RAD Server Technical Guide
Part 1

localhost:8080/rad-server/part/1

```
{
  "title": "RAD Server Technical Guide - Part 1",
  "edition": 2.0,
  "authors": [
    {
      "name": "Antonio Zapater",
      "revised": "2023-09-15"
    },
    {
      "name": "David I",
      "revised": "2019-04-05"
    }
  ],
  "examplesURL": "github.com/embarcadero/radserver-docs",
  "copyright": "©2019-2023"
}
```

embarcadero®

Preface

RESTful architectures are a key driving force behind modern API first application design. This book focuses on the RAD Server framework included with RAD Studio (Delphi/C++Builder) for developing such platforms.

RAD Server is a full backend MEAP (Mobile Enterprise Application Platform) that enables Desktop, Mobile and Web frontend development in any language, and this book is designed as a definitive guide for developers.

The benefit of a MEAP is that you have a pre-built cloud or on-prem server with many core capabilities (such as push notifications, user tracking and analytics) that you can plug into rapidly to deliver remote database and functional access.

This guide to Embarcadero RAD Server, originally authored by David I (2019), is in its second edition, revised by Antonio Zapater (2023), which includes many additional features added based on market demand since RAD Servers Launch. The second edition is also supported by a comprehensive video series supporting each chapter, along with source code examples on GitHub. https://github.com/embarcadero/radserver-docs

You can access all the video series linked to this paper you have it available on Youtube. Also, we strongly recommend downloading all the examples from this GitHub Repository.

Table of Contents

01

What is RAD Server? Introduction

Today's computing landscape is no longer confined to a desktop, device, server or data center. Applications are being moved from the desktop to multiple devices, network edge connections, and to on-premises, public and hybrid cloud services. With RAD Server and RAD Studio you can build solutions that cover the wide spectrum of your company's (and customers) computing needs and business requirements.

This documentation will show you how to quickly design, build, debug and deploy service based multi-tier applications using RAD Server's REST based API hosting engine, components and technologies that are available in RAD Studio, Delphi and C++Builder Enterprise and Architect editions.

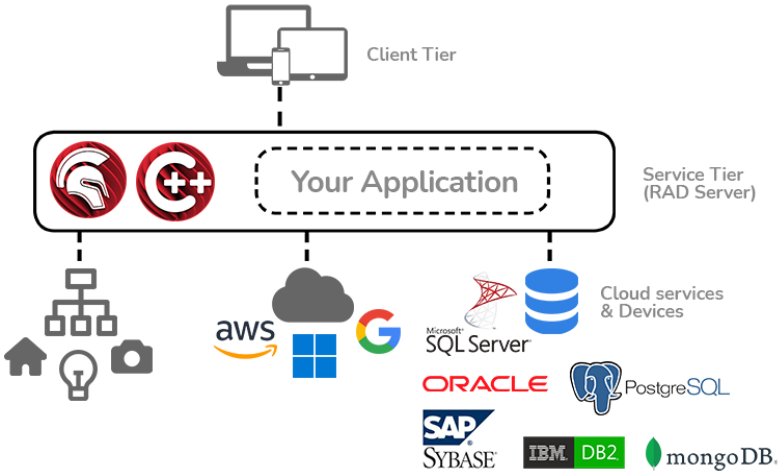
note

Throughout the RAD Server documentation and source code, you'll see references to EMS (Enterprise Mobility Services). EMS was the original name of what is now called the RAD Server product.

RAD Server Overview

Embarcadero's RAD Server provides a turn-key application foundation for rapidly building and deploying services-based applications using Delphi and C++Builder. RAD Server supports the REST (Representational State Transfer) protocol with JSON (or XML) parameter passing and return results. You can publish APIs, manage users and devices that are connected to the RAD Server, capture

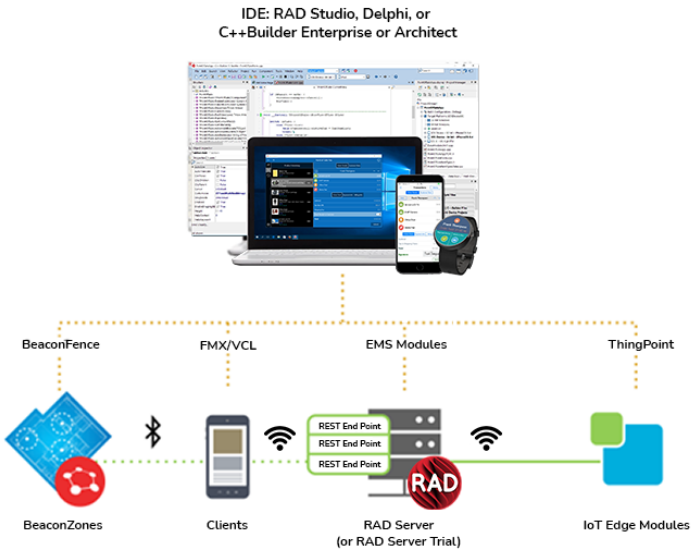
analytics about the use and users of applications, connect to local and enterprise databases using the FireDAC components and much more. RAD Server also supports user authentication, push notifications, geolocation and data storage.



Develop and Test REST endpoints and Location Tracking

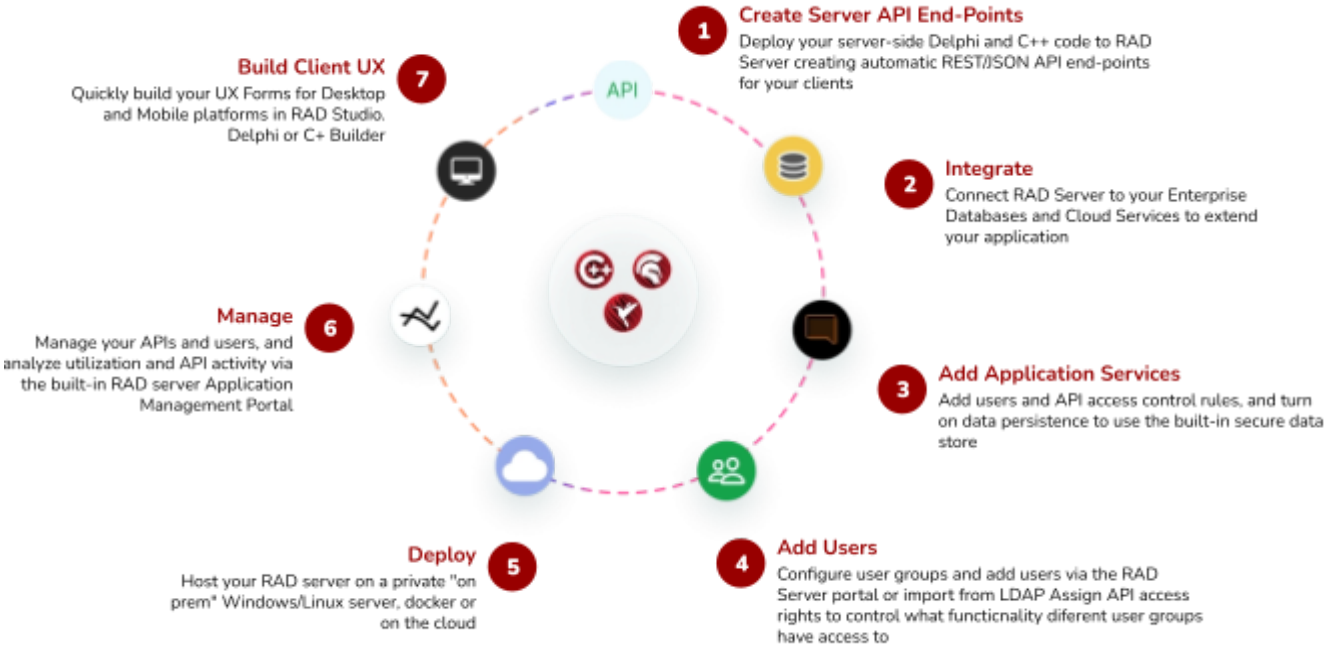
With RAD Server's wizards, components and tools you can quickly develop new middleware and back-end applications or migrate your existing Delphi and C++Builder client/server applications to a RAD Server based application to run on a server or in the cloud. You can publish your endpoints for REST calls from desktop, mobile, console, web and other types of applications. RAD Server comes with a full set of the tools, components, database connectivity and interfaces that you will rely upon in building your service applications.

RAD Server applications can be deployed on top of Microsoft Windows IIS and Apache web servers and you can deploy your Delphi based services to Linux Intel 64-bit servers. For C++Builder support for Linux stay tuned for updates to the Embarcadero RAD Studio blogs.



Building RAD Server based applications – Seven Key Aspects

To build RAD Server based applications, the diagram below guides developers through seven aspects and development phases.



Multi-Tier Development Made Easy

To start, create your server REST/JSON API-based endpoints (you can also use XML instead of JSON if required). Next you will extend your endpoints by integrating a wide range of databases, cloud services and other technologies.

You can add more application endpoints to users and create API access control rules. You can write code that leverages RAD Server's built-in secure data store to keep track of persistent data. You can create user groups and add users via console portal and import and authenticate users via LDAP-based API services.

After you have developed and debugged your applications you can host RAD Server applications on a private on-premises Windows and Linux servers. You can also migrate your applications to cloud systems like Amazon AWS, Microsoft Azure, Google and other cloud providers.

After your application is put into production, you can manage access to your APIs, control users access and analyze the utilization of your endpoint API activity with built-in application management interfaces. Finally, you can build desktop, mobile, Web, console and other application types supported by RAD Studio. You can also build modern Web client applications using the Sencha's Ext JS set of components and use other tools and programming languages to build client applications that support your RAD Server application's REST/JSON functionality.

Requirements for Building a RAD Server Applications

The following sections contain the product and technical requirements for building, testing and deploying RAD Server applications. Unless otherwise noted, “RAD Studio” and the IDE apply to the RAD Studio, Delphi and C++Builder products.

Using the RAD Studio IDE

A RAD Studio Enterprise or Architect Edition with a commercial license is required to build RAD Server applications. The trial edition of RAD Studio Enterprise can be used for 30 days for development and testing. The trial edition does not support deployment to a production server.

RAD Server Testing and Deployment Licenses

The free 30 day RAD Studio trial includes RAD Server 5-user development trial. RAD Server deployment licenses are included in Enterprise and Architect commercial editions of RAD Studio. RAD Studio Enterprise includes a single-site deployment license for RAD Server whereas RAD Studio Architect edition includes a multi-site deployment license for customers who are on active Update Subscription. Since RAD Studio Alexandria, Enterprise as well as Architect include the option of deploying RAD Server Lite in a multi-site environment.

RAD Server requires an InterBase encrypted database as part of deploying your applications in a Production Environment. You will need to use a valid RAD Server license to install this version of InterBase.

In case you want to deploy your application using InterBase as well, you will need 2 instances of InterBase running: one for your application and one for RAD Server.

Roundup of Core RAD Server Features

RAD Server provides developers with a wide spectrum of features for building REST-based service applications. RAD Server (formerly known as EMS) was first introduced in RAD Studio version XE7. Since that first release enhancements and new capabilities have been added to address the needs of developers and add support for new platforms, architectures and techniques.

Core Features

Here is a list of some of RAD Server’s core features that you’ll want to leverage in building your services-based applications.

- **REST End Point Publishing** – RAD Server implements a turnkey foundation for your application back end APIs and Services. RAD Server provides an easy to use API for publishing your business logic. Delphi or C++ code can be hosted as an API and auto-published as REST/JSON endpoints which are measured and managed by RAD Server. Endpoint publishing features include:

  - Access Control – You can set up group and user level access, with authentication, to all application APIs and control who has access to your application's API functionality. Create your own users and groups or import them automatically from your LDAP infrastructure.
  - API Analytics – All REST API end-point activity is recorded and measured for robust statistics tracking and analytics. You can analyze user, API, and services activity daily, monthly and yearly to gain insight into how your application is being utilized. You can also filter activity for all resources or by specific groups, users, device installations, and more. You can also export analytics to a CSV file for additional analysis with other tools.
  - Desktop, Mobile & Web Client Applications – All C++ and Delphi code hosted on RAD Server is published as REST/JSON end points consumable by any type of client application on multiple platforms for extreme flexibility and future-proofing.

- **Integration Middleware** – RAD Server provides multiple integrations out of the box with connectivity to external servers, applications, databases, smart devices, cloud services and other platforms. Integration capabilities include:
  - Enterprise Data – RAD Server delivers high performance built-in connectivity to all popular Enterprise RDBMS servers. Database connectivity uses the FireDAC set of components and libraries for easy connectivity with data from a variety of sources.
  - Cloud Services – With RAD Server you can easily integrate REST cloud services from a variety of cloud, social, and BaaS platforms such as Google, Amazon, Facebook, Kinvey, and more.

- **Application Services** – RAD Server includes a collection of ready to use built-in services to power your application. RAD Server includes core functions such as user directory services and user management, push notifications, user location tracking, and built-in data storage. Some of these application and device services include:
  - Push Notifications – using RAD Server you can send programmatic or on-demand notifications to your application users and their devices. RAD Server currently supports push notification systems including Apple Push Notification service (APNs) and Google FireBase Cloud Messaging (FCM). You can also write custom code to connect with other push notification systems.
  - Built-in Secure Datastore – With RAD Server's support for securing an InterBase server's encrypted datastore you can use built-in APIS TO store and retrieve JSON data without requiring a separate database server.
  - User/Group Management –Using RAD Server APIs you can create and manage your users, user groups, and control access via the RAD Server console (RSConsole.exe). Integrate your ActiveDirectory (LDAP) or develop your own custom authentication middleware.
  - User Location/Proximity – Your RAD Server applications can leverage RAD Studio's support for GPS, beacon and beacon fence technology. RAD Server applications can track user

movement, both indoors and outdoors, and respond to proximity events when users enter and exit custom beacon zones or approach designated beacon points.

    - Static Files Provider – Map URLs to folders and return the content of files like HTML, JS, CSS, images and more. This is extremely handy in small deployments (IE: using RAD Server Lite) or in development environments.

- **API Documentation** – Create easily documentation of your API using attributes and the built-in Swagger OpenAPI integration. Embed Swagger UI into RAD Server itself or configure it in remote instances through the auto-generated YAML and JSON files.
- **Easy to Deploy** – RAD Server is easy to develop, deploy and operate making it ideally suited for ISVs and OEMs building re-deployable solutions. Deploy it on Windows, Linux or Docker.

See Also

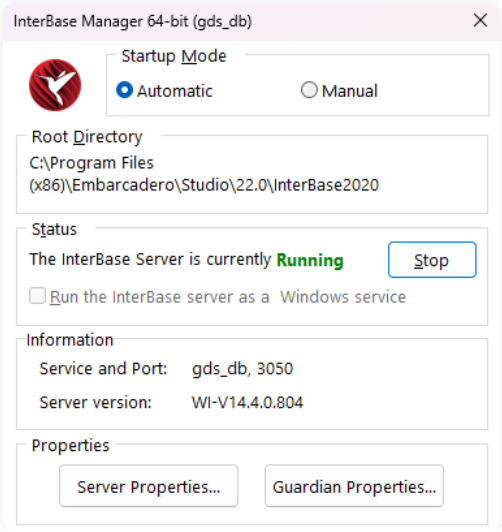
For the latest updated information about installation of RAD Studio and deployment of RAD Server based applications please refer to the following Embarcadero online links.

- RAD Server Product Overview
- RAD Studio Installation Notes
- RAD Studio and RAD Server Supported target platforms
- RAD Server Database Requirements for a Production Environment
- RAD Studio's Platform Status Page
- InterBase
- FireDAC
- FireDAC Supported Databases
- RAD Studio Enterprise Mobility Services
- RAD Studio Product Feature Matrix (PDF - Check RAD Server section)
- Swagger Open API
- EMS Push Notifications
- Apple Push Notification service (APNs)
- Firebase Cloud Messaging (FCM)

02

Using the RAD Wizard to Create a “Hello World”

It’s time to start programming. In this chapter, you’ll learn how to build your first RAD Server based service applications using Delphi and C++Builder. Before you begin, you’ll want to make sure that your InterBase database server is running. RAD Server uses an InterBase database for the storage of user information, user groups, analytics, registered devices, version information, registered Edge Modules, push notification messages and more.

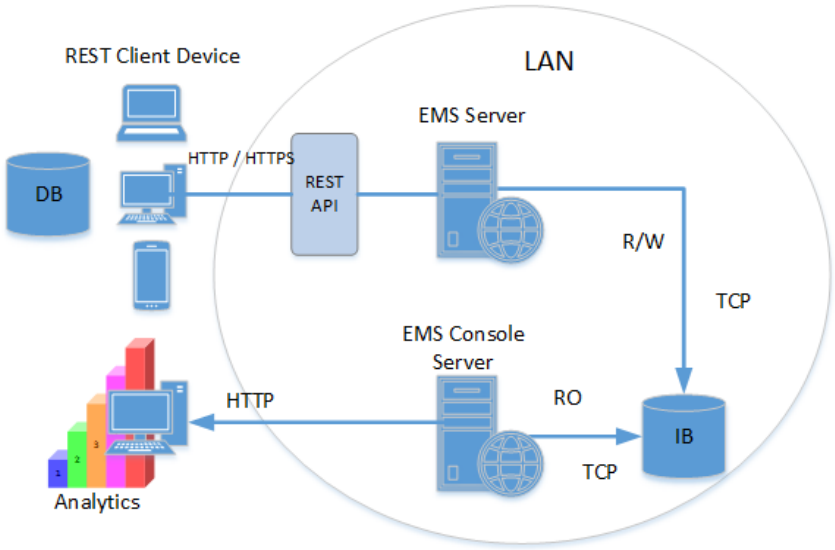


InterBase 2020 Server Manager

Building REST Based Services

RAD Server includes Enterprise Mobility Services (EMS) and offers a Mobile Enterprise Application Platform (MEAP) that you can host in a cloud or on-premises. Developers can use RAD Server to expose custom REST APIs and access enterprise database data using the FireDAC data access library and components.

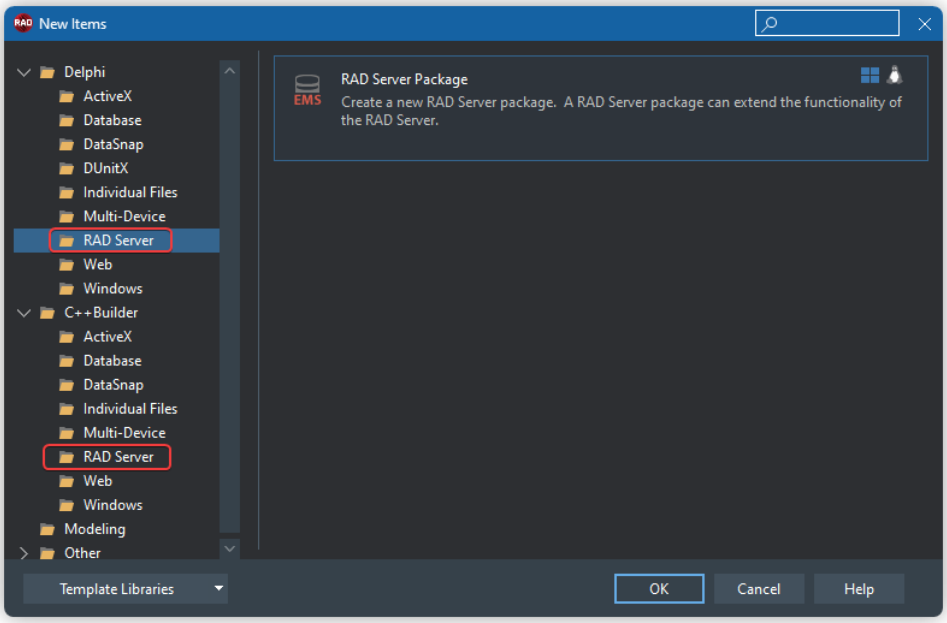
RAD Server provides developers with a comprehensive solution that includes remote database access, user tracking, device application management, use analytics and more. Compared to other solutions, RAD Server includes a pre-built application server that supports integration of custom packages. These custom packages can expose data sets, business logic and other REST-based resources. Components are also available for mobile, web and desktop application code to access RAD Server resources.



