有的 Delphi 和 C++Builder 客户端/服务器应用程序迁移到基于 RAD Server 的应用程序,以便在服务器或云端运行. 您可以为来自桌面,行动装置,控制面板,Web 和其他类型应用程序的 REST 呼叫发布服务端点. RAD Server 附带一整套工具,组件,数据库连接和接口接口,您将在建立服务应用程序时依赖它们.

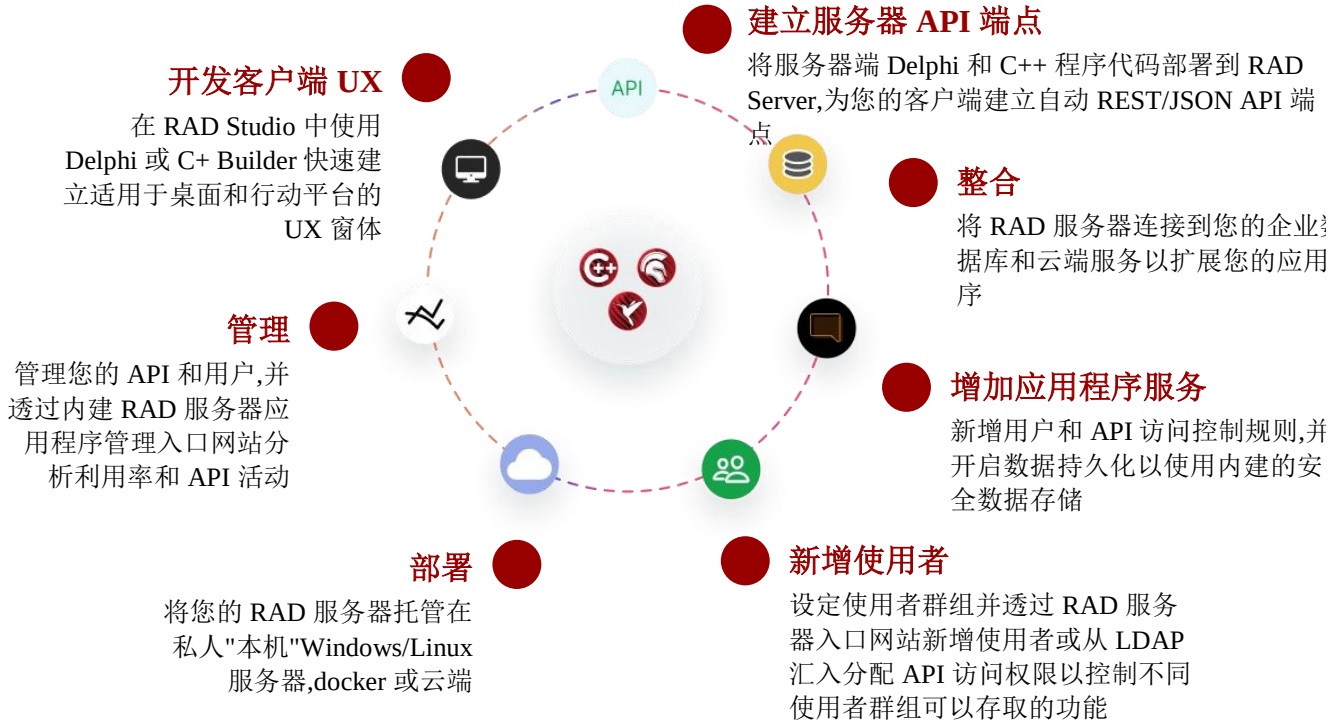
RAD 服务器应用程序可以部署在 Microsoft Windows IIS 和 Apache Web 服务器之上,您可以将基于 Delphi 的服务部署到 Linux Intel 64 位服务器. 有关 Linux 的 C++Builder 支持,请继续关注 Embarcadero RAD Studio 部落格的更新.



开发和测试 REST 端点、位置追踪和 IoT 边缘软件

开发基于 RAD 服务器的应用程序 – 七个关键面

为了建立基于 RAD 服务器的应用程序,下图指导开发人员完成七个面向和开发阶段.



让多层开发变得简单

首先,建立基于服务器 REST/JSON API 的端点(如果需要的话,您也可以使用 XML 而不是 JSON). 接下来,您将透过整合广泛的数据库,云端服务和其他技术来扩展端点.

您可以为用户新增更多应用程序端点并建立 API 访问控制规则. 您可以编写程序代码,利用 RAD Server 的内建安全数据储存来追踪持久性数据. 您可以透过控制台入口网站建立使用者群组并新增用户,并透过基于 LDAP 的 API 服务汇入和验证用户.

开发和侦错应用程序后,您可以在私人本地 Windows 和 Linux 服务器上托管 RAD 服务器应用程序. 您也可以将应用程序移转到 Amazon AWS,Microsoft Azure,Google 和其他云端供货商等云端系统.

应用程序投入生产后,您可以使用内建应用程序管理接口管理对 API 的存取,控制使用者存取并分析端点 API 活动的使用率. 最后,您可以建立 RAD Studio 支持的桌面,行动,Web,控制面板和其他应用程序类型. 您还可以使用 Sencha 的 Ext JS 组件集建立现代 Web 客户端应用程序,并使用其他工具和程序语言建立支持 RAD 服务器应用程序的 REST/JSON 功能的客户端应用程序.

建立 RAD 服务器应用程序的要求

以下部分包含建置,测试和部署 RAD 服务器应用程序的产品和技术要求.除非另有说明,否则"RAD Studio"和 IDE 适用于 RAD Studio,Delphi 和 C++Builder 产品.

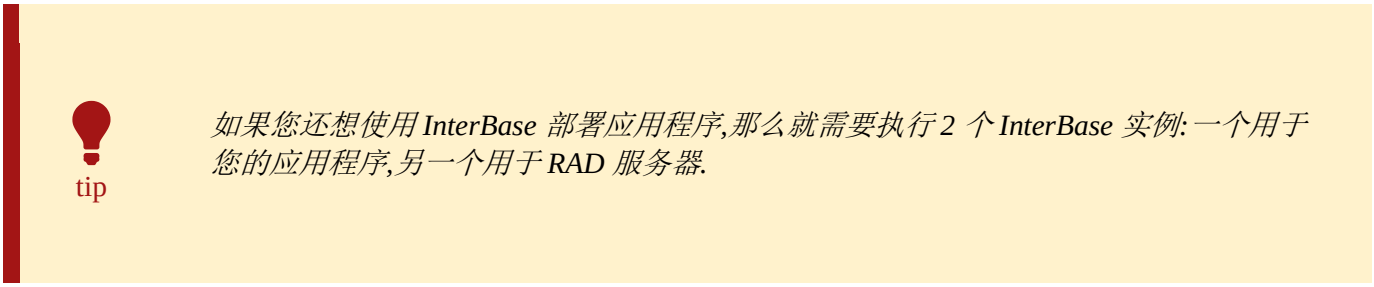
使用 RAD Studio IDE

建置 RAD 服务器应用程序需要具有商业许可证的 RAD Studio 企业或架构师版本. RAD Studio 企业试用版可使用 30 天用于开发和测试.试用版不支持部署到生产环境服务器.

RAD Server 测试和部署授权

30 天 RAD Studio 免费试用版包括 RAD Server 5 个用户开发试用版. RAD Server 部署授权包含在 RAD Studio 的 企业和架构师商业版本中. RAD Studio 企业版包括 RAD Server 的单一站点部署授权,而对于有效更新订阅 RAD Studio 架构师版本的客户则包含多站点部署授权. 自 RAD Studio Alexandria 版本起,企业和架构师版都可以选择在多站点环境中部署 RAD Server Lite.

RAD Server 需要 InterBase 加密数据库作为在生产环境中部署应用程序的一部分. 您需要使用有效的 RAD 服务器授权才能安装此版本的 InterBase.



核心 RAD 服务器功能综述

RAD Server 为开发人员提供了广泛的功能来建立基于 REST 的服务应用程序. RAD Server(以前称为 EMS)首次在 RAD Studio 版本 XE7 中引入. 自第一个版本以来,添加了增强功能和新功能,以满足开发人员的需求并添加对新平台,架构和技术的支持.

核心功能

以下是您在建立基于服务的应用程序时需要利用的一些 RAD Server 核心功能的列表.

- **REST 端点 发布** – RAD Server 为您的应用程序后端 API 和服务实现一站式方案基础. RAD Server 提供易于使用的 API 来发布您的业务逻辑. Delphi 或 C++ 程序代码可以作为 API 托管,并作为 REST/JSON 端点自动发布,由 RAD 服务器测量和管理.端点发布功能包括:
  - 访问控制 – 您可以透过身份验证设定对所有应用程序 API 的群组和使用者层级访问权限,并控制谁有权存取应用程序的 API 功能. 建立您自己的使用者和群组或从 LDAP 基础架构自动汇入它们.
  - API 分析 – 所有 REST API 端点活动都会被记录和测量,以进行可靠的统计追踪和分析. 您可以每天,每月和每年分析使用者,API 和服务活动,以深入了解应用程序的使用情况. 您也可以筛选所有资源的活动或按特定群组,用户,装置安装等筛选活动. 您也可以将分析结果汇出到 CSV 文件,以便使用其他工具进行额外分析.
  - 桌面,行动装置和 Web 客户端应用程序 – RAD Server 上托管的所有 C++ 和 Delphi 程序代码均作为 REST/JSON 端点发布,可供多个平台上的任何类型的客户端应用程序使用,以实现极高的灵活性和面向未来的需求.
- **整合中间件** – RAD Server 提供多种开箱即用的整合功能,可连接到外部服务器,应用程序,数据库,智能型装置,云端服务和其他平台.整合能力包括:
  - 企业数据 – RAD 服务器为所有流行的企业 RDBMS 服务器提供高效能的内建连接. 数据库连接使用 FireDAC 组件和函式库,可以轻松连接各种来源的数据.

  - 云端服务 – 透过 RAD Server，您可以轻松整合来自各种云端,社交和 BaaS 平台(例如 Google,Amazon,Facebook,Kinvey 等)的 REST 云端服务.

- **应用程序服务** – RAD 服务器包含一系列随时可用的内建服务来为您的应用程序提供支持. RAD Server 包括用户目录服务和用户管理、推播通知、用户位置追踪和内建数据储存等核心功能.其中一些应用程序和设备服务包括:

  - 推播通知 – 使用 RAD Server,您可以向应用程序用户及其装置发送程序设计或按需求通知. RAD Server 目前支持推播通知系统,包括 Apple 推播通知服务 (APN) 和 Google FireBase 云端讯息传递 (FCM).您也可以编写自定义程序代码来连接其他推播通知系统.

  - 内建安全资料存储 – 借助 RAD Server 对保护 InterBase 服务器加密数据储存的支持,您可以使用内建 APIS 来储存和检索 JSON 数据,而无需单独的数据库服务器.

  - 使用者/群组管理 – 使用 RAD 服务器 API,您可以建立和管理使用者、使用者群组,并透过 RAD 服务器控制台 (RSConsole.exe) 控制存取.整合您的 ActiveDirectory (LDAP) 或开发您自己的自定义身份验证中间件.

  - 用户位置/邻近度 – 您的 RAD 服务器应用程序可以利用 RAD Studio 对 GPS、信标和信标围栏技术的支持. RAD 服务器应用程序可以追踪用户在室内和室外的移动,并在使用者进入和退出自定义信标区域或接近指定信标点时响应接近事件.

  - 静态文件提供者 – 将 URL 对应到文件夹并传回 HTML、JS、CSS、图像等档案的内容.这在小型部署(便如：使用 RAD Server Lite)或开发环境中非常方便.

- **API 文件** – 使用属性和内建 Swagger OpenAPI 整合轻松建立 API 文件. 将 Swagger UI 嵌入 RAD Server 本身或透过自动产生的 YAML 和 JSON 档案在远程实例中配置它.

- **方便部署** – RAD 服务器易于开发,部署和操作,非常适合 ISV 和 OEM 建置可重新部署的解决方案.部署在 Windows,Linux 或 Docker 上.

请参考

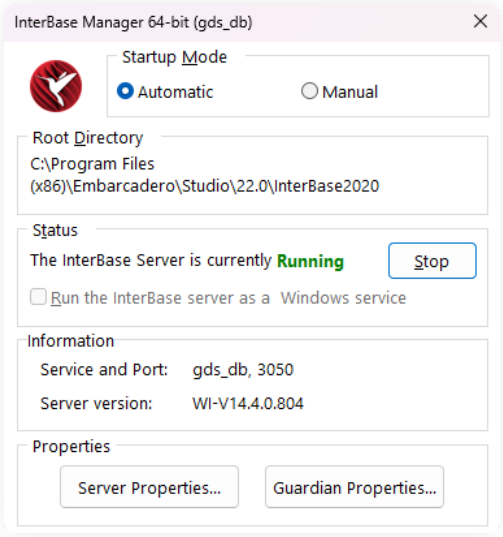
有关 RAD Studio 安装和基于 RAD Server 的应用程序部署的最新更新信息,请参阅以下 Embarcadero 在线链接.

- RAD 服务器产品概述
- RAD Studio 安装说明
- RAD Studio 和 RAD Server 支持的目标平台
- 生产环境的 RAD 服务器数据库要求
- RAD Studio 的平台状态页面
- InterBase
- FireDAC
- FireDAC 支持的数据库
- RAD Studio 企业行动服务
- RAD Studio 产品菜单(PDF - 查看 RAD 服务器章节说明)
- Swagger 开放 API
- EMS 推播通知
- Apple 推播通知服务 (APN)
- Firebase 云端讯息传递 (FCM)

02

使用 RAD 精灵建立“Hello World”

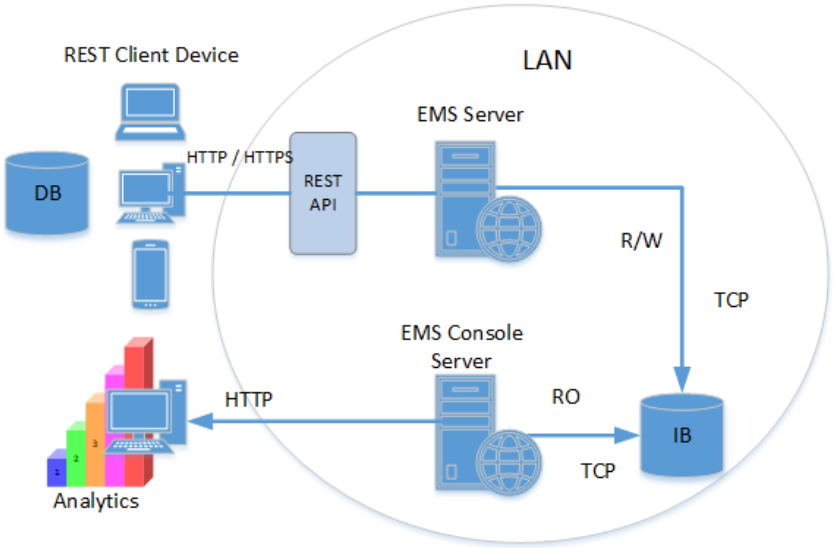
是时候开始程序设计了.在本章中,您将学习如何使用 Delphi 和 C++Builder 建立第一个基于 RAD 服务器的服务应用程序. 在开始之前,您需要确保您的 InterBase 数据库服务器正在执行. RAD 服务器使用 InterBase 数据库来储存用户信息,用户群组,分析,注册设备,版本信息,注册边缘模块,推播通知讯息等.



建构基于 REST 的服务

RAD Server 包括企业行动服务 (EMS) 并提供可在云端或本地端托管的行动企业应用程序平台 (MEAP). 开发人员可以使用 RAD Server 公开自定义 REST API,并使用 FireDAC 数据存取库和组件存取企业数据库数据.

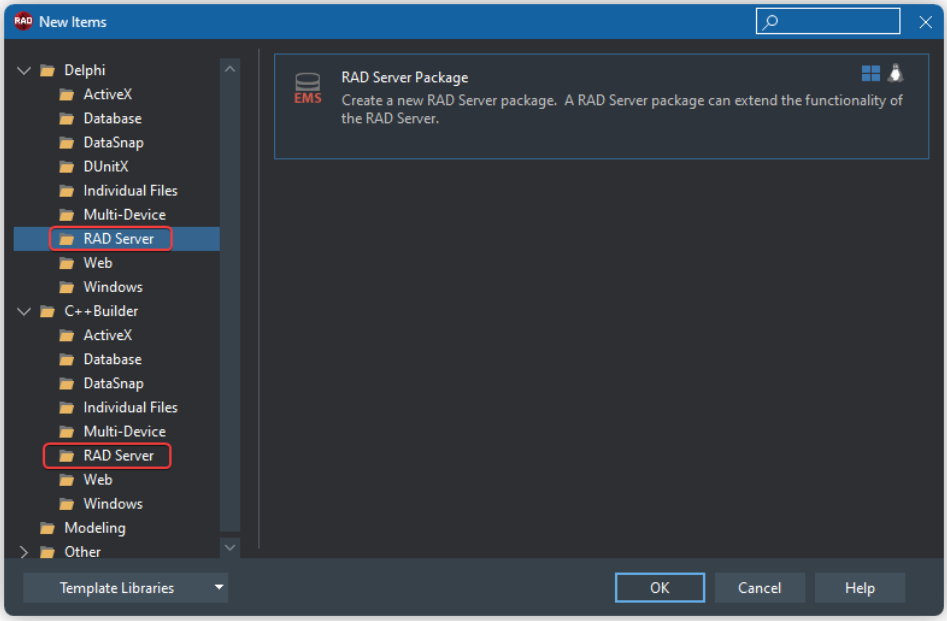
RAD Server 为开发人员提供了全面的解决方案,包括远程数据库存取,用户追踪,设备应用程序管理,使用分析等. 与其他解决方案相比,RAD Server 包含一个预先建置的应用程序服务器,支持自定义套件的整合. 这些自定义套件可以输出数据集,业务逻辑和其他基于 REST 的资源. 组件还可用于行动,Web 和桌面应用程序程序代码来存取 RAD 服务器资源.



RAD 服务器 REST API 架构

使用 RAD 服务器项目精灵

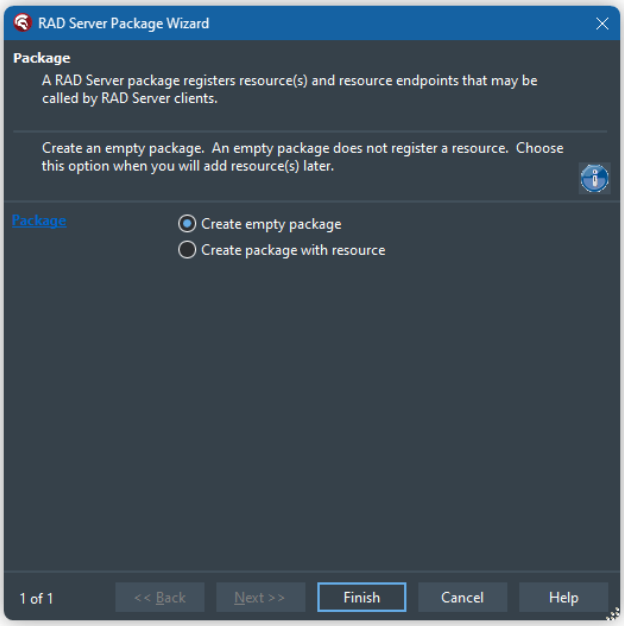
最快的入门方法是使用"新建项目"选单(File | New | Other⋯)并选择 Delphi 或 C++Builder 的 RAD Server | RAD Server Package 精灵.



适用于 Delphi 和 C++ 的 RAD 服务器项目精灵选项

选择 RAD Server Package 专案. 将出现一个精灵来帮助建立起始项目. 在第一页上,选择精灵如何建立将出现在 RAD 服务器应用程序中的资源和端点. TRAD 服务器套件精灵提供了两种继续选择.

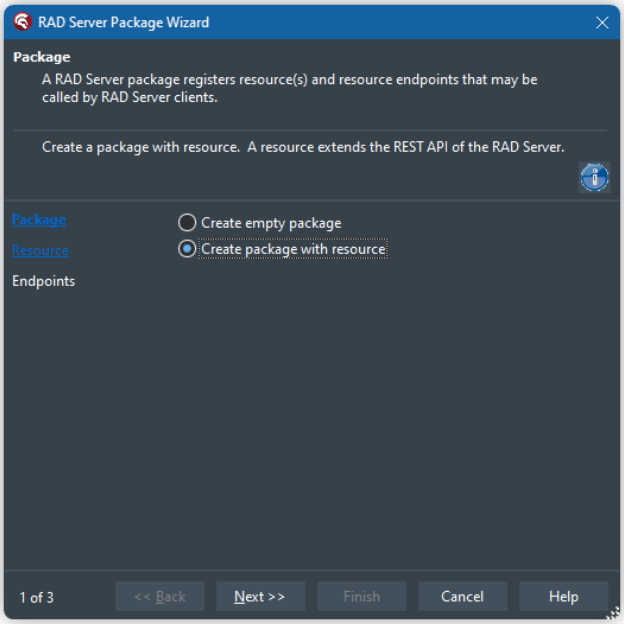
选择 1：建立一个不注册资源的空白套件. 如果您稍后才要新增资源,使用此选择,将建立起始主库并建立一个套件项目.



建立一个空的 RAD 服务器套件

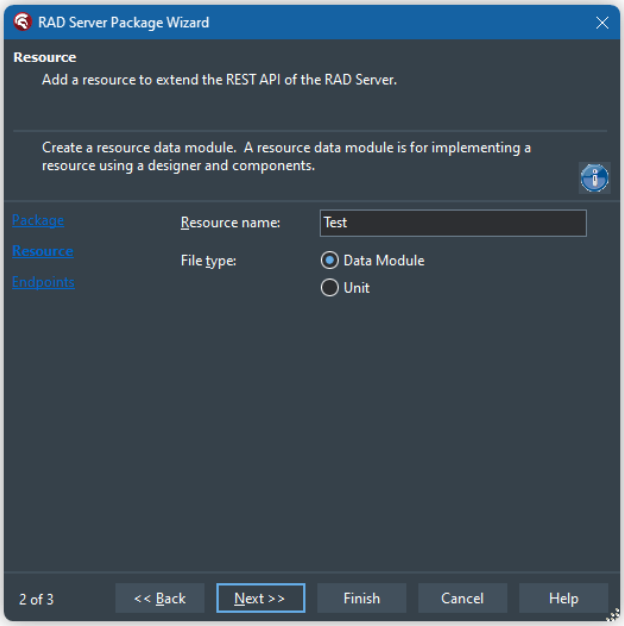
点击「完成」按钮将建立起始项目以进行额外的开发工作,以便建立最终的 RAD 服务器应用程序.

选择 2:使用扩充 RAD 服务器 REST API 的资源建立套件封包. 单击「下一步」按钮,将出现两个附加精灵步骤,以协助建立套件项目,资源和端点.要建立第一个 RAD 服务器项目,请做此选择.



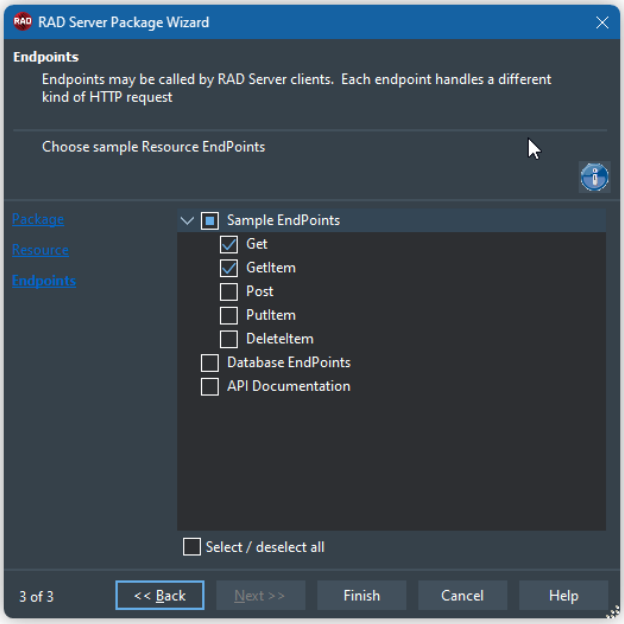
建立基于资源的 RAD 服务器封包

在精灵的第二页上,将资源名称设定为"Test" 文件类型单选按钮提供两个选项：1）建立用于在程序代码中实现资源的程序单元,2）建立用于使用 IDE 设计器,组件和程序代码编辑器实现资源的数据模块.对于第一个 RAD 服务器应用程序,请选择使用数据模块.



RAD 服务器套件精灵第 2 页 - 设定资源名称与文件类型

点选下一步按钮以建立一组起始端点.



RAD 服务器套件精灵第 3 页 - 选择起始端点

在精灵第三页上,保留建议的端点,如上图所示:Get (REST GET) 和 GetItem(REST GET 在 URL 末尾带有标识要取得的项目) 并且取消选取"API 文件" .若要建立您的起始项目,请单击"完成"按钮.

note

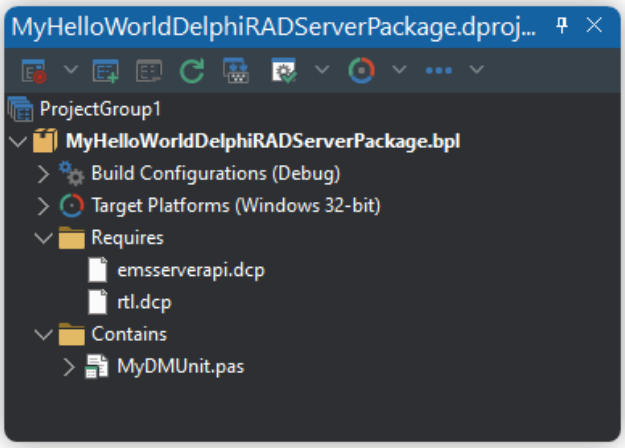
精灵中有两个额外选项:数据库端点(使用 FireDAC 将数据库链接到端点)和 API 文件(Swagger OpenAPI).我们将在接下来的章节中更详细地讨论这些内容.

