



RAD 서버 기술 가이드

```
localhost:8080/rad-server/details

1  {
2      "title": "RAD Server Technical Guide",
3      "edition": 2.1,
4      "authors": [
5          {
6              "name": "Antonio Zapater",
7              "revised": "2024-04-15"
8          },
9          {
10             "name": "David I",
11             "revised": "2019-04-05"
12         }
13     ],
14     "examplesURL": "github.com/embarcadero/radserver-docs",
15     "copyright": "©2019-2024"
16 }
```

머리말

RESTful 아키텍처는 최신 API 우선 애플리케이션 설계의 핵심 원동력이다. 이 책은 이러한 플랫폼 개발을 위한 RAD 스튜디오(델파이/C++빌더)에 포함된 RAD 서버 프레임워크 설명에 중점을 두었다.

RAD 서버는 모든 언어로 데스크탑, 모바일 및 웹 프론트 엔드 개발을 가능하게 하는 완전한 백엔드 MEAP(모바일 엔터프라이즈 애플리케이션 플랫폼)이며, 이 책은 개발자를 위한 최종 가이드로 설계됐다.

MEAP의 장점은 푸시 알림, 사용자 추적 및 분석과 같은 다양한 핵심 기능을 갖춘 사전 구축된 클라우드 또는 온프레미스 서버를 보유하고 있다는 점이다. 그리고 이를 통해 원격 데이터베이스 및 기능 액세스를 보다 빠르게 제공한다.

이 엠바카데로 RAD 서버 가이드는 원래 David I(2019)가 저술한 것으로, RAD 서버 출시 이후 시장 수요에 따라 추가된 많은 기능을 포함하여 Antonio Zapater(2023)에 의해 개정된 두 번째 에디션이다. 이 두 번째 에디션에서는 각 챕터를 지원하는 [포괄적인 동영상 시리즈](#)와 소스코드 예제가 깃허브에서 제공된다.
<https://github.com/embarcadero/radserver-docs>



동영상

이 책에 연결된 모든 동영상 시리즈는 [유튜브에서 확인할 수 있다](#). 또한 앞선 [깃허브 리포지토리](#)에서 모든 예제를 다운로드할 것을 적극 권장한다.

목차

01	
RAD 서버란 무엇인가?.....	7
도입.....	7
RAD 서버 개요.....	7
RAD 서버 기반 애플리케이션 구축 -7가지 핵심 방향.....	9
RAD 서버 애플리케이션 구축을 위한 요구 사항.....	9
RAD 스튜디오 IDE 사용.....	10
RAD 서버 테스트 및 배포 라이선스.....	10
핵심 RAD 서버 기능 요약.....	10
핵심 기능.....	10
참고 자료.....	12
02	
RAD 마법사를 사용하여	
"Hello World!" 만들기.....	13
REST 기반 서비스 구축.....	14
RAD 서버 프로젝트 마법사 사용.....	14
마법사 RAD 서버 프로젝트 및 소스 코드.....	18
첫 번째 애플리케이션에 대한 RAD 서버 구성.....	19
첫 번째 RAD 서버 애플리케이션 테스트.....	24
참고 자료.....	26
03	
첫 번째 CRUD	
애플리케이션 만들기.....	27
CRUD 기능을 사용하여 REST 기반 서비스 빌드.....	27
생성된 프로젝트 설명.....	30
프로젝트 빌드 및 테스트.....	32
TEMSDatasetResource의 추가 기능.....	33
04	
REST 디버거.....	36
REST 디버거란 무엇이며 어디에서 찾을 수 있는가.....	36
REST 디버거로 첫 번째 PUT 요청 보내기.....	37
REST 디버거에 포함된 기타 기능.....	39
05	
FireDAC 일괄 처리 및 JSONWriter 사용.....	40
메모리 스트리밍을 사용하여 JSON 데이터베이스의 데이터 반환하기.....	40
파이어닥의 BatchMove, BatchMoveDataSetReader 그리고 BatchMoveJSONWriter 사용하기.....	43
참고 자료.....	46

06

JSONValue, JSONWriter 및 JSONBuilder.....	47
JSON 데이터 처리를 위한 프레임워크.....	47
JSONValue 사용하기.....	48
JSON 클래스 사용 예제.....	49
JSONWriter 사용하기.....	51
JSONWriter 사용 예제.....	51
JSONBuilder 사용.....	53
참고 자료.....	54