RAD Server REST API Architecture

Using the RAD Server Project Wizard

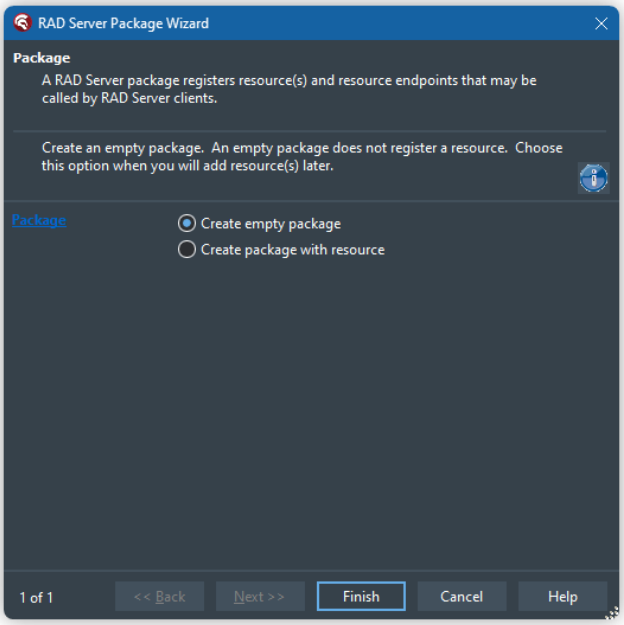
The fastest way to get started is to use the New Projects menu (File | New | Other...) and choose the RAD Server | RAD Server Package wizard for Delphi or C++Builder.



RAD Server Project Wizard choices for Delphi and C++

Select the RAD Server Package project. A wizard will appear to help create the starting project. On the first page choose how the wizard will create the resources and endpoints that will appear in the RAD Server application. The RAD Server Package Wizard provides two choices to proceed.

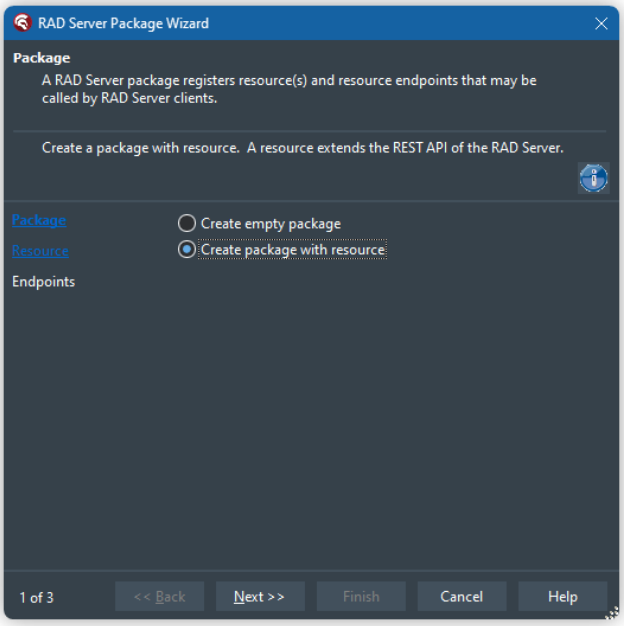
Choice 1: Create an empty package that does not register a resource. Choose this option if you are going to add your resources later on. Using this choice, a package project will be created with a starting main library.



Create an empty RAD Server package

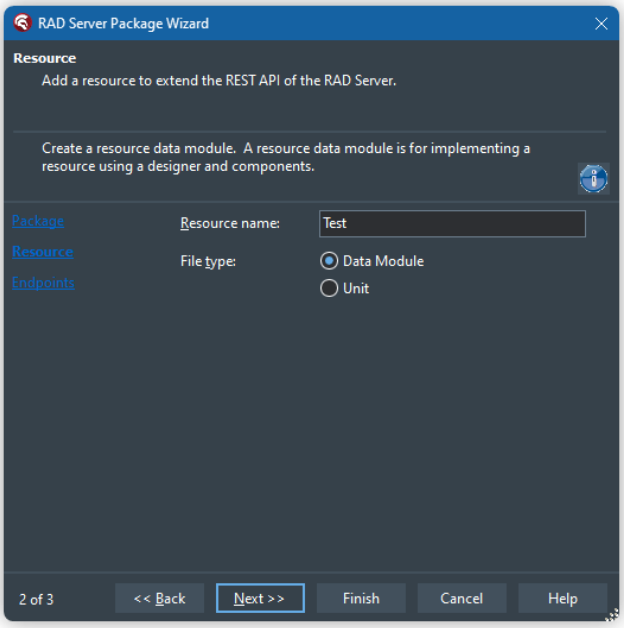
Clicking the Finish button will create the starting project for additional development work to create the finished RAD Server application.

Choice 2: Create a package with a resource that extends the REST API for the RAD Server. Click the Next Button and two additional wizard steps will appear to help create the package project, resource and endpoints. To build the first RAD Server project make this choice.



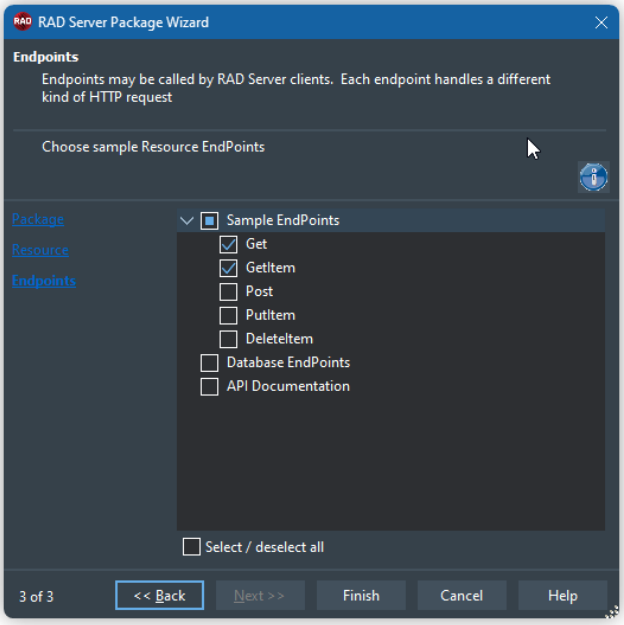
Create a resource based RAD Server package

On the wizard’s second page set the Resource name to “Test”. The File type radio buttons presents two options: 1) create a unit for implementing the resource in code, and 2) create a data module for implementing the resource using the IDE’s designer, components and code editor. For this first RAD Server application chose to use a Data Module.



RAD Server Package Wizard page 2 - set the resource name and file type

Click the Next button to create a starting set of endpoints.



RAD Server Package Wizard page 3 - choose starting EndPoints

On the wizard third page leave the suggested endpoints as you can see it in the picture above: Get (REST GET) and GetItem (REST GET with a segment at the end of the URL that identifies the item to get) and uncheck “API Documentation”. To create your starting project click the Finish button.

note

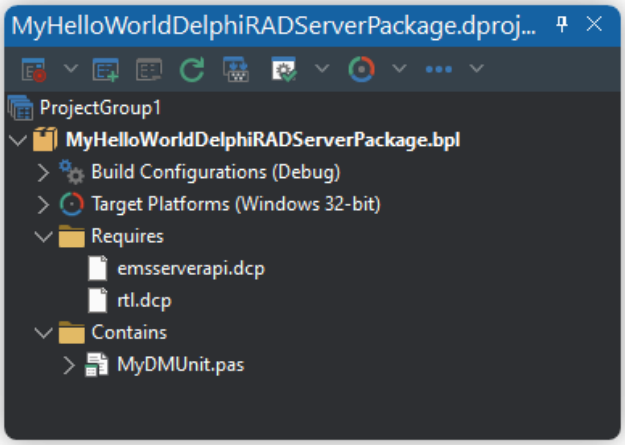
There are two extra options in the wizard: Database EndPoints (link your Database to endpoints using FireDAC) and API Documentation (Swagger OpenAPI). We will talk about these in the next chapters in more detail.

After the wizard completes, you are placed back in the IDE. The first thing to do is to save the project. For the C++ and Delphi data module, use the name “MyDMUnit”. For the C++ project and package use the name “MyHelloWorldCppRADServerPackage”. For the Delphi project and package use the name “MyHelloWorldDelphiRADServerPackage”.

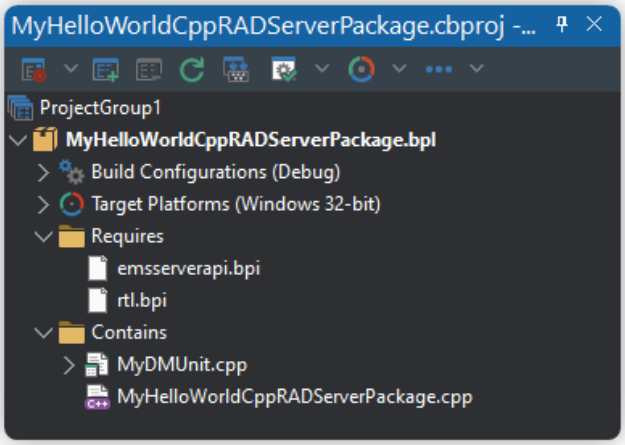
The Wizard RAD Server Project and Source Code

The project created is very low-code and it has just a couple of methods linked to each endpoint. RAD Studio populates automatically those with some default code that we’ll tweak a bit to make it more Hello World-y.

Here’s how the projects on Delphi and C++ should look like. The DataModule created will be empty with no components in it. After the screenshots you can find the modifications done in the sample code auto-generated.



Generated Delphi project



Generated C++ Project

note

All the source code and samples used in this document are hosted on GitHub and split in chapters. We strongly recommend you to download the whole repository to follow the documentation in a better way.

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;
```

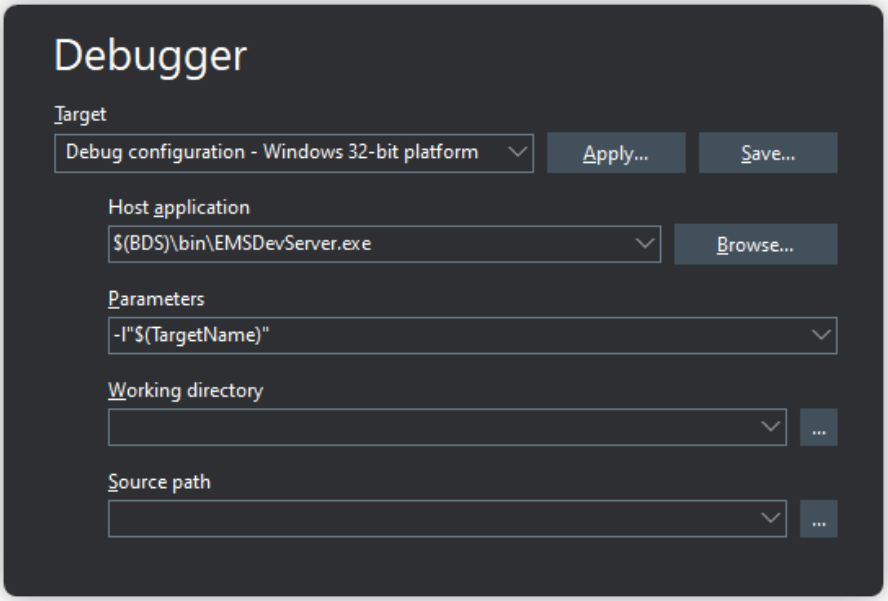
MyDMUnit.cpp:

```
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);
}
```

Configuring RAD Server for your first Application

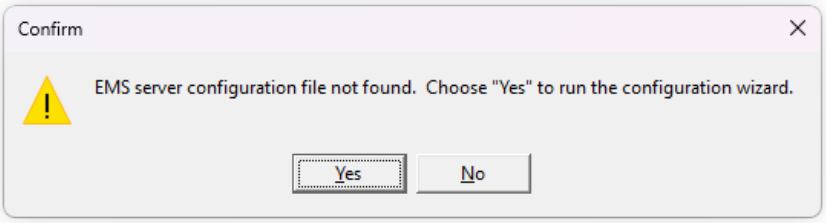
Now that you’ve used the wizard to build your first RAD Server application, you can use the IDE to compile and test the application. The IDE uses the EMSDevServer as the host executable (EMDDevServer.exe) to start execution with the parameter being the package file to load. There are two versions of the EMSDevServer for Win32 and Win64 development ($(BDS)\bin\EMSDevServer.exe and $(BDS)\bin64\EMSDevServer.exe). This is all configured automatically by the IDE.



Run | Parameters... dialog showing EMSDevServer.exe as the Host application

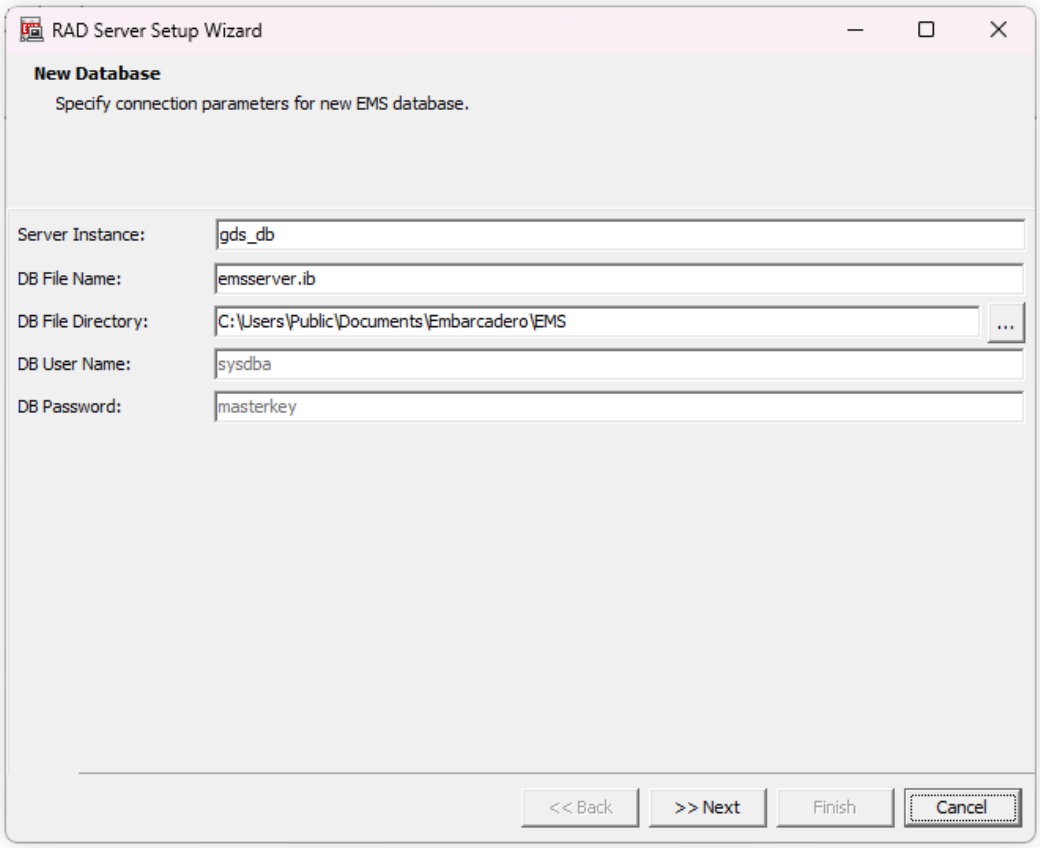
RAD Studio also includes EMSDevConsole.exe which will start the EMSDevConsole server and open the EMS Console Server Window. The EMS Console provides a web application which displays analytics, provides for user/group management, and more for your RAD Server application. This console is addressed in more detail in a next chapter.

Choose the Run | Run menu item or hit F9 to compile, link and run the starting application. The RAD Server Development Server will start, by default, using TCP port 8080. If this is the first time run of a RAD Server Application a dialog box will appear that says the RAD Server configuration file, emsserver.ini, was not found. This happens when there is no RAD Server registry key or if the configuration file does not exist.



Trying to start the RAD Server Development Server without a configuration file

Click the Yes button to run the RAD Server configuration wizard.



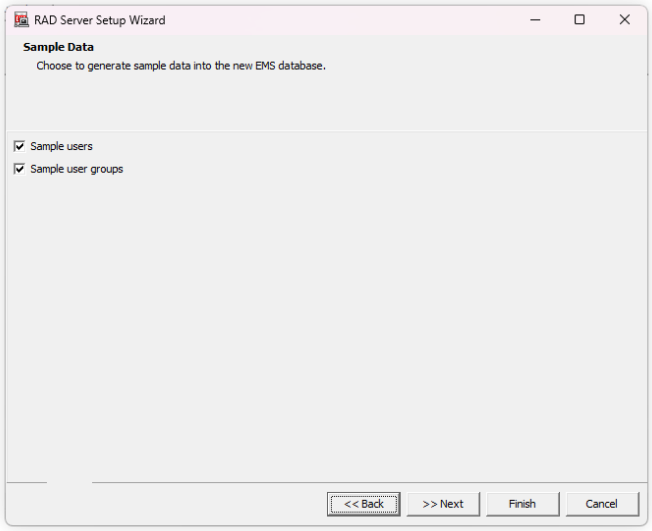
Setup wizard page 1 - specify the new EMS database connection parameters

In the first wizard step, enter the InterBase server instance (by default RAD Studio’s development version of InterBase Server uses gds_db). This wizard page also contains the name for the RAD Server database (emsserver.ib), and directory that will contain the database and configuration file.

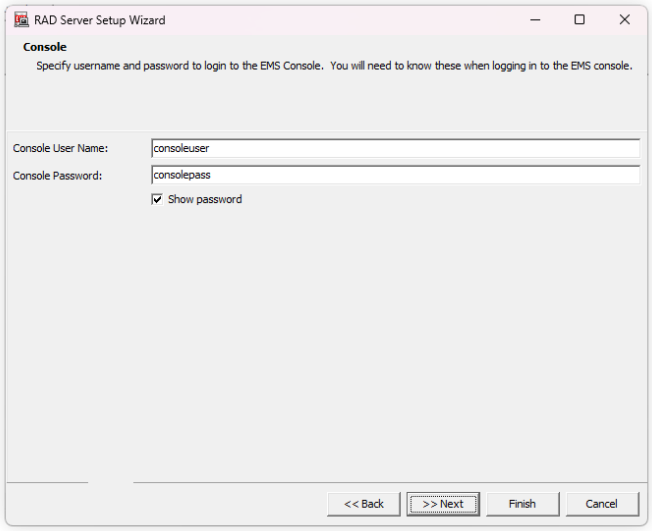
If you previously installed InterBase in your computer using a different server instance name, enter that name string.

Click the Next button to tell the wizard whether you to create sample RAD Server database data for users and user groups. For development and testing we’ll set both check boxes.

Click the Next button to set up the user name and password for logging into the RAD Server console.

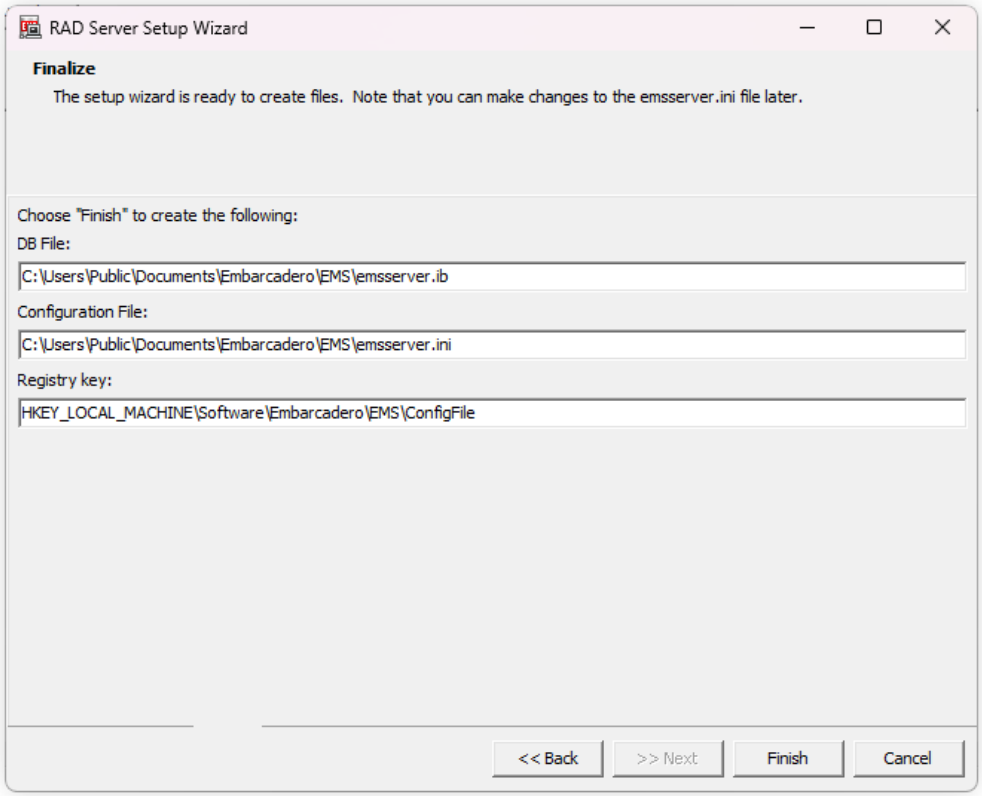


Setup wizard page 2 - Sample users and groups



Setup wizard page 3 - choose a Console user name and password

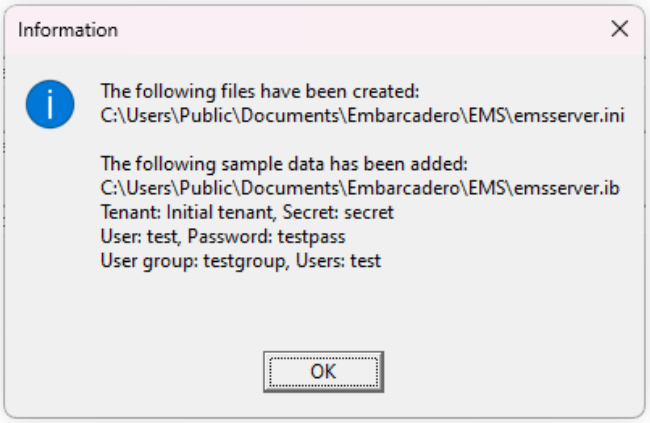
And lastly click the Next button to go to the final wizard step. The wizard is ready to create the RAD Server database file, configuration file, and set the Windows registry key for the currently logged in user.



Final RAD Server configuration wizard page

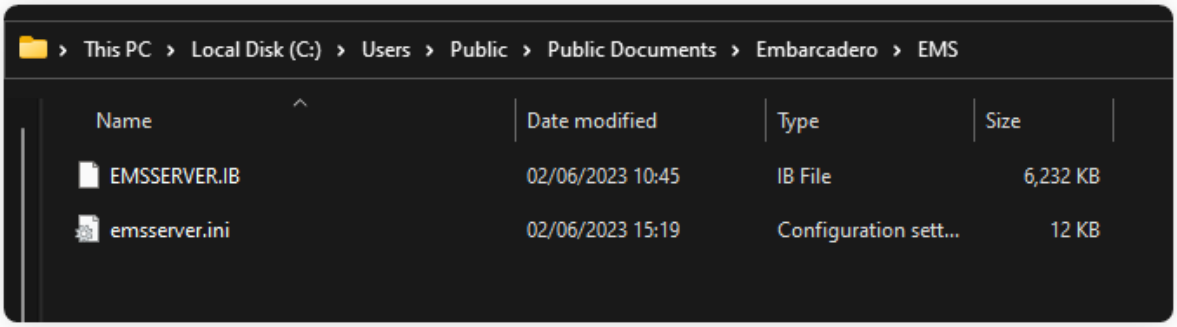
This page displays the database file path and name, configuration path and name and the Windows Registry key. You can always make changes to the RAD Server configuration file (emsserver.ini) at any time. Click the Finish button. A confirmation dialog will appear with a reminder that the configuration will use an instance of InterBase that does not have a RAD Server license. The development license limits your RAD Server application to a maximum of 5 users. When you are ready to deploy your RAD Server application you'll be able to use your deployment licenses for RAD Server and InterBase.

Click the Yes button. The wizard will display location of the emsserver.ini configuration file. It also lists the sample data that has been added to the database.



List of RAD Server files created by the configuration wizard

Click the OK button. Two files now appear in the C:\Users\Public\Documents\Embarcadero\EMS directory.



Files on disk created by the RAD Server configuration wizard

note

The emsserver.ini file is where all the default parameters of RAD Server are defined. Even though it will be addressed in next chapters feel free to check its content and the extense documentation specified into the file itself.

In general, the RAD Studio IDE is started without administrator privileges. So the Windows registry entry for HKEY_LOCAL_MACHINE get virtualised to HKEY_CURRENT_USER\Software\Classes\VirtualStore\MACHINE\SOFTWARE\WOW6432Node\Embarcadero\EMS on a Windows 64 bit operating system.

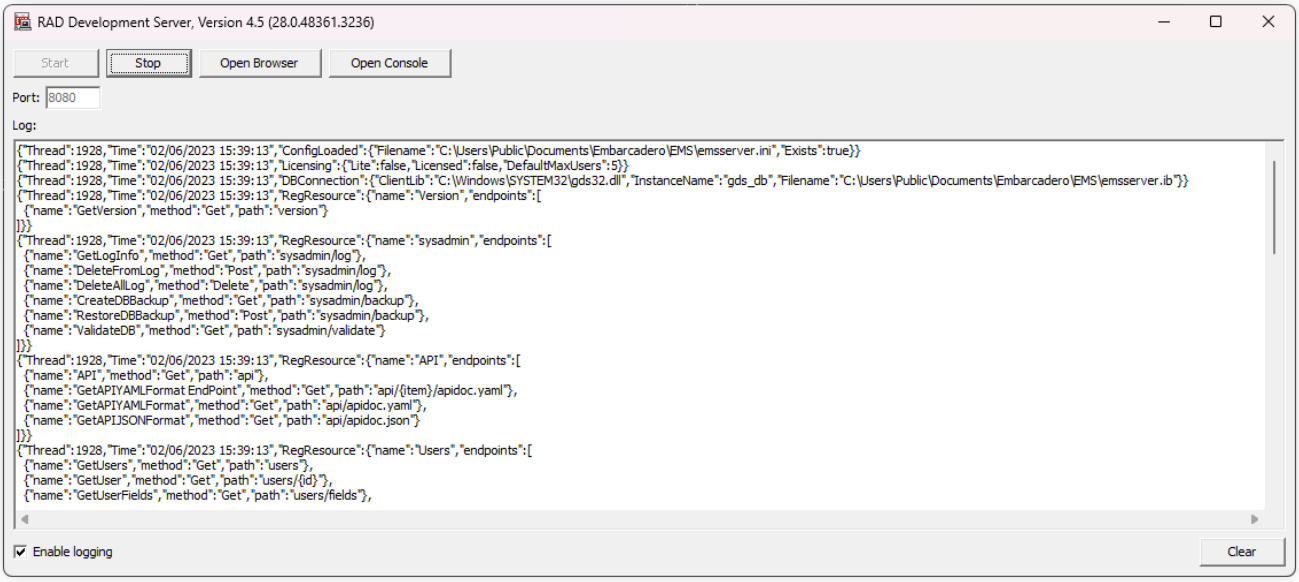
Testing your first RAD Server Application

When the RAD Server configuration server is created, the RAD Server Development Server will start executing.

warning

The default port where EMSDevServer runs is 8080. If your computer uses that port for another service you can change the default port used changing it on the “Port” field for whichever suits your needs. To make this change permanent modify in the emsserver.ini file the parameter `[Server.Connection.Dev]`

When you hit run in the IDE your RAD Server Development Server will start executing and the Log will show the operations that take place for your package application. The Development Server is exactly the same on Delphi and C++.

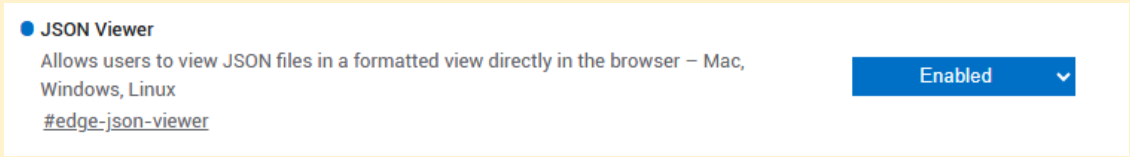


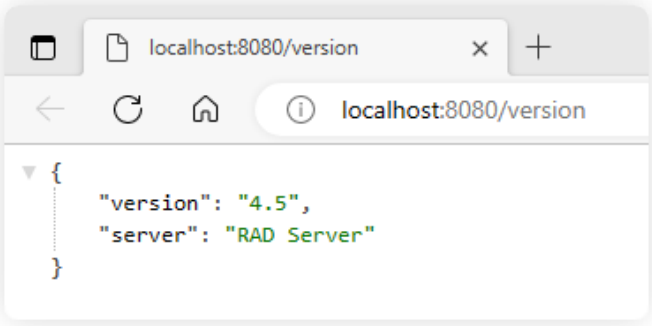
RAD Server Development Server starting the first application package

The RAD Server Development Server log will display the configuration, database connection, licensing information, the application package that was loaded, the resources that were registered, and endpoints that were created.

Clicking the Open Browser button will start your default browser and display the JSON result of calling the GetVersion built-in endpoint. You’ve now used your first RAD Server REST endpoint!

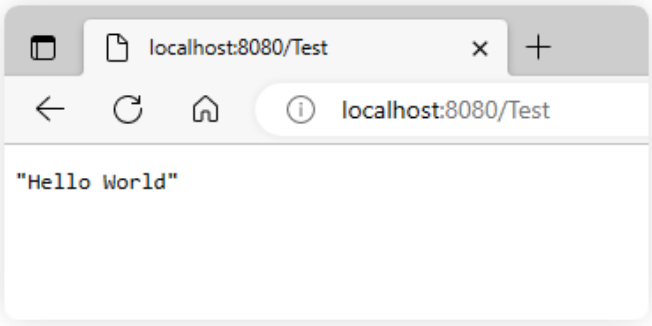
To get a more human readable JSON responses on your browser you can install the extension “JSON Parser”. It is available for all the major browsers.
*On **Microsoft Edge** there is a setting under “edge://flags” called “JSON viewer”. Enabling this gives you a readable JSON response without installing an extension*





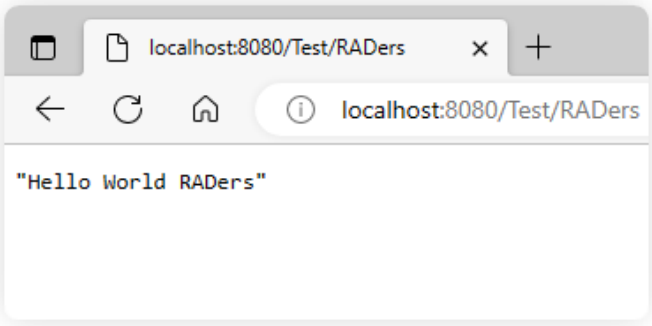
Browser showing the output from calling the version endpoint

In the browser, change the URL to localhost:8080/Test and hit enter. The browser will receive the JSON response from the Get endpoint.



Browser showing the JSON output from the Test resource's Get method

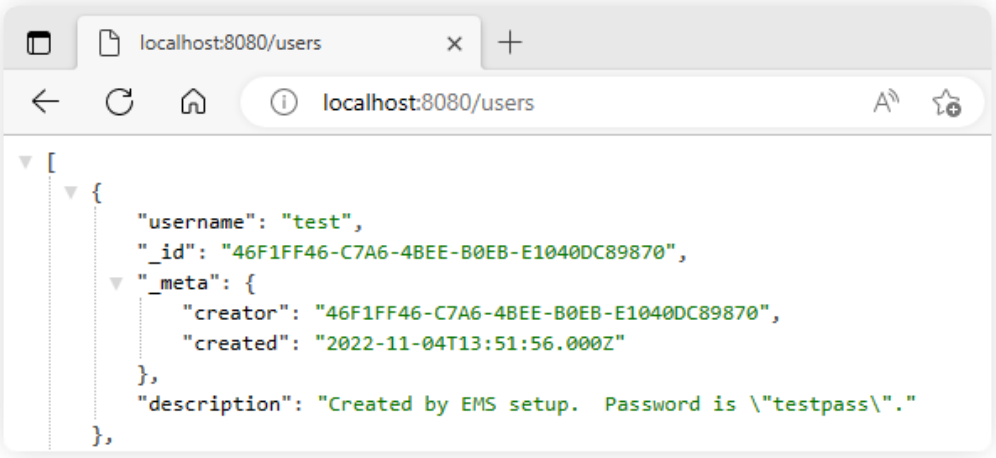
If you pass an additional item on the URL, the GetItem endpoint will be called, and the code behind will return a JSON string containing the resource name plus the item you typed.



Browser showing the JSON output from the Test resource's GetItem method

For simple examples it's okay to return a JSON string, but for larger, more complex data structures, you may not want to return a large JSON string. RAD Studio provides many other ways to generate JSON data including using JSON objects, JSON streams and the JSON writer.

Edit the URL to use the “users” resource which will call the default GetUsers endpoint to display JSON for the user generated by the RAD Server configuration wizard in the RAD Server datastore (there is only one to start).



JSON response in the Browser for a call to the GetUsers end point

You’ve now used four of the endpoints that were generated by the RAD Server project wizard.

See Also

- Code Samples on in this chapter
- RAD Server Engine (EMS Server)
- Setting Up Your RAD Studio (EMS) Server
- Configuring Your EMS Server or EMS Console Server on Windows
- RAD Server Administrative API

03

Creating your first CRUD Application

RAD Studio provides multiple ready to use components but one of the most useful ones when it comes to create CRUD APIs is EMSDataSetResource. This component allows you to link a FireDAC query to it and expose not only the data but also manipulate it. The component automatically creates all the required endpoints for CRUD and provides extra functionality like pagination, sorting and more.

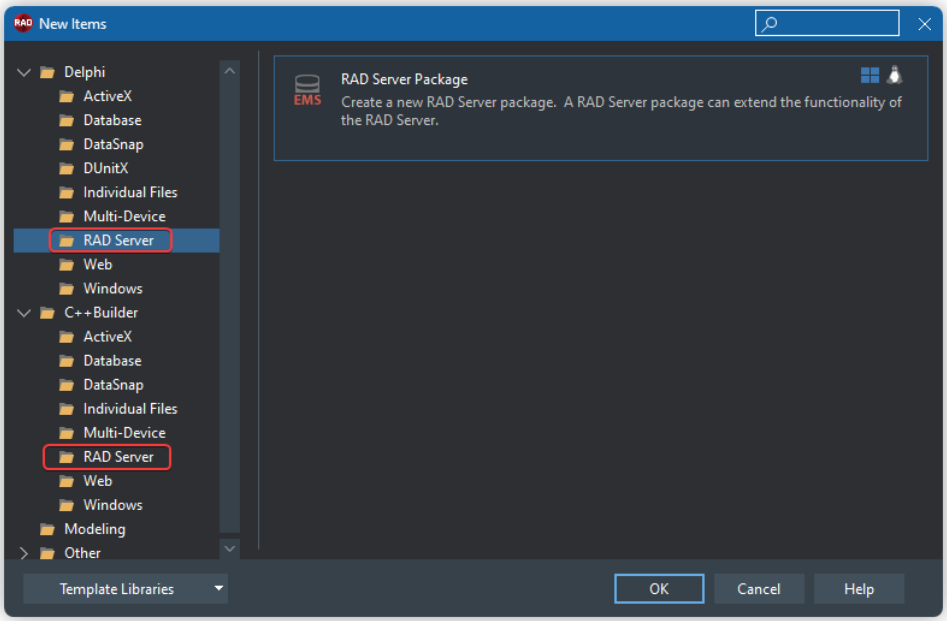
The EMSDataSetResource can be created in any of your current units or even easier, use the RAD Server Wizard to create all the required components automatically linked to a FDConnection.

note

On this demo we will use the employee InterBase database but feel free to use any other database compatible with FireDAC. The only requirement to use the RAD Server Wizard is that the database connection is preconfigured in the "Data Explorer" so it's recognised by RAD Studio.

Building REST Based Services with CRUD functionalities

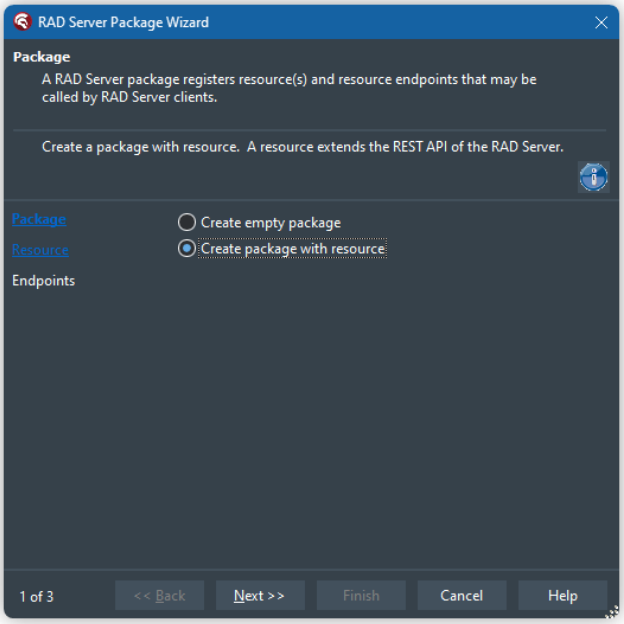
As we did in the previous chapter, the fastest way to get started is to use the New Projects menu (File | New | Other...) and choose the RAD Server | RAD Server Package wizard for Delphi or C++Builder.



RAD Server Project Wizard choices for Delphi and C++

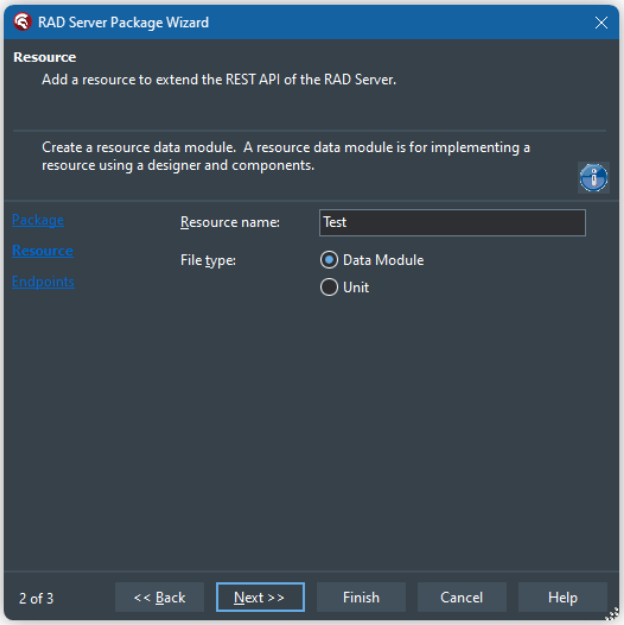
Select the RAD Server Package project. A wizard will appear to help create the starting project. On the first page choose how the wizard will create the resources and endpoints that will appear in the RAD Server application. The RAD Server Package Wizard provides two choices to proceed.

Now create a package with a resource that extends the REST API of the RAD Server. Click the Next Button and two additional wizard steps will appear to help create the package project, resource and endpoints. To build the first RAD Server project make this choice.



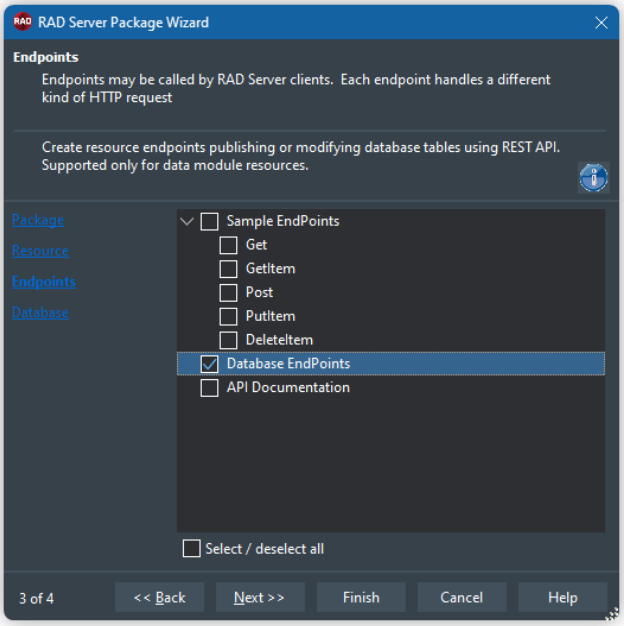
Create a resource based RAD Server package

On the wizard's second page set the Resource name to "Test". The File type radio buttons present two options: 1) create a unit for implementing the resource in code, and 2) create a data module for implementing the resource using the IDE's designer, components and code editor. For this first RAD Server application chose to use a Data Module.



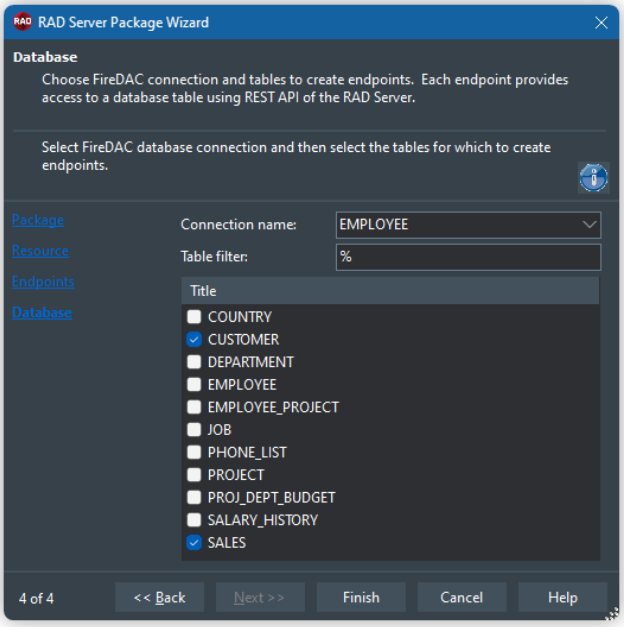
RAD Server Package Wizard page 2 - set the resource name and file type

Click the Next button to create a starting set of endpoints.



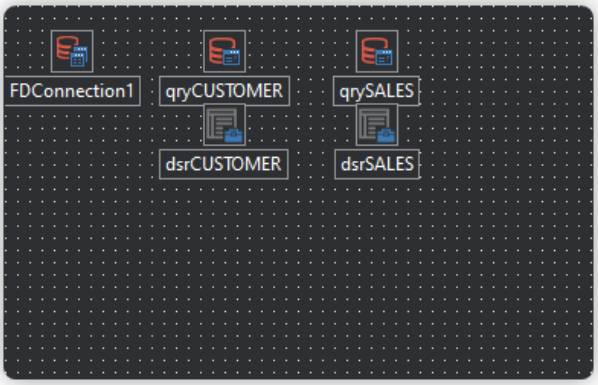
RAD Server Package Wizard page 3 - choose starting EndPoints

On the wizard third page we will choose different options compared with the previous chapter. In this case we are going to uncheck “Sample EndPoints” and check “Database Endpoints”. Now click “Next”.



RAD Server Package Wizard page 4 - choose database and tables

Once the project is generated we should see a FDConnection, 2 FDQueries and 2 EMSDataSetResource.



Data Module generated by the Wizard

Explaining the project generated

The beauty of this demo is that we could build it already and we will be able to access the endpoints auto generated for us, but first, let’s fix something: Access the code of the DataModule and change the attributes of the endpoints. This is not really relevant but it’s common good practice to keep your endpoints lowercase as well as pluralized.

Delphi:

```
  [ResourceName('test')]
  TTestResource1 = class(TDataModule)
    FDConnection1: TFDConnection;
    qryCUSTOMER: TFDQuery;
    [ResourceSuffix('customers')]
    dsrCUSTOMER: TEMSDataSetResource;
    qrySALES: TFDQuery;
    [ResourceSuffix('sales')]
    dsrSALES: TEMSDataSetResource;
```

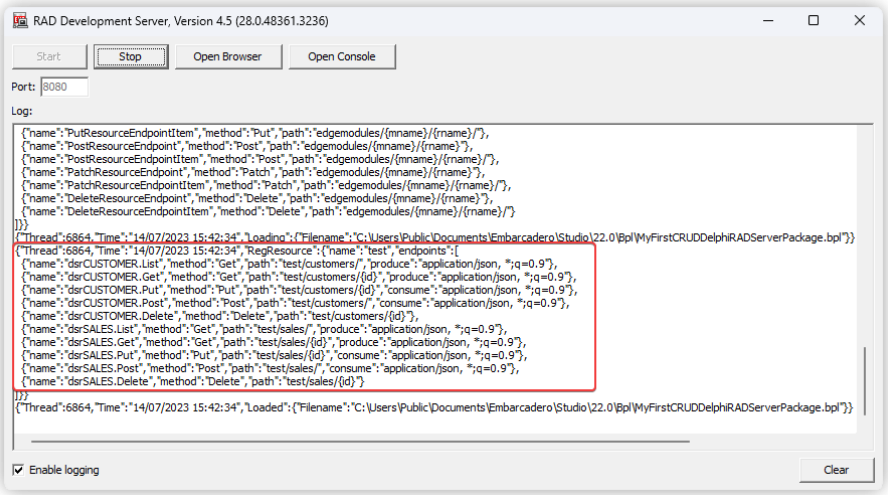
C++:

```
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());
}
```

As we can see, the ResourceSuffix attributes are linked to the EMSDatasetResources. That means that the query linked to that DatasetResource will be exposed under that endpoint.

The project couldn't be simpler. Not one line of logic coded by us so far and we have a fully functional CRUD system linked to 2 tables. Let's build the project and analyze it in further detail.

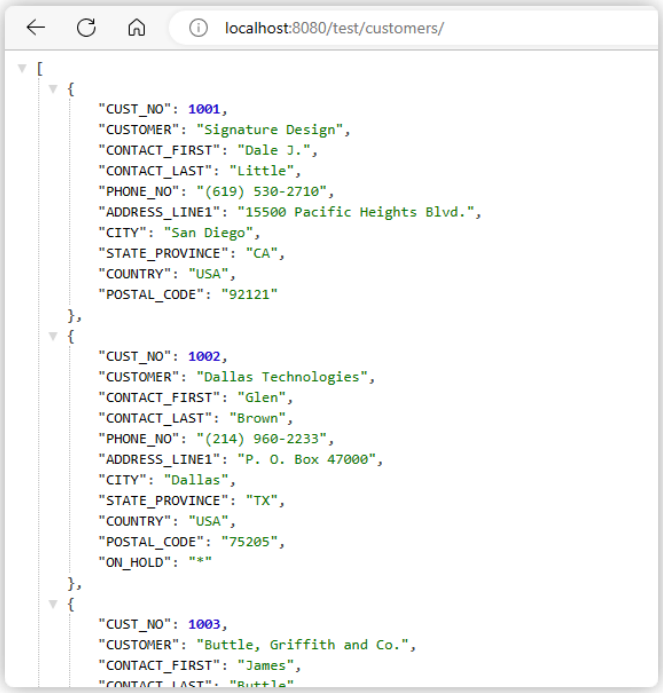
Building and testing the project



RAD Server Log showing all the endpoints created automatically

We can see on the RAD Server log that the endpoints “customers” and “sales” have been created but also with the parameters {id} available in the URI to get/put/delete individual records or post new ones.

If we open the browser and access the URL *http://localhost:8080/test/customers/* it returns an array with all the records in the table customers.



Array of customers returned by RAD Server

To access a specific customer using their ID (Cust_No) we just need to send the request *http://localhost:8080/test/customers/1004*. As you have probably already guessed, if you want to access the sales endpoint you simply need to call *http://localhost:8080/test/sales/* and so on.

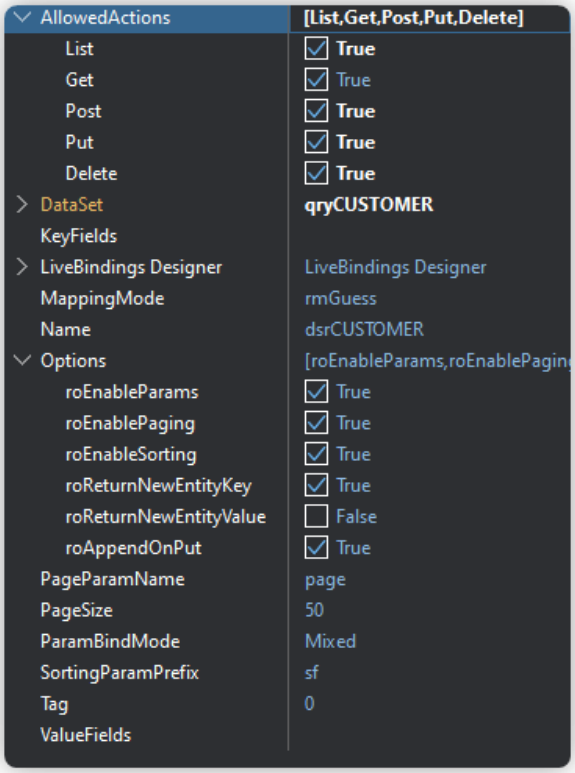
Accessing a specific customer using their ID

warning

When using TEMSDatasetResource it is crucial to keep the slash / at the end of the endpoint. Accessing the endpoint without the / RAD Server will drop a "not found" exception.

Additional features of TEMSDatasetResource

What we've seen so far is very impressive already. We can expose in just a few clicks as many datasets as we need and develop our API very rapidly, but TEMSDatasetResource offers even more features built-in. Let's analyze the key ones:



AllowedActions – built in control for allowing or preventing List, Get, Post, Put, and Delete on endpoints.

DataSet – connect to a dataset: Query, Table etc.

KeyFields – choose dataset fields that must be matched when doing a lookup.

PageParamName – name of the parameter to use pagination through the URL. IE: ?page=1

PageSize – when accessing LIST action, define the pagination size of the payload.

SortingParamPrefix – text string that will be pre-pended to a data setValueFields entry.

ValueFields – choose fields to use in a parameterized query and also to appear in the JSON response

Options – sub-property settings to enable/disable param use, row paging, data set field sorting, etc.

Now that we know a bit more about this component, let's try to use some of these features in our API. If we access *http://localhost:8080/customers/?sfCONTACT_LAST=A&page=1* we will get the first page of customers in ascending order by the field CONTACT_LAST. If we change the value =A for =D the response will be in descending order.

But how is that "order by" injected into the SQL? If we open the FDQuery we will see the next statement:

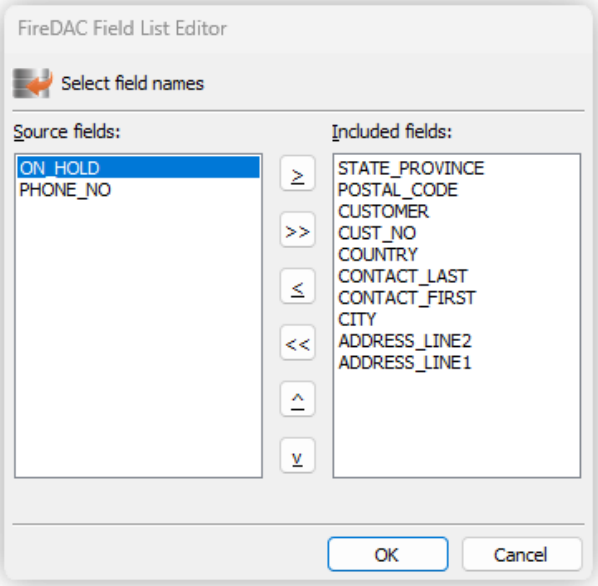
```
select * from customer
{IF &SORT} order by &SORT {FI}
```

As we can see, the SQL statement is very simple, but that macro is key for this to work. The EMSDatasetResource uses that macro for being able to mix pagination and ordering in the same query for us.

note

When pagination is being used and we reach the end of the Dataset, RAD Server will simply return an empty array to let us know that there is nothing else in that page.

Another very useful functionality is the option of fetching fields from our database to be used in our logic but we don't want to expose those fields in our API. Using the property ValueFields we can easily choose which fields we want to publish.



Selected fields to publish in our API endpoint

04

REST Debugger

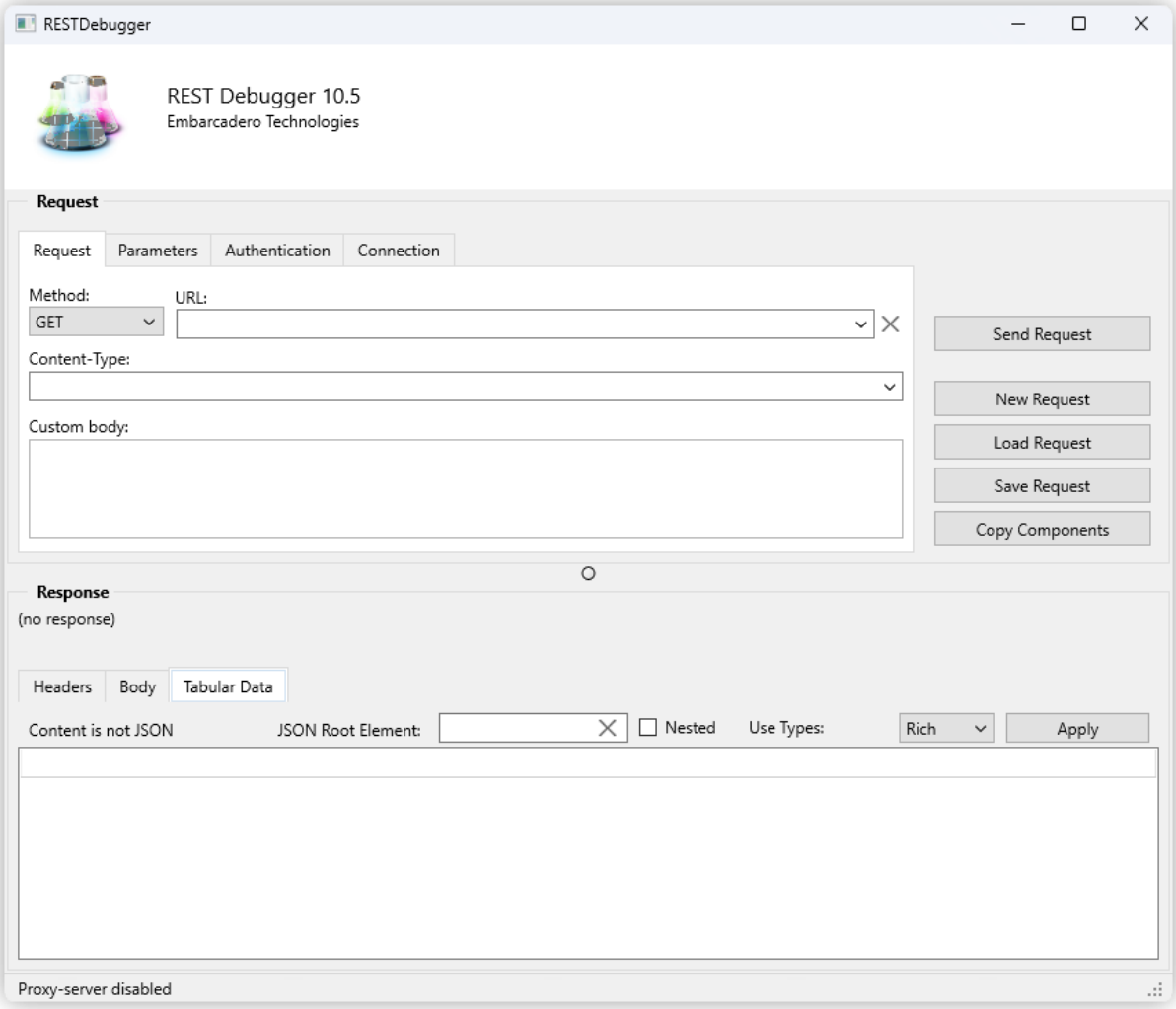
During these first chapters we have talked about the available actions in the REST API ecosystem, but so far, we have only used GET through a browser. If you are wondering how to use the actions POST, PUT and DELETE, in this chapter we will see a tool that comes with RAD Studio called REST Debugger and not only it will simplify your testing process, but also will help you to develop applications in a faster way.

REST Debugger is not a product only to be used with RAD Server. You can use it to access any other third-party REST API service and speed up your development process taking advantage of its integration with RAD Studio.

What is REST Debugger and where to find it

REST Debugger is Embarcadero's free solution for exploring, understanding, and integrating RESTful web services with Delphi and C++Builder apps. It empowers developers to explore, test, and ultimately understand how a RESTful web service works with features such as filterable JSON blobs, streamlined OAuth 1.0/2.0 authentication, and configurable request/resource parameters. Not only that, but also offers you the possibility to copy and paste REST components directly into your projects in just a few clicks.

If you want to give it a try, you can find it in RAD Studio under the menu **Tools/REST Debugger** or you can also download a standalone version for free on this link.



REST Debugger UI

Sending our first PUT Request with REST Debugger

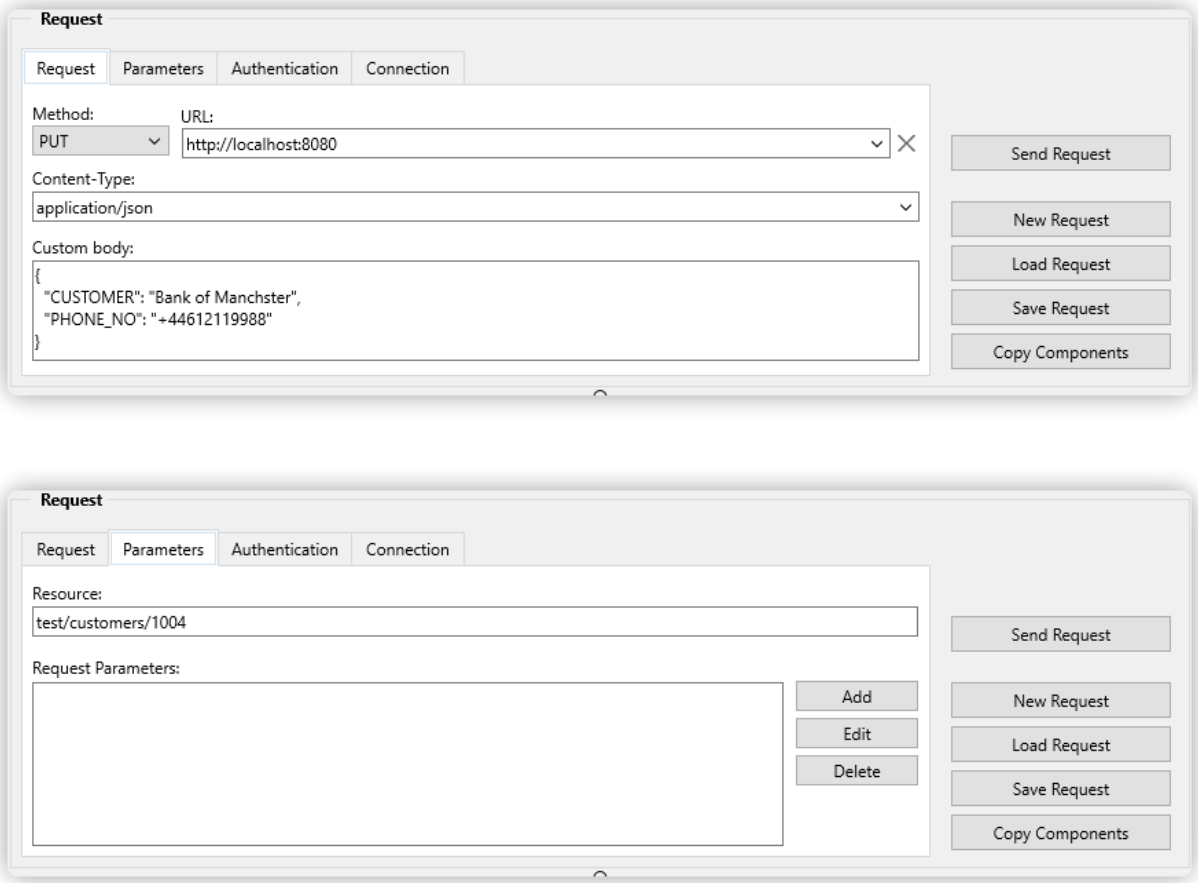
We can see on the dropdown on the left that the default value is GET, but we can now choose other ones we can't on a browser.

Using the same project we created on chapter 3 let's modify a customer.

warning

It's indispensable to have the RAD Server project from chapter 3 up and running. Otherwise we won't be able to call the API endpoint.

To modify a customer we just need to call the customers endpoint with the ID of the one we want to modify. Also, we need to specify in the body of the request the new values of the properties.

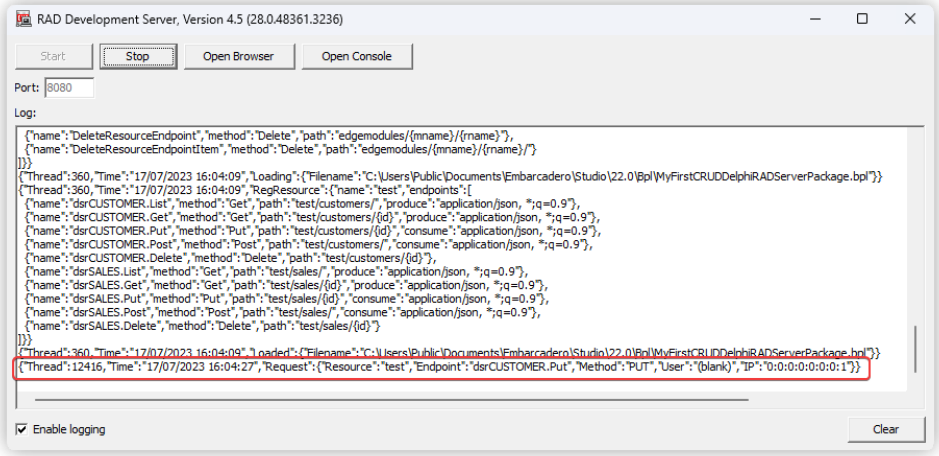


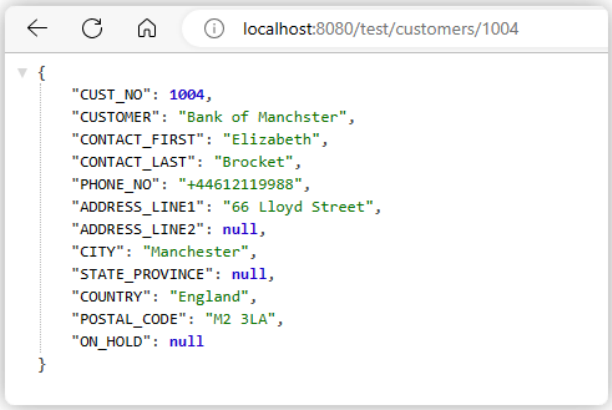
Defining the required values to send the PUT Request

To configure the request we have defined the Method PUT, URL, the resource we are pointing to (in this case the customer with ID 1004) as well as the JSON body with the properties we want to update: the customer's name as well as their phone number.

The only last step is to press "Send Request" and if we get a 200 HTTP response, now we can check on the RAD Server log that the request went through. Also, if we send a GET request for this particular customer (we can do this on REST Debugger or in the browser) we will see that the data has been successfully updated.

The procedure is the same in case we want to create a new customer, although we will need to provide all the required information in the body and change the method to POST.





RAD Server log with the registered PUT request and the modified data

Other features included with REST Debugger

Even though it is not fully related with RAD Server, it is worth mentioning a very powerful feature that REST Debugger offers. Once you have defined the URL, parameters etc use the button “copy components” to generate in the clipboard all the required RAD studio components to just paste them in any of your projects. In that way, you can prototype even faster UIs to access RAD Server or any other third party API.

In the GitHub repository of this chapter you’ll find a basic FMX example of this. All the components required for access to the API were copied and pasted using the button “Copy Components”. To test it out you just need to run first the RAD Server application and then run the FMX one and press “Send Request”.

Another important topic when it comes to REST API is authentication. If you need to authenticate to the API you can use multiple methods under the “authentication” tab, Also, you can use the “Add Parameters” button in the “parameters” tab to include specific parameters in your request, like for example, an api-key parameter in the header.

05

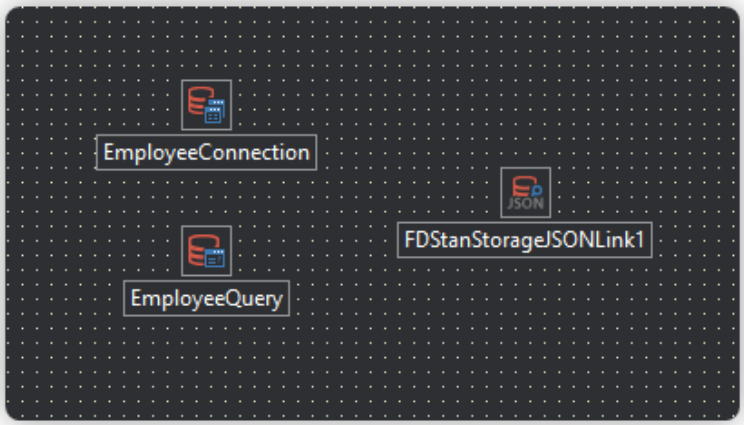
Using FireDAC Batch Move and JSONWriter

Depending on the requirements for your projects or what technology you are more familiar with, RAD Studio allows you to use even more tools to create your REST API.

FireDAC components are available to produce and consume a stream containing database metadata and data encoded in JSON for a response from one of your RAD Server endpoints. This approach is great if your client applications are going to be VCL or FMX. You can use MemoryTables to automatically map all the database information and metadata to map it automatically. Other client applications that use languages, like JavaScript, could have a problem dealing with the database information and data that would be included in the response, but RAD Studio provides a way to generate clean JSON that JavaScript or other languages would expect to receive.

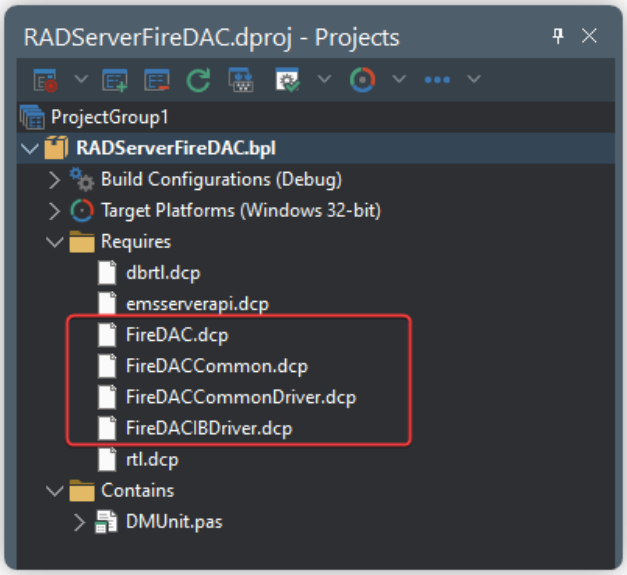
Returning JSON Database Data Using a Memory Stream

FireDAC includes components to access a database table information and produce the result as a JSON string. Create a RAD Server application with a resource module. Add a FDConnection component and connect it with the InterBase sample Employee.gdb database. Add a FDQuery component and set the EmployeeQuery SQL string to select * from employee. Add a FDStanStorageJSONLink component to facilitate creating the JSON.

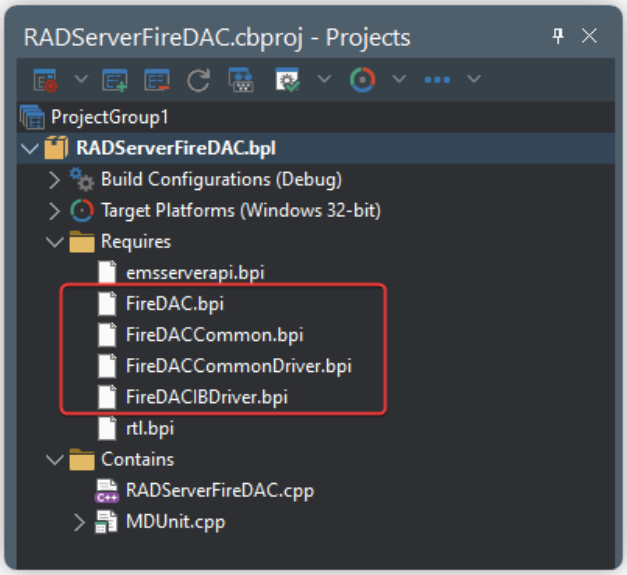


RAD Server project's resource module

When building a RAD Server Delphi based application, a set of warnings may appear and a dialog box will pop-up to allow the application package to be compatible with other installed packages. Clicking the OK button will add the required package files to the requires section in the project. For C++Builder the packages can be added manually (right mouse click on the Requires node in the project manager window and select Add Reference... from the pop-up menu).



Delphi RAD Server FireDAC project



C++ RAD Server FireDAC project

You'll find these files in the C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\lib for each of your target platforms.

Here is the RAD Server Get method implementation that uses a memory stream to send the JSON response with employee table data.

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);
}
```

Use a browser and the URL *http://localhost:8080/FireDAC* to get the response containing the JSON data for the database's employee table. The JSON contains much more information than just the data. Also included in the response is metadata information about the table, columns, types, etc.

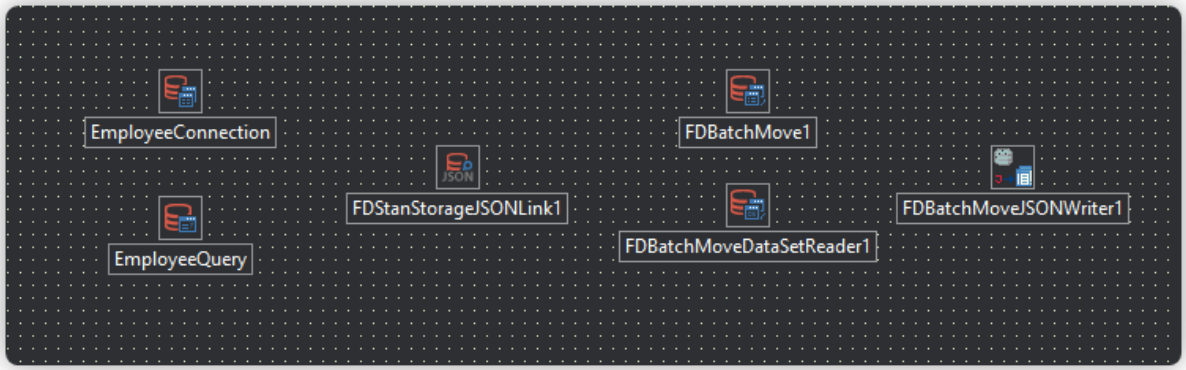
Browser window containing the JSON response

This is definitely not the simple column and value JSON that other languages could consume without parsing the response using code, but it can become very handy if your clients are going to be developed with RAD Studio.

Using FireDAC's BatchMove, BatchMoveDataSetReader and BatchMoveJSONWriter

For a complex database, using approaches like those mentioned in the previous chapter and above would involve writing a lot more code. Taking advantage of FireDAC's FDBatchMove, FDBatchMoveDataSetReader and FDBatchMoveJSONWriter components greatly simplifies the creation of the JSON response.

We are going to upgrade the same project we have created and add the components FDBatchMove, FDBatcMoveDataSetReader and FDBatchMoveJSONWriter to the resource module.



Resource module with FireDAC Query, BatchMove, DataSetReader and JSONWriter

Set the FDBatchMoveDataSetReader's DataSet property to EmployeeQuery.

We are going to create a new endpoint named GetBatchMove.

Delphi:

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

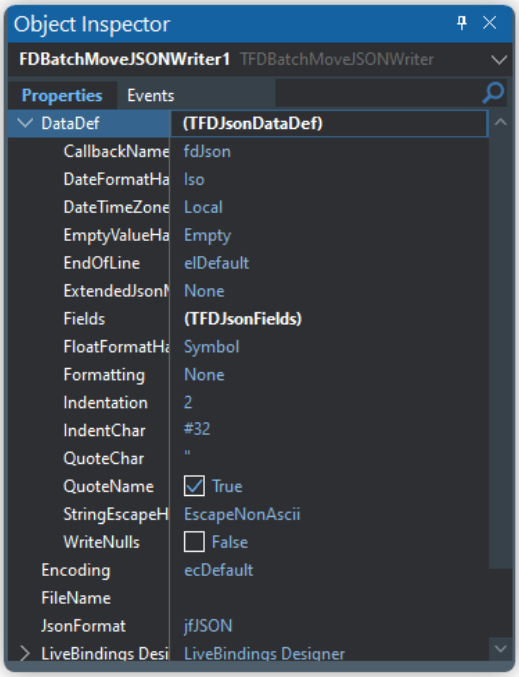
C++:

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

Calling the GET method using the URL *http://localhost:8080/BatchMove* returns the JSON data result:

Browser with JSON result using BatchMove

FDBatchMoveJSONWriter provides multiple options for formatting the JSON result regarding DateFormats, endlines, writeNulls etc.



BatchMoveJSONWriter DataDef sub-properties in the ObjectInspector

The FDBatchMove component also allows you to create mappings to setup source and destination columns mapping and to get the current source record values.

The Mappings property may be filled:

- Manually at design or run times. This allows to specify custom mappings, conversion expressions, etc.
- Automatically at Execute call, if it is empty. The source and destination columns matching performed by the column names. If a destination column has no corresponding source column, then the destination column will be excluded from mapping and will be not filled at data movement.

See Also

- Marco Cantu blog: DataSet Mapping to JSON - RAD Server web service Delphi example
- FireDAC.Comp.BatchMove.TFDBatchMove
- FireDAC.Comp.BatchMove.JSON.TFDBatchMoveJSONWriter
- Readers and Writers JSON Framework
- FireDAC.TFDBatchMove Sample
- RTL.JSONWriter

06

JSONValue, JSONWriter and JSONBuilder

RAD Server provides support for handling JSON data that can be consumed by different programming languages and tools. Creating a JSON string, transmitting the string as a response, and having the client application code process the return is okay for smaller amounts of data. Imagine how large a JSON array response would be for an entire database or a complex data structure? RAD Studio provides three main frameworks for working with JSON data. This chapter covers a few of the many ways RAD Server applications can return JSON to a calling application.

Frameworks for Handling JSON Data

RAD Studio provides multiple frameworks to handle JSON data. The three most common are:

- JSON Objects Framework – creates temporary objects to read and write JSON data.
- Readers and Writers JSON Framework – allows you to read and write JSON data directly.
- JSONBuilder – using writers, create complex structures in a more maintainable way.

The JSON objects framework requires the creation of a temporary object to parse or generate JSON data. To read or write JSON data, you have to create an intermediate memory object such as TJSONObject, TJSONArray, or TJSONString before reading and writing the JSON.

The Readers and Writers JSON Framework allows applications to read and write JSON data directly to a stream, without creating a temporary object. Not having to create a temporary object to read and write the JSON provides better performance and improved memory consumption.

JSON Builder is a combination of the two previous ones. It was created to make your code more readable and maintainable. It also follows a more modern approach where you can chain methods one after another.

In the demo project of this chapter you will find 3 different endpoints that generate exactly the same response but using these three frameworks available. Feel free to use in your projects whichever you feel more comfortable with.

Same JSON response obtained on each endpoint

Using JSONValue

Use the JSON Objects Framework to create JSON strings by assembling them in code. JSONValue is the ancestor class for all the JSON classes used for defining JSON string, object, array, number, Boolean, true, false, and null values. Included in the RAD Studio JSON implementation are the following classes and methods:

TJSONObject – implements a JSON object. Methods in TJSONObject Include:

- Parse – method to parse a JSON data stream and store the encountered JSON pairs into a TJSONObject instance.
- ParseJSONValue – method to parse a byte array and create the corresponding JSON value from the data.
- AddPair method – Adds a new JSON pair to a JSON object.
- GetPair method – Returns the key-value pair that has the specified I index in the list of pairs of a JSON object, or nil if the specified I index is out of bounds.
- GetPairByName method – returns a key-value pair, from a JSON object, that has a key part matching the specified PairName string, or nil if there is no key matching PairName.
- SetPairs – Defines the list of key-value pairs that this JSON object contains.