使用精灵完成后,您将返回 IDE.首先要做的是保存项目.对于 C++ 和 Delphi 数据模块,请使用名称"MyDMUnit".对于 C++ 项目和套件,请使用名称"MyHelloWorldCppRADServerPackage".对于 Delphi 项目和套件,使用名称"MyHelloWorldDelphiRADServerPackage".

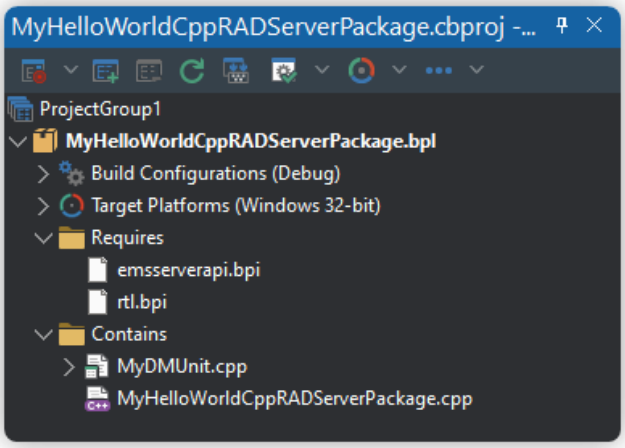
精灵 RAD 服务器项目和原始码

创建的项目程序代码非常少,并且只有几个连结到每个端点的方法. RAD Studio 会自动填入一些预设程序代码,我们将对其进行一些调整,使其更加 Hello World 化.

Delphi 和 C++ 上的专案应如下所示.创建的 DataModule 将是空白的,其中没有任何组件.在截图之后您可以发现自动产生的范例程序代码中所做的修改.



产生的 Delphi 项目



产生的 C++项目

备注

本书中使用的所有源代码和范例都托管在 GitHub 上并区分为章节.我们强烈建议您下载整个储存库,以便更好地遵循本书的进度.

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;
```

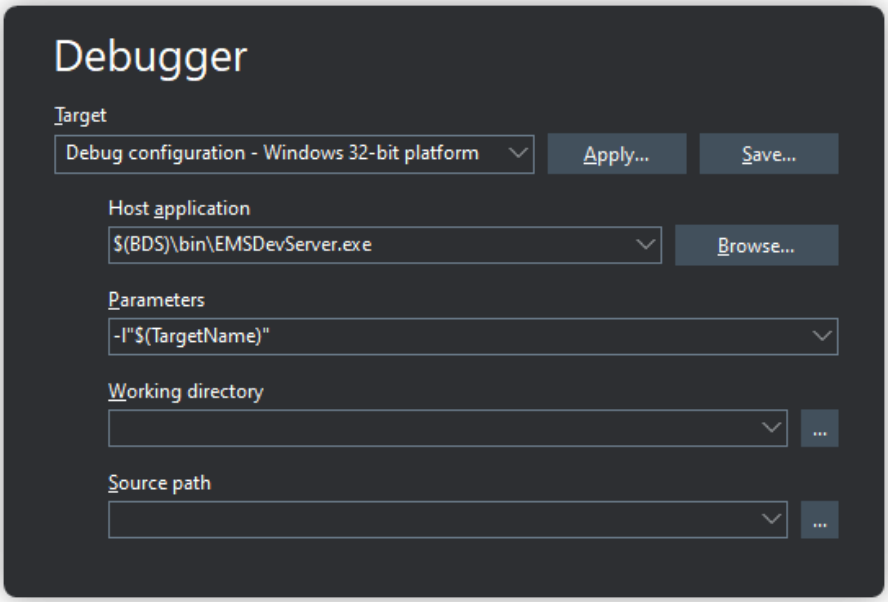
MyDMUnit.cpp:

```
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);
}
```

为您的第一个应用程序设定 RAD 服务器

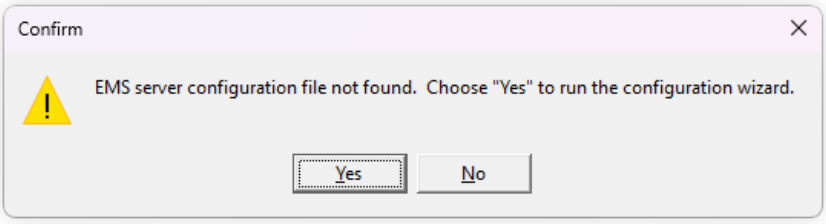
现在您已经使用精灵建立了第一个 RAD 服务器应用程序,您可以使用 IDE 来编译和测试该您的应用程序. IDE 使用 EMSDevServer 作为主机可执行文件(EMDDevServer.exe)开始执行,其参数为要加载的套件文件, 在此处也就是您刚才建立的套件. 用于 Win32 和 Win64 开发的 EMSDevServer 有两个版本($(BDS)\bin\EMSDevServer.exe 和 $(BDS)\bin64\EMSDevServer.exe).这一切都是由 IDE 自动配置的.



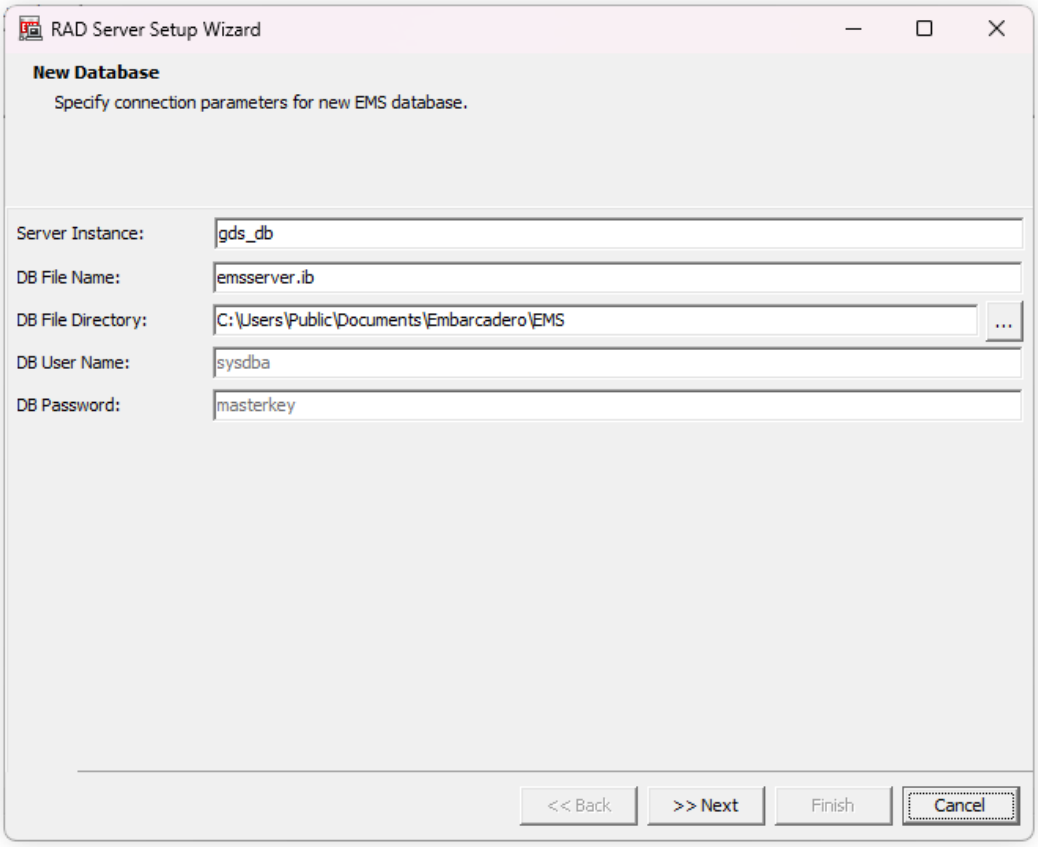
Run | Parameters...对话框显示 EMSDevServer.exe 作为主机应用程序

RAD Studio 还包括 EMSDevConsole.exe,它将启动 EMSDevConsole 服务器并开启 EMS 控制面板服务器窗口. EMS 控制面板提供了一个 Web 应用程序,可显示分析,提供用户/群组管理以及 RAD 服务器应用程序的更多功能.该控制台将在下一章中更详细地介绍.

选择 Run | Run 选单项目或按 F9 编译,链接并执行启动应用程序. 预设情况下,RAD 服务器开发服务器将使用 TCP 端口 8080 启动.如果这是您第一次执行 RAD 服务器应用程序,则会出现一个对话框,提示未找到 RAD 服务器配置文件 emsserver.ini.当没有 RAD 服务器注册表项或配置文件不存在时,就会出现这情况.



尝试在没有配置文件的情况下启动 RAD Server 开发服务器. 请点选"是"按钮执行 RAD 服务器设定精灵.



设定精灵第 1 页 - 指定新的 EMS 数据库联机参数

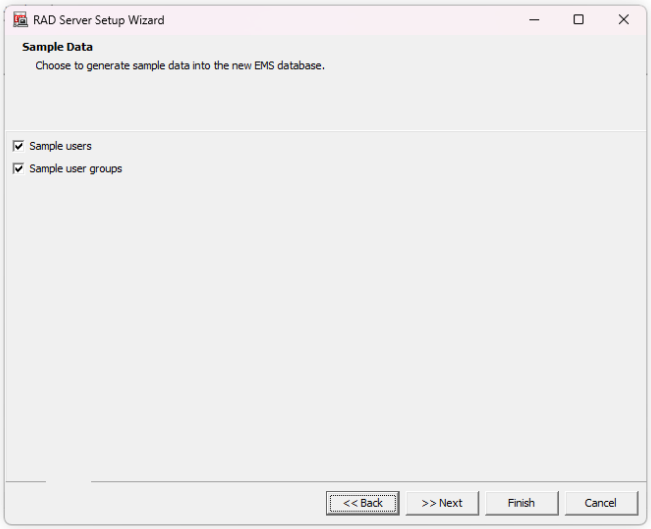
在第一个精灵步骤中,输入互联网服务器实例名称(默认情况下 RAD Studio 的 InterBase 服务器的开发版本使用 gds_db).此精灵页面还包含 RAD 服务器数据库的名称 (emsserver.ib) 以及包含数据库和配置文件的目录.

⚠□ 警告

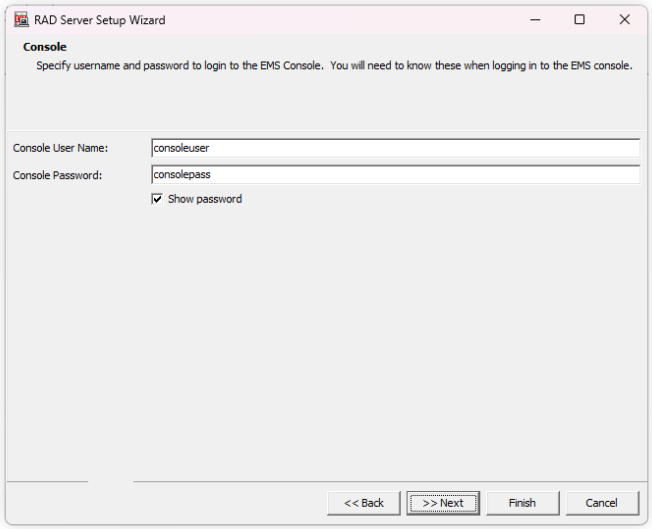
如果您之前使用不同的服务器实例名称在计算机中安装了 InterBase,请输入该名称字符串.

点选下一步按钮告诉精灵是否为用户和用户群组建立范例 RAD 服务器数据库数据.对于开发和测试,我们将勾选此两个选择项目.

点选"下一步"按钮设定用于登入 RAD 服务器控制面板的用户名称和密码.

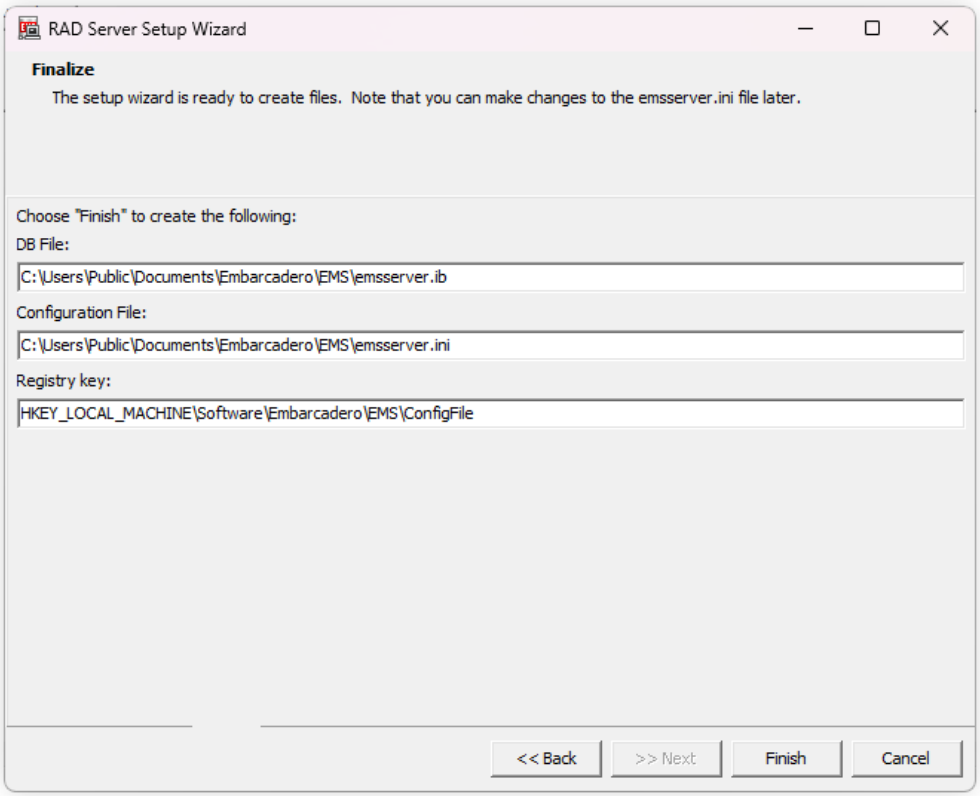


设定精灵第 2 页 - 用户和群组范例



设定精灵第 3 页 - 选择控制面板用户名称和密码

最后点选"下一步"按钮前往精灵的最后一步.此精灵已准备好建立 RAD 服务器数据库档案,设定文件,并为目前登入的使用者设定 Windows 注册表项.

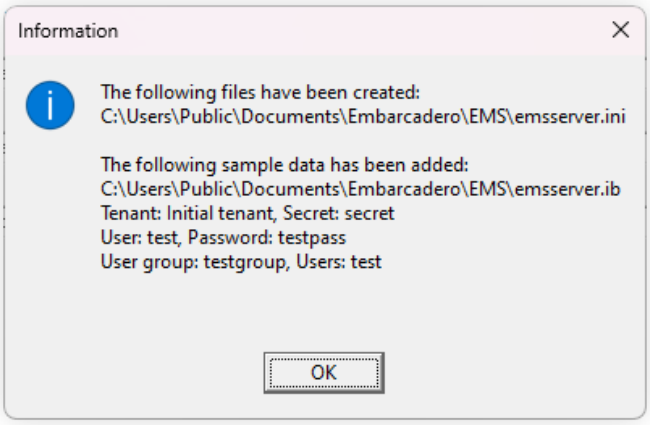


最终 RAD 服务器设定精灵页面

此页面显示数据库档案路径和名称,设定路径和名称以及 Windows 登录项目. 您可以随时变更 RAD 服务器配置文件 (emsserver.ini). 点选完成按钮. 此时将出现确认对话框,并提醒您设定将使用没有 RAD 服务器许可

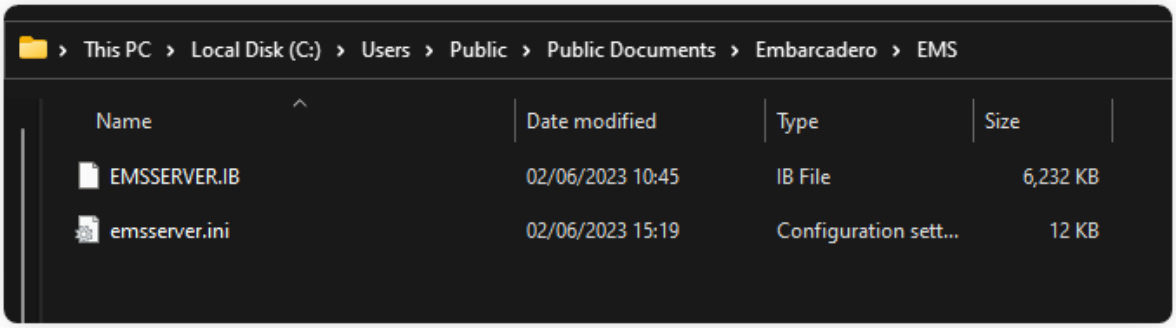
证的 InterBase 实例. 开发授权将您的 RAD 服务器应用程序限制为最多 5 个用户. 当您准备好部署 RAD 服务器应用程序时,您将能够使用 RAD 服务器和 InterBase 的部署授权.

点选"是"按钮.此精灵将显示 emsserver.ini 配置文件的位置.它还列出了已添加到数据库的范例数据.



设定精灵建立的 RAD 服务器档案列表

单击“确定”按钮.现在,有两个档案出现在 C:\Users\Public\Documents\Embarcadero\EMS 目录中.



RAD 服务器设定精灵在磁盘上建立的文件档案

备注

emsserver.ini 档案是定义 RAD 服务器所有预设参数值的地方. 尽管它将在下一章中讨论, 但请随意检查其内容以及文件本身中指定的扩展文档.
一般来说,RAD Studio IDE 是在没有管理员权限的情况下启动. 因此,HKEY_LOCAL_MACHINE 的 Windows 登录机码在 Windows 64 位操作系统上虚拟化为 HKEY_CURRENT_USER\Software\Classes\VirtualStore\MACHINE\SOFTWARE\WOW6432Node\Embarcadero\EMS.

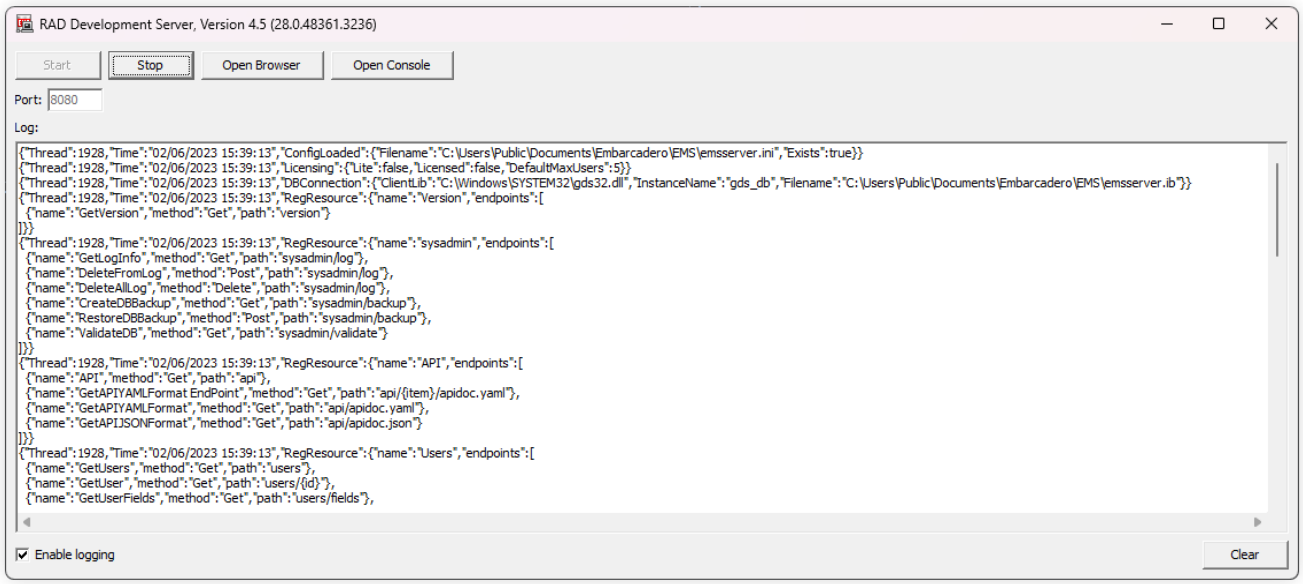
测试您的第一个 RAD 服务器应用程序

当 RAD Server 设定服务器建立完成后,RAD Server 开发服务器将开始执行.

警告

EMSDevServer 运行的预设端口是 8080.如果您的计算机将该端口用于其他服务,您可以更改使用的预设端口,在"Port"字段中更改它以满足您的需求. 若要使此变更永久有效, 请修改 emsserver.ini 档案中的参数`[Server.Connection.Dev]`

当您在 IDE 中点击"执行"时,您的 RAD 服务器开发服务器将开始执行,并且日志将显示您的套件应用程序执行的流程. Delphi 和 C++ 上的开发服务器完全相同.



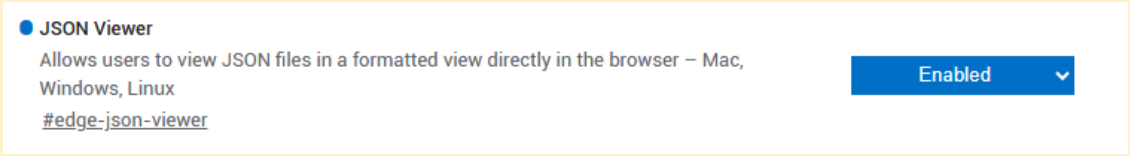
RAD Server 开发服务器启动第一个应用程序套件

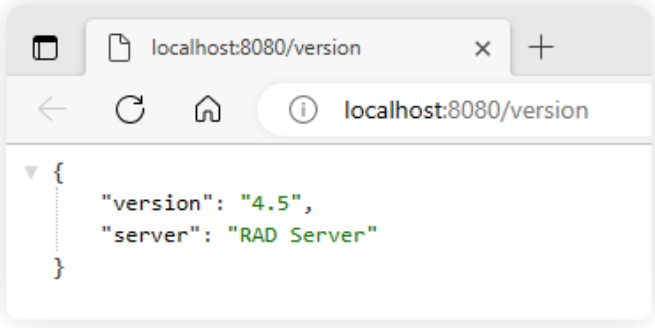
RAD Server 发服务器日志将显示设置、数据库联机、授权信息、已加载的应用程序套件、注册的资源以及已建立的端点.

点击"Open Browser"按钮将启动默认浏览器并显示呼叫 GetVersion 内建端点的 JSON 结果.您现在已经使用了第一个 RAD 服务器 REST 端点!

窍门

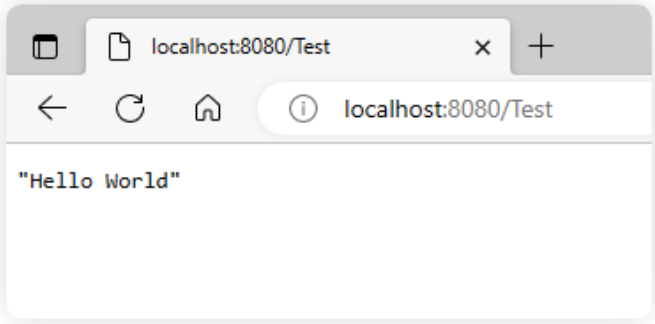
要在浏览器上获得更易于阅读的 JSON 响应,您可以安装扩充功能"JSON Parser". 它适用于所有主要浏览器.
在 Microsoft Edge 上,"edge://flags" 下有一个名为"JSON Viewer"的设置. 启用此功能可为您提供可读的 JSON 响应,而无需安装扩充





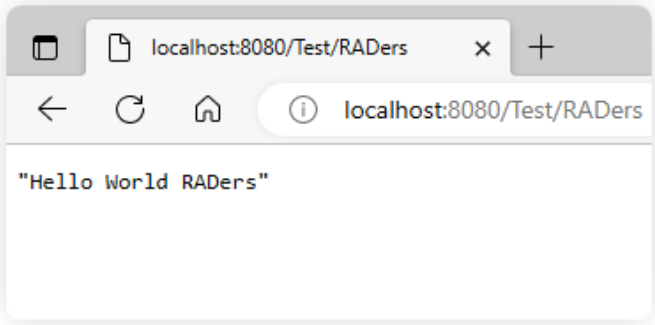
显示呼叫版本端点的输出的浏览器

在浏览器中,将 URL 变更为 localhost:8080/Test 并按 Enter 键.浏览器将从 Get 端点接收 JSON 回应.



浏览器显示 Test 资源的 Get 方法的 JSON 结果

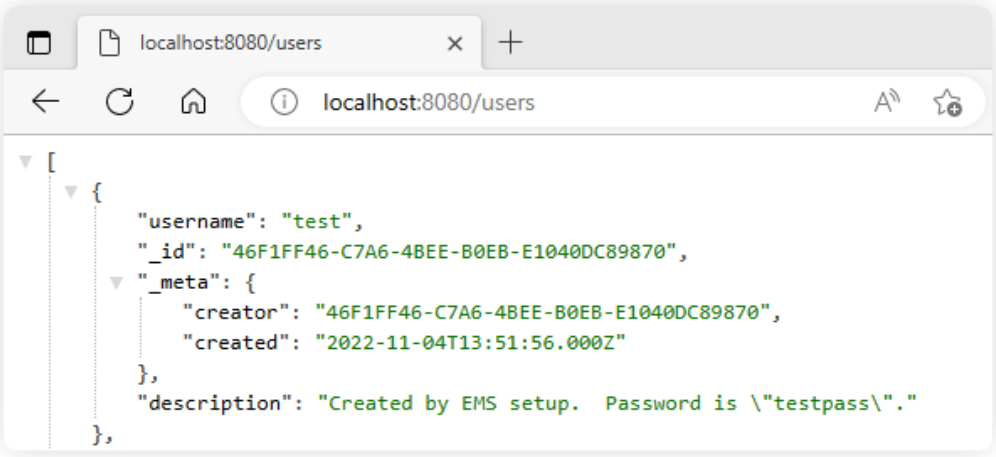
如果您在 URL 上传递附加项目,则会呼叫 GetItem 端点,并且后面的程序代码将传回 JSON 字符串,其中包含资源名称以及您键入的项目.



浏览器显示 Test 资源的 GetItem 方法的 JSON 结果

对于简单的范例,可以传回 JSON 字符串,但对于更大,更复杂的数据结构,您可能不希望传回大量的 JSON 字符串.RAD Studio 提供了许多其他方法来产生 JSON 数据,包括使用 JSON 对象,JSON 流和 JSON 编写器.

编辑 URL 以使用"users"资源,该资源将呼叫预设的 GetUsers 端点以显示由 RAD 服务器数据储存中的 RAD 服务器配置精灵产生的用户的 JSON(只有一个可以启动).



浏览器中呼叫 GetUsers 端点的 JSON 响应

现在您已经使用了 RAD 服务器项目精灵产生的四个端点服务.

请参考

- 本章的程序代码范例
- RAD 服务器引擎(EMS 服务器)
- 设定您的 RAD Studio (EMS) 服务器
- 在 Windows 上设定 EMS 服务器或 EMS 控制台服务器
- RAD 服务器管理 API

03

建立您的第一个 CRUD 应用程序

RAD Studio 提供了多个随时可用的组件,但在建立 CRUD API 时最有用的组件之一是 EMSDataSetResource. 该组件允许您将 FireDAC 查询连结到它,不仅公开数据,还可以对其进行操作. 此组件会自动建立 CRUD 所需的所有端点,并提供额外的功能,如分页、排序等.

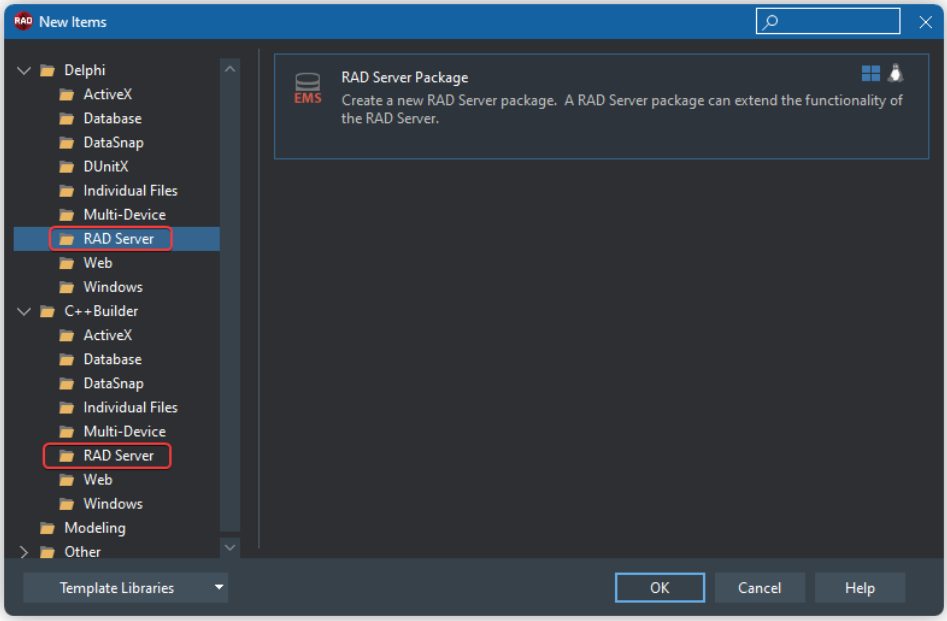
EMSDataSetResource 可以在您目前的任何单位中创建,或者甚至更容易一点,您可使用 RAD 服务器精灵建立自动连结到 FDConnection 的所有必要组件.

备注

在此范例中,我们将使用 InterBase employee 数据库,但也可以随意使用与 FireDAC 兼容的任何其他数据库. 使用 RAD 服务器精灵的唯一要求是在「Data Explorer」中预先配置数据库连接,以便 RAD Studio 识别它.

建立基于 REST 且具备 CRUD 功能的服务

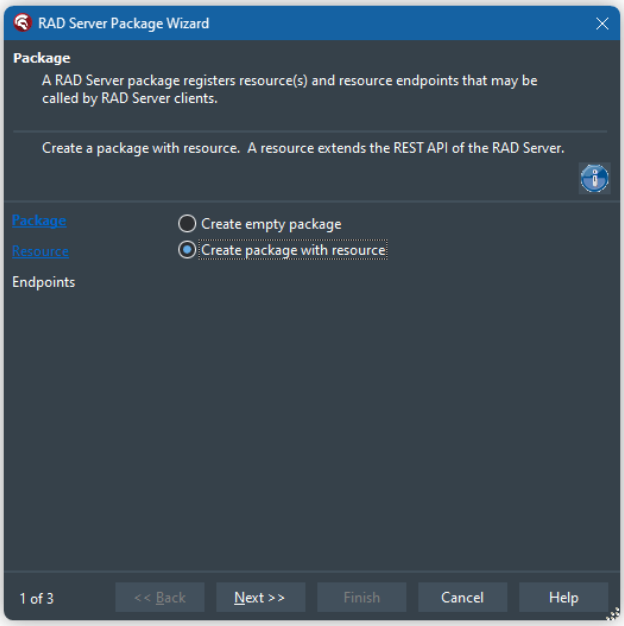
正如我们在上一章中所做的那样,最快的入门方法是使用“New Projects”选单(File|New|Other...}并选择 Delphi 或 C++Builder 的 RAD Server|RAD Server Package wizard...选单选项.



适用于 Delphi 和 C++ 的 RAD 服务器项目精灵选项

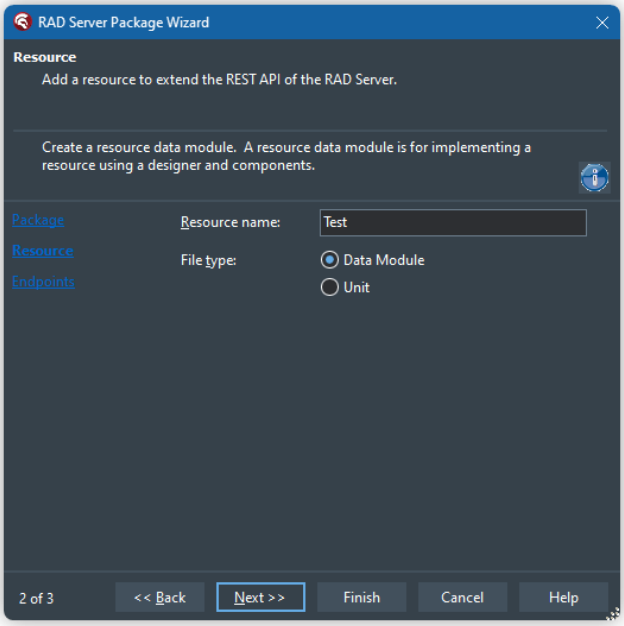
选择 RAD 服务器套件项目.此时将出现一个精灵来帮助建立起始项目. 在第一页上,选择精灵如何建立将出现在 RAD 服务器应用程序中的资源和端点. RAD 服务器套件精灵提供了两种继续选项.

现在建立一个包含扩展 RAD 服务器 REST API 资源的套件. 点选"下一步"按钮,将出现两个附加精灵步骤,以协助建立套件项目、资源和端点.要建立第一个 RAD 服务器项目,请做出此选择.



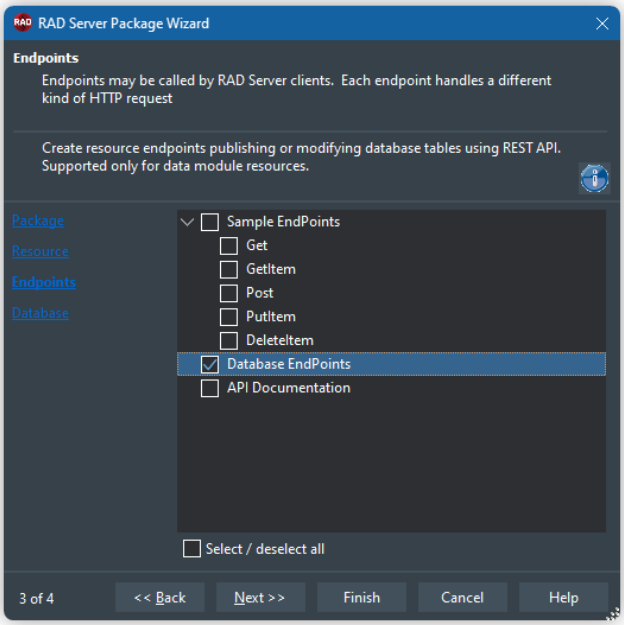
建立基于资源的 RAD 服务器套件

在精灵的第二页上，将资源名称设定为"Test". 文件类型单选按钮提供两个选项：1)建立用于在程序代码中实现资源的单元，2)建立用于使用 IDE 设计器、组件和程序代码编辑器实现资源的数据模块.对于第一个 RAD 服务器应用程序,选择使用数据模块.



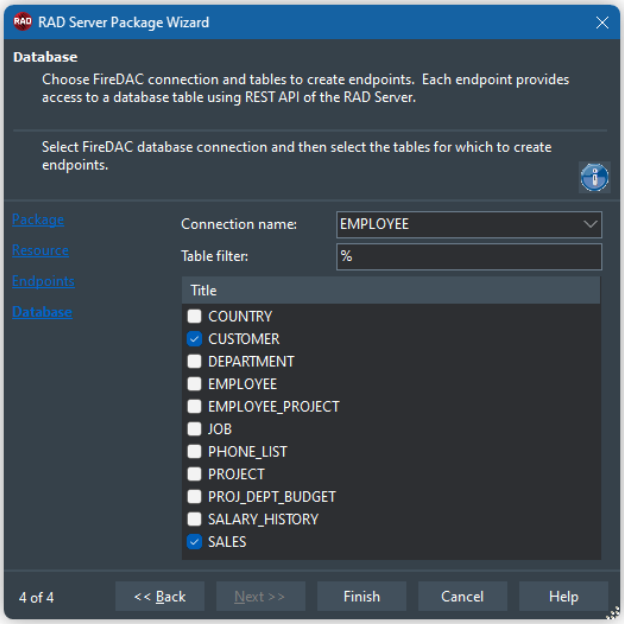
RAD 服务器套件精灵第 2 页 - 设定资源名称与文件类型

点选下一步按钮以建立一组起始端点.



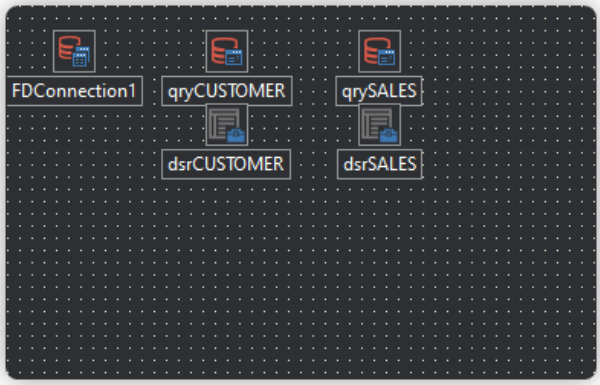
RAD 服务器套件精灵第 3 页 - 选择起始端点

在精灵的第三页上,我们将选择与上一章相比不同的选项.在这种情况下,我们将取消选取"范例端点"并选取"数据库端点". Now click “Next”.



RAD 服务器套件精灵第 4 页 - 选择数据库和表格

项目产生后,我们应该会看到一个 FDConnection、2 个 FDQueries 和 2 个 EMSDataSetResource.



精灵产生的数据模块

解释产生的项目

这个范例的美妙之处在于我们已经可以构建它,并且我们将能够访问为我们自动生成的端点,但首先,让我们修复一些问题: 存取 DataModule 的程序代码并更改端点的属性. 这并不真正相关,但保持端点小写和复数是常见的良好做法.

Delphi:

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

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

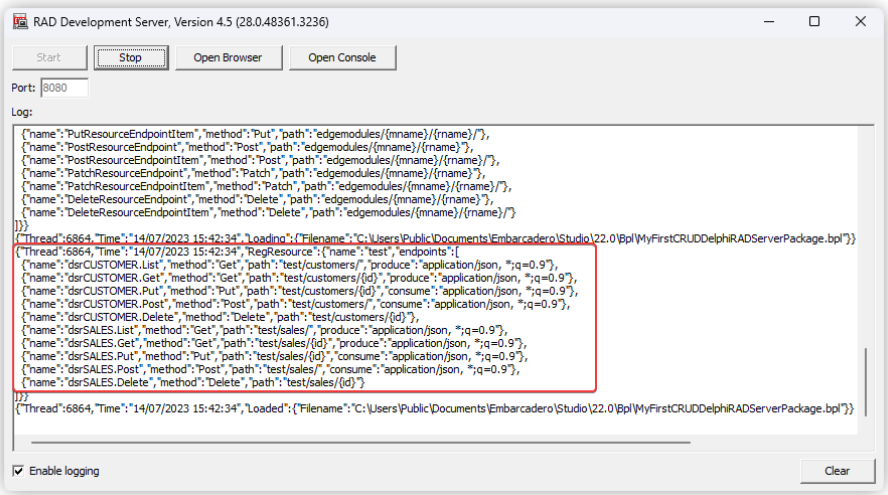
C++:

```
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());
}
```

正如我们所看到的,ResourceSuffix 属性链接到 EMSDatasetResources.这意味着连结到该 DatasetResource 的查询将在该端点下公开.

该项目再简单不过了.到目前为止,我们没有编写任何逻辑程序代码,但我们已经拥有一个连结到 2 个表的功能齐全的 CRUD 系统. 让我们建立该项目并进一步详细分析它.

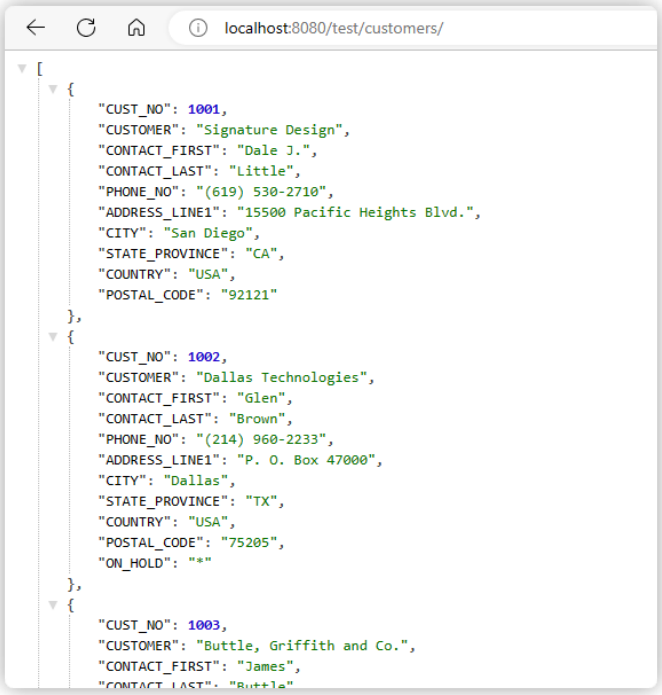
建置和测试项目



RAD 服务器日志显示自动建立的所有端点

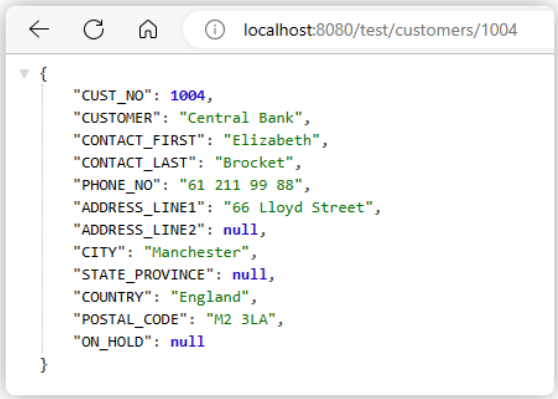
我们可以在 RAD 服务器日志中看到端点"customers"和"sales"已创建,而且还可以使用 URI 中的参数 {id} 来取得/储存/删除单一记录或发布新记录.

如果我们开启浏览器并造访 URL http://localhost:8080/test/customers/ 它会传回一个数据集,其中包含 customer 表中的所有记录.



RAD 服务器传回的客户数据集

要使用 ID (Cust_No) 存取特定客户,我们只需发送请求 http://localhost:8080/test/customers/1004. 正如您可能已经猜到的那样,如果您想访问销售端点,您只需呼叫 http://localhost:8080/test/sales/ 等.

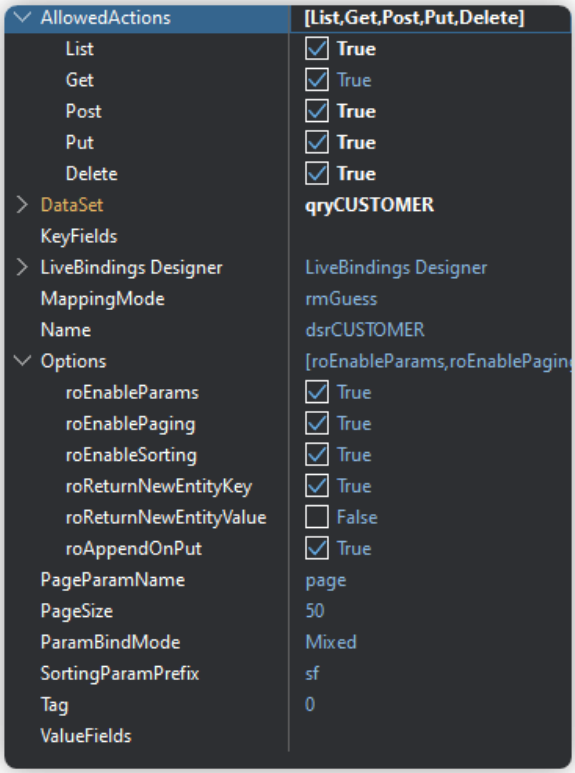


使用 ID 存取特定客户

> ⚠□ 警告
>
> *使用 TEMSDatasetResource 时,将斜线 / 保留在端点末端至关重要.在没有使用 "/" 存取端点时 RAD 服务器将抛出"未找到"异常.*

TEMSDatasetResource 的附加功能

到目前为止我们所看到的已经非常令人印象深刻了.只需点击几下,我们就可以公开所需数量的数据集,并且非常快速地开发我们的 API,但 TEMSDatasetResource 提供了更多内建功能.让我们来分析一下关键的功能:



AllowedActions – 用于允许或阻止在端点上列出、存取、发布、放置和删除的内建控制.

DataSet – 连接到数据集：查询、数据表等.

KeyFields – 选择进行查找时必须匹配的数据集字段.

PageParamName – 透过 URL 使用分页的参数名称.即：?page=1

PageSize – 存取 LIST 作业时,定义有效负载的分页大小.

SortingParamPrefix – 将会新增到数据集 ValueFields 项目之前的文字字符串.

ValueFields – 选择要在参数化查询中使用并显示在 JSON 响应中的字段字段

Options – 用于启用/停用参数使用、行分页、数据集字段排序等的子属性设置.

现在我们对这个组件有了更多的了解,让我们尝试在我们的 API 中使用其中一些功能如果我们造访 http://localhost:8080/test/customers/?sfCONTACT_LAST=A&page=1,我们将透过字段 CONTACT_LAST 按升序排列取得客户的第一页资料. 如果我们将值 =A 更改为 =D,则回应将按降序排列.

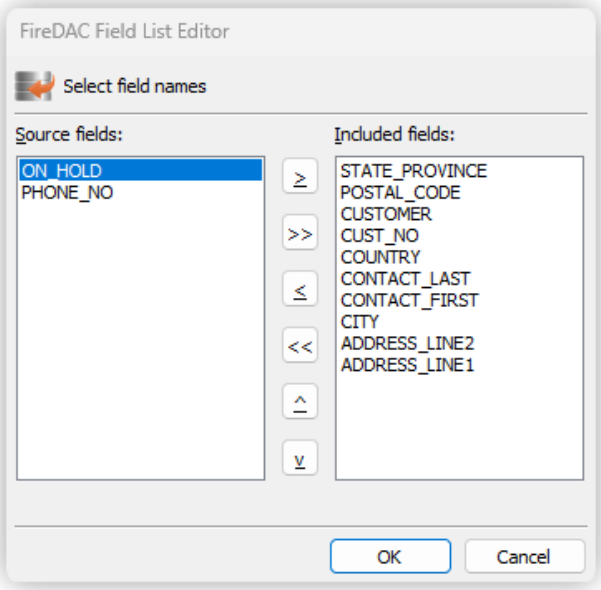
但是"order by"是如何注入到 SQL 中的呢?如果我们开启 FDQuery 我们将看到下一语句:

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

正如我们所看到的,SQL 语句非常简单,但该宏是其工作的关键. EMSDatasetResource 使用该宏能够在同一查询中混合分页和排序.

> **备注**
>
> *当使用分页并且我们到达数据集的末尾时,RAD 服务器将简单地传回一个空数组,让我们知道该页面中没有其他内容.*

另一个非常有用的功能是从数据库中获取字段字段以在我们的逻辑中使用的选项,但我们不想在 API 中公开这些字段字段. 使用 ValueFields 属性,我们可以轻松选择要发布的字段字段.



要在我们的 API 端点中发布的选定字段字段

04

REST 除错器

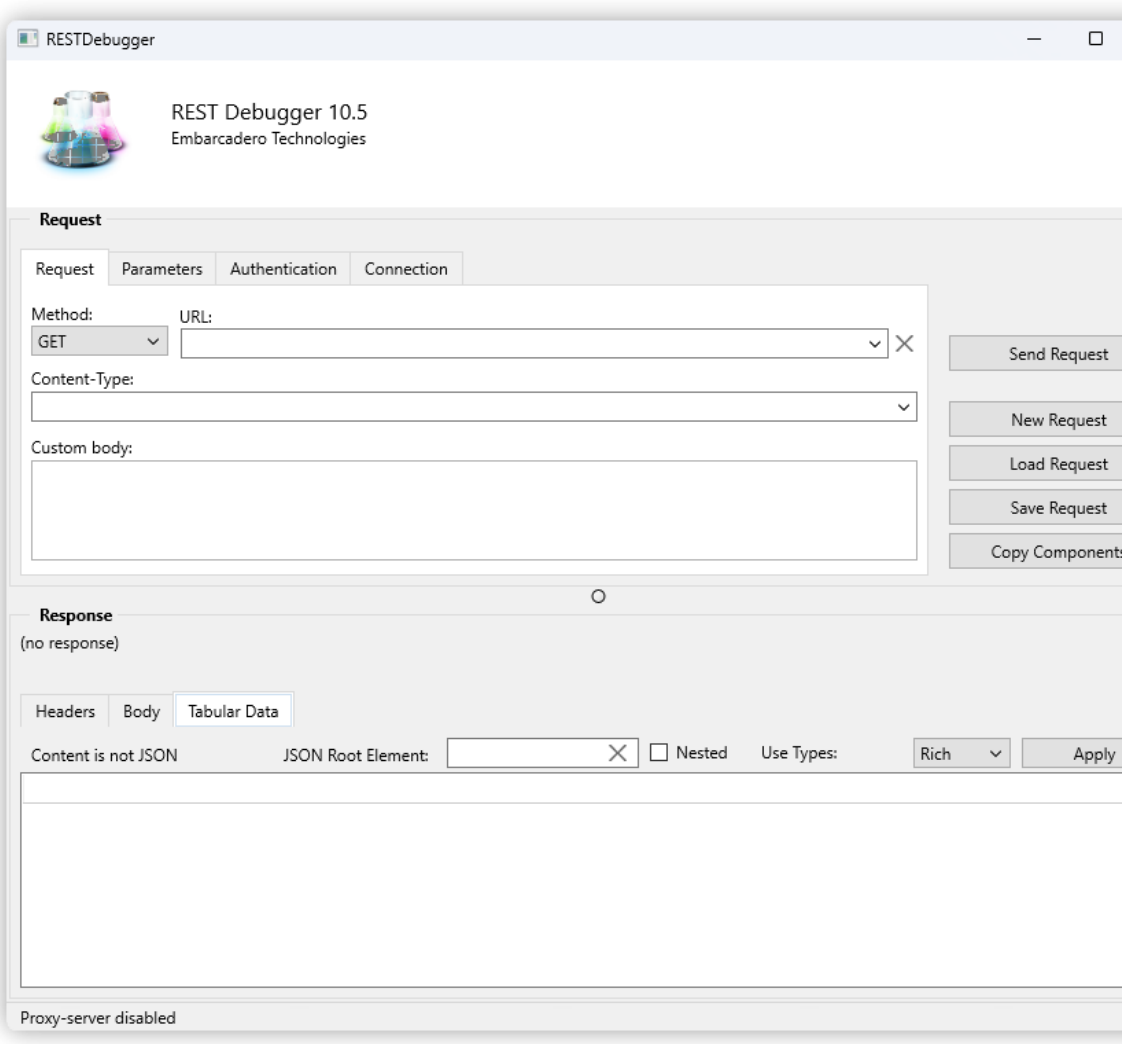
在第一章中,我们讨论了 REST API 生态系统中的可用操作,但到目前为止,我们仅透过浏览器使用了 GET. 如果您想知道如何使用 POST、PUT 和 DELETE 操作,在本章中我们将看到 RAD Studio 附带的一个名为 REST Debugger 的工具,它不仅可以简化您的测试过程,还可以帮助您更快的开发应用程序.

REST Debugger 并不是只能与 RAD Server 一起使用的产品.您可以使用它来存取任何其他第三方 REST API 服务,并利用其与 RAD Studio 的整合来加快您的开发流程.

什么是 REST Debugger 以及在哪里可以找到它

REST Debugger 是 Embarcadero 的免费解决方案,用于探索、理解 RESTful Web 服务并将其与 Delphi 和 C++Builder 应用程序整合. 它使开发人员能够探索、测试并最终了解 RESTful Web 服务如何与可过滤的 JSON blob、简化的 OAuth 1.0/2.0 身份验证和可配置的请求/资源参数等功能一起使用. 不仅如此,您还可以透过几次点击将 REST 组件直接复制并贴上到您的项目中.

如果您想尝试一下,可以在 RAD Studio 的"Tools/REST Debugger"菜单下找到它,或者您也可以免费下载独立版本点选此连结.



REST Debugger 用户接口

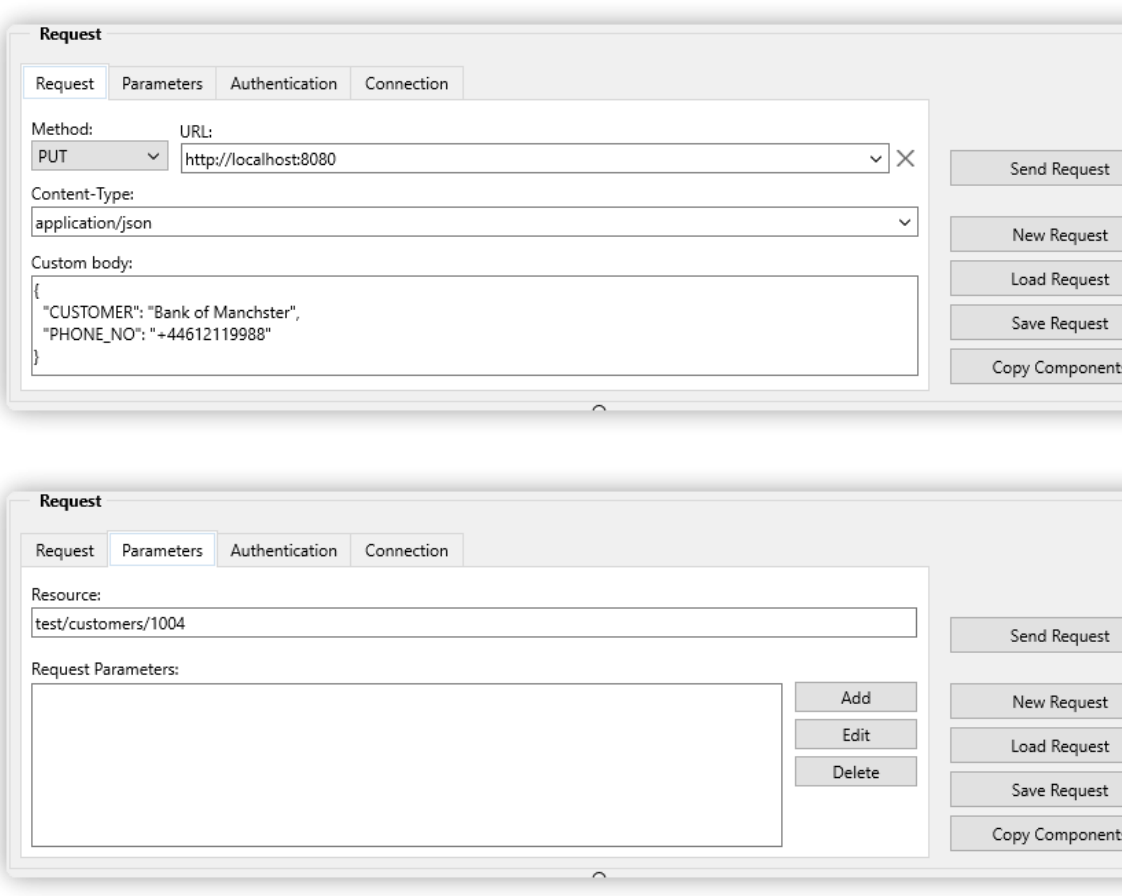
使用 REST Debugger 发送我们的第一个 PUT 请求

我们可以在左侧的下拉列表中看到默认值是 GET,但我们现在可以选择浏览器上无法选择的其他值.

使用我们在第 3 章中建立的相同项目来修改客户.

> ⚠ **警告**
>
> *启动并执行第 3 章中的 RAD 服务器项目是必不可少的.否则我们将无法呼叫 API 端点.*

要修改客户,我们只需使用要修改的 ID 呼叫客户端点.此外.我们需要在请求正文中指定属性的新数值.

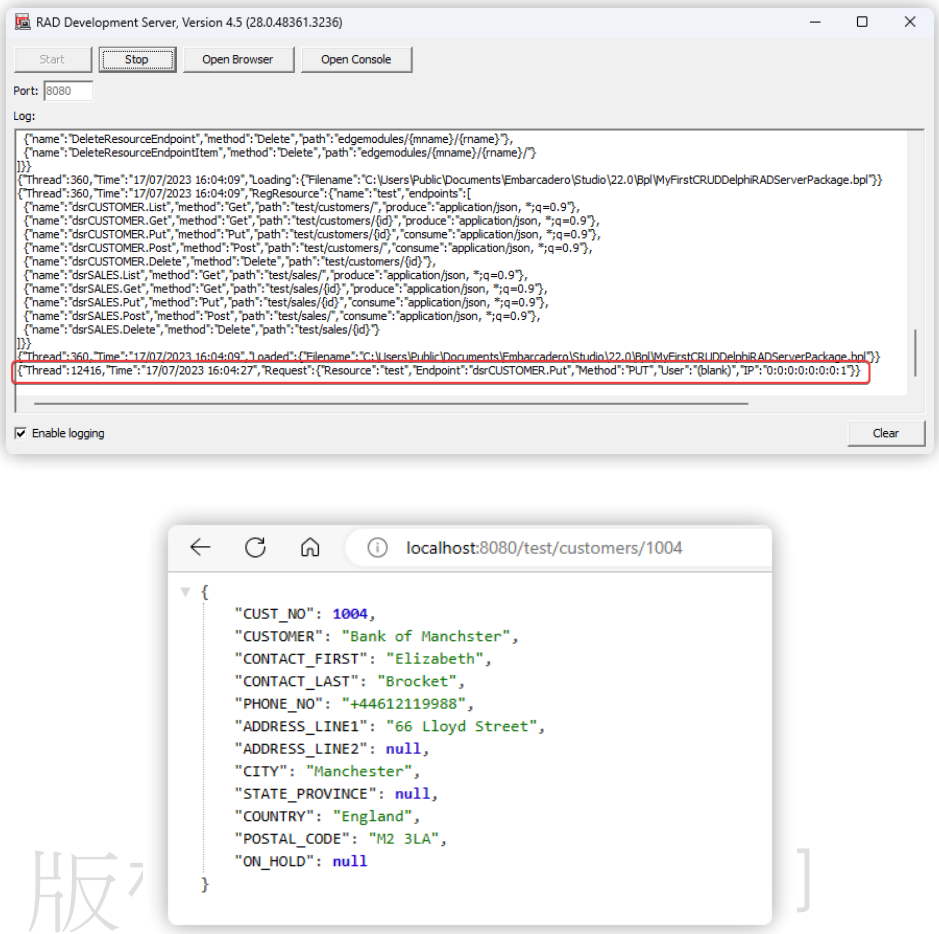


定义发送 PUT 请求所需的数值

为了配置请求,我们定义了 PUT 方法、URL、我们指向的资源(在本例中是 ID 为 1004 的客户)以及包含我们要更新的属性的 JSON 正文:客户的姓名及其电话号码.

最后一步是点选“Send Request”,如果我们收到 200 HTTP 响应,现在我们可以检查请求经过的 RAD 服务器日志. 此外,如果我们向该特定客户发送 GET 请求(我们可以在 REST 侦错器或浏览器中执行此操作),我们将看到数据已成功更新.

如果我们想要建立新客户,流程是相同的,尽管我们需要在正文中提供所有必需的信息并将方法更改为 POST.



RAD 服务器日志，包含注册的 PUT 请求和修改的数据

REST Debugger 包含的其他功能

尽管它与 RAD Server 并不完全相关,但值得一提的是 REST Debugger 提供的一个非常强大的功能.定义 URL、参数等后,使用"Copy Components"按钮在剪贴簿中产生所有必要的 RAD studio 组件,然后将它们贴上到您的任何项目项目中. 透过这种方式,您可以设计更快的 UI 原型来存取 RAD 服务器或任何其他第三方 API.

在本节的 GitHub 储存库中,您将找到一个基本的 FMX 范例.使用"Copy Components"按钮复制并贴上存取 API 所需的所有组件. 要测试它,您只需先执行 RAD 服务器应用程序,然后执行 FMX 应用程序并按"Send Request".

REST API 的另一个重要主题是身分验证. 如果您需要对 API 进行身份验证，您可以使用"Authentication"标签下的多种方法, 此外,您还可以使用"Parameters"标签中的"Add"按钮在请求中包含特定参数,例如标头中增加 api-key 参数.

05

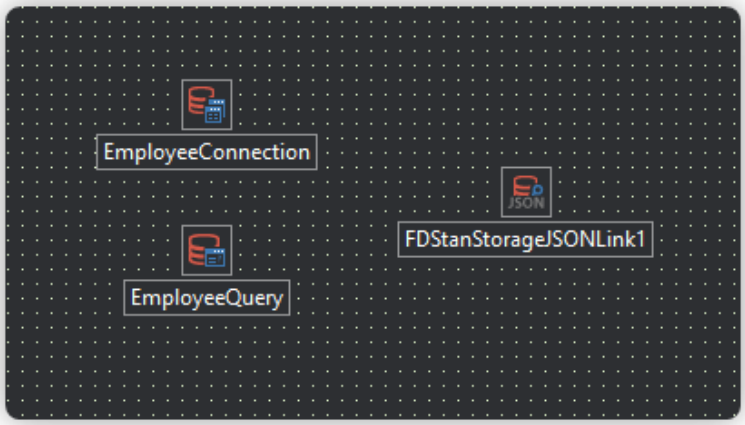
使用 FireDAC 批次移动和 JSONWriter

根据您的项目要求或您更熟悉的技术,RAD Studio 允许您使用更多工具来建立 REST API.

FireDAC 组件可用于产生和使用包含数据库元数据和以 JSON 编码的数据流,以响应来自 RAD 服务器端点之一的回应. 如果您的客户端应用程序将采用 VCL 或 FMX,则此方法非常有用. 您可以使用 MemoryTables 自动映像所有数据库信息和元数据. 使用 JavaScript 等语言的其他客户端应用程序在处理响应中包含的数据库信息和数据时可能会出现问题,但 RAD Studio 提供了一种产生 JavaScript 或其他语言期望接收的干净 JSON 的方法.

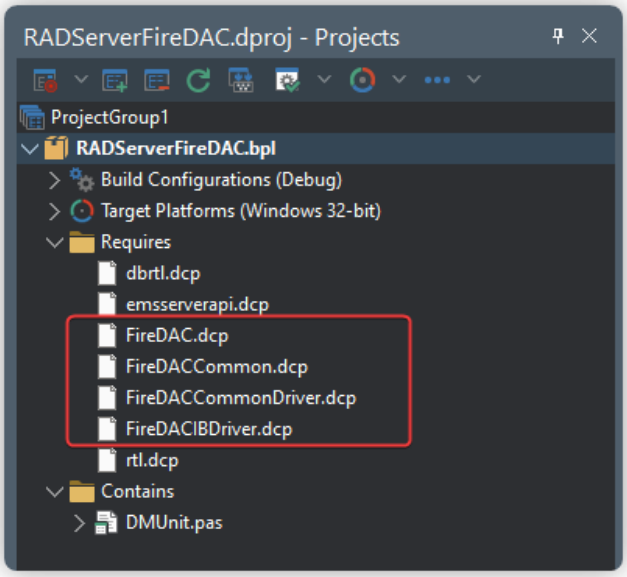
使用内存流返回 JSON 数据库数据

FireDAC 包含用于存取数据库表格信息并产生 JSON 字符串结果的组件.请使用资源模块建立 RAD 服务器应用程序. 新增 FDConnection 组件并将其与 InterBase 范例 Employee.gdb 数据库连接.新增 FDQuery 组件并设定 EmployeeQuery SQL 字符串为 select * from employee.新增 FDStanStorageJSONLink 组件以方便建立 JSON.

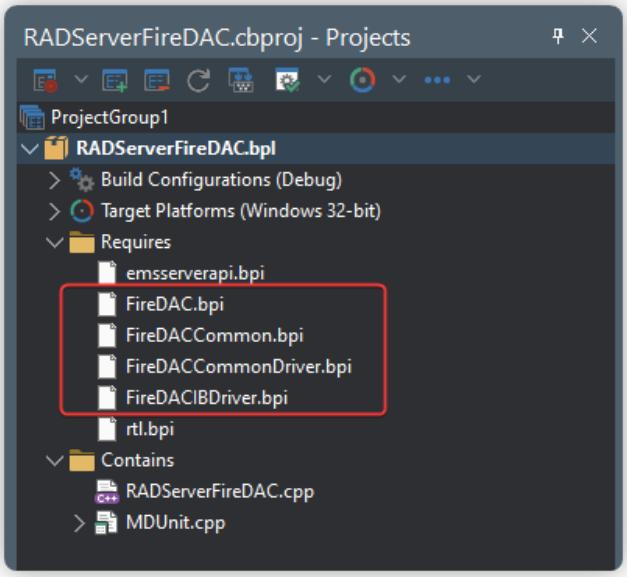


RAD Server 项目的资源模块

建置基于 RAD Server Delphi 的应用程序时,可能会出现一组警告,并会弹出一个对话框以允许应用程序套件与其他已安装的套件兼容. 单击"OK"按钮会将所需的套件档案新增至项目中的"requires"部分.对于C++Builder,可以手动新增套件(在项目管理器窗口中的 Requires 节点上单击鼠标右键,然后从弹出式菜单中选择 Add Reference...).



Delphi RAD 服务器 FireDAC 项目项目



C++ RAD 服务器 FireDAC 项目项目

您可以在每个目标平台的 C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\lib 中找到这些文件.

以下是 RAD Server Get 方法实现,它使用内存流发送带有员工表数据的 JSON 响应.

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);
}
```