07

사용자 지정.....	55
엔드포인트 만들기.....	55
모범 사례의 예.....	55
API의 잡은 통신을 제어하는 방법.....	55
하위 리소스 추가하기.....	56
응답에 중첩된 데이터 추가하기 (마스터/디테일).....	57
새 구현 테스트.....	62
사용자 정의 GET, POST, PUT, DELETE 메서드 구현.....	64
응답 오류 처리하기.....	66
참고 자료.....	66

08

내장된 분석도구에	
액세스.....	67
주요 특징.....	67
RAD 서버 콘솔에 액세스.....	68

09

RAD 서버 배포.....	71
RAD 서버를 배포할 수 있는 위치.....	71
겟잇의 설치 프로그램 사용.....	71
RAD 서버를 수동으로 배포하기 위한 전제 조건.....	72
Windows에 수동으로 배포.....	73
인터넷 서비스 서버 엔진.....	73
RAD 서버 설치.....	74
웹 서버(IIS 또는 Apache).....	77
리눅스에서 수동으로 배포.....	77
호환되는 배포판.....	77
인터넷 서비스 서버 엔진 설치.....	77
인터넷 서비스 서버 등록 및 시작.....	78

인터넷 서비스 실행.....	78
RAD 서버 설치.....	79
아파치용 RAD 서버 설정.....	80
도커에 배포.....	81
옵션 1: PA-RADServer-IB.....	82
옵션 2: PA-RADServer.....	82
RAD 스튜디오로 컴파일된 RAD 서버 모듈 복사.....	83
EMSServer.ini 파일 구성.....	84
10	
RAD 서버 라이트.....	85
Lite 버전이란?.....	85
RAD 서버 라이트 라이선스 취득 방법.....	86
RAD 서버 라이트 프로젝트 배포.....	86
배포할 파일들.....	87
수동으로 배포.....	87
배포 마법사 사용.....	87
MSVS 런타임.....	88
프로덕션 데이터베이스 만들기.....	88
프록시 구성.....	89
리눅스의 경우.....	89
11	
인증 및 권한부여.....	90
내장된 인증: 사용자 및 그룹 관리하기.....	90
로그인 하기.....	92
로그아웃 하기.....	93
회원가입 하기.....	94
그룹(Group) 관리하기.....	95
내장된 권한부여(Integrated authorization).....	95
전역 자격 증명(Global Credentials).....	95
사용자들 그리고 그룹 권한부여.....	96
사용자 정의 인증 (Custom authentication).....	97
사용자 정의 권한부여 (Custom authorization).....	100
RAD 서버 관리 콘솔.....	101
새 프로파일 생성하기.....	102
사용자들과 그룹들을 다루기.....	102
RSConsole 안으로 더 깊이 들어가기.....	104
12	
여러분의 엔드포인트들에 대한 문서화와 테스트를 위해	
OpenAPI (Swagger) 사용하기.....	105
OpenAPI/Swagger란 무엇인가 그리고 우리는 왜 이것을 사용하는가?.....	105

Swagger UI를 RAD 서버 안에 심어 넣기(embedding).....	105
사용자 지정 도움말 문서를 만들기.....	107
예시.....	107
EndPointRequestSummary.....	108
EndPointRequestParameter.....	109
EndPointResponseDetails.....	110
EndPointObjectsDefinitions.....	111
EMSDatasetResource에 애트리뷰트들을 정의하기.....	112
13 파일 관리 및 스토리지.....	114
TEMSSFileResource.....	114
예시.....	115
코드에서 파일을 다루기.....	116
Content-Type HTTP 헤더들.....	117
간단한 예시.....	117

01

RAD 서버란 무엇인가? 도입



오늘날의 컴퓨팅 환경은 더 이상 데스크탑, 디바이스, 서버 또는 데이터 센터에 국한되지 않는다. 애플리케이션은 데스크탑에서 여러 디바이스, 네트워크 에지 연결, 온프레미스, 퍼블릭 및 하이브리드 클라우드 서비스로 이동하고 있다. RAD 서버와 RAD 스튜디오를 사용하면 회사(및 고객)의 광범위한 컴퓨팅 요구 사항과 비즈니스 요구 사항을 충족하는 솔루션을 구축할 수 있다.