- FindValue – Finds and returns a TJSONValue instance located at the specified JSON path. Otherwise, returns nil.
- Get Value – Returns the value part from a key-value pair specified by the Name key in a JSON object, or nil if there is no key that matches Name.
- Pairs – Accesses the Key-value pair that is located at the specified Index in the list of pairs of the JSON object, or nil if the specified Index is out of bounds.
- GetCount – Returns the number of key-value pairs of a JSON object.

TJSONArray – Implements a JSON array. JSONArray methods include:

- Add – Adds a non-null value given through the Element parameter to the current element list.
- Get – Returns the element at the given index in the JSON array.
- Pop – Removes the first element from the JSON array.
- Size – Returns the size of the JSON array.
- ToBytes – Serializes the current JSON array content into an array of bytes.
- ToString – Serializes the current JSON array into a string and returns the resulting string.

Additional JSON classes include:

- TJSONString – Implements a JSON string.
- TJSONNumber – Implements a JSON number.
- TJSONBool – JSON Boolean value.
- TJSONTrue – Implements a JSON true value.
- TJSONFalse – Implements a JSON false value.
- TJSONNull – Implements a JSON null value.

Example using JSON classes

The following GetJSON method of the demo project implements a Get endpoint that uses several of the JSON classes to create, parse and display the results of JSONObjects and JSONArray.

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);
}
```

Using JSONWriter

Using JSONWriter simplifies RAD Server application development to craft custom JSON that delivers data for programming language clients to consume. Use JSONWriter to start your JSON object, write a property name and a value, keep writing properties and values until you end the JSON object.

Example using JSONWriter

Here is an implementation of a Get endpoint that returns data using JSONWriter's WriteStartArray, WriteStartObject, WritePropertyName, WriteValue, WriteEndObject, WriteEndArray methods. The AResponse argument has a built in JSONWriter that is very handy to create more complex structures directly on the response.

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();
}
```

Using JSONBuilder

This framework is a JSONWriter wrapper that allows you to build JSON in a faster and more readable way. It follows a fluent interface (also known as method chaining) approach that in case of very complex JSON structures simplifies your code and makes it easier to maintain and read.

In the same project example you will find another endpoint that uses a JSON Builder to create the response. Let's see the code:

Delphi:

```
procedure TTestResource1.GetJSONBuilder(const AContext: TEndpointContext; const
ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
begin
  var Writer := Aresponse.Body.JSONWriter;
  // link the JSONWriter from the response to the builder
  var Builder := TJSONObjectBuilder.Create(Writer);
  try
    Builder
      .BeginObject
        .BeginArray('colors')
          .BeginObject
            .Add('name', 'red')
            .Add('hex', '#ff0000')
            .Add('default', False)
            .AddNull('customId')
          .EndObject
          .BeginObject
            .Add('name', 'blue')
            .Add('hex', '#0000ff')
            .Add('default', True)
            .Add('customId', 653992)
          .EndObject
      .EndArray
    .EndObject;
  finally
```

```
    Builder.Free;
  end;
end;
```

C++:

```
void TTestResource1::GetJSONBuilder(TEndpointContext* AContext, TEndpointRequest*
ARequest, TEndpointResponse* AResponse)
{
  TJsonWriter* Writer = AResponse->Body->JSONWriter;
  // link the JSONWriter from the response to the builder
  TJSONObjectBuilder* Builder = new TJSONObjectBuilder(Writer);
  try {
  Builder
       ->BeginObject()
         ->BeginArray("colors")
              ->BeginObject()
                ->Add("name", "red")
                ->Add("hex", "#ff0000")
                ->Add("default", False)
                ->AddNull("customId")
              ->EndObject()
              ->BeginObject()
                ->Add("name", "blue")
                ->Add("hex", "#0000ff")
                ->Add("default", True)
                ->Add("customId", 653992)
              ->EndObject()
         ->EndArray()
       ->EndObject();
  } __finally {
       delete Builder;
  }
}
```

You can find a very useful sample project provided with RAD Studio called fmWorkBench (although it's available only for Delphi). You can find it in the path: C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\Samples\Object Pascal\RTL\Json

See Also

- JSON
- Readers and Writers JSON Framework
- JSONBuilder
- WorkBench sample project

- Tutorial: Using the REST Client Library to Access REST-based Web Services

07

Creating your own customized endpoints

Until this chapter we have seen basic JSON structures: arrays, objects... with fairly simple URIs as well: /customers, /sales... but it's very common when it comes to REST API best practices to find sub-resources URIs and also nested arrays/objects inside other objects in the JSON responses. In this chapter we will talk about how to accomplish those kinds of structures with RAD Server as well as creating your own GET, POST, PUT or DELETE methods.

An example of good practices

Even though you can structure your API the way you want, there are thousands of articles talking about the best practices when it comes to standardization or REST API. At the end of the day it is up to you how you want to structure your API, but it's worth reading a bit to know the basics of these standards and try to not reinvent the wheel.

For example: when accessing a specific customer, we have learnt that we use the URI **/customers/{id}** but what if we want to access the sales of that particular customer? A very common option would be to define another endpoint like: **/customers/{id}/sales**. This endpoint will return the sales orders of the specific customer defined by their id.

This is considered good practice and is usually called nested resources or sub-resources. This defines a hierarchical relationship between your endpoints that can help third-parties or your own dev team to understand your API in an easier way.

Avoiding APIs to be too chatty

When developing an application it's common to access a form/webpage and finding yourself needing data from multiple endpoints. Let's say that you access a sales order from a customer: You will probably need the customer's details, order information, shipping address, the lines of that order, maybe payments, invoice... The list of requests can get quite long, and when it comes to REST APIs there is a problem: requests are expensive. With this I don't mean expensive not only for the server to handle all these requests, but also the latency that the internet introduces to this equation. Each request will take a few milliseconds to go back and forth and if we need to send 10 or even more requests to just access one page in particular, there is a lot of room for improvement there. This is when nested JSON responses make more sense.

Imagine that you can request RAD Server in only one request a summary of one customer with all their sales. This could be equivalent to a classic master-detail relationship, but all returned in one request. We will also see how we can accomplish this.

Adding sub-resources

For sub-resources we can still use the same TEMSDatasetAdapter we have seen in previous chapters. We only need to tweak a few things in the attributes and RAD Studio and FireDAC will do the rest for us.

Using the same project we have used so far (with 2 queries: qryCUSOTMER, qrySALES) let's modify the SQL of qrySALES like this:

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

And the attributes on top of the drsSALES EMSDatasetAdapter, let's change the attributes for these ones:

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}";
```

On the SQL statement we have just added a WHERE clause with a parameter that filters the sales of a specific customer.

If we check the attributes, something more interesting is happening. RAD Server will automatically inject the value from {CUST_NO} into the FireDAC query and will filter the sales of that customer. Also, we needed to specify the rest of the methods (List, Get, Post etc) because now 2 keys are involved in the same endpoints and it's mandatory to specify the names of them to make it work. The good news is that we can still use these endpoints to create, modify or delete specific sales as we would do with any other endpoints created using an EMSDatasetAdapter.

Adding nested data in a response (Master/Detail)

Now that we have our first sub-resource endpoint, let's create a nested response with multiple values. For this, we finally need to write some code.

Let's create one published method in our TTestResource1 data module class.

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();
    }
}
```

We can see in this simple code that we just retrieve the details of the specific customer ID we are getting from the URL using a macro, and passing it as a parameter to the same qrySALES we were using already. No extra queries required.

Although using components like EMSDatasetAdapter helps a lot in a low-code approach, sometimes we need specific requirements and we need to code our own implementations. As we have seen in previous chapters, we can use JSONWritters to customize our response as we please.

If we check the code of this example, on every incoming request we have access to the ARequest and AResponse arguments to access all the required information and build our own responses. In this example we use the method “WriteStartObject” to create a new object, then use two methods from a TQuerySerializer class (more about this later) and then we end the object.

There are multiple useful methods and properties we can use with JSONWriters and JSONReaders that will make your coding experience much easier. I strongly encourage you to check the documentation to learn all the available features.

Now you are probably asking yourself: But what are those 2 methods: **`ExcludeMasterFieldFromFields`** and **`SerializeMasterDetail`**? These two methods have been coded for this particular demo example, but you have access to them in the GitHub repository demos, and of course, feel free to use the code in your projects as well. Their task is very well documented in the methods but summing up, they just convert a master/detail relationship to a JSON object with the detail query inserted in as a JSON array in the main JSON object. **`ExcludeMasterFieldFromFields`** could be not necessary, but for avoiding redundant data, we exclude from the detail the MasterField.


tip

Check out the unit Data.DBJson It includes multiple classes to help you out converting Datasets to JSON and vice versa. In this example we have used the class `TDataSetToJSONBridge` *that allows us to serialize these much faster and granularly.*

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;
}
```

note

In a real project it would make more sense to abstract these methods in another class/unit because they can be easily reusable, but for simplicity we have kept them in this same DataModule.

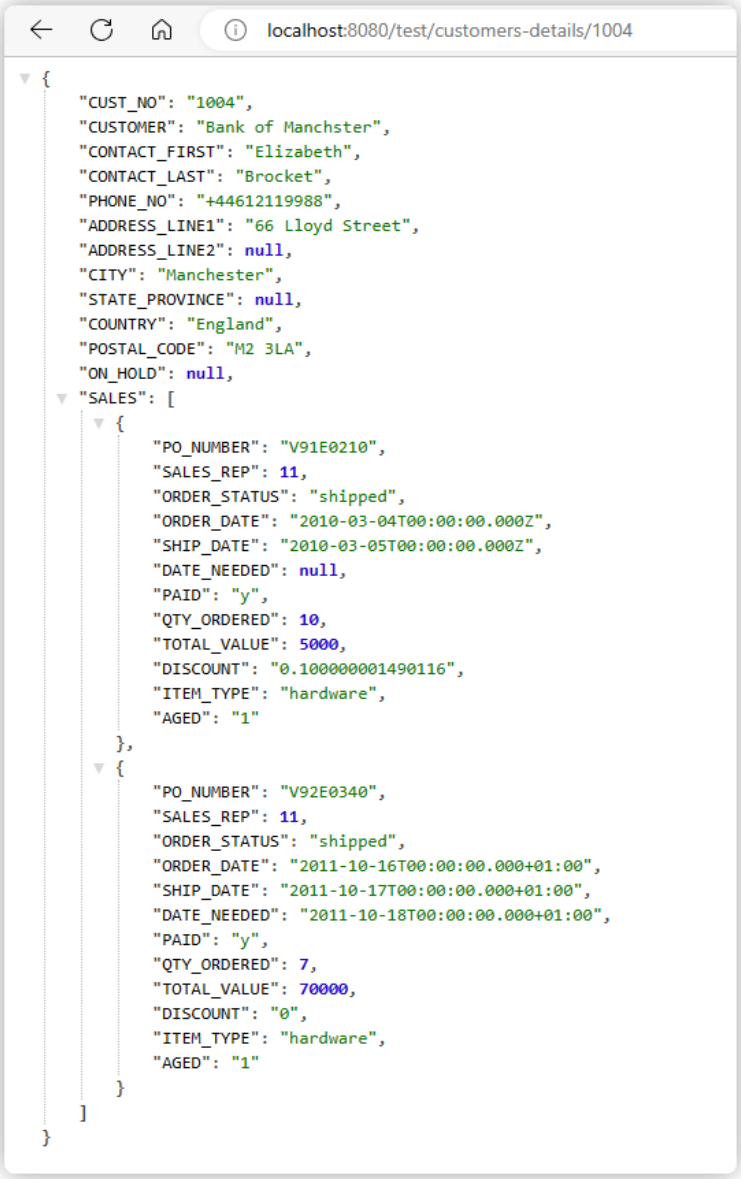
Testing the new implementations

Let's run the demo project and access the URL http://localhost:8080/customers/1040/sales/

Accessing sales from a specific customer using sub-resources approach

We are now filtering the sales of the customer 1004, but if we want to access/modify/delete one sale in particular, we can also access through this same URI. We just need to add to the end the order Id, for example: http://localhost:8080/test/customers/1004/sales/V91E0210

Let's access now the other endpoint we defined: http://localhost:8080/test/customers-details/1004



Accessing an endpoint with sub-resources (Customer and their sales)

In the same request we are getting all the sales from the specific customer which means that instead of two calls to RAD Server, we got all the information we needed in just one. Obviously these kind of requests can get more complex with multiple nested levels etc.

note

In the github repository demo project associated with this chapter you will find an extra endpoint to access a list of customers and their associated sales. Instead of filtering one in particular we will obtain all of them. Notice that most of the code has been reused making it very simple to implement in further developments.

Creating custom GET, POST, PUT, DELETE methods

So far we've seen how to create GET custom methods, but in some scenarios we will need to use other verbs like POST, PUT and DELETE. To code this kind of implementation we just need to start the method name by the verb we want to use IE: **"procedure PutMethodName(.."**. As you have probably seen in the previous examples, all the methods que customized started with "Get": if we change that for, let's say, "Post" we will be defining a POST method. Let's see an example:

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;
```

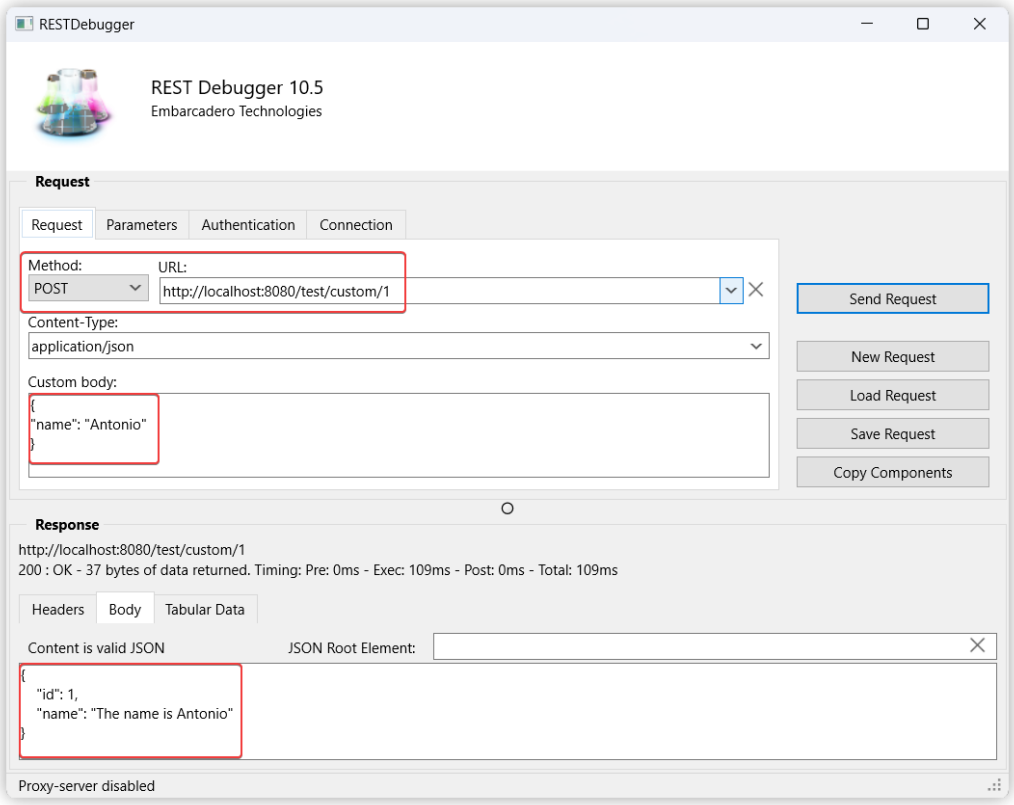
C++:

```
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();
}
```

Now we have available a new extra endpoint ./custom/{id}. If we send a POST request from REST Debugger adding in the body the expected “name” property we will get this.



Response from the custom POST method

Handling response errors

As we could see in the previous example of a customized POST endpoint, we raised an error in case we didn't get all the data we expected. RAD Server offers a built-in solution to raise and return the most common errors straight away from the AResponse object. You can find more detailed information about the available errors in this link.

See also

- JSON Writers and Readers
- REST API Best practices

08

Accessing the built-in analytics

The RAD Server Console is a service that provides a pre-configured web application which displays multiple data as well as analytics from the RAD Server Engine. It allows you to have a more in-depth view of the activity on your RAD Server instances and make decisions based on real data. Analyze user, API, and services activity to gain insight into how your application is being utilized.

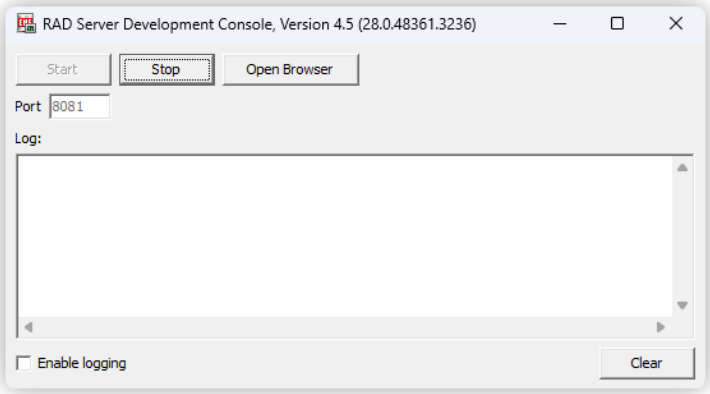
Main Characteristics

The RAD Server Console accesses the database server in read-only mode.

- It gives feedback on the API calls with statistics from the RAD Server Engine resources: Users, Groups, Installations, modules and its resources.
- You can use the console as a stand-alone application for testing purposes or set up the console on a Microsoft IIS Server for a production environment.
- Note: Microsoft IIS Server is not available on Linux. You can use Apache for a production environment on Linux
- The RAD Server Console offers analytics for new resources by extending the functionality of the server.
- The RAD Server Console offers analytics for the registered RAD Server users.
- You can export and save the analytics data to a .csv file in your system.

Accessing the RAD Server Console

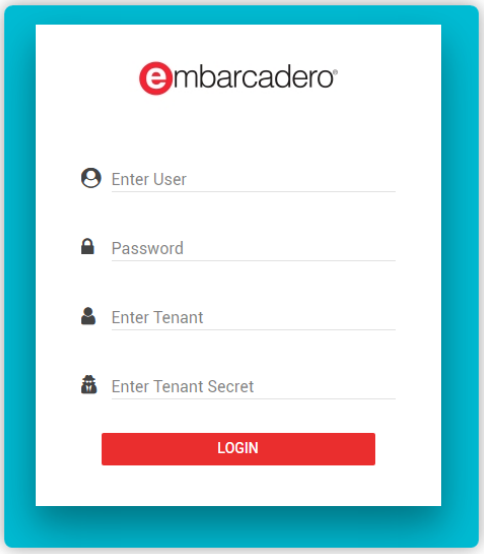
Go back to the RAD Server Development Server and click the Open Console button. This will start the RAD Server Development Console Server running automatically on port 8081 and It will also open a browser with the analytics console login window.



RAD Server Development Console Server UI

note

If the default port 8081 is being used by your computer, you just need to change 8081 for any port available on your machine, press "Start" and then "Open Browser".