使用浏览器和 URL http://localhost:8080/FireDAC 取得包含数据库员工表的 JSON 数据的响应.JSON 所包含的信息远不止于数据.响应中还包括有关数据表、数据表字段、型态等的元数据信息.

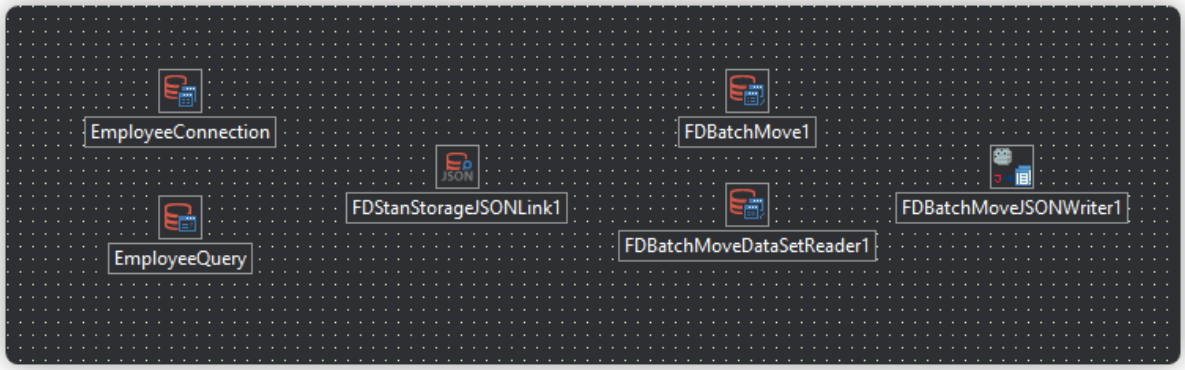
包含 JSON 响应的浏览器窗口

这绝对不是其他语言可以在不使用程序代码解析响应的情况下使用的简单字段和值的 JSON 数据,但如果您的客户端要使用 RAD Studio 进行开发,它会变得非常方便.

使用 FireDAC 的 BatchMove、BatchMoveDataSetReader 和 BatchMoveJSONWriter

对于复杂的数据库应用而言,使用上一章和上面提到的方法将需要编写更多的程序代码. 利用 FireDAC 的 FDBatchMove、FDBatchMoveDataSetReader 和 FDBatchMoveJSONWriter 组件可大幅简化 JSON 回应的创建.

我们将升级我们创建的同一个项目,并将组件 FDBatchMove、FDBatchMoveDataSetReader 和 FDBatchMoveJSONWriter 加入到资源模块中.



具有 FireDAC Query、BatchMove、DataSetReader 和 JSONWriter 的资源模块

S 将 FDBatchMoveDataSetReader 的 DataSet 属性设为 EmployeeQuery.

我们将建立一个名为 GetBatchMove 的新端点.

Delphi:

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

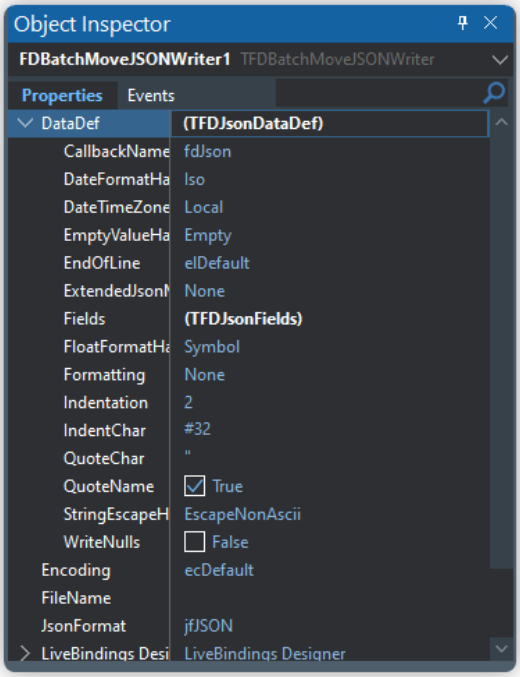
C++:

```
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 方法传回 JSON 数据结果:

使用 BatchMove 提供 JSON 结果的浏览器

FDBatchMoveJSONWriter 提供了多个选项来格式化 JSON 结果,涉及 DateFormats、endlines、writeNulls 等.



对象查看器中的 BatchMoveJSONWriter DataDef 子属性

FDBatchMove 组件还允许您建立映像以设定来源和目标字段映像并从目前来源记录取得数据数值.

Mappings 属性可以填为:

- 设计或执行时期手动设定映像.这允许指定自定义映射、转换表达式等.
- 如果映像设定为空,则在执行 Execute 呼叫时自动执行. 这将透过执行来源列和目标匹配的域名来映像. 如果目标列没有对应的来源字段,则目标字段将从对应中排除,并且在数据移动时不会被填入数据.

请参考

- Marco Cantu 部落格:数据集对应到 JSON - RAD 服务器 Web 服务 Delphi 范例
- FireDAC.Comp.BatchMove.TFDBatchMove
- FireDAC.Comp.BatchMove.JSON.TFDBatchMoveJSONWriter
- Readers 和 Writers JSON 框架
- FireDAC.TFDBatchMove 范例
- RTL.JSONWriter

06

JSONValue, JSONWriter 和 JSONBuilder

RAD Server 支持处理可由不同程序语言和工具使用的 JSON 数据.对于较小的数据量而言,建立 JSON 字符串、传输字符串作为响应,然后让客户端应用程序程序代码处理回传结果是可行的. 但想象一下,对于整个数据库或复杂的数据结构来说,JSON 数组回应有多大? RAD Studio 提供了三个用于处理 JSON 资料的主要框架.本章介绍 RAD 服务器应用程序向呼叫应用程序传回 JSON 的多种方法中的几种.

处理 JSON 数据的框架

RAD Studio 提供了多个框架来处理 JSON 数据.最常见的三种是:

- JSON 对象框架 – 建立临时对象来读取和写入 JSON 数据.
- Readers 和 Writers JSON 框架 – 允许您直接读写 JSON 数据.
- SONBuilder – 使用编写器,以更易于维护的方式建立复杂的结构.

JSON 对象框架需要建立临时对象来解析或产生 JSON 数据.要读取或写入 JSON 数据,您必须在读取和写入 JSON 之前建立一个中间内存对象,例如 TJSONObject、TJSONArray 或 TJSONString.

Readers 和 Writers JSON 框架允许应用程序直接读取 JSON 数据并将其写入串行流,而无需建立临时对象. 无需建立临时对象来读取和写入 JSON,可提供更好的效能并改善内存消耗.

JSON Builder 是前两个的组合.它的创建是为了使您的程序代码更具可读性和可维护性.它还遵循一种更现代的方法,您可以将方法一个接一个地连结起来.

在本章的范例项目中,您将发现 3 个不同的端点,它们产生完全相同的响应,但使用这三个可用的框架.您可以在您的项目中随意使用您觉得更舒服的方式.

在每个端点上获得相同的 JSON 回应

使用 JSONValue

使用 JSON 对象框架透过在程序代码中组装 JSON 字符串来建立它们. JSONValue 是用于定义 JSON 字符串、对象、数组、数字、布尔值、true、false 和 null 值的所有 JSON 类别的祖先类别. RAD Studio JSON 实作中包含以下类别和方法:

TJSONObject – 实作一个 JSON 物件。.TJSONObject 中的方法包括:

- Parse – 解析 JSON 数据流并将遇到的 JSON 对储存到 TJSONObject 对象的方法.
- ParseJSONValue – 解析字节数组并从数据建立对应 JSON 值的方法.
- AddPair 方法 – 将新的 JSON 对新增至 JSON 物件.
- GetPair 方法 – 传回 JSON 对象对列表中具有指定 I 索引键值的对,如果指定 I 索引越界,则传回 nil.
- GetPairByName 方法 – 从 JSON 对象传回一个键值对,该对象的键部分与指定的 PairName 字符串匹配,如果没有与 PairName 匹配的键,则传回 nil.
- SetPairs – 定义此 JSON 对象包含的键值对列表.
- FindValue – 寻找并传回位于指定 JSON 路径的 TJSONValue 实例.否则返回 nil.
- GetValue – 传回 JSON 对象中 Name 键指定的键值对中的值部分,如果没有与 Name 相符的键,则传回 nil.
- Pairs – 存取 JSON 对象对列表中指定 Index 处的键值对,如果指定 Index 越界,则传回 nil.
- GetCount – 传回 JSON 物件的键值对数量.

TJSONArray – 实作 JSON 数组. JSONArray 方法包括:

- Add – 将透过 Element 参数给定的非空值新增至目前元素列表.

- Get – 传回 JSON 数组数组中给定索引处的元素.
- Pop – 从 JSON 数组数组中删除第一个元素.
- Size – 传回 JSON 数组数组的大小.
- ToBytes – 将目前 JSON 数组数组内容串行化为字节数组数组.
- ToString – 将目前 JSON 数组数组串行化为字符串并传回结果字符串.

其他 JSON 类别包括:

- TJSONString – 实作 1 个 JSON 字符串.
- TJSONNumber – 实作 1 个 JSON 数值.
- TJSONBool – JSON 布尔值.
- TJSONTrue – 实作 1 个 JSON true 数值.
- TJSONFalse – 实作 1 个 JSON false 数值.
- TJSONNull – 实作 1 个 JSON null 数值.

使用 JSON 类别的范例

以下的范例项目的 GetJSON 方法实作了一个 Get 端点,该端点使用多个 JSON 类别来建立、解析和显示 JSONObjects 和 JSONArray 的结果.

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 简化了 RAD 服务器应用程序开发,以制作自定义 JSON,为程序语言客户端提供可供使用的数据. 使用 JSONWriter 启动 JSON 对象,写入属性名称和值,继续写入属性和值,直到结束 JSON 对象.

使用 JSONWriter 的范例

以下是 Get 端点的实现,它使用 JSONWriter 的 WriteStartArray、WriteStartObject、WritePropertyName、WriteValue、WriteEndObject、WriteEndArray 方法传回数据. AResponse 参数有一个内建的 JSONWriter,它可以非常方便地直接在响应上建立更复杂的结构.

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

该框架是一个 JSONWriter 包装器,可让您以更快、更易读的方式建立 JSON. 它遵循流畅的接口(也称为方法连结)方法,在非常复杂的 JSON 结构的情况下,可以简化程序代码并使其更易于维护和阅读.

在同一项目范例中,您将发现另一个使用 JSON 产生器来建立响应的端点.我们来看看程序代码:

Delphi:

```
procedure TTestResource1.GetJSONBuilder(const AContext: TEndpointContext; 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*
AResponse, 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 提供的一个非常有用的范例项目项目,称为 fmWorkBench(尽管它仅适用于 Delphi).您可以在下面路径中找到它：

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

请参考

- JSON
- Readers 和 Writers JSON 框架
- JSONBuilder
- WorkBench 范例项目
- 教学：使用 REST 客户端函式库存取基于 REST 的 Web 服务

07

建立您自己的自定义端点

在本章之前,我们已经了解了基本的 JSON 结构:数组数组、对象…还有相当简单的 URI：/customers、/sales…但在 REST API 最佳实践中查找子资源 URI 和嵌套数组/是很常见的. 在本章中,我们将讨论如何使用 RAD Server 完成这些类型的结构以及建立您自己的 GET、POST、PUT 或 DELETE 方法.

良好做法的范例

尽管您可以按照自己想要的方式建立 API,但仍有数千篇文章讨论标准化或 REST API 的最佳实践. A 归根结底,这取决于您想要如何建立 API,但了解这些标准的基础知识,并尽量不要重新发明轮子, 是值得阅读和思考的.

例如:当我们存取特定客户数据时,我们了解到我们使用 URI /customers/{id} 但如果我们想存取该特定客户的销售情况该怎么办? 一个非常常见的选项是定义另一个端点,例如：/customers/{id}/sales.该端点将传回由其 ID 定义的特定客户的销售订单.

这被认为是良好实践,通常称为嵌套资源或子资源. 这定义了端点之间的层次关系,可以帮助第三方或您自己的开发团队以更简单的方式理解您的 API.

避免 API 过于啰嗦

开发应用程序时,访问窗体/网页并发现自己需要来自多个端点的数据是很常见的. 假设您存取客户的销售订单:您可能需要客户的详细数据、订单信息、送货地址、订单行,可能还需要付款、发票…请求清单可能会很长,当涉及 REST API 时,会出现一个问题:请求的成本很高. 我的意思并不是说服务器处理所有这些请求的成本很高,而且互联网给这个方程式带来的延迟也很昂贵. 每个请求都需要几毫秒来回,如果我们需要

发送 10 个甚至更多请求来访问一个页面,那么这里还有很大的改进空间.这就是嵌套 JSON 响应更有意义的时候.

想象一下,您可以在一次请求中向 RAD 服务器请求一个客户及其所有销售额的摘要.这可能相当于一种经典的主从关系,但全部在一个请求中返回.我们也将了解如何实现这一目标.