이 설명서에서는 RAD 스튜디오, 델파이, C++빌더 엔터프라이즈 및 아키텍트 에디션에서 제공되는 RAD 서버의 REST 기반 API 호스팅 엔진, 구성 요소 및 기술을 사용하여 서비스 기반 다계층 애플리케이션을 신속하게 설계, 구축, 디버그 및 배포하는 방법을 설명한다.

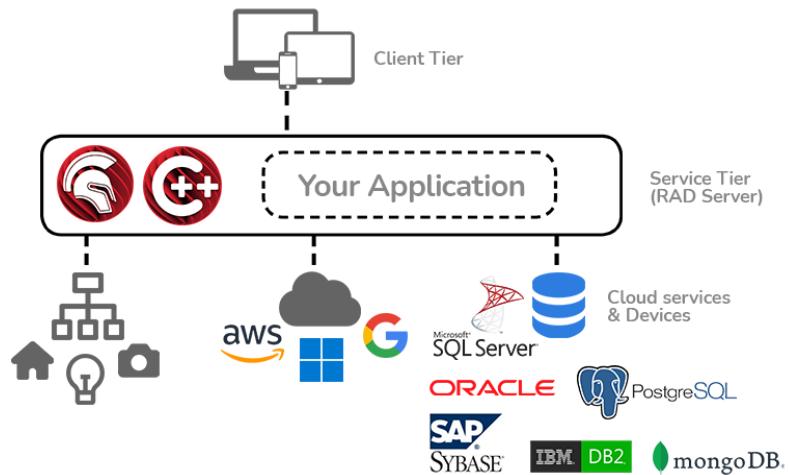


참고

RAD 서버 설명서 및 소스 코드 전체에서 EMS(엔터프라이즈 모빌리티 서비스)에 대한 참조를 볼 수 있다. EMS는 현재 RAD 서버의 원래 이름이었다.

RAD 서버 개요

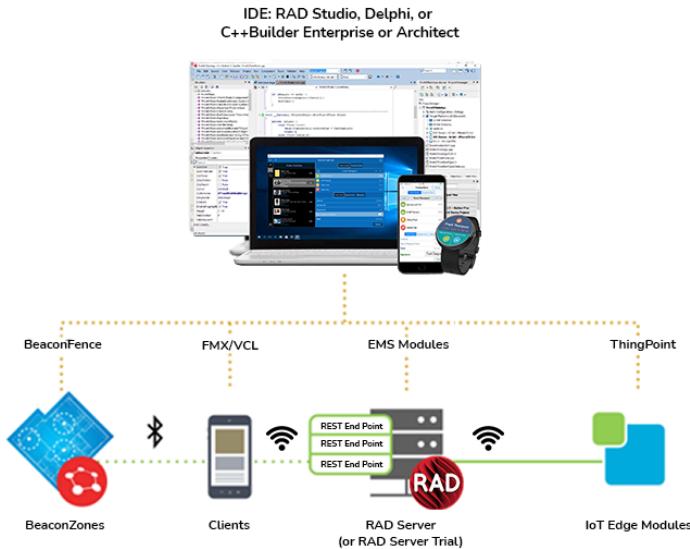
엠바카데로 RAD 서버는 델파이 및 C++빌더를 사용하여 서비스 기반 애플리케이션을 빠르게 빌드하고 배포할 수 있는 터키 방식의 백엔드 솔루션이다. RAD 서버는 REST(Representational State Transfer) 프로토콜을 지원하며, JSON(또는 XML) 매개변수를 통해 전달과 반환을 수행한다. API들을 게시하고, RAD 서버에 연결된 사용자들과 장비들을 관리하고, 앱 사용에 대한 분석을 획득하고, FireDAC 컴포넌트를 사용하여 로컬 및 엔터프라이즈 데이터베이스에 연결한다. 또한 RAD 서버는 사용자 인증, 푸시 알림, 위치 파악, 데이터 저장소도 지원한다.