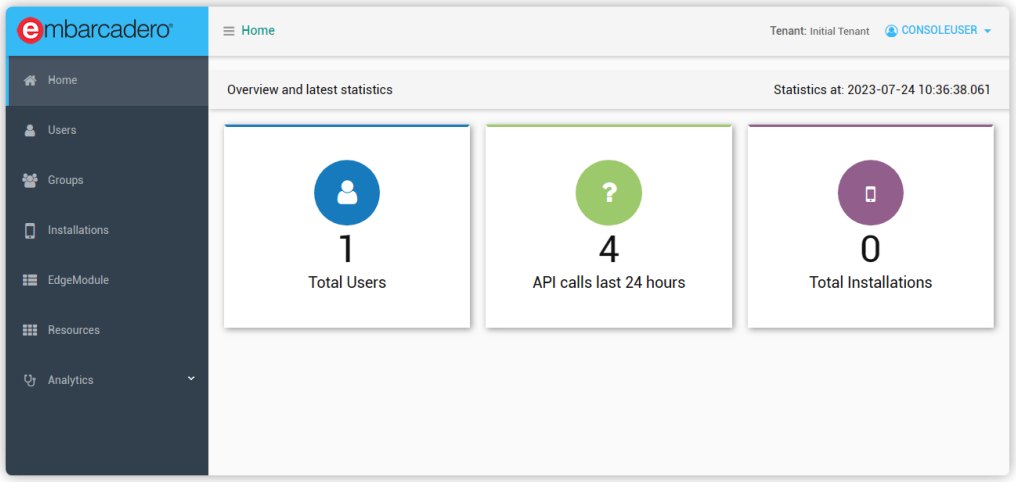


RAD Server Console user login screen

To access the console, RAD Server comes with default credentials preconfigured (leave the tenant info empty):

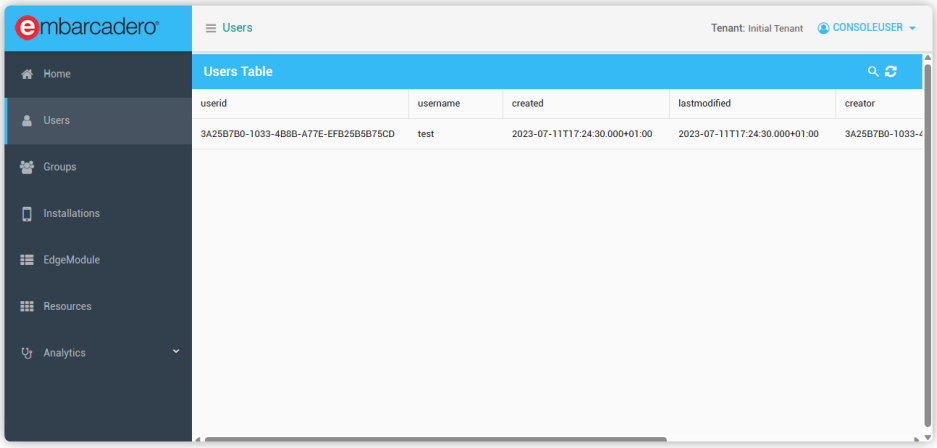
user: **consoleuser**
password: **consolepass**

RAD Server provides a default user and password to access the console. Remember to change these credentials in the emsserver.ini configuration file (check the chapter about this configuration file for more detailed information).



RAD Server Console home page

After logging in you'll see a graphics view of the RAD Server console single page JavaScript app with menu on the left and content on the right. The menu provides information of users, groups, device installations, EdgeModules, Resource Modules and Analytics. Here is the screen that shows a list of users and their information including when the user was created and when the user information was last modified.

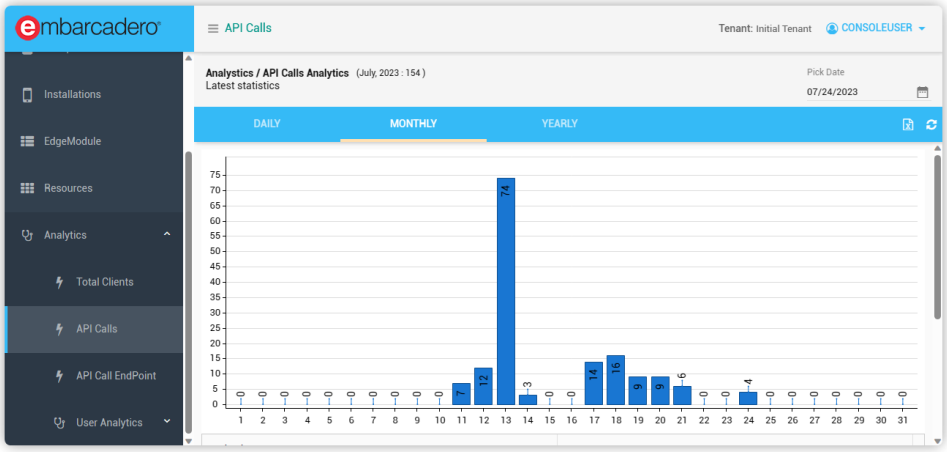


RAD Server Console users table

Clicking on the Analytics menu item opens a menu to select from a range of analytics including total clients, API calls, API endpoints called, and more. Analytics can be chosen by day, month and year. The analytics can also be filtered by user, group, etc and specific endpoints. Analytics results can also be saved to a .CSV file for additional processing by external applications.

These analytics provide great information for the decision making process and auditories. Seeing when your services are more or less used for planning updates, or seeing which endpoints are rarely used are just examples of very valuable insights.

The following chart shows an API calls chart for a selected month.



RAD Server Console Ext JS API calls analytics page

09

Deploying RAD Server

Previously, the first RAD Server applications were tested using the development versions of the RAD Server (EMSDevServer.exe) and Console (EMSDevConsole.exe) applications. This chapter covers the multiple platforms where you can deploy RAD Server in production. If you are interested in RAD Server Lite, jump to the next chapter.

warning

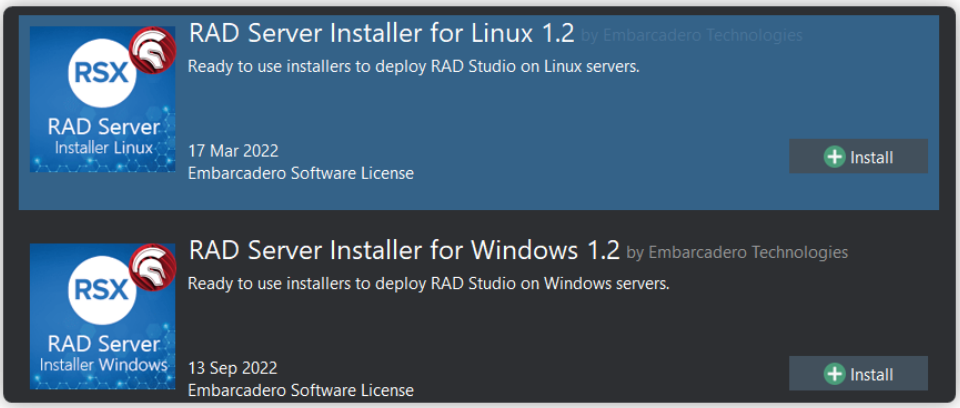
When a new bpl or dcp resource is compiled, this won't be included in the "export" folder of your project (where binaries usually go). These resources will be created by default in your Embarcadero Studio installation path:
C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\"Bpl or Dcp"
Inside the Bpl or Dcp folder there are platform specific ones .

Where can RAD Server be deployed

RAD Server is compatible with the platforms **Windows**, **Linux** and **Docker**. Although conceptually speaking the services required for each platform are the same, there are some differences that we will see in this chapter, but first, let's talk about the similarities and how RAD Server works under the hood.

Using the installers from GetIt

If you are deploying your RAD Server application on Windows or Linux, the fastest way to do the installation is using the installers that you can download from GetIt. You just need to search "RAD Server" and you will find these two:



RAD Server installer from GetIt

Once the "installation" has finished (even though it's more a download) you can find the installers in the path: C:\Users\<YourUserName>\Documents\Embarcadero\Studio\<XX>.0\CatalogRepository and copy the whole folder to your production machine. The folder contains all the required files for a full installation of RAD Server (InterBase installer included).

Before executing the installer in your production environment, you must have installed IIS or Apache so the installer can configure all the requirements accordingly.

The installer will guide you through the different options you need to install.

note

During the installation you'll be requested to provide a valid license for InterBase. Use your EDN account and the RAD Server serial number to register the InterBase Instance.

Prerequisites to deploy RAD Server manually

This chapter is focused on understanding all the parts that need to be installed or configured to deploy RAD Server. Even if you use the installer, it's important to understand all the requirements for a better debugging and problem solving. Also, if you need to update your RAD Server to a newer version, you won't need to reinstall the whole application and updating just a few dlls and bpls/so should be enough.

These are the mandatory requirements for a RAD Server installation working in production:

- InterBase Server engine
- RAD Server license

- RAD Server installation
- Web Server (IIS 7+ or Apache 2.4+)
- Resource files compiled with RAD Studio
- Configuring the EMSServer.ini file

Regardless the platform you choose to deploy, you will need to install/configure all these. For example, on Windows you will need to configure Microsoft's Web Server IIS or Apache for Windows and on Linux you will need Apache. It's important to understand that RAD Server is not an executable as such (apart from the Lite version, but more on that later). Resources are compiled in the form of BPLs for Windows or SO libraries for Linux. That's why we need a web server to access those resources.

When it comes to InterBase, a database instance needs to be used by RAD Server internally. A lot of information is saved (statistics, users, roles etc) and that's why it requires its own database to store all that information.

The fact that RAD Server uses InterBase internally doesn't imply that you must use this database engine for your own data. FireDAC connects to a wide range of databases and you can choose whichever suits your needs.


note

In case that InterBase is your database of choice and it's going to be deployed in the same machine, you'll need two instances running on different ports. It's common practice to keep the port 3050 for your own database instance and install RAD Server InterBase instance in another port. IE: 3051. The same instance can't be used because RAD Server uses its own encryption system.

Deploying on Windows manually

InterBase Server engine

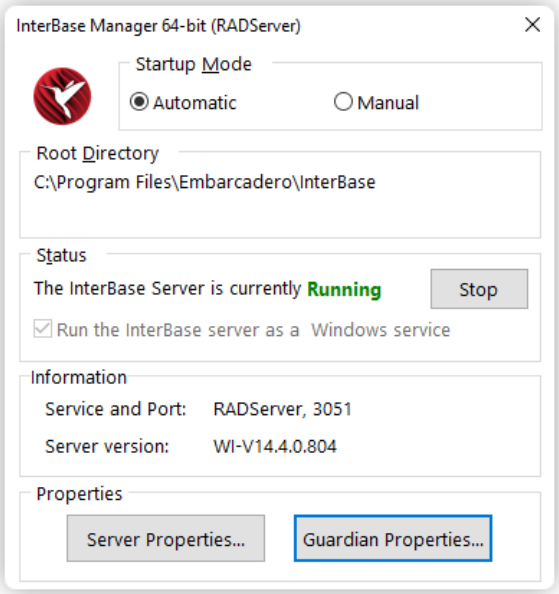
Download the latest InterBase installer for Windows from https://my.embarcadero.com and install it in your production machine. You can follow this tutorial in case you've never installed it before. Here you can also find RAD Server Database Requirements for a Production Environment on Windows.

Specific details for the installation:

- Choose "Server and Client"
- Allow running multiple instances of InterBase on the same machine
- Change the default port suggested to 3051
- Name the instance RADServer (instead of the default gds_db)
- For registering InterBase, use the same RAD Server serial number you were provided and your EDN account.

Once the installation has finished, you need to start the RADServer instance of InterBase server. Choose Start | Programs | Embarcadero InterBase | 64-bit instance = RADServer | InterBase Server

Manager. If you want InterBase to run as a service (the default) check that box. If you want InterBase to run when your computer starts, click the Automatic radio button. Then click the Start button.



InterBase Manager 64-bit for RADServer

You can find InterBase Manager in your programs list under "Embarcadero InterBase" or in the path where you installed it, under the folder ".\bin\IBMgr.exe" and specifying the instance name IE: ".\IBMgr.exe RADServer". Another option is simply using the windows search and type "InterBase Manager".

RAD Server installation

For installing RAD Server on a Windows machine we need to follow very similar steps to when we configured it in our dev machine. Most of the files that will be required for this process can be found in these folders:

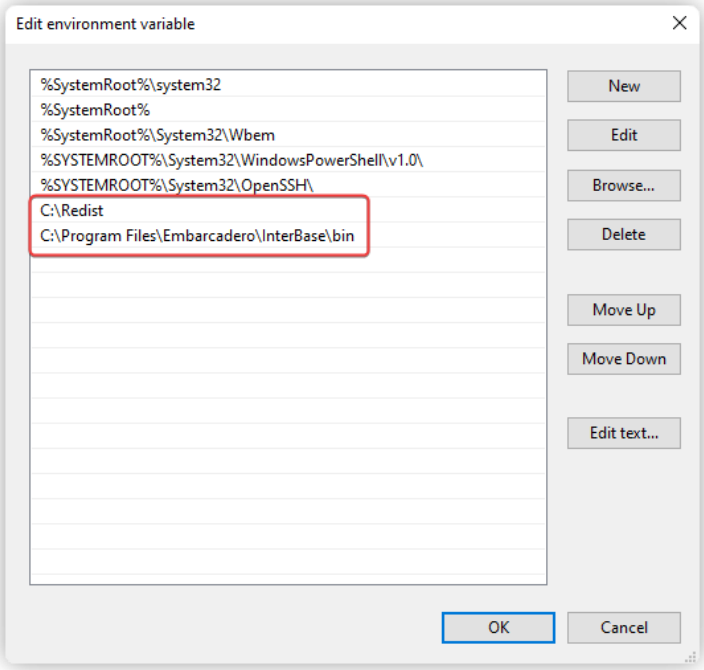
- C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\bin64
- C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\redist\win64

In the docwiki there is a very detailed tutorial about how to install RAD Server on Windows. You can access it here. Nevertheless, we will explain the basic steps here:

Follow these steps to prepare the production server to test and use the InterBase RAD Server instance and EMSDevServer.EXE to create the RAD Server database and configuration file.

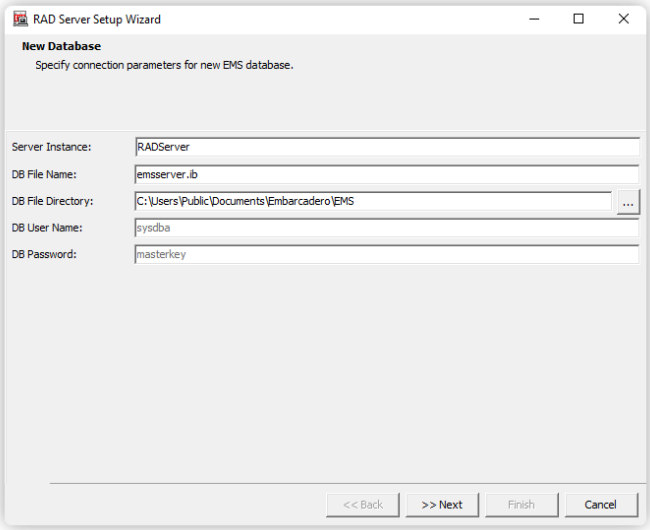
1. Copy the 64-bit EMSDevServer.exe to the production server in a folder called c:\installs\EMS

2. Copy the required files from the RAD Studio Redist/win64 folder to the production server in a folder called c:\Redist
3. Edit the system path environment variable on the production server to add the c:\Redist and c:\Program Files\InterBase\bin folders.

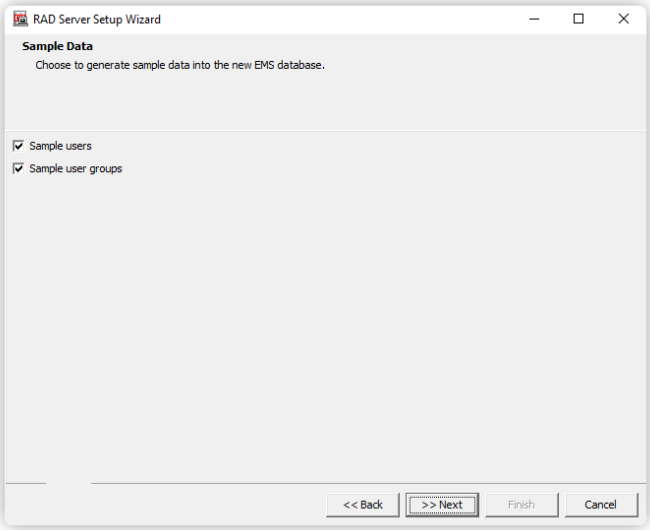


Add two folders to your system path

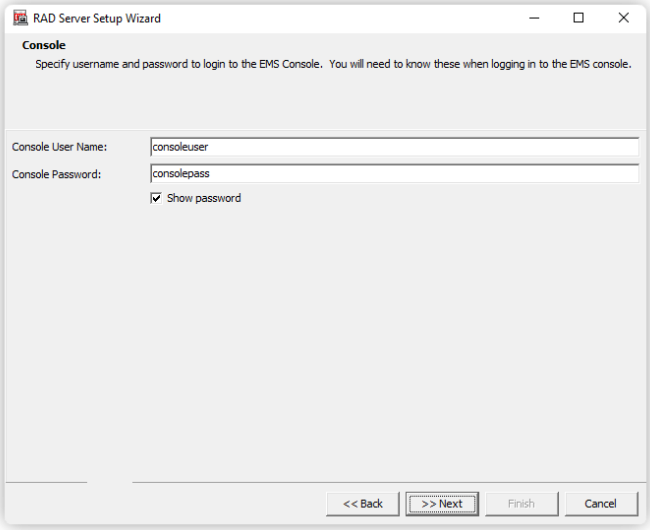
4. Copy the EMS template and web resources files on the development computer from C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\ObjRepos\en\EMS to the production server folder called c:\installs\ObjRepos\EMS (EMSDevServer.EXE will look for the template and web resource files in an ObjRepos\EMS sub-folder under the same parent folder you place the EMSDevServer folder)
5. Make sure that the InterBase server with RAD Server license is started on the production server.
6. Run the EMSDevServer.exe (as you did in the first RAD Server development configuration) to setup the production RAD Server configuration file and InterBase RAD Server database. The following screens show the steps.