新增子资源

对于子资源,我们仍然可以使用我们在前面的章节中看到的相同的 TEMSDatasetAdapter.我们只需要调整属性中的一些内容,RAD Studio 和 FireDAC 将为我们完成剩下的工作.

使用我们迄今为止使用过的相同项目(有 2 个查询：qryCUSOTMER、qrySALES),让我们像这样修改 qrySALES 的 SQL:

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

还有 drsSALES EMSDatasetAdapter 之上的属性,让我们改变这些属性:

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 语句中,我们刚刚新增了一个 WHERE 子句,其中包含一个过滤特定客户销售的参数.

如果我们检查属性,就会发生更有趣的事情. RAD 服务器会自动将 {CUST_NO} 中的值注入 FireDAC 查询中,并过滤该客户的销售额. 另外,我们需要指定其余的方法(List、Get、Post 等),因为现在相同的端点涉及 2 个键值,并且必须指定它们的名称才能使其工作. 好消息是,我们仍然可以使用这些端点来建立、修改或删除特定销售,就像我们对使用 EMSDatasetAdapter 建立的任何其他端点所做的那样.

在响应中新增嵌套资料(主/从详细资料)

现在我们有了第一个子资源端点,让我们建立一个具有多个值的巢状响应.为此,我们最后需要写一些程序代码.

让我们在 TTestResource1 数据模块类别中建立一个已发布的方法.

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();
    }
}
```

我们可以在这段简单的程序代码中看到,我们只是使用宏检索从 URL 获取的特定客户 ID 的详细信息,并将其作为参数传递给我们已经使用的相同 qrySALES. 如此做就无需额外的查询.

尽管使用像 EMSDatasetAdapter 这样的组件对低程序代码方法有很大帮助,但有时我们需要特定的要求,并且需要编写自己的实作程序代码. 正如我们在前面的章节中所看到的,我们可以使用 JSONWritters 来自定义我们的响应.

如果我们检查此范例的程序代码,对于每个传入请求,我们都可以存取 ARequest 和 AResponse 参数来存取所有必需的信息并建立我们自己的响应. 在此范例中,我们使用"WriteStartObject"方法建立一个新对象,然后使用 TQuerySerializer 类别中的两个方法(稍后将详细介绍),然后结束该对象.

我们可以将多种有用的方法和属性与 JSONWriters 和 JSONReaders t 一起使用,这将使您的编码体验变得更加轻松. 我强烈建议您查看文件以了解所有可用功能.

现在你可能会问自己: 但这两种方法是什么: **ExcludeMasterFieldFromFields** 和 **SerializeMasterDetail**? 这两种方法已经针对这个特定的演示范例进行了编码,但是您可以在 GitHub 储存库演示中取得它们,当然,也可以在您的项目中随意使用这些程序代码. 他们的任务在方法中有很好的说明,但总而言之,他们只是将主/详细关系转换为 JSON 对象,并将详细查询作为 JSON 数组数组插入到主 JSON 对像中. **ExcludeMasterFieldFromFields** 可能不是必需的,但为了避免冗余数据,我们从详细信息中排除了 MasterField.


窍门

您可查看 Data.DBJson 单元,它包含多个类别来帮助您将数据集转换为 JSON,反之亦然. 在此范例中,我们使用了 TDataSetToJSONBridge 类别,它允许我们更快,更精细地串行化这些内容.

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;
}
```


备注

在实际项目中,将这些方法抽像到另一个类别/单元中会更有意义,因为它们可以轻松重复使用,但为了简单起见,我们将它们保留在同一个 DataModule 中.

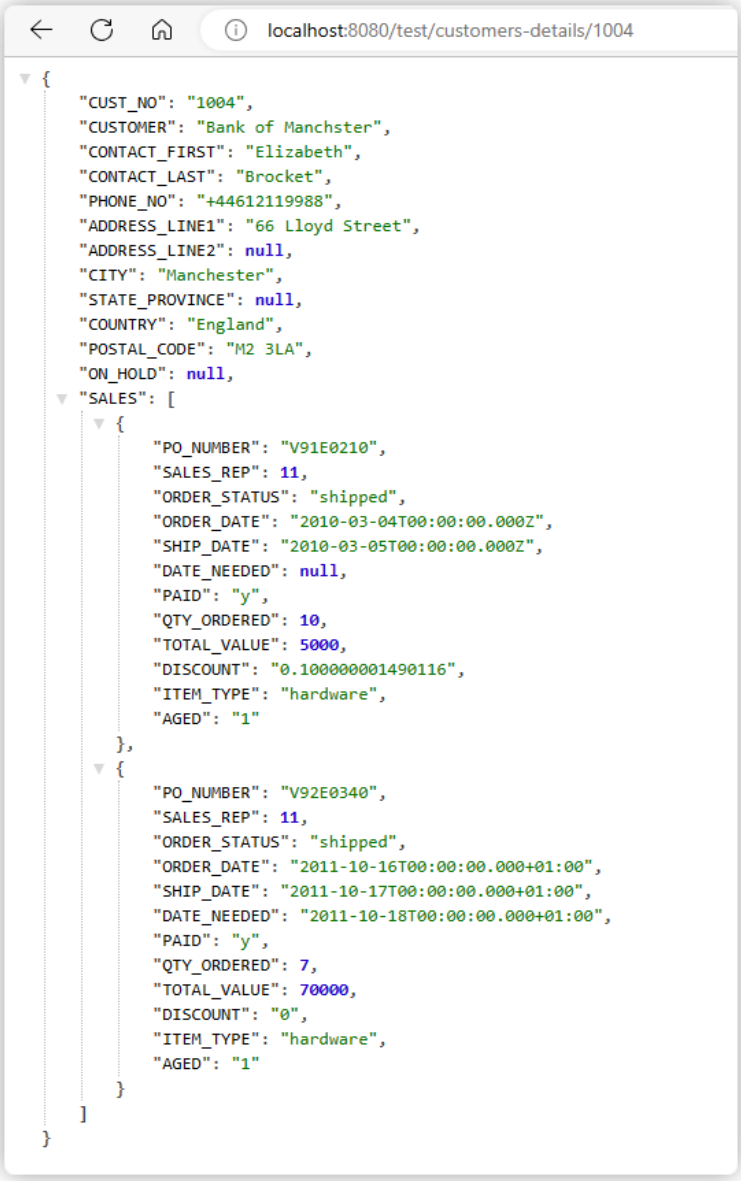
测试新的实作

让我们执行此范例项目项目并存取 URL http://localhost:8080/test/customers/1004/sales/

使用子资源方法存取特定客户的销售情况

我们现在正在过滤客户 1004 的销售,但是如果我们想要存取/修改/删除特定的一项销售,我们也可以透过这个相同的 URI 进行存取.我们只需要在末尾添加订单 Id 即可,例如: http://localhost:8080/test/customers/1004/sales/V91E0210

现在让我们存取我们定义的另一个端点: http://localhost:8080/test/customers-details/1004



使用子资源(客户及其销售)存取端点

在同一个请求中,我们获得了特定客户的所有销售额,这意味着我们只需一次即可获得所需的所有信息,而不是两次呼叫 RAD 服务器. 显然,透过多个嵌套层级等,此类请求可能会变得更加复杂.

备注

在与本章相关的 github 储存库示范项目中,您将找到一个额外的端点来存取客户清单及其相关销售. 我们不会过滤特定的一个,而是取得所有的数据. 请注意,大部分程序代码已被重复使用,这使得在进一步的开发中让实作非常简单.

建立自定义 GET、POST、PUT、DELETE 方法

到目前为止,我们已经了解如何创建 GET 自定义方法,但在某些情况下,我们需要使用其他动词,例如 POST、PUT 和 DELETE. 要编写这种实作程序代码,我们只需要用我们想要使用的动词来命名方法名称, 例如: **“procedure PutMethodName(..”**. 正如您可能在前面的范例中看到的那样,所有自定义的方法都以"Get"开头:如果我们将其更改为"Post", 我们将定义一个 POST 方法.让我们来看一个例子:

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;
```

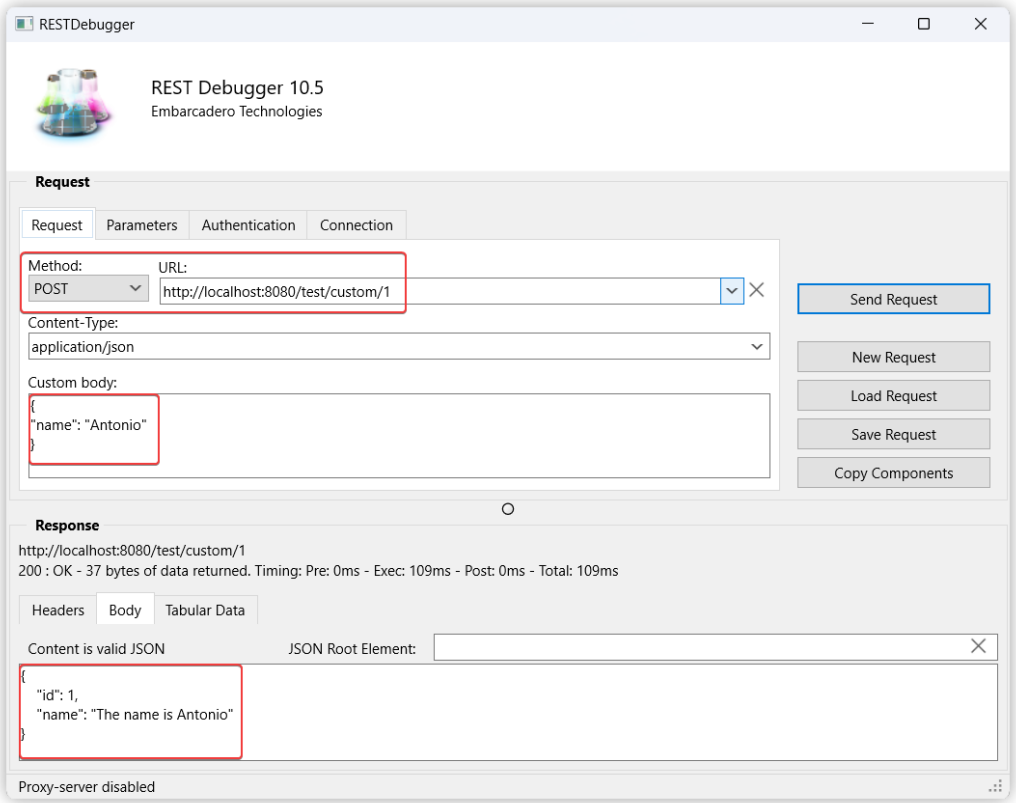
C++:

```
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();
}
```

现在我们有了一个新的额外端点 ./custom/{id}.如果我们从 REST 侦错器发送 POST 请求, 并在正文中添加预期的"name"属性,我们将得到如下的结果.



来自自定义 POST 方法的回应

处理响应错误

正如我们在前面的自定义 POST 端点范例中所看到的,如果我们没有获得预期的所有数据,我们会触发错误. RAD 服务器提供了一个内建解决方案, 可以直接从 AResponse 对象触发并传回最常见的错误. 您可以在以下位置找到有关可用错误的更多详细信息此连结.

请参考

- JSON Writers 和 Readers
- REST API 最佳实践

08

存取内建的分析功能

RAD 服务器控制面板是一项提供预先配置 Web 应用程序的服务,该应用程序显示来自 RAD 服务器引擎的多个数据以及分析数据. 它允许您更深入地了解 RAD 服务器实例上的活动,并根据真实资料做出决策.分析使用者、API 和服务活动,深入了解应用程序的使用情况.

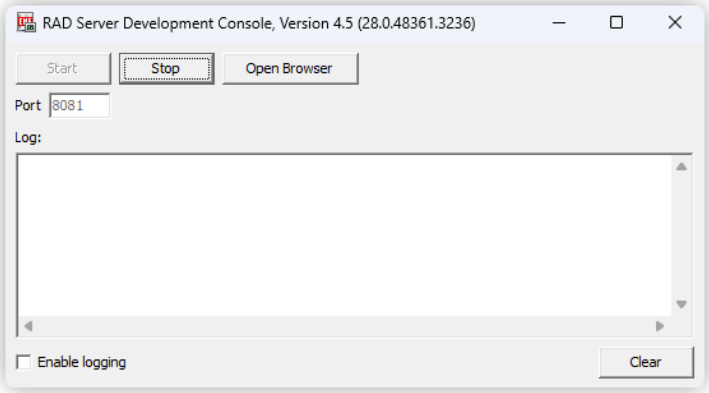
主要特点

RAD 服务器控制面板以只读模式存取数据库服务器.

- 它透过 RAD 服务器引擎资源的统计数据提供 API 呼叫的回馈:用户、群组、安装、模块及其资源.
- 您可以将控制面板作为独立应用程序用于测试目的,或在 Microsoft IIS 服务器上设定控制台以用于生产环境.
- 注意:Microsoft IIS Server 在 Linux 上不可用.您可以于 Linux 的生产环境中使用 Apache
- RAD 服务器控制面板透过扩充服务器功能提供新资源分析.
- RAD 服务器控制台为注册的 RAD 服务器用户提供分析数据.
- 您可以将分析数据导出并储存到系统中的.csv 档案中.

存取 RAD 服务器控制台

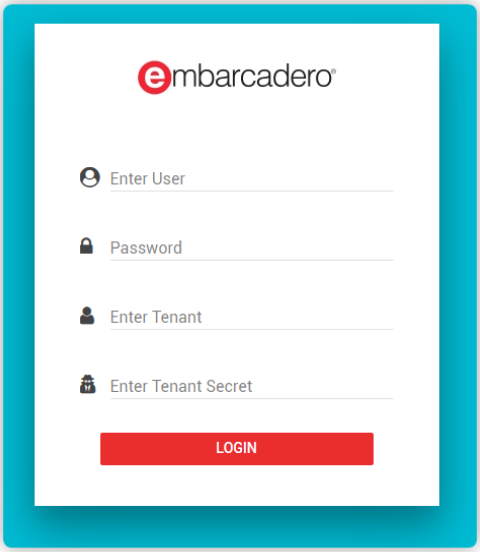
返回 RAD Server 开发服务器并点选 Open Console 按钮. 这将启动在端口 8081 上自动执行的 RAD 服务器开发控制台服务器,并且还将开启具有分析控制面板登入窗口的浏览器.



RAD 服务器开发控制台服务器 UI

备注

如果您的计算机已经使用了默认端口 8081,您只需将 8081 变更为计算机上可用的任何端口,按"开始"按钮,然后"开启浏览器".

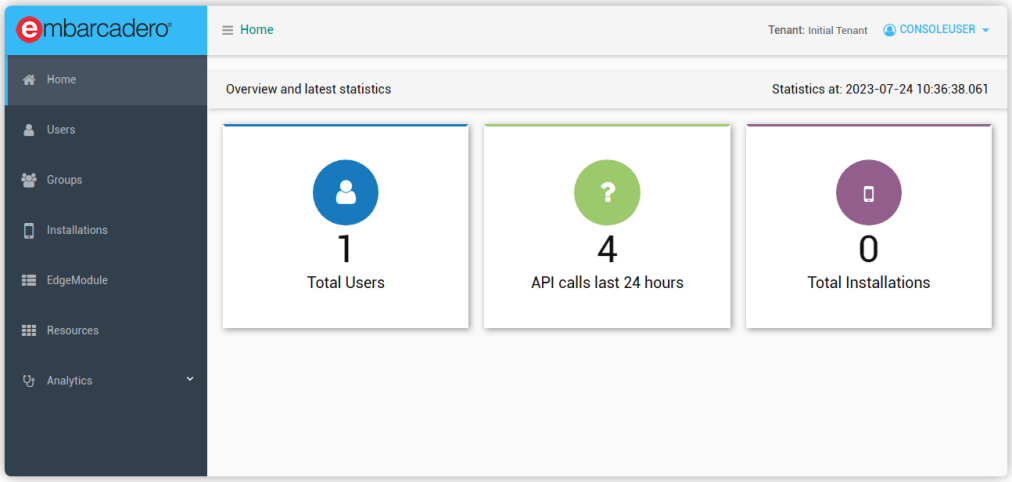


RAD 服务器控制台使用者登入画面

若要访问控制台,RAD 服务器附带了预先设定的默认凭证(将 tenant 信息留空): user: **consoleuser**
password: **consolepass**

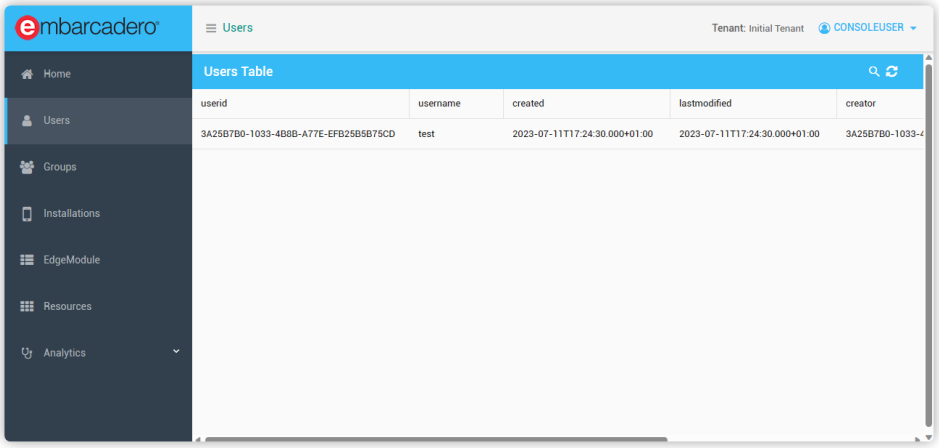
警告

RAD 服务器提供默认用户和密码来访问控制台. R 请记住在 emsserver.ini 配置文件中更改这些凭证(查看有关此配置文件的章节以取得更多详细信息).



RAD 服务器控制台首页

登入后,您将看到 RAD 服务器控制面板单页 JavaScript 应用程序的图形视图,左侧是选单,右侧是内容. 选单提供了如下凡信息: users, groups, device installations, EdgeModules, Resource Modules 和 Analytics. 这是显示用户列表及其信息的屏幕,包括用户的建立时间和上次修改用户信息的时间.



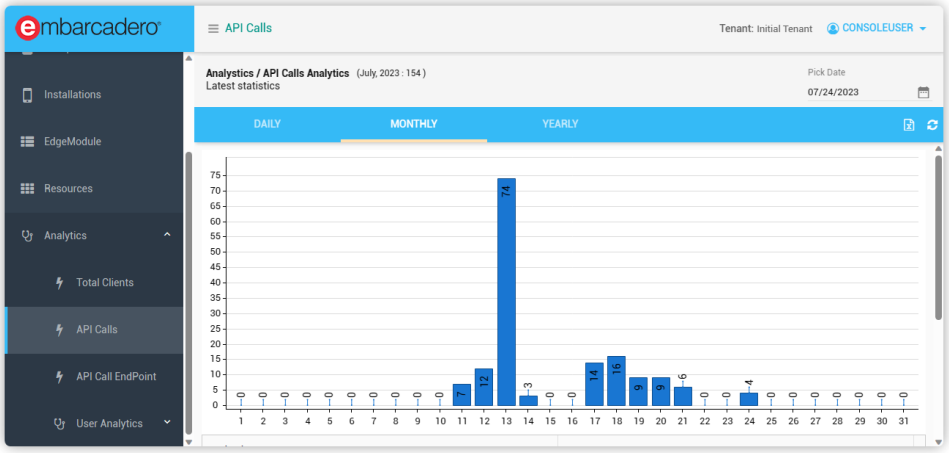
RAD 服务器控制台使用者列表

点选"Analytics"选单项目会开启一个选单,可从一系列分析中进行选择,包括客户端总数、API 呼叫、呼叫的 API 端点等. 分析数据可依日、月、年选择分析. 分析还可以按使用者,群组等和特定端点进行过滤.分析结果还可以保存到 .CSV 档案中,以便外部应用程序进行额外处理.

窍门

这些分析数据为决策过程和审计提供了重要讯息. 查看您的服务的使用量可提供您用于规划更新,或查看哪些端点很少使用,这些都是非常有价值的见解的范例.

下图显示了选定月份的 API 呼叫图表.



RAD 服务器控制台 Ext JS API 呼叫分析页面

09

部署 RAD 服务器

在此之前,第一个 RAD 服务器应用程序是使用 RAD 服务器 (EMSDevServer.exe) 和控制台 (EMSDevConsole.exe) 应用程序的开发版本进行测试的. 本章介绍了您可以在生产环境中部署 RAD 服务器的多个平台.如果您对 RAD Server Lite 有兴趣,请跳至下一章.

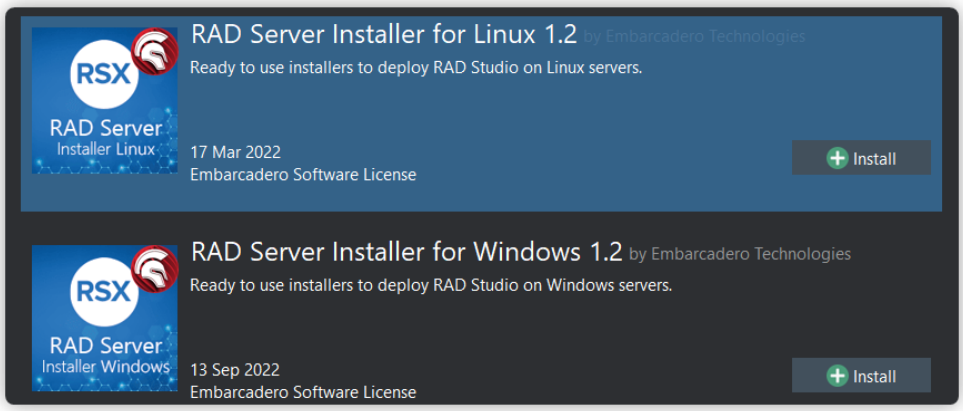
编译新的 bpl 或 dcp 资源时,它不会包含在项目的"export"文件夹中(二进制文件通常所在的位置).这些资源将预设建立在您的 Embarcadero Studio 安装路径中: C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\”Bpl 或 Dcp”Bpl 或 Dcp 文件夹内有特定于平台的专属文件夹.

RAD 服务器可以部署在那些平台

RAD Server 与 **Windows**、**Linux** 和 **Docker** 平台相容. 虽然从概念上讲,每个平台所需的服务是相同的,但我们将在本章中看到一些差异,但首先,让我们讨论一下相似之处以及 RAD Server 在幕后如何运作.

使用 GetIt 中的安装程序

如果您要在 Windows 或 Linux 上部署 RAD 服务器应用程序, 最快的安装方法是使用可从 GetIt 下载的安装程序. 你只需要搜寻"RAD Server",你就会找到这两个安装程序:



来自 GetIt 的 RAD 服务器安装程序

一旦"安装"完成后（其实这只是一个下载的动作),您可以在下列路径中找到安装程序:C:\Users\<YourUserName>\Documents\Embarcadero\Studio\<XX>.0\CatalogRepository 并将整个文件夹复制到您的生产环境计算机,此文件夹包含完全安装 RAD Server 所需的所有档案(包括 InterBase 安装程序).

在生产环境中执行安装程序之前,您必须安装 IIS 或 Apache,以便安装程序可以相应地配置所有要求.

安装程序将引导您完成安装所需的不同选项.

备注

在安装过程中,系统会要求您提供有效的 InterBase 授权.请使用您的 EDN 账户和 RAD 服务器序号注册 InterBase 实例.

手动部署 RAD 服务器的先决条件

本章重点在于讨论部署 RAD 服务器所需安装或设定的所有部分. 即使您使用安装程序,为了更好地除错和解决问题,了解所有要求也很重要. 另外,如果您需要将 RAD 服务器更新到较新的版本,则无需重新安装整个应用程序,只需更新一些 dll 和 bpls/so 档案就足够了.

下面是在生产环境中安装 RAD 服务器的强制性要求:

- InterBase 服务器引擎
- RAD 服务器授权
- RAD 服务器安装
- Web 服务器(IIS 7+ 或 Apache 2.4+)
- 使用 RAD Studio 编译的资源文件
- 配置 EMSServer.ini 文件

无论您选择部署哪个平台,您都需要安装/配置所有这些步骤. 例如,在 Windows 上,您需要为 Windows 设定 Microsoft 的 Web Server IIS 或 Apache,而在 Linux 上，,将需要设定 Apache. 重要的是要了解 RAD Server 本身并不是可执行文件(除了 Lite 版本,稍后会详细介绍). 资源以 Windows 的 BPL 或 Linux 的 SO 函式库的形式编译.这就是为什么我们需要一个网页服务器来存取这些资源.

对于 InterBase,RAD Server 内部需要使用一个数据库实例. 保存了大量信息需要被储存(统计数据、用户、角色等),这就是为什么它需要自己的数据库来储存所有这些信息.

RAD Server 在内部使用 InterBase 的事实并不意味着您必须为自己的数据使用此数据库引擎。.FireDAC 连接到广泛的数据库,您可以选择适合您需求的数据库.


备注

如果您选择使用 InterBase 数据库并且将其部署在同一台计算机上,则您将需要在不同端口上执行两个实例. 通常的做法是为您自己的数据库实例保留端口 3050,并在另一个端口 3051 安装 RAD Server InterBase 实例.. 无法使用相同实例,因为 RAD Server 使用自己的加密系统.

在 Windows 上手动部署

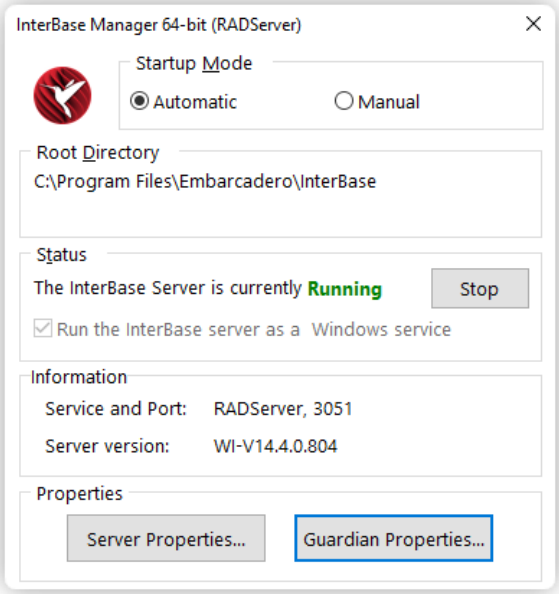
InterBase Server 引擎

从 https://my.embarcadero.com 下载适用于 Windows 的最新 InterBase 安装程序并将其安装在您的生产计算机上. 如果您以前从未安装过,可以按照 本教学 进行操作. 您也可以在此处找到 Windows 上生产环境的 RAD 服务器数据库要求.

安装的具体细节:

- 选择“Server 和 Client”
- 允许在同一台计算机上执行 InterBase 的多个实例
- 建议将预设端口变更为 3051
- 将实例命名为 RADServer(而不是预设的 gds_db)
- 若要注册 InterBase,请使用为您提供的相同 RAD 服务器序号和您的 EDN 账户.

安装完成后,您需要启动 InterBase 服务器的 RADServer 实例. 选择 Start | Programs | Embarcadero InterBase | 64-bit instance = RADServer | InterBase Server Manager. 如果您希望 InterBase 作为服务执行(默认值),请选取该方块.如果您希望 InterBase 在计算机启动时运行,请单击"Automatic"单选按钮.然后点击开始按钮.



适用于 RADServer 的 64 位 InterBase 管理员

窍门

您可以在程序列表中的"Embarcadero InterBase"下或安装它的路径中的文件夹“.\bin\IBMgr.exe”下找到 InterBase Manager,并指定实例名称，例如:".\IBMgr.exe RADServer".另一个选择是简单地使用 Windows 搜寻并输入"InterBase Manager".

RAD Server 安装

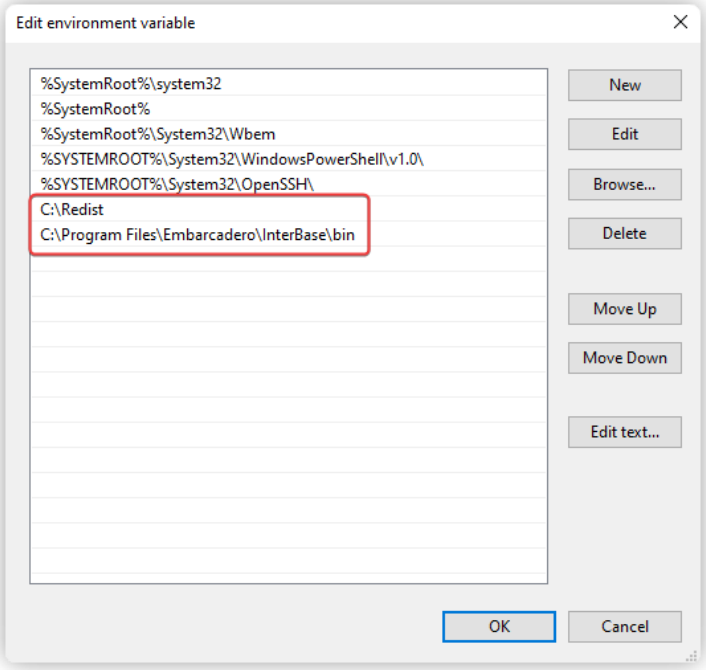
要在 Windows 计算机上安装 RAD 服务器,我们需要遵循与在开发计算机中设定它时非常相似的步骤. 此过程所需的大部分文件都可以在这些文件夹中找到:

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

在 docwiki 中有一个关于如何在 Windows 上安装 RAD Server 的非常详细的教学课程. 您可以 在这里找到它 尽管如此,我们将在这里解释基本步骤:

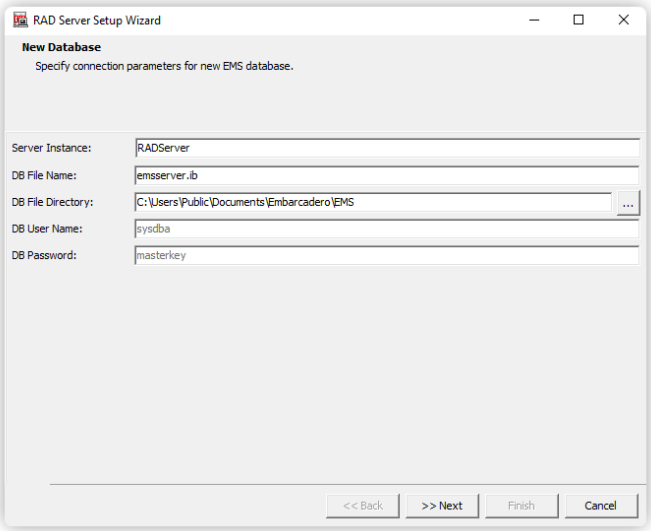
请依照下列步骤准备生产服务器以测试并使用 InterBase RAD Server 实例和 EMSDevServer.EXE 以便建立 RAD Server 数据库和配置文件.

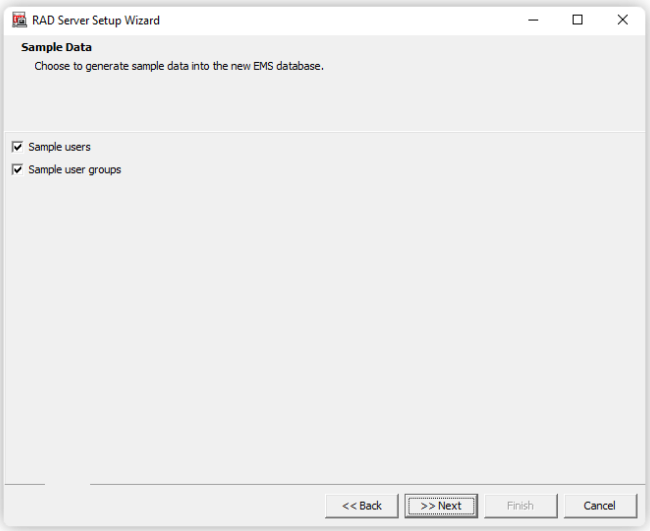
1. 将 64 位 EMSDevServer.exe 复制到生产服务器的 c:\installs\EMS 文件夹中
2. 将所需档案从 RAD Studio Redist/win64 文件夹复制到生产服务器上名为 c:\Redist 的文件夹中
3. 编辑生产服务器上的系统路径环境变量,新增 c:\Redist 和 c:\Program Files\InterBase\bin 文件夹.



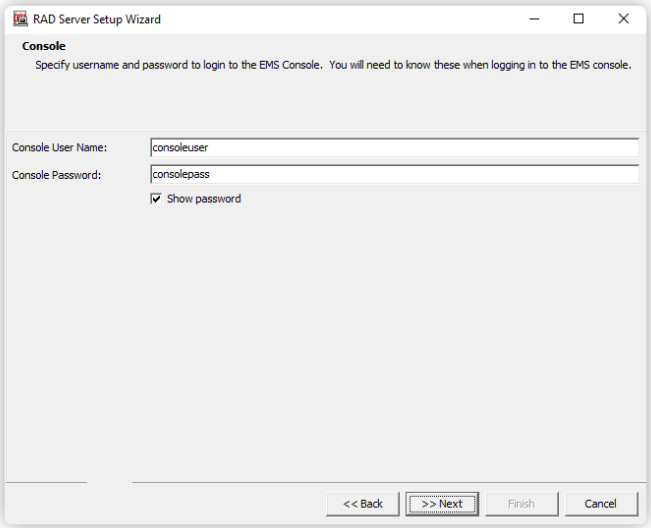
将两个文件夹新增至您的系统路径

4. 将开发计算机上的 EMS 模板和 Web 资源文件从 C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\ObjRepos\en\EMS 复制到名为 c:\installs\ObjRepos\EMS 的生产服务器文件夹中(EMSDevServer.EXE 会在放置 EMSDevServer 文件夹的相同父文件夹下的 ObjRepos\EMS 子文件夹中寻找范本和 Web 资源文件)
5. 确定在生产服务器上启动了具有 RAD Server 许可证的 InterBase 服务器.
6. 执行 EMSDevServer.exe(如同第一个 RAD 服务器开发设定中所做的)以设定生产 RAD 服务器配置文件和 InterBase RAD 服务器数据库.以下的画面显示这些步骤.

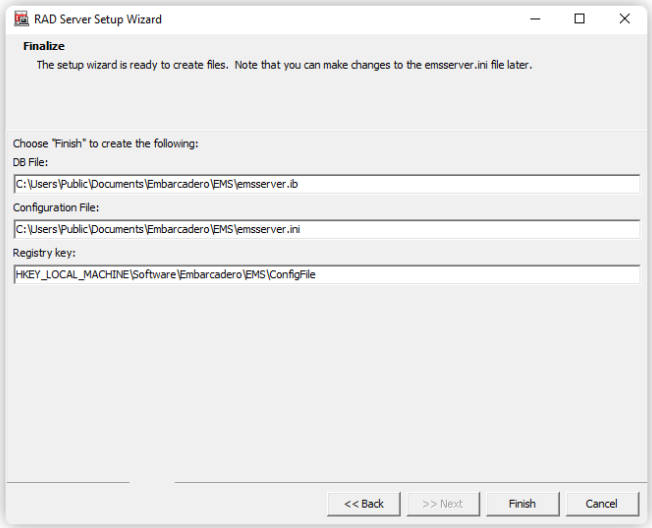




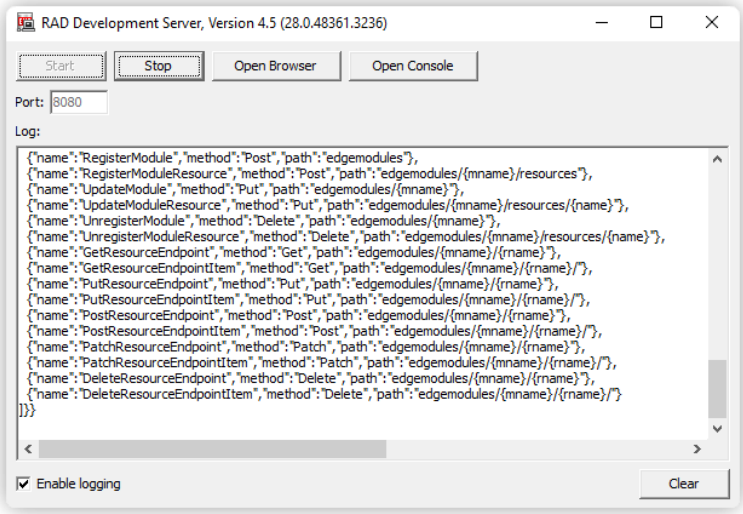
RAD 服务器设定精灵 - 设定 RAD 服务器的联机参数



RAD 服务器设定精灵 - 选择产生范例数据

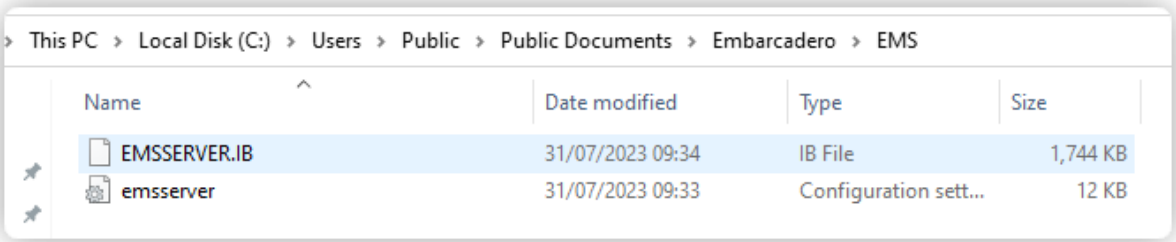


RAD 服务器设定精灵 - 设定 RAD 服务器的联机参数 RAD 服务器安装精灵 - 检视将建立的档案和登录项



RAD Server 开发服务器正在执行

RAD 服务器精灵将在 Public Documents 文件夹下的预设文件夹中建立两个文件.



RAD 服务器安装精灵 - 在公用文件文件夹中建立的两个文件

Web 服务器(IIS 或 Apache)

如果您在 Windows 上部署,则可以使用 Microsoft 随 Windows 提供的 Web 服务器(又称为 IIS),也可以使用 Apache for Windows. 在 本连结您可以查看详细指南,它不仅详细说明需要复制到生产环境的文件,还说明如何在 Windows 计算机上设定 IIS 或 Apache.

I 如果您选择 IIS,Microsoft 有不同的版本,且服务的安装过程可能会有所不同. 如果您以前没有使用此服务的经验,您可以在以下位置找到信息: 连结.

最后一步是复制我们编译的资源.到本章结尾查看如何操作.

如果您在服务器管理员上使用"新增角色或功能"选项来新增 Web 服务器 (IIS),则必须在"应用程序开发"部分下勾选安装"ISAPI 扩充功能"和"ISAPI 过滤器"

在 Linux 上手动部署

将 RAD 服务器和应用程序部署到 Linux 服务器提供了以下选项:

- 若要建立独立 RAD 服务器,请参阅 RAD 服务器安装章节
- 若要建立适用于 Apache 的 RAD 服务器,请参阅设定适用于 Apache 的 RAD 服务器章节

相容的 Distros

RAD Server 正式支持 Ubuntu 18+ 和 RHEL 7+.这并不意味着它不能安装在 RockyLinux、Debian 等其他发行版中,但内部测试始终使用官方支持的发行版进行测试.

安装 InterBase Server 引擎

从 https://my.embarcadero.com 下载适用于 Linux 的最新 InterBase 安装程序. 在 zip 档案内,您将找到安装程序. 在这里您还可以找到 Linux 上生产环境的 RAD 服务器数据库要求.

解压缩下载的档案后,为安装程序指派执行权限并执行它:

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

安装的具体细节:

- 选择 “Server 和 Client”
- 允许在同一台计算机上执行 InterBase 的多个实例
- 建议将预设端口变更为 3051

- 将实例命名为 RADServer(而不是预设的 gds_db)
- 安装路径: /opt/interbase

窍门

InterBase 安装程序将自动侦测您的Linux 安装是否有桌面环境. 如果您想强制控制面板模式执行安装程序, 请使用此参数:sudo ./install_linux_x86_64.sh ***-i Console***

备注

您可以使用您喜欢的名称定义实例和路径的名称.如果这样做,请记住在配置过程中相应地参考这些内容.

注册并启动 InterBase Server

若要启动注册精灵,请执行指令:

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

这将启动授权精灵. 对于控制面板模式,我们建议选择选项 2“Direct register”,您可以在其中指定您的 RAD 服务器序号以及您的 EDN 账户. 助理将完成其余的工作,并验证您连接到 Embarcadero 服务器的授权.

如果您现在想验证授权是否已正确加载,您可以使用上一个选单"List license"中的选项 1 来确认一切按预期进行.

InterBase 实例已安装并获得授权,但需要启动它.为此,我们需要进入 InterBase 控制台执行此命令

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

为了简化其他应用程序和服务与 InterBase 数据库的连接,最简单的方法是建立一个到 InterBase 函式库的符号链接并将其指向 /usr/lib. 这将避免您需要将函式库复制到需要 InterBase 连接的每个服务的位置.

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

将 InterBase 作为服务执行

InterBase 也可以设定为服务,以便在 Linux 启动时运作.在终端机窗口中使用以下命令.

存取您安装 interbase 的路径中的"examples」"文件夹,并将 ibserverd 脚本档案复制到您安装的服务器实例版本的位置:

```
sudo cp ibserverd ibserverd_RADServer
```

透过以'sudo'或'root'身分执行上述脚本来设定自动服务启动.

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

第二个参数是安装文件夹,第三个参数是实例名称.现在,当您重新启动系统时,只要获得正确许授权, 服务就会自动启动.

检查以确保 InterBase 设定为下次重新启动时作为服务启动.

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

当将 InterBase 作为服务运行时,只要计算机在多用户模式下执行,InterBase 服务器就会自动启动.

如果您喜欢手动建立服务(或者您的 Linux 发行版使用稍微不同的方法),您可以在以下位置找到有关此设定的详细信息连结

备注

若要将 InterBase 作为服务删除,请执行:
sudo /opt/interbase/examples/ibservice.sh -r[emove]

安装 RAD Server

在安装了 RAD Studio 的计算机上,您可以在下列路径中找到 RAD Server Linux 安装程序：C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\EMSServer

将这些档案复制到您的 Linux 计算机并执行安装程序.您可能需要授予它执行权限.

警告

*确保已安装 libcurl.如要安装它,请使用您的发行版套件管理器.例如基于 Debian 的可执行：**apt install libcurl4***

安装 shell 脚本将建立一个目录 /usr/lib/ems,其中包含 EMSDevServerCommand、EMSDevConsoleCommand 以及执行命令档案所需的几个执行时期函式库 (.so) 文件. 您也可以在 此连结 中的 docwiki 中找到包含详细信息的教学课程.

安装完成后,在设定模式下执行 EMSDevConsole:

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

输入 **start** a 并且按下 enter.

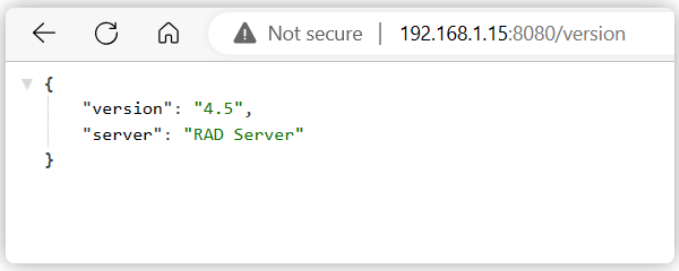
透过输入以下值指定联机参数:

- 服务器实例：键入下列默认实例名称 **RADServer**
- DB 文件名: 内定名称是 **emsserver.ib**
- DB 档案目录: **/usr/lib/ems**
- DB 使用者名称: 内定参数是 **sysdba**
- DB 密码: 内定参数是 **masterkey**
- Console 使用者名称: 内定值是 **consoleuser**
- Console 密码: 内定值是 **consolepass**

输入 **“n”** 如果配置选项是正确的. emsserver.ini 和 emsserver.ib 档案已创建,RAD 服务器在端口 8080 上开始执行..

配置过程完成后,您可以在 **/usr/lib/ems** 中找到 RADServer 数据库,在 **/etc/ems** 中找到配置文件.

保持 RADServer 执行,现在让我们测试一下我们是否可以正确存取它并得到响应.存取: *http://<LinuxMachineIP>:8080/version*



显示呼叫版本端点输出的浏览器

EMSDevServerCommand 和 EMSDevConsoleCommand 可用于开发和测试 Linux RAD 服务器应用程序, 而无需使用 Apache. 下一步是设定和测试 RAD 服务器和 Delphi/C++ 编译的应用程序模块,以便在 Linux 和 Apache 上以生产模式执行.

窍门

如果您想在 Linux 上部署 RAD 服务器,同时也使用 InterBase 作为您的数据选择数据库,您可以参照本教学.

为 Apache 设定 RAD 服务器

使用 InterBase iSQL 指令（在 /opt/interbase/bin 目录中）确保 RAD 服务器能够连接到 emsserver.ib 数据库文件.

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

设定 Apache HTTP Server 以加载 Apache RAD 服务器 (libmod_emsserver.so) 和 Apache RAD 服务器控制台 (libmod_emsconsole.so) 模块. 尽管无论您使用哪种 Linux 发行版,Apache 的配置都非常相似,但请记住 RHEL 和基于 Debian 的发行版之间存在一些差异.

备注

检查 Linux 发行版的文文件,以验证加载模块和定义位置卷标的建议方法.

新增下面的设定以加载 RAD Server Apache 服务器模块 (libmod_emsserver.so) 和 RAD Server Apache 控制面板模块 (libmod_emsconsole.so).

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

新增位置卷标以建立容器,您可以在其中指定给定 URL 的访问控制规则.

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

若要测试您的 RAD 服务器是否正确执行,请使用浏览器透过存取来显示 RAD 服务器版本号: http://<LinuxMachineIP>/radserver/version

最后一步是复制我们编译的资源,请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 简单得多。 Embarcadero 在 dockerhub 中有多种可用于此平台的映像.

您会发现 2 个与 RAD Server 相关的镜像：这两个镜像之间的唯一区别是，一个在容器内运行 InterBase 服务器引擎，另一个假设运行 RAD Server 所需的 InterBase 服务器将托管在其他地方.

InterBase 服务器也与 Docker 兼容，而 Embarcadero 提供了一个映像来进行容器化。这是 DockerHub 的连结.

这些 Docker 映像的建置方式是完全开源的，并可在 GitHub 上公开取得. 这只是一种方法，但如果您对 Docker 足够熟悉，请随意使用这些作为模板并根据您的特定需求进行调整.

下面的 DockerHub 和 GitHub 连结中有大量有关如何部署和自定义这些映像的信息.

选项 1: PA-RADServer-IB

这个镜像就是我们所说的"包含所有电池".容易，但请记住，第一次执行此容器时，您无法在分离模式下执行此操作. 需要执行第一个精灵来设定 RAD 服务器授权和一些额外的详细信息。一切设定完毕后，您可以独立运行它.

另一件需要记住的事情是，如果您没有计划扩展应用程序并且希望将所有内容放在一个地方，那么这个容器非常方便, 另一件需要记住的事情是，如果您没有计划扩展应用程序并且希望将所有内容放在一个地方，那么这个容器非常方便

但如果您想在未来扩展应用程序，也许最好的方法是将 RAD 服务器与 InterBase 服务器分开，并将它们放在单独的容器/机器中.

DockerHub 连结

GitHub 连结

本镜像包含:

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

选项 2: PA-RADServer

该容器需要连接到安装了有效 RAD 服务器授权的 InterBase 服务器，否则它将无法运作. 如果您想要扩展应用程序并部署连接到相同 InterBase Server 的多个实例，它是一个理想的容器.

DockerHub 连结

GitHub 连结

本镜像包含:

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

> 窍门
>
> *请记住，对于简单的环境，您可以使用 PAServer 将资源更新直接上传到容器，而无需重新产生它.*

存取 本连结以获取有关在 Docker 上部署 RAD 服务器的更多信息.

复制使用 RAD Studio 编译的 RAD 服务器模块

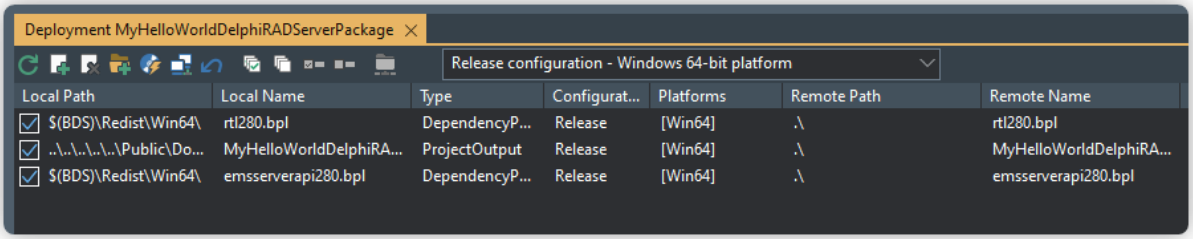
无论您选择哪种操作系统，将模块或所需的附加函式库部署到生产计算机的过程都几乎相同. 对于您自己的资源，您只需将.bpl/.so 档案复制到您的生产机器上.

RAD 服务器应用程序套件档案根据项目设定编译到相对应的文件夹。默认的 Delphi 套件输出和 C++ 最终输出目录是:

- 对于 Delphi:
  - 32 位 Windows - C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\Bpl
  - 64 位 Windows - C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\Bpl\Win64
  - Linux - C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\Bpl\Linux64
-
- 对于 C++，所有 RAD 服务器应用程序包档案都编译到 .\$(Platform)\$(Config) 文件夹.

有多种方法可以将所需的 RAD 服务器应用程序和执行时期 DLL 档案部署到生产服务器。三种常见的传输方法是:

- 将套件档案复制到 RADServer 安装的生产服务器路径下
- 透过 FTP 将文件传输到生产服务器
- 使用平台助理(PAServer) 和"Project| Deployment"选单项目让 IDE 将档案移到生产服务器。此屏幕截图显示了 Windows 64 的范例.



Project | Deployment PAServer 可以传输的文件

将编译好的 RAD Server 扩充套件档案（例如，第 1 章 MyHelloWorldDelphiRADServerPackage 中的项目）复制到生产 RADServer 文件夹.


窍门

使用 PAServer 时，可以透过在生产计算机中编辑档案 paserver.config 来变更部署档案的默认路径. 执行 PAServer 时可能需要提升权限，这取决于写入档案所需的路径.

配置 EMSServer.ini 文件

现在我们已将新资源新增至生产文件夹中，我们需要在 EMSServer.ini 档案中指定有可用的新资源.

编辑 emsserver.ini 档案以便在 [Server.Packages] 部分下新增每个 RAD 服务器扩充包.

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

若要设定已执行实例的 emsserver.ini 文件，请执行 ./config.sh 脚本。该脚本将自动重新启动 Apache.

10

RAD Server 精简版(Lite)

什么是精简版?

RAD Server 需要一个基于 InterBase 的后端数据库,并且通常部署为 IIS 或 Apache 的 Web 服务器 DLL 模块. 因此,标准部署需要:

- RAD 服务器模块的 Web 服务器及其配置
- RAD 服务器部署与配置
- 使用特殊用途 RAD 服务器授权安装的 InterBase(用户需要在目标装置上注册才能启动的授权)

对于开发来说,我们长期以来提供了基于 Indy HTTP 服务器的独立版本的 RAD Server,它提供有限的效能,但更容易部署并且能够在除错器下执行(因此您可以除错 RAD Server 模块程序代码). 开发版本并不能使用来部署.它对您可以建立的使用者数量有限制,并且可以与本机 InterBase 开发者版本一起使用(它的授权是 RAD Studio 授权的一部分).

RAD Server 精简版 (**RSLite**) 为不需要大量吞吐量的测试服务器和场景提供更简单的部署模型,它透过使用 InterBase 嵌入式数据库引擎 IBToGo(而不是成熟的服务器)来实现这一点,并将其与简化的授权模型相结合.

RSLite 使用与开发版本相同的二进制文件(随 RAD Studio 一起提供)以及 IBToGo 二进制文件和授权文件,您可以连同您的解决方案一起部署(无需在部署到的计算机上注册). 由于它使用嵌入式数据库并且使用 Indy HTTP Server 组件,因此它无法提供与常规完整 RAD Server 安装相同数量的每秒请求服务数量,并且无法透过多个 RAD Server 前端进行扩展.

RSLite 使用的底层架构只有有限的可扩展性,但我们希望它足以满足许多简单的部署场景 - 请记住,服务吞吐量也取决于 RAD 服务器模块执行的特定程序代码.


窍门

对于公共系统上的部署,我们建议避免直接公开 RSLite HTTP 服务器,而是透过代理配置对其进行访问,以便您仍然拥有一个 Web 服务器(如 Apache 或 IIS),为传入的 HTTPS 呼叫提供安全上下文并转发这些呼叫到 RSLite.

如何取得 RAD Server Lite 授权

您可以使用 RAD Studio 11(包括 Delphi 11 和 C++Builder 11)的任何企业版或 架构师授权来兑换授权.. 请造访此页面 并按照提供的说明进行操作.


备注

您需要注册密钥和 EDN 账户.

这里的流程不仅仅是接收 RSLite 的授权密钥,而是一个可以在安装时部署的 slip 档案(储存在 .TXT 档案中的授权). 此授权对安装数量没有限制,但您不能在同一台计算机上执行两个实例.


备注

授权文件需要放置在特定的子文件夹中,这与兑换网站上的一般信息似乎暗示的不同.

部署 RAD Server 精简版项目项目

在部署项目之前获得授权后,有两个不同的注意事项:

- 首先,您需要使用 RSLite、所需的运行时间套件和 IBToGo 部署配置 (以下是取得它的步骤)
- 其次,您需要产生一个适合生产的数据库文件,与 IBToGo 授权兼容 - 由 RAD Server 开发者版本建立的本机数据库将不兼容

要部署的文件档案

手动部署

实际上,这些是部署 RSLite 解决方案所需的档案(除了您的应用程序套件及其相依性之外):

1. RSLite 执行文件,与开发人员版本相同:RAD Studio bin 文件夹中提供的 EMSDevServer.exe(或类似的 64 位版本)
2. 所需的 RAD Studio 运行时间套件包,其中包括最小安装所需的套件包(此处列出并在 RAD Studio win32 或 win64 redist 文件夹中提供)以及 RAD 服务器模块中的程序代码所需的任何其他运行时套件包:
   - 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 文件夹(例如,C:\Users\Public Documents\Embarcadero\Interbase redist\InterBase2020)下的子文件夹 win32_togo 或 win64_togo 中找到 InterBase ToGo 部署档案— 对于 Linux,您可以找到 libibtogo. so 档案位于正确的 InterBase redist 文件夹中
4. 将上面获得的许可证文件加入到 interbase/license 文件夹中(IBToGo redist 配置的一部分)

使用部署精灵

请依照下列步骤使用部署精灵中的 RSLite 功能部署文件:

1. 新增 RSLite 功能.
2. 接下来新增 IBToGo 功能.
3. 取消 iblite 注册文件的勾选
4. 在如何取得 RAD Server Lite 授权部分中,新增要部署的档案：产生 rslite 启动档案并将其目标设定为 "interbase/license".
5. 接下来,新增从建立生产数据库部分获得的文件,以将我的 **emsserver.ini** 部署到 ./.
6. 最后,新增从建立生产数据库部分获得的文件,以将我的 **emsserver.ib** 部署到 ./.

MSVC 执行时期档案

要在目标 Windows 计算机上执行 IBToGo(以及使用 IBToGo 的 RSLite),需要安装 Visual C++ 2013 执行时期链接库. 在装有 RAD Studio 的开发计算机上,您很可能已经安装了它.但是,在通用目标部署计算机上,您可能必须安装它, 您可以从这里下载 Microsoft.

建立生产数据库

使用此配置,您可以透过执行 EMSDevServer.exe 应用程序来启动 RSLite. 请注意,如果目标计算机有 InterBase 客户端,它将作为更高优先级的选择,并且如果 InterBase 客户端是 RAD Studio 附带的开发者版本,则一切都将正常工作,但是在标准 RAD Server 开发者模式配置中.

您可以透过查看 RAD 服务器启动时日志中的前几行来弄清楚这一点.如果是"RSLite"配置,前几行将如下所示:

```
{"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"}}
```

如果程序代码显示"Lite"设定为 false,您可能需要手动停用 gds32.dll(或其 64 位版本)InterBase 客户端链接库的加载,该链接库通常位于 C:\Windows\SysWOW64 中(如果 InterBase 客户端找不到库它, 它就会加载本地 **ibtogo.dll**).

现在,如果您启动 RSLite(使用正确的配置)并且没有 emsserver.ini 文件和 emsserver.ib 数据库文件,它将提示您建立一个. 为此,RSLite 必须在 RAD Studio 的对象储存库文件夹(产品文件夹下的 ObjRepos)寻找配置. 更简单的方法是将 Program Files (x86)\Embarcadero\Studio\<XX>.0\ObjRepos\en\ems 下的档案复制到具有 **emsdevserver.exe** 相对路径的文件夹中："../ObjRepos /EMS".换句话说,您需要一个与包含 RSLite 安装的文件夹(项目部署目录)处于相同等级目录的 ObjRepos 文件夹.

备注

每个 RSLite 部署都不需要重复这样做,只需产生一次生产数据库,您可以稍后按原样复制到目标计算机上. 事实上,在开发环境中建立的数据库与 RSLite 部署不兼容.

我们建议您指定与 RSLite 部署相同的目标文件夹,以便精灵将在您的部署文件夹中建立 **emsserver.ini** 档案和 **emsserver.ib** 数据库文件. 现在,取得 RSLite、这些配置文件、运行时间套件和 IBToGo(包括授权)和整个文件夹,您就拥有了在目标 Windows 计算机上部署所需的所有内容.

Proxy 配置

由于 RSLite 在保护和加密方面的局限性,不建议直接将 RSLite 公开为公共 Web 应用程序. 我们建议使用代理层和专用服务或使用流行的 Web 服务之一作为前端. 例如,在 Apache 中,您设定虚拟主机,启用 HTTPS,并使用下列设定将流量重新导向至 RSLite 实例:

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

对于 Linux

对于 Linux,您可以按照与上面类似的步骤操作,一切都应该按预期工作.作为替代方案,您也可以考虑安装完整的 RAD 服务器,然后将 IBToGo 新增至安装中:

- 使用 RAD 安装文件夹中提供的 ems_install.sh 安装 RAD 服务器
- 将 IBToGo 档案从 InterBase"redist"文件夹复制到 Linux 上的 EMS 文件夹 (/usr/lib/ems)
- 执行 EMSDevServerCommand,依照精灵建立 EMS 数据库和配置文件

备注

您可能需要透过 sudo 运行应用程序以获得适当的权限

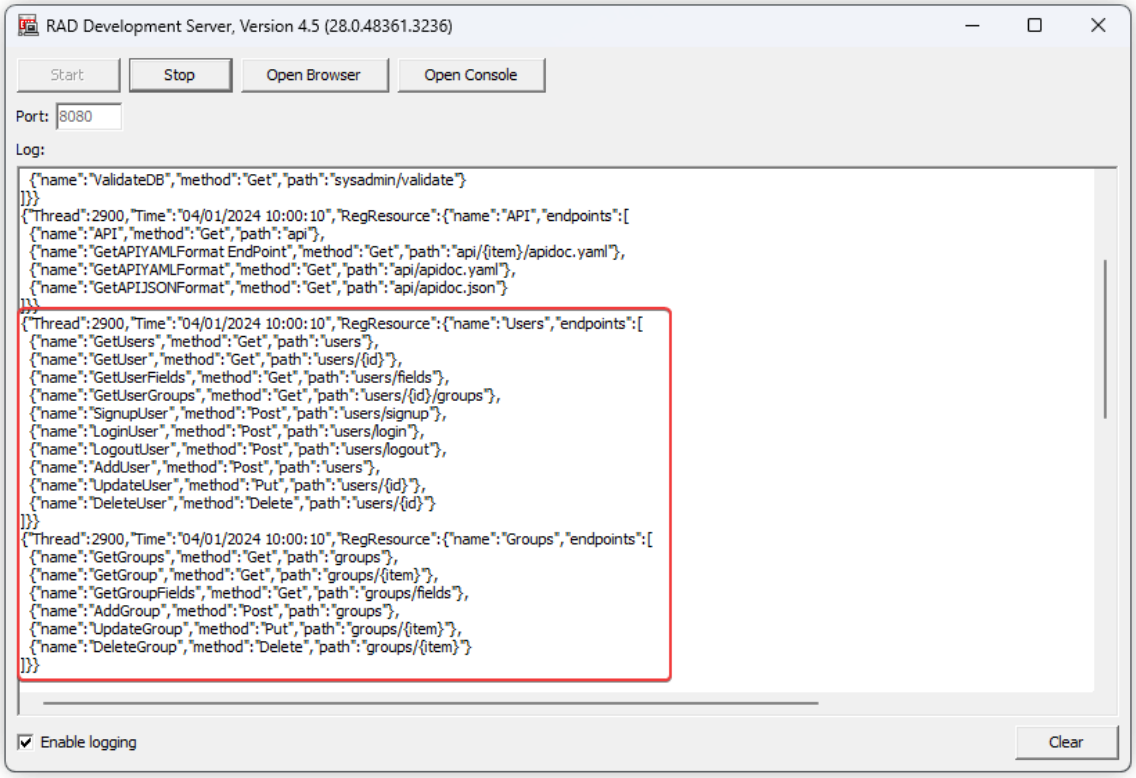
11

认证与授权

RAD Server 为使用者验证和授权提供开箱即用的解决方案. 多个端点可用于处理所有这些事项,并且不仅可以透过请求存取它们,还可以从使用 RAD Studio 开发的其他基于 RAD 服务器的应用程序或以程序设计方式从 RAD 服务器资源存取它们,如果您想整合自己的身份验证系统并且覆盖 RAD 服务器提供的服务的话, ,这样的开放功能非常的有用..

整合性的身份验证:管理使用者和群组

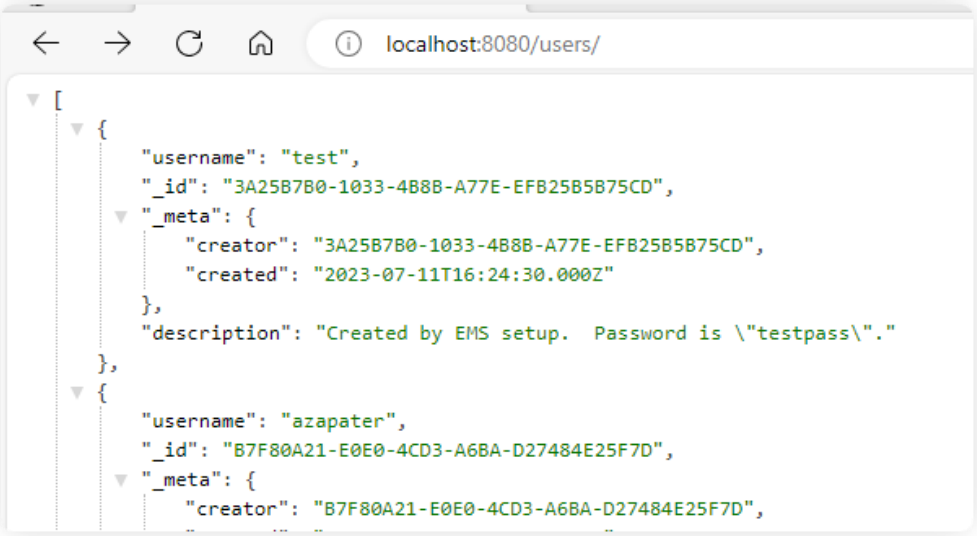
在我们的开发机器中执行 RAD 服务器实例后,如果我们分析日志,我们可以看到一些重要的服务端点:



RAD 服务器为使用者和群组管理所建立的预设端点

有 2 个与身分验证相关的资源: 使用者 和 群组. RAD 服务器不仅允许我们定义用户,还可以将这些用户分配到特定群组,以便可以以更精细的方式定义角色并授予或拒绝对特定端点和/或资源的权限.

让我们访问端点 users/ 看看我们能得到什么.



存取 RAD 服务器中建立的使用者列表

RAD 服务器传回一个数组数组,其中包含数据库中建立的所有用户.正如我们在屏幕截图中看到的,预设情况下,RAD 服务器会建立一个测试用户,密码为"testpass"(在安装精灵期间定义).


备注

如果您想查看所有资源的更多详细信息,您可以在 OpenAPI json/yaml apidoc 文件规范中定义的文档中找到它们.

使用者和群组资源遵循标准 CRUD 约定,因此如果您想要建立新记录,则需要向此端点发送 POST 请求.要修改一个记录,则需要 PUT 请求(在正文请求中包含新值)等等.

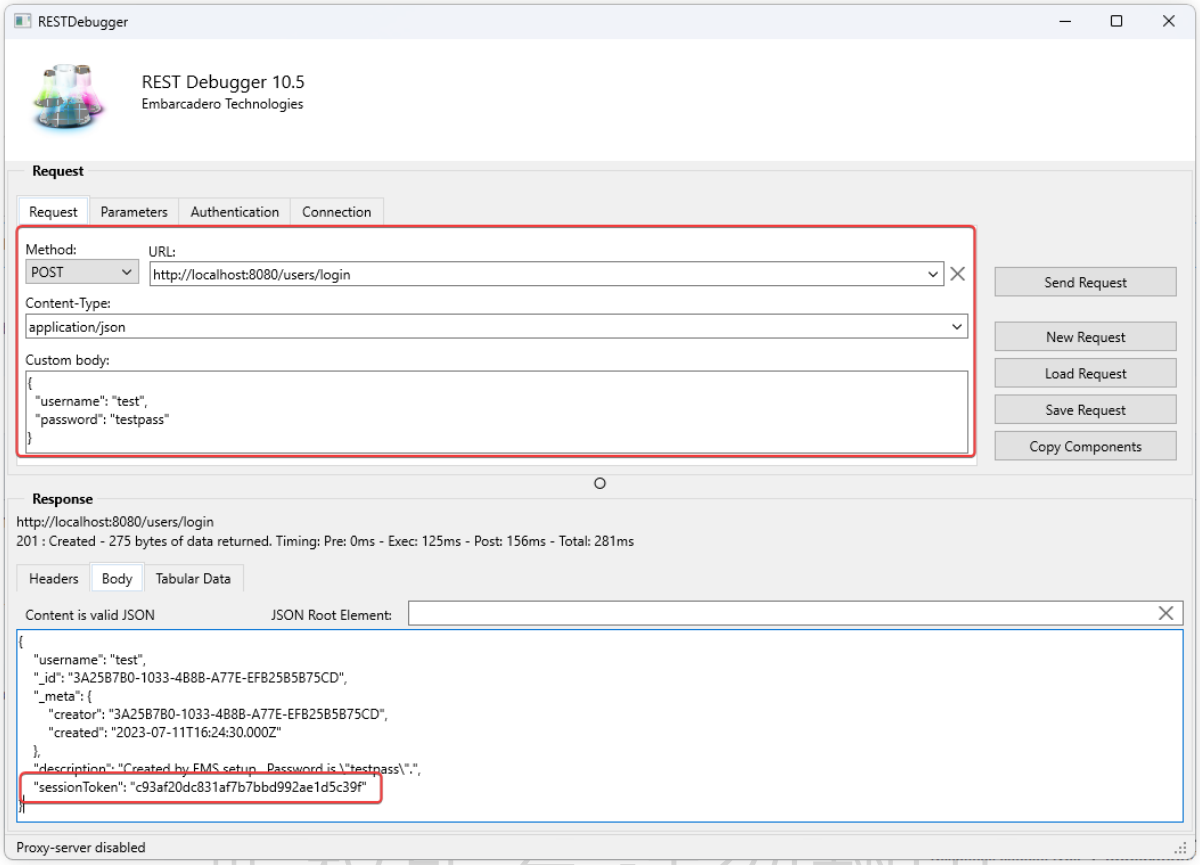

窍门

可以向使用者和群组新增自定义字段.只需在请求正文中添加任意字段字段,它将储存为额外的字段字段.

登录

创建的用户将能够登入 RAD 服务器并获取标志,该标志将用于在将来的请求中识别他们.

正如我们在自动建立的端点中看到的,要登录 RAD Server,我们只需透过 POST 请求存取 users/login 端点,并在正文中包含用户名称/密码.



存取 POST users/login 端点以取得 sessionToken

一旦我们有了 sessionToken,我们必须将该值插入到未来的请求标头中,以便 RAD 服务器可以正确识别用户.

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

注销

默认情况下,会话标志的到期日期是无限的.然而,出于安全原因,这可能不是最好的设定.可以使用 EMSServer.ini 中的下列设定来微调标志到期日期/时间.有一些参数可以配置这个:

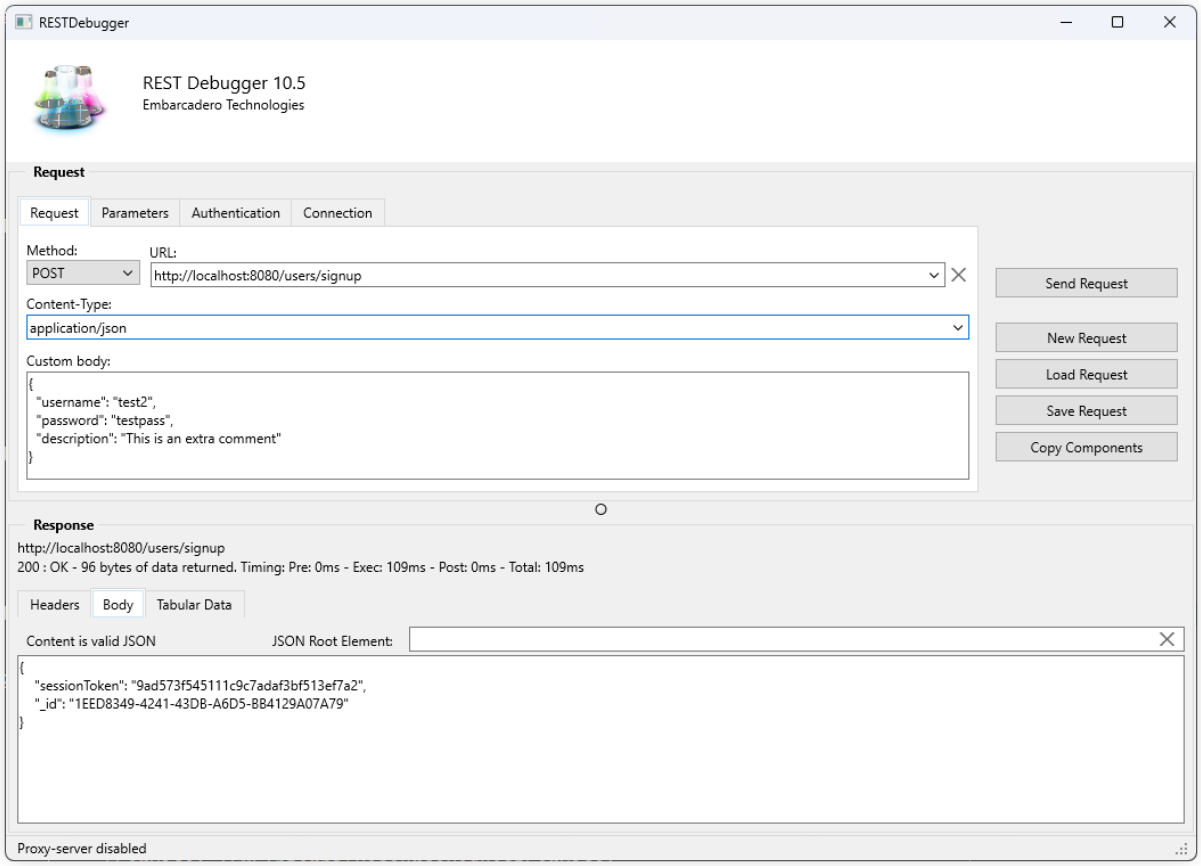
```
SessionInactivityTimeout=
;# Set SessionInactivityTimeout=60 to limit maximum time of a session inactivity in
seconds.
;# This operates on a session token.  Default is 0 (no timeout).
SessionLiveTimeout=
;# Set SessionLiveTimeout=60 to limit maximum time of a session live in seconds.
;# This operates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

在这两种情况下,超时都必须以秒为单位指定.一旦超过其中一个超时时间,标志将被自动停用.

如果我们想要强制注销并手动停用活动标志,我们可以向端点 users/logout 发送请求(在标头中包含用户的标志).

报名

如果我们向 /users/signup 发送请求,我们将在 RAD Server 中注册用户,同时我们将取得 ID 以及会话标志.



存取 POST users/singup 端点以建立新用户

在上面的范例中,需要注意的是,可以在请求正文中发送任意数量的自定义字段(例如，我们使用了'description'自定义字段).

使用者名称是唯一的,因此如果发送使用现有使用者名称的注册请求,RAD 服务器将响应 409 错误.

备注

不想要公开注册程序?不要忘记透过将其限制为角色部分中的授权用户和群组来限制对 users.signup 端点的存取.欲了解更多信息,请参阅授权章节.

管理群组

也可以使用整合端点来管理群组.此资源也遵循标准 CRUD 约定.

一个用户可以属于多个群组,一个群组当然可以包含多个用户.

重要提示:要将使用者指派到群组,必须将 POST 请求设定为 /users/groups/{id},其中包含该群组的所有成员的数组数据.

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

即使您不打算在群组层级定义角色,我们也鼓励您将使用者指派到群组. 这在分析中非常有用,因为可以根据这些群组显示特定的详细信息.

整合性授权

全局凭证

可以定义多个安全等级的全局凭证. 在 RAD 服务器引擎配置文件(emsserver.ini 档案)中,[Server.Keys] 群组下有 3 个与此相关的参数:

MasterSecret
它授权您完全存取储存在 RAD 服务器数据库中的所有 RAD 服务器数据.

来执行管理任务.您可以使用它来存取 RAD 服务器引擎(EMS 服务器)中的所有 RAD 服务器资源.

AppSecret
它授权您从 RAD 服务器客户端应用程序存取授权端点.

ApplicationID
应用程序 ID 用于识别来自基于 RAD 服务器的客户端的请求,因此仅处理具有有效应用程序 ID 的请求,而拒绝具有无效应用程序 ID 的请求. 此标识符可用于区分不同的 RAD 服务器实例.

您可以在此处找到更详细的信息 docwiki link.

使用者和群组授权

保护 RAD 服务器端点的最简单方法是透过 EMSServer.ini 档案.所有规则必须在[Server.Authorization]部分下定义.让我们看看 ini 档案中默认的范例:

```
[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
```

尽管该文件已经提供了非常不言自明的注释,让我们看几个范例.

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

销售组的所有成员都可以存取资源 Sales 中的所有端点.

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

在上面的范例中,我们限制了对名为'Users'的整个资源的公共访问. 但是,我们特别允许公共存取端点'LoginUser'和'SignupUser'. 这是限制对整个资源进行存取, 但一些必需的例外的访问的非常有用的方法.密钥'public'意味着将可以在没有任何身份验证目标情况下处理请求.

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

在上面的范例中,所有具有有效证目标请求都可以存取资源 Customers,无论其角色如何,但它们必须注册.

自定义认证

如果您已经实现了身份验证服务,例如 ActiveDirectory/LDAP 或任何其他第三方服务,则可以将其与 RAD Server 整合. 在范例目录中,您可以找到 2 个自定义登入范例以及与 AD 基本整合的范例(只要 RAD 服务器在 Windows 计算机上执行).

要考虑的主要前提是我们需要验证我们的身份验证服务所提供的凭证是否正确. 验证后,我们必须在 RAD 服务器内建验证中对使用者进行身份验证(如果是第一次连接,则建立使用者),然后返回 RAD 服务器标志.

为了从我们的程序代码存取 RAD 服务器的内部 API,我们可以建立它的一个新实例,指定应使用的上下文.让我们来看一个例子:

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);
        }
}
```

为了让 RAD 服务器知道该请求来自授权用户,如果外部身分验证服务验证了请求中包含的凭证,我们只需注册或登入该用户即可.

您可能已经注意到,我们没有使用相同的密码来建立 RAD 服务器用户.这是为什么?好吧,如果用户在外部身份验证服务中更改密码,我们也需要更新 RAD 服务器密码,这会增加额外的复杂性以保持所有内容为最新. 对用户密码进行哈希处理将使我们能够完全依赖身分验证外部服务,同时仍在 RAD 服务器实例中保留安全机制.您可以检查范例项目项目以了解更详细的实施情况.


备注

customLogin 和 AD 整合的范例可以在 RAD Studio 的范例文件夹中找到:
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

是否必须在 RAD 服务器中单独建立每个用户? 从技术上来说是不用的，但强烈推荐. 主要原因是为了分析和日志记录. 如果我们在 RAD Server 中单独建立每个用户,我们将能够利用嵌入式分析并单独获取每个用户的精细数据. 从技术上讲,可以为所有登入重复使用相同的 RAD 服务器"常规"用户,并为每个登入建立一个标志,但需要明确了解的是,分析功能将变得不太有用.

在前面的范例中,我们了解如何使用内部 RAD 服务器 API 以程序设计方式注册和登入用户,但这只是此 API 的多种可能性中的 2 种. 建立群组,将使用者指派给群组…透过我们在前面几节中讨论的 API REST 实现的几乎所有功能都可以透过程序设计方式实现.

所有 RAD 服务器内部 API 均位于 EMS.Services 程序单元中. 请查看程序代码或这份 文件以便查看所有可用的方法.

客制化授权

另一种选择是以程序方式保护端点.您可以在每个端点中定义特定规则以允许或禁止存取使用者和/或群组.

连结到请求的每个方法都有一个 AContext 参数,我们可以使用它来检查分配给请求的使用者或群组.

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
```

经过所需的验证后,如果我们不想授予对该方法的访问权限,我们可以简单地使用参数 AResponse 传回未经授权的错误.