REST 엔드포인트 및 위치 추적 개발 및 테스트

RAD 서버에는 마법사들, 컴포넌트들, 도구들이 들어 있으므로, 여러분은 새 미들웨어와 백-엔드 애플리케이션을 빠르게 개발할 수 있다. 또한 여러분의 델파이 및 C++빌더로 만든 기존 클라이언트/서버 애플리케이션을 RAD 서버 기반 애플리케이션으로 이전하고 서버 또는 클라우드에서 운영할 수도 있다. 여러분의 엔드포인트들을 REST 호출용으로 공개한다. 따라서, 데스크탑, 모바일, 콘솔, 웹 및 기타 다양한 유형의 애플리케이션들이 호출할 수 있다. RAD 서버에는 서비스 애플리케이션을 구축하는 데 필요한 도구들, 컴포넌트들, 데이터베이스 연결, 인터페이스들이 모두 포함되어 있다. 따라서 여러분이 믿고 의지할 수 있다.

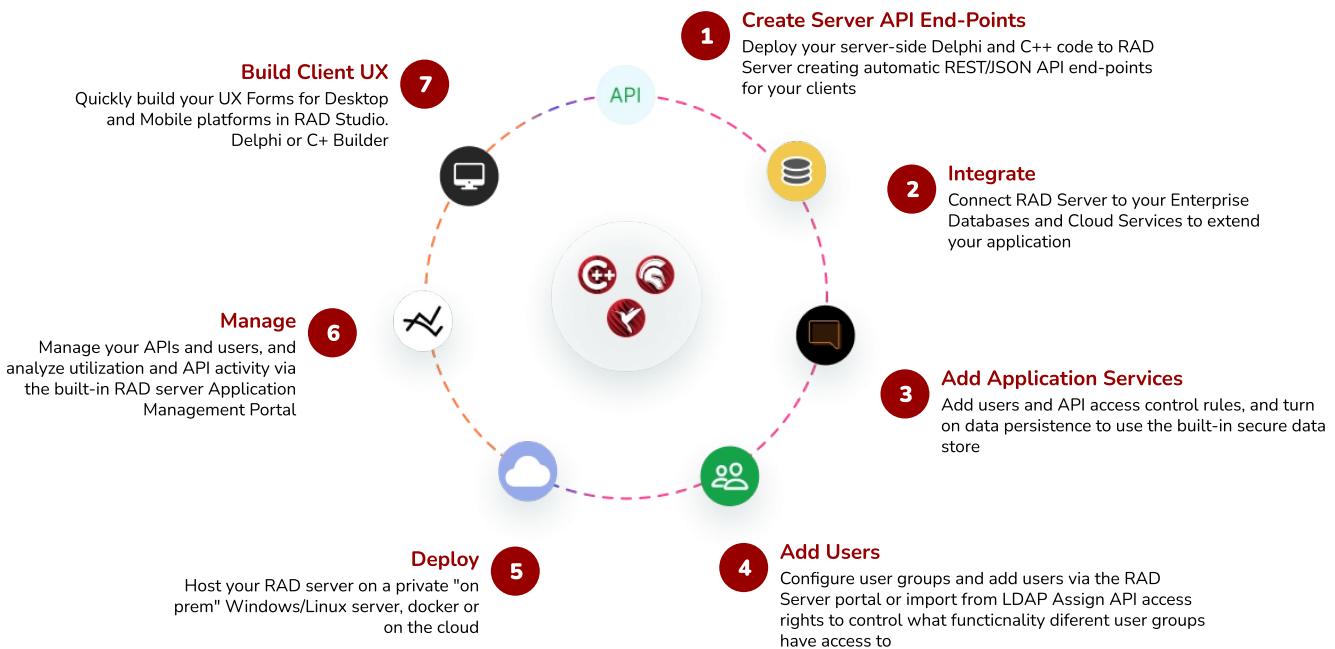
RAD 서버 애플리케이션은 마이크로소프트 윈도우 IIS 및 아파치 웹 서버 위에 배포할 수 있으며, 델파이 기반 서비스를 리눅스 인텔 64비트 서버에 배포할 수 있다. 리눅스용 C++빌더 지원에 대한 자세한 내용은 엠바카데로 RAD 스튜디오 블로그의 업데이트를 계속 지켜봐 주길 바란다.



REST 엔드포인트, 위치 추적 및 IoT 에지웨어 개발 및 테스트

RAD 서버 기반 애플리케이션 구축 -7가지 핵심 방향

아래 다이어그램은 RAD 서버 기반 애플리케이션을 빌드하기 위해 개발자들에게 7가지 방향과 개발 단계를 안내한다.