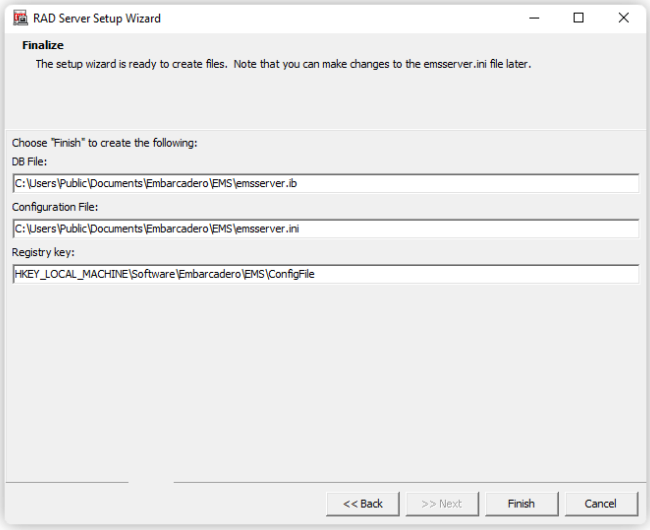
RAD Server Setup Wizard - set connection parameters for RAD Server



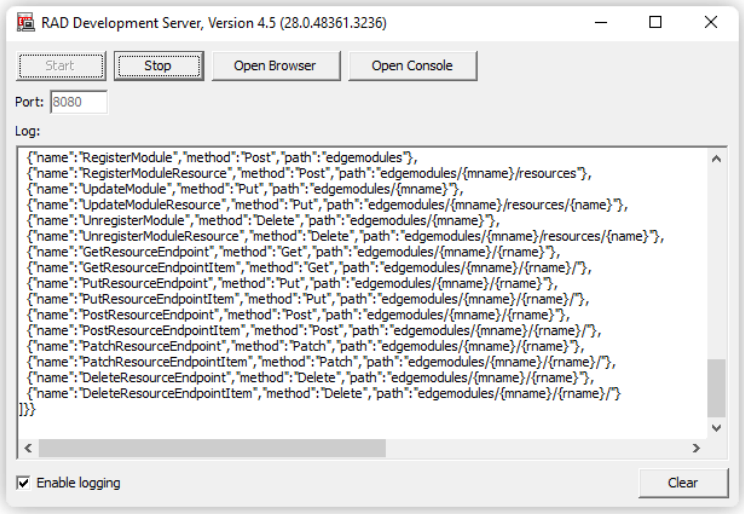
RAD Server Setup Wizard - choose to generate sample data



RAD Server Setup Wizard - set connection parameters for RAD Server

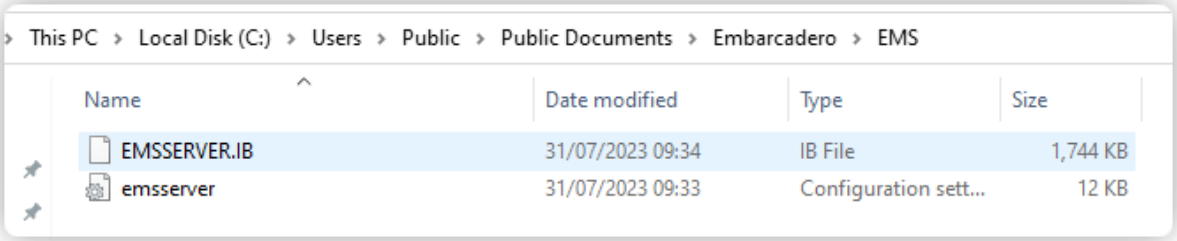


RAD Server Setup Wizard - review the files and registry key that will be created



RAD Server Development Server is running

The RAD Server wizard will create two files in the default folder under the Public Documents folder.



RAD Server Setup Wizard - two files created in a public documents folder

Web Server (IIS or Apache)

If you deploy on Windows you can use the web server that Microsoft provides with Windows a.k.a. IIS or you can also use Apache for Windows. In this link you can see a detailed guide about not only the files you need to copy to your production environment but also how to configure IIS or Apache on a Windows machine.

If you choose IIS, Microsoft has different versions of it and the installation process of the service can differ a bit. If you don't have previous experience with this service, you can find information in this link.

The last step will be copying our compiled resources. Go to the end of this chapter to see how to do it.

If you use on Server Manager the "add role or feature" option for adding Web Server (IIS), it's mandatory to install "ISAPI Extensions" and "ISAPI Filters" under the "Application Development" section

Deploying on Linux manually

Deploying RAD Server and your applications to a Linux server provides the following options:

- To create a Stand-alone RAD Server, see the RAD Server Installation section
- To create RAD Server for Apache, see the Setting Up RAD Server for Apache section

Compatible Distros

RAD Server officially supports Ubuntu 18+ and RHEL 7+. This doesn't mean that it can't be installed in other distros like RockyLinux, Debian etc but the internal testing is always done with the officially supported ones.

Installing InterBase Server engine

Download the latest InterBase installer for Linux from https://my.embarcadero.com. Inside the zip file you'll find the installer. Here you can also find RAD Server Database Requirements for a Production Environment on Linux.

Once you have unzipped the file you downloaded, assign execution permissions to the installer and execute it:

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

Specific details for the installation:

- Choose "Server and Client"
- Allow running multiple instances of InterBase on the same machine
- Change the default port suggested to 3051
- Name the instance RADServer (instead of the default gds_db)
- Install Folder: /opt/interbase

InterBase installer will detect automatically if your Linux installation has Desktop environment or not. In case you want to force Console mode execute the installer use this argument:
sudo ./install_linux_x86_64.sh ***-i Console***

You can define the name for the instance and path with the name you prefer. If you do, keep in mind to reference to those accordingly during the configuration process.

Registering and starting InterBase Server

To launch the registration wizard execute the command:

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

This will launch the license wizard. For console mode, we recommend option 2 “Direct register” where you’ll be able to specify your RAD Server serial number plus your EDN account. The assistant will do the rest of the work and will verify your license connecting to the Embarcadero servers.

If you want to verify now if the license has been loaded correctly, you can use option 1 in the previous menu “List license” to confirm everything went as expected.

The InterBase instance is already installed and licensed but it needs to be started. For that, we need to enter the InterBase console executing this command

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

To simplify connecting other applications and services to InterBase databases the simplest approach is to create a symbolic link to the InterBase library and point it to /usr/lib. This will avoid you to copy the lib to every service that needs an InterBase connection.

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

Running InterBase as a Service

InterBase can also be set up as a service so that it runs when Linux is started. Use the following commands in a terminal window.

Access the “examples” folder in the path where you installed interbase and copy the ibserverd script file to a version for the server instance you installed:

```
sudo cp ibserverd ibserverd_RADServer
```

Setup the automatic service launch by executing the above script as 'sudo' or 'root'.

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

The 2nd argument is the install folder, and the 3rd argument is the instance name. Now, when you restart the system, the service should start automatically as long it is properly licensed.

Check to make sure InterBase is set to start as a service on next reboot or startup.

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

When running InterBase as a service, the InterBase server starts automatically whenever the machine is running in a multi-user mode.

If you prefer to create the service manually (or your Linux distribution uses a slightly different approach) you can find detailed information about this setup in this link.

To remove InterBase as a service, run:
sudo /opt/interbase/examples/ibservice.sh -r[emove]

Installing RAD Server

On the machine where you have RAD Studio installed you'll find the RAD Server Linux installer in the path: C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\EMSServer

Copy those files to your Linux machine and execute the installer. You may need to give execution permissions to it.

Ensure that libcurl is installed. To install it use your distro package manager. IE for Debian based: *apt install libcurl4*

The install shell script will create a directory, /usr/lib/ems, containing the EMSDevServerCommand, EMSDevConsoleCommand along with several runtime library (.so) files required for the command files to execute. You can also find a tutorial from the docwiki in this link with detailed information.

Once the installation has finished, run the EMSDevConsole in setup mode:

```
/usr/lib/ems/EMSDevConsoleCommand -setup
```

type **start** and press enter.

Specify the connection parameters by entering the following values:

- Server instance: type the following default instance name **RADServer**
- DB File Name: the default name is **emsserver.ib**

- DB File Directory: /user/lib/ems
- DB User Name: the default parameter is **sysdba**
- DB Password: the default parameter is **masterkey**
- Console User Name: the default value is **consoleuser**
- Console Password: the default value is **consolepass**

Type **"n"** if the configuration options are correct. The emsserver.ini and emsserver.ib files are created and RAD Server starts execution on port 8080. The configuration file can always be manually edited.

Once the configuration process has finished, you can find the RADServer database in **/usr/lib/ems** and the configuration files in **/etc/ems.**

Keeping the DevServer executed, let's test now that we can access it correctly and we get a response. Access http://<LinuxMachineIP>:8080/version.

Browser showing the output from calling the version endpoint

EMSDevServerCommand and EMSDevConsoleCommand can be used for developing and testing Linux RAD Server applications without using Apache. The next step is to set up and test RAD Server and Delphi/C++ compiled application modules to run in production mode on Linux and Apache.

In case you want to deploy RAD Server on Linux but also use InterBase as your database of choice for your data, you can follow this tutorial.

Setting Up RAD Server for Apache

Use the InterBase iSQL command (in the /opt/interbase/bin directory) to make sure that RAD Server will be able to connect to the emsserver.ib database file.

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

Configure the Apache HTTP Server to load the Apache RAD Server (libmod_emsserver.so) and Apache RAD Server Console (libmod_emsconsole.so) modules. Even though Apache's configuration is very

similar regardless of the Linux distro you are using, keep in mind that there are some differences between RHEL and Debian based distributions.

note

Check the documentation of your Linux distro to verify what's the recommended way to load modules as well as define location tags.

Add the following lines to load the RAD Server Apache server module (libmod_emsserver.so) and the RAD Server Apache console module (libmod_emsconsole.so).

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

Add the Location tags to create a container where you can specify access control rules for a given URL.

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

To test that your RAD Server is correctly running, use a browser to bring up the RAD Server version number accessing: http://<LinuxMachineIP>/radserver/version

The last step it will be copying our compiled resources. Go to the end of this chapter to see how to do it.

Deploying on Docker

The deployment of RAD Server in Docker is much simpler than using Windows and Linux. Embarcadero has various images in dockerhub available for this platform.

You will find 2 images related with RAD Server: The only difference between these two is that one has an InterBase Server Engine running inside the container and the other one assumes that the InterBase Server required for running RAD Server will be hosted somewhere else.

tip

InterBase Server is also compatible with Docker and Embarcadero provides an image to containeraize it. Here's the link to DockerHub.

How these Docker images are built is fully open source and are publicly available on GitHub. This is just one approach, but if you are comfortable enough with Docker, feel free to use these as a template and adapt them to your specific needs.

There is plenty of information about how to deploy and customize these images in the DockerHub and GitHub links underneath.

Option 1: PA-RADServer-IB

This image is what we could call “all batteries included”. Having InterBase built in makes things much easier, but keep in mind that the first time you run this container you can’t do it in detach mode. A first wizard needs to be executed for configuring the RAD Server license and a few extra details. Once everything is set up, you can run it detached.

Another thing to keep in mind is that this container is very handy if you don’t have plans in scaling your application and want to have everything in one place, but if you want to scale the application in the future, maybe the best approach is to separate RAD Server from the InterBase Server and have them in separated containers/machines.

DockerHub link

GitHub link

This image includes:

- InterBase Server
- PAServer
- RADServer required files
- Apache pre-configured

Option 2: PA-RADServer

This container will need to connect to an InterBase Server with a valid license of RAD Server installed, otherwise it won’t work. It’s an ideal container in case that you want to scale your application and deploy multiple instances connected to the same InterBase Server.

DockerHub link

GitHub link

This image includes:

- PAServer
- RADServer required files
- Apache pre-configured

Remember that for simple environments, you can use PAServer to upload your resources updates straight to the container without the need of regenerating it.

Access this link for further information about deploying RAD Server on Docker.

Copying RAD Server modules compiled with RAD Studio

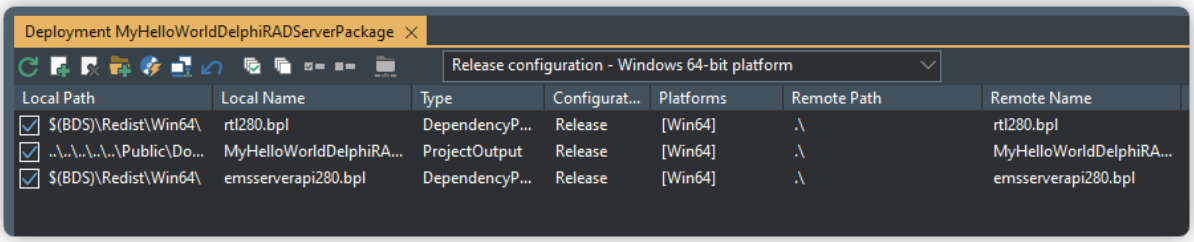
The process of deploying modules or required additional libraries to your production machine is nearly identical regardless of the operating system you've chosen. For your own resources, you only need to copy the .bpl/.so files to your production machine.

RAD Server application package files are compiled into folders depending on the project settings. The default Delphi package output and C++ final output directories are:

- For Delphi:
  - 32-bit Windows - C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\Bpl
  - 64-bit Windows - C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\Bpl\Win64
  - Linux - C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\Bpl\Linux64
- For C++ all RAD Server application package files are compiled to the .\$(Platform)\$(Config) folders.

There are several ways to deploy the required RAD Server application and run time DLL files to the production server. Three common transfer methods are:

- Copy the package files to the production server path where RADServer is installed
- FTP the files to the production server
- Use the Platform Assistant (PAServer) with the "Project | Deployment" menu item to have the IDE move the files to the production server. This screenshot shows an example for windows 64.



Project | Deployment files that PAServer can transfer

Copy the compiled RAD Server extension package files (for example, the project from Chapter 1 MyHelloWorldDelphiRADServerPackage) to the production RADServer folder.

When using PAServer, the default path where the files are deployed can be changed editing the file paserver.config in the production machine. Elevated privileges may be needed when PAServer is executed, depending on the needed path to write files.

Configuring the EMSServer.ini file

Now that we have added a new resource to our production folder, we need to specify in EMSServer.ini file that there is a new resource available.

Edit the emsserver.ini file to add each of your RAD Server extension packages under the [Server.Packages] section.

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

To configure the emsserver.ini file of an already running instance, run the **./config.sh** script. The script will restart apache automatically.

10

RAD Server Lite

What is the Lite version?

RAD Server requires a backend database, based on InterBase, and it is generally deployed as a web server DLL module for either IIS or Apache. For this reason, a standard deployment requires:

- The web server and its configuration of the RAD Server module
- The RAD Server deployment and configuration
- An installation of InterBase with a special purpose RAD Server license (a license the user needs to register on the target device to activate)

For development, we have long offered a stand alone version of RAD Server, based on the Indy HTTP server, which offers limited performance but much easier deployment and the ability to be executed under the debugger (so you can debug your RAD Server modules code). The development version is not meant, and it's not licensed for deployment. It has a limit in the number of users you can create, and it can work with a local InterBase Developer edition (the license for it is part of the RAD Studio license).

RAD Server Lite (**RSLite**) offers a simpler deployment model for test servers and scenarios not requiring a lot of throughputs, and it offers this by using the InterBase embedded database engine, IBToGo, instead of the full-blown server and combines it with a simplified licensing model.

RSLite uses the same binary of the development edition (that ships with RAD Studio) along with IBToGo binaries and a license slip file you can deploy with your solution (requiring no registration on the computer you deploy it to). Because it uses an embedded database and because it uses the Indy HTTP Server component, it cannot serve the same number of requests per second of a regular full-blown RAD Server installation, and it cannot scale with multiple RAD Server front ends.

The underlying architecture used by RSLite has much more limited scalability, but we expect it to be sufficient for many simple deployment scenarios — keeping in mind that the throughput also depends on the specific code your RAD Server modules execute.

For deployment on a public system, we recommend avoiding exposing the RSLite HTTP server directly, but making it accessible via a proxy configuration so you still have a web server (like Apache or IIS) providing the security context for the incoming HTTPS calls and forwarding those to RSLite.

How to get a RAD Server Lite License

You can redeem a license with any Enterprise or Architect license for RAD Studio 11 (including Delphi 11 and C++Builder 11). Visit this page and follow the instructions provided.

You need your registration key and EDN account.

The process here is not just to receive a license key for RSLite, but a slip file (a license stored in a .TXT file) that you can deploy alongside your installation. This license has no limitations in terms of the number of installations, but you cannot have two instances running on the same machine.

The license file needs to be placed in a specific sub folder, unlike what the general information on the redemption site might seem to imply.

Deploying a RAD Server Lite project

Once you have the license before you deploy a project, there are two different considerations:

- First, you need to create a deployment configuration with RSLite, the required runtime packages, and IBToGo deployment (These are the steps)

- Second you need to generate a proper database file for production, compatible with the IBToGo license — a local database created by RAD Server Developer edition won't be compatible

The Files to Deploy

Deploying manually

In practical terms, these are the files needed to deploy an RSLite solution (in addition to your application packages and their dependencies):

1. The RSLite executable, which is the same of the developer edition: **EMSDevServer.exe** available in the RAD Studio bin folder (or the similar 64-bit version)
2. The required RAD Studio runtime packages, which include those required for a minimal installation (listed here and available in the RAD Studio win32 or win64 redist folder) plus any other runtime package required by the code in your RAD Server modules:
    - 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. The InterBase ToGo deployment files found under the public document InterBase redist folder (for example, C:\Users\Public Documents\Embarcadero\Interbase redist\InterBase2020) in the subfolders win32_togo or win64_togo — for Linux you can find the **libibtogo.so** file in the proper InterBase redist folder
4. Add the license file obtained above to the interbase/license folder (part of the IBToGo redist configuration)

Using the Deployment Wizard

Deploy your files using the RSLite feature in the Deployment Wizard by following these steps:

1. Add the RSLite feature.
2. Next, add the IBToGo feature.
3. Uncheck the **iblite** registration file

4. In the How to get a RAD Server Lite License section, add the file to deploy: generate a **rslite** activation file and set its destination as "interbase/license".
5. Next, add the file obtained from the Creating the Production Database section to deploy my **emsserver.ini** to ./.
6. Finally, add the file obtained from the Creating the Production Database section to deploy my **emsserver.ib** to ./.

MSVC runtime

In order to run IBToGo (and so RSLite using IBToGo) on a target Windows machine it needs to have the Visual C++ 2013 runtime library installed. On a developer machine with RAD Studio, you'd most likely have it already installed. On a general target deployment machine, however, you might have to install it, after downloading it from Microsoft.

Creating the Production Database

With this configuration, you can start the RSLite by executing the **EMSDevServer.exe** application. Note that if the target machine has an InterBase client, it will pick it up as a higher priority, and if the InterBase client is the Developer edition that comes with RAD Studio, everything will work but in a standard RAD Server Developer configuration.

You can figure this out by looking at the first few lines in the log when RAD Server starts. If it is an "RSLite" configuration, the first few lines will look like this:

```
{"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"}}
```

If the code indicates that "Lite" is set to false, you might need to manually disable loading of **gds32.dll** (or its 64-bit version) InterBase client library, generally found in C:\Windows\SysWOW64 (if the InterBase client library cannot be found it loads the local **ibtogo.dll**.

Now, if you start RSLite (with the proper configuration) and there is no **emsserver.ini** file and no **emsserver.ib** database file, it will prompt you to create one. For this to work, RSLite must find the configuration in RAD Studio's Object Repository folder (ObjRepos under the product folder). The easier way to do this is to copy the files under Program Files (x86)\Embarcadero\Studio\<XX>.0\ObjRepos\en\ems in a folder with this relative path from **emsdevserver.exe**: "../ObjRepos/EMS". In other words, you need an ObjRepos folder at the same level directory as the folder containing your RSLite installation, the project deployment directory.

note

This is not needed for each RSLite deployment, only once to generate a production database, you can later copy on the target computers as is. In fact, the database created in a development environment is not compatible with RSLite deployment.

We recommend you specify as the target folder the same as your RSLite deployment so that the wizard will create an **emsserver.ini** file and an **emsserver.ib** database file in your deployment folder. Now grab the entire folder with RSLite, these configuration files, the runtime packages, and IBToGo, including the license, and you have all you need to deploy on a target Windows computer.

Proxy Configuration

It is not recommended to expose RSLite directly as a public web application due to its limitations in terms of protection and encryption. We recommend using a proxy layer, with a dedicated service or using one of the popular web services as a front end. In Apache, for example, in you configure a virtual host, enable HTTPS, and redirect the traffics to the RSLite instance with a configuration like the following:

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

For Linux

For Linux, you can follow similar steps as above, and everything should work as expected. As an alternative, you can also consider installing the full RAD Server and then adding IBToGo to the installation:

- Install the RAD Server using the **ems_install.sh** available in the RAD installation folder. See Here
- Copy the IBToGo files from the InterBase “redist” folder to the EMS folder on Linux (/usr/lib/ems)
- Execute the EMSDevServerCommand and follow the wizard to create the EMS database and configuration file

note

You might need to run the application via sudo to have the proper permissions