Delphi

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

C++

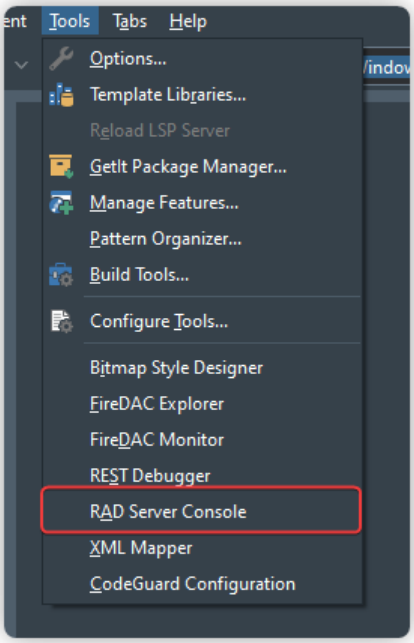
```
AResponse->RaiseUnauthorized("Unauthorized access", "");
```

RAD 服务器管理控制台

RAD 服务器控制台又名 RSConsole 已经存在很多年了,但它有点隐藏在路径中:

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

从 RAD Studio 12 Athens 开始，可以在"Tools"菜单下找到它.



用于存取 RAD 服务器控制台的选单项

建立新的配置文件

此控制台可用于连接到本机开发环境或生产中的远程服务器.它允许您定义多个配置文件并按需连接到它们.

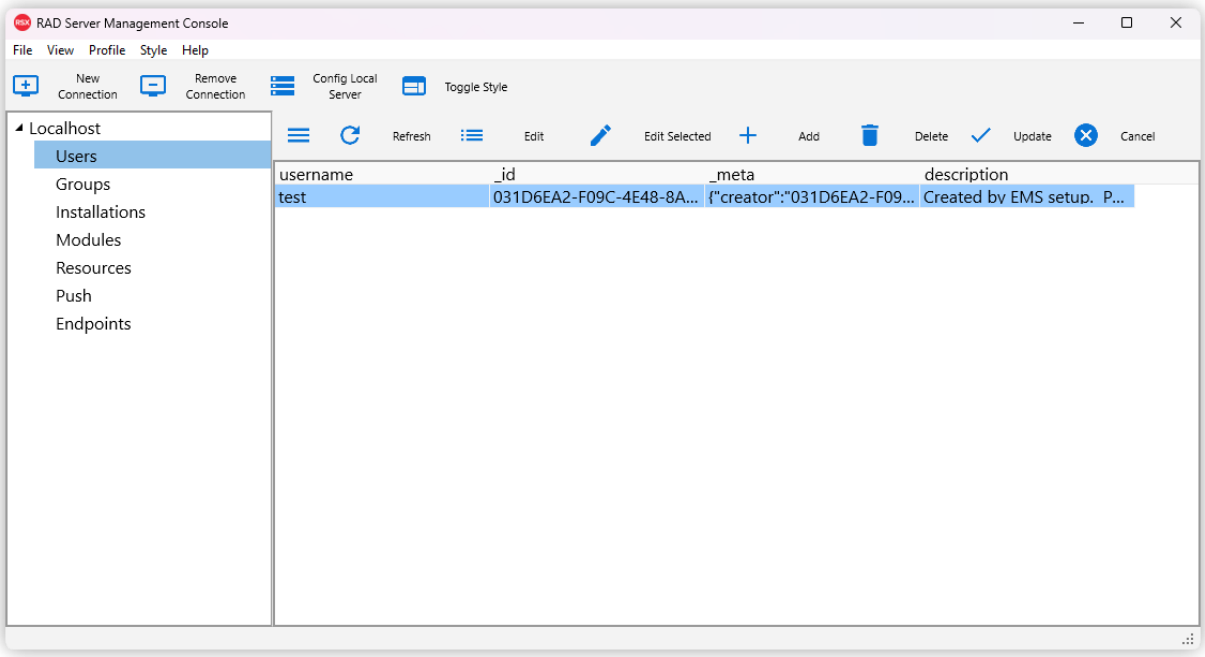
To create a new profile/connection simply press the button “New Connection” or access the menu “要建立新的配置文件/连接,只需按下“New Connection”按钮或存取菜单“Profile/New Profile…”.配置非常简单，只需要连接到主机的基本细节.

警告

如果 RAD 服务器开发未执行或未在生产环境中,则无法建立与 RAD 服务器实例的联机（Windows 上的 EMSDevServer.exe 和 Linux 上的 EMSDevServerCommand）.

管理使用者和群组

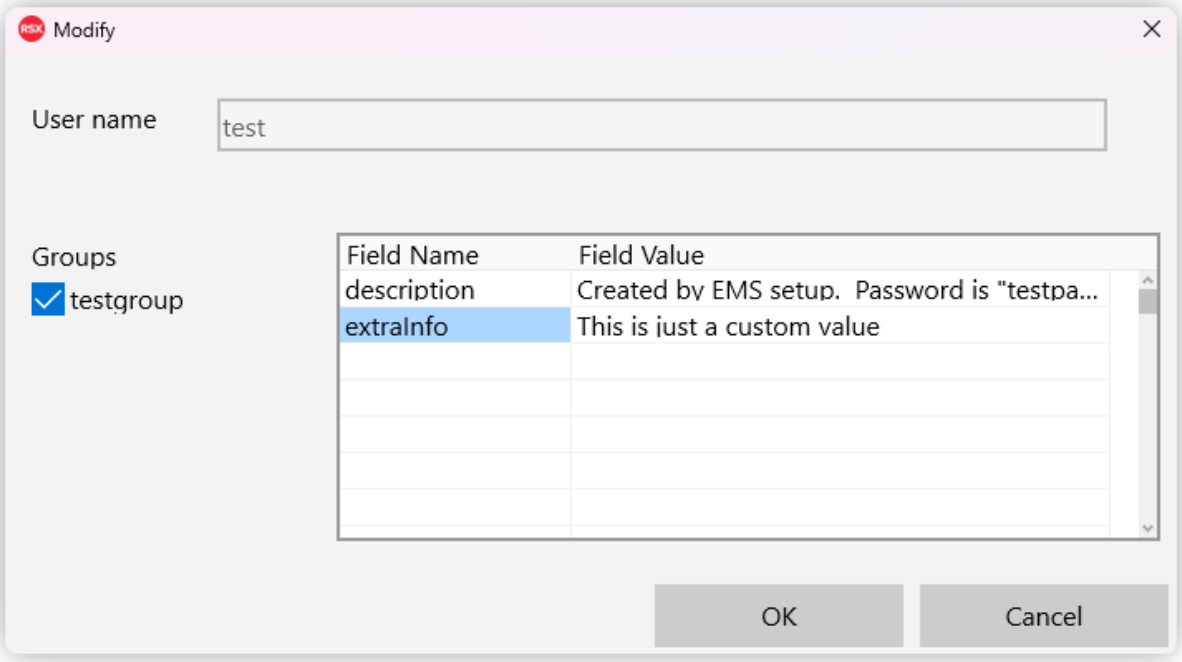
连接到所需实例后，左侧将显示所有可用资源的清单。在这种情况下，我们将讨论“Users”和“Groups”.



使用 RAD 服务器控制台建立的使用者列表

若要建立、编辑或删除使用者或群组，只需点击左侧的特定部分，您将看到一个带有多个选项的工具栏: Edit, Add, Delete 等等.

作为范例，让我们看一下"modify user"窗口（尽管"create user"功能几乎相同）:



“Modify User” 窗口

窍门

建立用户时，密码字段可见，但编辑使用者时无法存取. 若要修改用户的密码，只需在自定义字段部分新增"password"字段并将新密码作为值.

建立新群组的过程也非常简单。单击“groups”左侧部分，然后单击“Add”。在此窗口中，可以指定哪些使用者将成为群组的成员.

深入了解 RSConsole

如果您有兴趣了解 RAD Server 控制台在幕后如何运作，该 FMX 应用程序的所有源代码都包含在 RAD Studio 中。你可以在这条路径上找到它:

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

尽管它是一个相当复杂的应用程序，但它是查看集成 RAD 服务器 API 提供远程访问的所有可能性以及每个操作发送的请求类型的完美场所.

12

使用 OpenAPI 记录和测试您的端点 (Swagger)

什么是 OpenAPI/Swagger 以及为什么要使用它?

P 以前称为 Swagger 规范, OpenAPI 是一个很好的规范，可以以 JSON/YAML 格式动态建立 API 文文件，使用它，开发人员可以建立其 API 的 Swagger 实例. 凭借最大的 API 工具生态系统之一，支持 Swagger 的 API 提供交互式文件、客户端 SDK 生成和可发现性.

RAD Studio 让开发人员直接使用属性（包括摘要、参数和详细信息/响应）记录利用这些规范的 RAD 服务器.


备注

如果您使用 C++ Builder，您可能知道 C++ 本身不像 Delphi 那样支持属性，但如前面的章节所示，有一个解决方法可以在 C++ Builder 中提供此功能。此方法也用于以该语言记录您的 API.

将 Swagger UI 嵌入 RAD 服务器

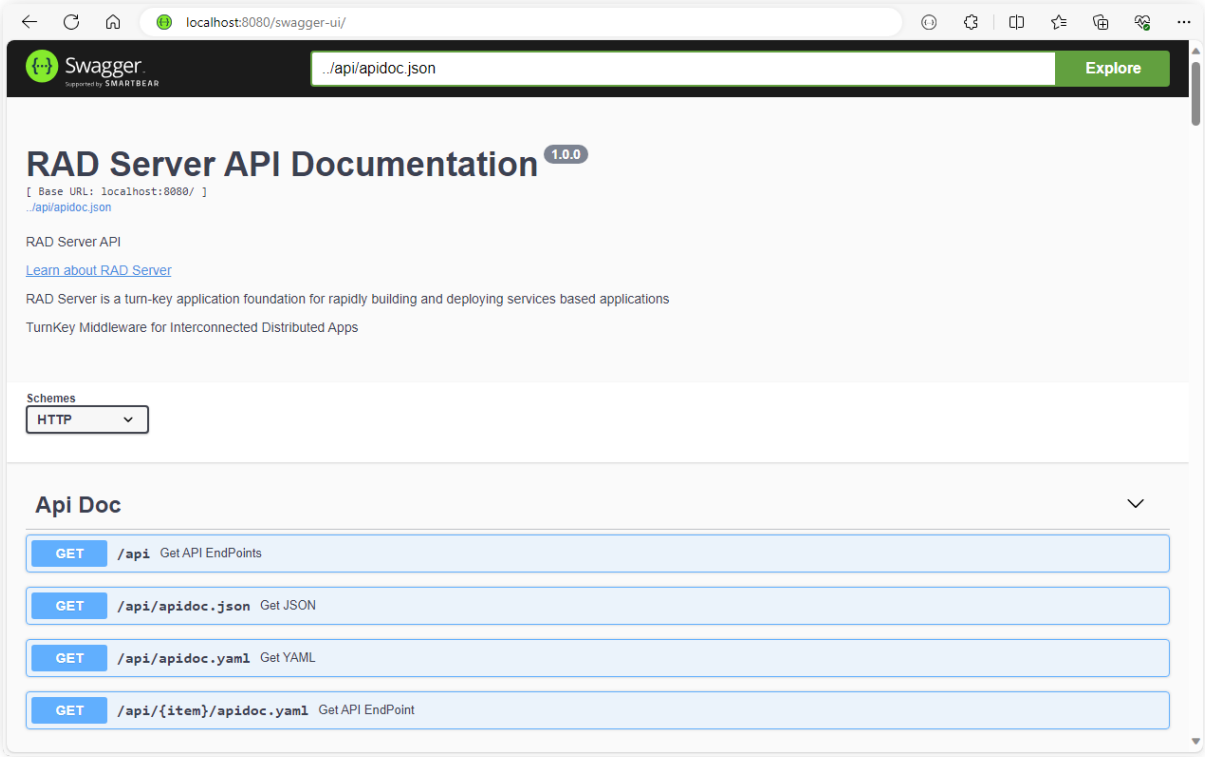
RAD Server 可以提供静态档案（例如：托管网站的前端），但此功能也可用于直接在 RAD Server 中托管 Swagger UI.

为此，我们只需访问 EMSServer.ini 档案并搜寻键值 [Server.PublicPaths]。在此之下，您将找到引用 Swagger UI 路径位置的特定行。取消批注该行并指示正确的路径.

正如前面章节中提到的，该文件可以在您的开发机器的路径中找到：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"}
```

定义后，启动 RAD 服务器并存取 URL/swagger-ui/。范例: http://localhost:8080/swagger-ui/



使用 apidoc.json 定义的 Swagger 首页

如我们在屏幕截图中看到的，swagger UI 会自动引用 JSON 规格。我们可以将其更改为 /api/apidoc.yaml，并且将显示相同的规范.

此外，RAD 服务器中嵌入的所有可用端点均已完整记录在产生的规格文件中.

窍门

Swagger UI 档案随 RAD Studio 提供在路径中："C:\Program Files (x86)\Embarcadero\Studio\XX.0\ObjRepos\en\EMS\swagger-ui".它们也可以从官方储存库下载.

建立自定义文档

范例

RAD Studio 提供了一个很好的范例，可以详细了解所有可能性以及每个可用属性.

此范例适用于 Delphi 和 C++，可以在路径中找到:

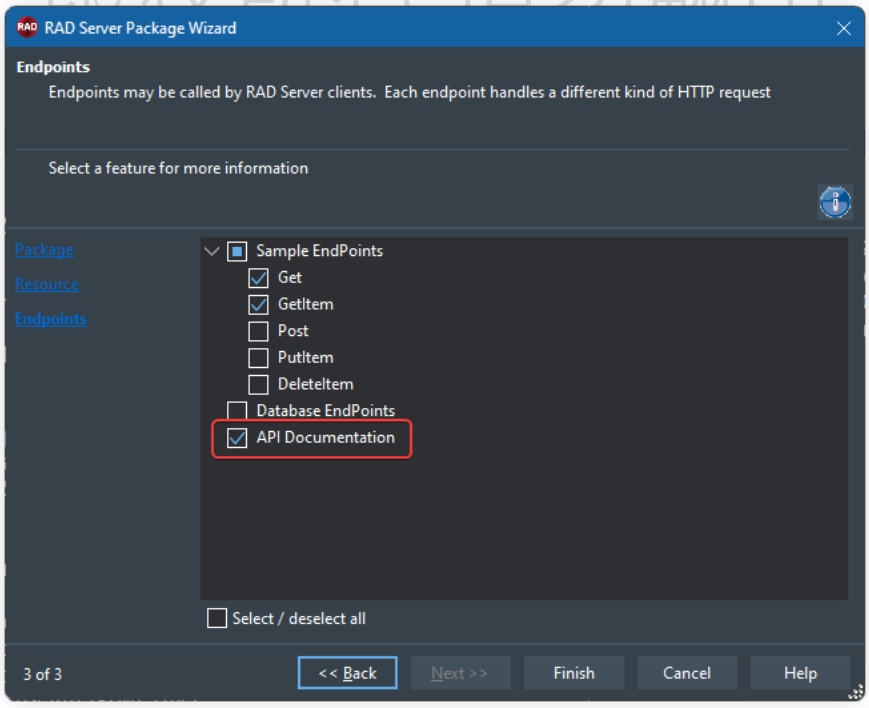
Delphi:
C:\Users\Public\Documents\Embarcadero\Studio\XX.0\Samples\Object Pascal\Database\EMS\APIDocAttributes

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

备注

本章的范例项目是 RAD Studio 提供的范例项目的更新版本. 由于正在使用新的多字符串文字功能，因此该项目将仅与 12 Athens 或更高版本兼容.

另一种选择是使用精灵建立新的 RAD 服务器并选择 API 文件复选框。在这种情况下，基本属性将自动填充.



自动建立 API 文件范例的精灵选项

EndPointRequestSummary

方法描述:

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

- **Tags**: 定义一个标签.
- **Summary**: 方法标题.
- **Description**: 方法说明.
- **Produces**: API 可以产生的 MIME 类型。这对于所有 API 都是全局的，但可以在特定 API 呼叫上被覆写。该值必须如 Mime 类型中所述.
- **Consume**: API 可以使用的 MIME 类型。这对于所有 API 都是全局的，但可以在特定 API 呼叫上被覆写。该值必须如 Mime 类型中所述.

EndPoint 的 GET 方法的描述声明范例:

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

请求中使用的参数的描述.

唯一参数由名称和位置的组合定义.

有五种可能的参数类型: Path, Query, Header, Body, 和 Form.


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

- **ParamIn**: 参数的位置: Path, Query, Header, Body 或 Form.
- **Name**: 参数的名称。参数名称区分大小写.
  - 当参数位置为‘Body’时，名称必须为'body'.
  - 当参数位置为'Path'时，名称必须对应于 Paths 对象中路径字段中关联的路径段.
  - 其余情况，名称与 ParamIn 对应.
- **Description**: 参数的简要说明。这可能包含使用范例。 GFM 语法可用于丰富文本表示.
- **Required**: 确定该参数是否为强制参数。如果参数在‘Path’中，则该属性为必填项，其值必须为 True，否则如果是可能包含该属性，则其默认值为 False.
- **ParamType**: 参数的型别。对于'Body'以外的其他值，该值必须是以下值之一：‘spArray’，‘spBoolean’，‘spInteger’，‘spNumber’，‘spNull’，‘spObject’，‘spString’，‘spFile’. 如果 ParamType 为'spFile'，则使用的 MIME 类型必须为“multipart/form-data”或“application/x-www-form-urlencoded”，且参数必须在“form-data”中。对于值“Body”，需要 JSONSchema 和 Reference.
- **ItemFormat**: ParamType 的扩展格式：‘None’，‘Int32’，‘Int64’，‘Float’，‘Double’，‘Byte’，‘Date’，‘DateTime’，‘Password’.
- **ItemType**: 如果 ParamType 为'array'，则为必要。它描述了数组数组中项目的类型.
- **Schema**: 发送到服务器的原语的架构定义。主体请求结构的定义。如果 ParamType 是’Array’或’Object’，则可以以 JSON 和/或 YAML 格式定义模式.
- **Reference**: 发送到服务器的原语的架构定义。主体请求结构的定义。如果类型是“'Ayyay'或'Object'，则可以定义模式。例如：'#/definitions/pet'

参数定义范例:

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

请求回应的描述.

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

- **Code**: 回应代码.
- **Description**: 响应代码说明.
- **PrimitiveType**: 回传的 原始数据类型 型态. Swagger 规范中的原始数据类型是基于 JSON-Schema Draft 4 支持的类型. JSON Schema 为 JSON 值定义了七种基本型别: ‘spArray’, ‘spBoolean’, ‘spInteger’, ‘spNumber’, ‘spNull’, ‘spObject’, ‘spString’. 请参考 JSON 模式原始类型. 附加的原始数据型态, ‘spFile’, 由参数对象和响应对象使用来将参数类型或响应设定为文件.
- **PrimitiveFormat**: 传回的原始数据的格式，例如: ‘None’, ‘Int32’, ‘Int64’, ‘Float’, ‘Double’, ‘Byte’, ‘Date’, ‘DateTime’, ‘Password’.
- **Schema**: 发送到服务器的原语的架构定义。主体请求结构的定义。如果 ParamType 是'Array'或'Object'，则可以以 JSON 和/或 YAML 格式定义模式.
- **Reference**: 发送到服务器的原语的架构定义。主体请求结构的定义。如果类型是'Array'或'Object'，则可以定义模式。例如: “#/definitions/pet”

响应定义范例:

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 格式定义对象定义。让我们来看一个例子:

Delphi

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

C++


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

但这些定义是如何指定的呢？由于篇幅较长，我们在这里只会看到一个简短的 Delphi 范例。对于完整的范例，您可以查看 RAD Studio 提供的范例项目.

Delphi

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



窍门

请记住，定义 YAML 档案对于注意缩排至关重要。这种格式没有像 JSON 那样使用大括号的对象结构. 在 YAML 中，缩排定义了结构的层次结构，因此在使用多行字符串的情况下，请确保在何处定义结束三引号.

定义 EMSDatasetResource 的属性

EMSDatasetResource 是一个很棒的组件，可以帮助非常快速地创建标准 CRUD 端点，但是由于该组件在幕后完成了大部分繁重的工作，因此最初我们无法存取每个端点来定义我们想要的属性.

默认情况下，仅定义通用 EndPointRequestSummary 属性，RAD 服务器将使用 {id} 作为主键建立通用 CRUD 定义，且端点将无法测试。为了解决这个问题，我们不仅可以定义主键名称，还可以定义每个操作及其自定义详细信息.

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());
```

在连结到本章的 GitHub 储存库中，您可以找到我们在第 3 章中使用的相同项目。. 您可以在 README.md 档案以及单元的注释中找到有关所遵循约定的更多详细信息.


窍门

要提供 YAML 和 JSON 规范，需要单独定义这两个规范。一个简单的解决方法是在 YAML 中建立所有定义（更容易阅读），然后使用在线转换器或任何 GPT 机器人为您产生等效的 JSON 定义.

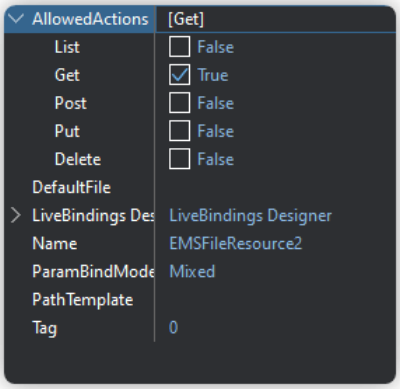
13

文件管理和储存

在本章中，我们将分析 RAD Server 提供的多种管理文件和存取基础架构的方式. 此外，我们还将了解如何指定每个端点可以在全局范围内产生或使用的文件类型，或者如何在每个端点上以精细的方式定义它.

TEMSFileResource

TEMSFileResource 组件的工作方式与 TEMSDatasetResource 非常相似，它可让我们的生活更加轻松。它抽象化了文件管理的大部分业务逻辑，所有需要的程序代码都由属性来完成.



自动建立 API 文件范例的精灵选项

AllowedActions: 在预设情况下，只有 GET 处于活动状态，但可以自定义组件以允许任何操作，以便可以取得档案清单并上传、更新或删除文件.

DefaultFile: 如果 GET 档案请求不包含指定档案的参数，则传回定义为 DefaultFile 的文件.

PathTemplate: 这是一个简单而强大的属性。最基本的范例可能是像 *c:\temp\{id}*这样的值. 这定义了储存该组件处理的档案的绝对路径, {id} 是一个通配符，我们将在属性中使用它作为包含文件名的参数. 在此属性中使用大括号允许选择建立更复杂的路径，例如为子文件夹甚至扩充档案定义通配符: *c:\uploads\{folder}\{file}.{extension}*.在这种情况下，需要在属性中定义 3 个参数：文件夹、档案和扩展名。我们稍后会看到一个例子.


备注

通常，PathTemplate 将根据环境(除错或发布)具有不同的值。建议使用编译指令更改此属性.

范例

对于 Delphi，在数据模块中的 EMSFileResource 定义之前新增三个 ResourceSuffix 定义项目.

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());
}
```

在此范例中，我们定义了 3 个操作：list、get 和 post，但 put 和 delete 也可以按照相同的模式实现. 端点 test/fileResource/ 现在允许列出"PathTemplate"属性中的所有字段字段，使用 {id} 通配符存取一个特定文件

或上传新文件. 重要的是要知道要使用此组件上传文件，正文必须包含文件本身的二进制输出. 不能使用多部分窗体上传多个档案（我们将在稍后看到如何完成此操作的范例）.


警告

要在 C++ 上使用 TEMSFileResource，您必须在"require"部分中新增函式库 emsserverresource.bpi。你可以在路径中找到这个文件: C:\Program Files (x86)\Embarcadero\Studio\XX.0\lib\{Platform}\release\emsserverresource.bpi

从程序代码管理文件

对于稍微复杂一点的应用、TEMSFileResource 可能不敷使用，那么您也可以透过程序代码来处理存取文件. 每个请求上的 ARequest 参数允许存取正文中的档案。这些必须以多部分表格形式发送。让我们来看看一些程序代码:

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);
```

在这个基本范例中，我们存取正文的第 0 部分（当然它可以包含多个部分/档案），并透过将正文流写入 TFile 变量本身，使用 TFile 类别将其储存在给定路径中. 真的就是这么简单。在储存库范例中，您可以找到一个稍微复杂的情况，其中我们循环遍历主体的所有可能部分以允许将多个档案上传到端点.

Content-Type HTTP 表头

RAD Server 对 Content-Type 和基于 Accept 的映像的 EndPoint 属性的新支持为资源映像提供了更好的支持，而不仅仅依赖 URL. 这意味着您可以将两种不同的方法对应到相同的 URL 和 HTTP 动词，同时根据请求传回不同类型的数据 t.

新增了两个新的 EndPoint 属性:

- EndpointProduce: 指定此端点可以产生的 MIME 类型/档案扩展名作为对 GET 方法的回应。端点选择将基于 Accept HTTP 请求标头.
- EndpointConsume: 指定此端点可为 PUT、POST、PATCH 方法使用的 MIME 类型/档案扩展名。端点选择将基于 Content-Type HTTP 请求标头.

本章将展示 RAD 服务器应用程序资源如何使用 EndpointProduce 属性进行基于 image/jpeg 和 application/xml MIME 类型的多个 REST Get 请求.

一个简单的范例

使用项目包精灵启动 RAD 服务器项目。选择数据模块档案类型并将资源名称设定为 AcceptTypes. 取消选择所有标准端点并点击“Finish”按钮.

在路径 c:\temp\page 中建立一个文件夹，并将名为 content.jpeg 的任何 jpeg 档案和名为 content.txt 的文本文件复制到其中.

Delphi

在资源类别的已发布部分中新增两个以 ResourceSuffix 和 EndpointProduce 属性修饰的方法声明.

```
[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;
```

在实作部分为 GetImage 和 GetText 端点新增以下程序代码.

```
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

为 GetImage 和 GetText 端点新增以下程序代码。在 Register 函数中，为 GetImage 和 GetText 端点新增 ResourceSuffix(s) 和 EndpointProduce 属性.

```
//---------------------------------------------------------------------------
#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
```

储存并建置 RAD 服务器项目。现在，您可以根据发送的 Content-Type 标头存取 content.jpeg 和 content.txt.