간편한 멀티-티어 개발

먼저, 서버 REST/JSON API 기반 엔드포인트를 만든다(필요하다면 JSON 대신 XML을 사용할 수도 있다). 그런 다음 다양한 데이터베이스, 클라우드 서비스 및 기타 기술을 통합하여 엔드포인트를 확장할 수 있다.

당신은 사용자에게 더 많은 애플리케이션 엔드포인트를 추가하고 API 액세스 제어 규칙을 만들 수 있다. 또한 RAD 서버의 기본 제공 보안 데이터 저장소를 활용하여 영구 데이터를 추적하는 코드를 작성할 수 있다. 그리고 콘솔 포털을 통해 사용자를 가져오고 인증할 수 있다.

애플리케이션을 개발하고 디버깅한 후에는 사설 온프레미스 윈도우 및 리눅스 서버에서 RAD 서버 애플리케이션을 호스팅할 수 있다. 또한 애플리케이션을 아마존 AWS, 마이크로소프트 Azure, 구글 및 기타 클라우드 제공업체와 같이 클라우드 시스템으로 마이그레이션할 수도 있다.

애플리케이션이 제품으로 배치된 후에도 기본으로 제공되는 애플리케이션 관리 인터페이스를 통해 API에 대한 액세스를 관리하고, 사용자 액세스를 제어하며, 엔드포인트 API 활동의 활용도를 분석할 수 있다. 마지막으로 데스크톱, 모바일, 웹, 콘솔 및 RAD 스튜디오에서 지원하는 다른 애플리케이션 유형을 빌드할 수 있다. 또한, Sencha의 Ext JS 구성 요소 세트를 사용하여 최신 웹 클라이언트 애플리케이션을 빌드하고 다른 도구 및 프로그래밍 언어를 사용하여 RAD 서버 애플리케이션의 REST/JSON 기능을 지원하는 클라이언트 애플리케이션을 빌드할 수 있다.

RAD 서버 애플리케이션 구축을 위한 요구 사항

다음 섹션에는 RAD 서버 애플리케이션을 빌드, 테스트 및 배포하기 위한 제품 및 기술 요구 사항이 포함되어 있다. 별도의 언급이 없는 한, "RAD 스튜디오" 및 IDE는 RAD 스튜디오. 델파이 및 C++빌더 제품에 적용된다.

RAD 스튜디오 IDE 사용

RAD 서버 애플리케이션을 빌드하려면 상용 라이선스가 포함된 RAD 스튜디오 엔터프라이즈 또는 아키텍트 에디션이 필요하다. RAD 스튜디오 엔터프라이즈 평가판은 개발 및 테스트를 위해 30일 동안 사용할 수 있다. 평가판의 경우 제품 서버로의 배포를 지원하지 않는다.

RAD 서버 테스트 및 배포 라이선스

30일간 무료로 제공되는 RAD 스튜디오 평가판에는 RAD 서버 5인 사용자 개발 체험판이 포함되어 있다. RAD 서버 배포 라이선스는 RAD 스튜디오의 엔터프라이즈 및 아키텍트 상용 에디션에 포함되어 있다. RAD 스튜디오 엔터프라이즈에는 RAD 서버용 단일 사이트 배포 라이선스가 포함된 반면, RAD 스튜디오 아키텍트 에디션에는 업데이트 서브스크립션이 활성화된 고객을 위한 멀티 사이트 배포 라이선스가 포함되어 있다. RAD 스튜디오 알렉산드리아부터 엔터프라이즈와 아키텍트에는 멀티 사이트 환경에 RAD 서버 라이트를 배포할 수 있는 옵션이 포함되어 있다.

프로덕션 환경에 애플리케이션을 배포하려면 RAD 서버에 인터베이스로 암호화된 데이터베이스가 필요하다. 이 버전의 인터베이스를 설치하려면 유효한 RAD 서버 라이선스를 사용해야 한다.



팁 인터베이스를 사용하여 애플리케이션을 배포하려는 경우 애플리케이션용과 RAD 서버 용의 두 가지 인터베이스의 인스턴스가 실행 중이어야 한다.

핵심 RAD 서버 기능 요약

RAD 서버는 개발자에게 REST 기반 서비스 애플리케이션을 구축하기 위한 다양한 기능을 제공한다. RAD 서버(과거 EMS)는 RAD 스튜디오 버전 XE7에서 처음 소개되었다. 첫 번째 출시 이후 개발자의 요구 사항을 해결하고 새로운 플랫폼, 아키텍처 및 기술에 대한 지원을 추가하기 위해 개선 사항과 새로운 기능이 추가되었다.

핵심 기능

다음은 서비스 기반 애플리케이션을 구축할 때 활용할 수 있는 RAD 서버의 핵심 기능 목록이다.

- **REST 엔드포인트 퍼블리싱** - RAD 서버는 애플리케이션 백엔드 API 및 서비스를 위한 터미널 방식의 솔루션을 제공한다. RAD 서버는 비즈니스 로직 게시를 위한 사용하기 쉬운 API를 제공한다. 델파이 또는 C++ 코드를 API로 호스팅하고 RAD 서버에서 측정 및 관리되는 REST/JSON 엔드포인트로 자동으로 게시할 수 있다. 엔드포인트 게시 기능에는 다음이 포함된다:
 - **액세스 제어** - 인증을 통해 모든 애플리케이션 API에 대한 그룹 및 사용자 수준 액세스를 설정하고 애플리케이션의 API 기능에 액세스할 수 있는 사용자를 제어할 수 있다. 사용자 그룹을 직접 만들거나 LDAP 인프라에서 자동으로 가져올 수 있다.
 - **API 분석** - 강력한 통계 추적 및 분석을 위해 모든 REST API 엔드포인트 활동이 기록되고 측정된다. 사용자, API 및 서비스 활동을 일/월/연간으로 분석하여 애플리케이션이 어떻게 활용되고 있는지에 대한 통찰력을 얻을 수 있다. 모든 리소스 또는 특정 그룹, 사용자, 디바이스 설치 등을 기준으로 활동을 필터링할 수도 있다. 다른 도구를 이용한 추가 분석을 위해 기록된 분석을 CSV 파일로 내보낼 수 있다.

- 데스크탑, 모바일 및 웹 클라이언트 애플리케이션 - RAD 서버에서 호스팅 되는 모든 C++ 및 델파이 코드는 여러 플랫폼의 모든 유형의 클라이언트 애플리케이션에서 사용할 수 있는 REST/JSON 엔드포인트로 게시되어 뛰어난 유연성과 미래 보장성을 제공한다.
- **통합 미들웨어** - RAD 서버는 외부 서버, 애플리케이션, 데이터베이스, 스마트 디바이스, 클라우드 서비스 및 기타 플랫폼에 대한 연결을 통해 다양한 통합 기능을 즉시 제공한다. 통합 기능은 다음과 같다:
 - 엔터프라이즈 데이터 - RAD 서버는 모든 인기 있는 엔터프라이즈 RDBMS 서버에 대한 고성능의 내장된 연결을 제공한다. 데이터베이스 연결은 다양한 소스의 데이터에 쉽게 연결할 수 있도록 FireDAC 구성 요소 및 라이브러리 세트를 사용한다.
 - 클라우드 서비스 - RAD 서버를 사용하면 구글, 아마존, 페이스북, Kinvey 등과 같은 다양한 클라우드, 소셜 및 BaaS 플랫폼의 REST 클라우드 서비스를 쉽게 통합할 수 있다.
- **애플리케이션 서비스** - RAD 서버에는 애플리케이션을 강화하기 위해 바로 사용할 수 있는 기본 제공 서비스 모음이 포함되어 있다. RAD 서버에는 사용자 딕렉토리 서비스 및 사용자 관리, 푸시 알림, 사용자 위치 추적, 내장 데이터 저장소와 같은 핵심 기능이 포함되어 있다. 이러한 애플리케이션 및 디바이스 서비스 중 일부는 다음과 같다:
 - 푸시 알림 - RAD 서버를 사용하여 애플리케이션 사용자와 해당 디바이스에 프로그래밍 방식 또는 온디맨드 알림을 보낼 수 있다. RAD 서버는 현재 애플의 푸시 알림 서비스(APN) 및 구글 파이어 베이스 클라우드 메시징(FCM)을 포함한 푸시 알림 시스템을 지원한다. 사용자 지정 코드를 작성하여 다른 푸시 알림 시스템과 연결할 수도 있다.
 - 내장된 보안 데이터스토어 - RAD 서버가 인터베이스 서버의 암호화된 데이터스토어 보안을 지원하므로 별도의 데이터베이스 서버 없이도 내장된 API를 사용하여 JSON 데이터를 저장하고 검색할 수 있다.
 - 사용자/그룹 관리 - RAD 서버 API를 사용하여 사용자, 사용자 그룹을 생성 및 관리하고 RAD 서버 콘솔(RSConsole.exe)을 통해 액세스를 제어할 수 있다. ActiveDirectory(LDAP)를 통합하거나 사용자 지정 인증 미들웨어를 개발할 수 있다.
 - 사용자 위치/접근성 - RAD 서버 애플리케이션은 GPS, 비콘, 및 비콘 펜스 기술에 대한 RAD 스튜디오의 지원을 활용할 수 있다. RAD 서버 애플리케이션은 실내와 실외 어디에서나 사용자의 움직임을 추적하고 사용자가 사용자 지정 비콘 구역에 출입하거나 지정된 비콘 지점에 접근하면 근접 이벤트에 응답할 수 있다.
 - 정적 파일 공급자 - URL을 폴더에 매핑하고 HTML, JS, CSS, 이미지 등과 같은 파일 컨텐츠를 반환한다. 이 기능은 소규모 배포(예: RAD 서버 라이트 사용) 또는 개발 환경에서 매우 유용하다.
- **API 문서** - 어트리뷰트와 내장된 Swagger OpenAPI 통합을 사용하여 API에 대한 문서를 쉽게 작성할 수 있다. Swagger UI를 RAD 서버 자체에 임베드하거나 자동 생성된 YAML 및 JSON 파일을 통해 원격 인스턴스에서 구성할 수 있다.
- **손쉬운 배포** - RAD 서버는 개발, 배포 및 운영이 간편하여 재구축할 수 있는 솔루션을 구축하는 ISV 및 OEM에 이상적이다. 윈도우, 리눅스 또는 도커에 배포할 수 있다.

참고 자료

RAD 스튜디오 설치 및 RAD 서버 기반 애플리케이션 배포에 대한 최신 업데이트 정보는 다음 엠바카데로 온라인 링크를 참조하길 바란다.

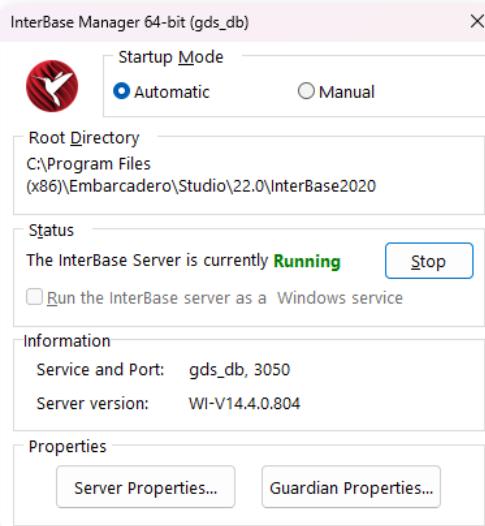
- [RAD 서버 제품 개요](#)
- [RAD 스튜디오 설치 노트](#)
- [RAD 스튜디오 및 RAD 서버 지원 대상 플랫폼](#)
- [프로덕션 환경을 위한 RAD 서버 데이터베이스 요구 사항](#)
- [RAD 스튜디오의 플랫폼 상태 페이지](#)
- [인터베이스](#)
- [FireDAC](#)
- [FireDAC 지원 데이터베이스](#)
- [RAD 스튜디오 엔터프라이즈 모빌리티 서비스](#)
- [RAD 스튜디오 제품 기능 매트릭스\(PDF - RAD 서버 섹션 확인\)](#)
- [Swagger 오픈 API](#)
- [EMS 푸시 알림](#)
- [애플 푸시 알림 서비스\(APN\)](#)
- [파이어베이스 클라우드 메시징\(FCM\)](#)

02

RAD 마법사를 사용하여 “Hello World!” 만들기



이제 본격적인 프로그래밍을 시작한다. 이 장에서는 RAD 서버 기반 서비스 애플리케이션을 델파이와 C++ 빌더를 사용하여 빌드하는 방법을 배운다. 시작하기 전, 인터베이스 데이터베이스 서버가 실행 중인지 확인해야 한다. 이는 RAD 서버가 인터베이스 데이터베이스를 사용하여 사용자 정보, 사용자 그룹, 분석, 등록된 디바이스, 버전 정보, 등록된 에지 모듈, 푸시 알림 메시지 등을 저장하기 때문이다.

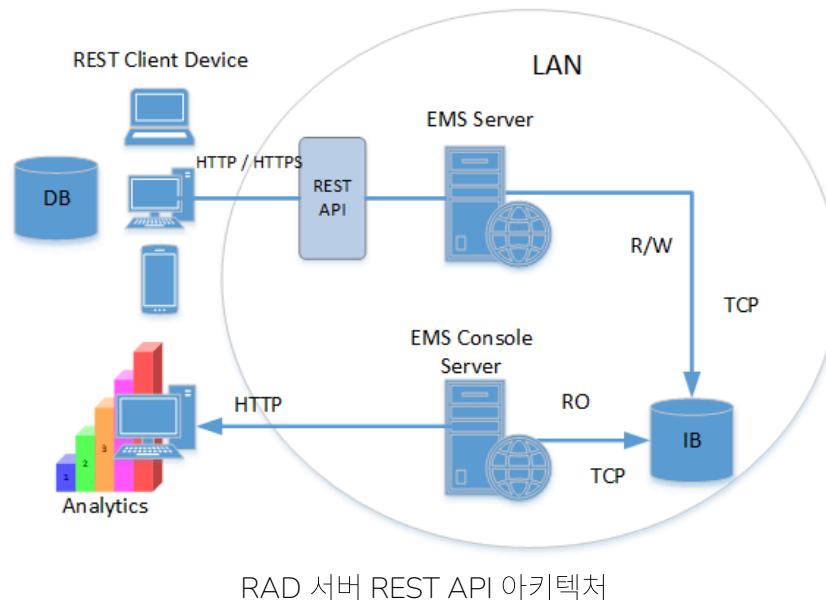


인터베이스 2020 서버 관리자

REST 기반 서비스 구축

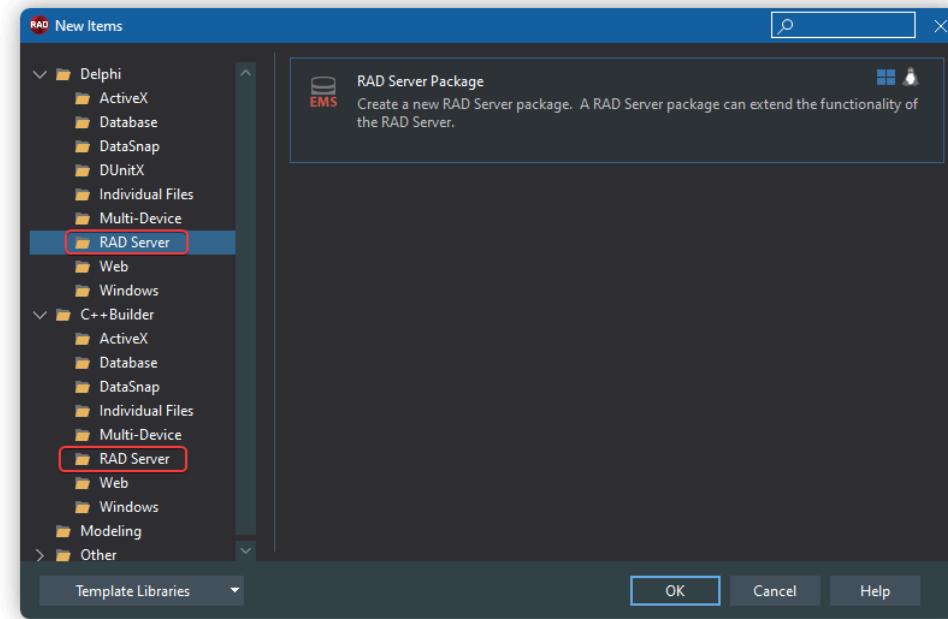
RAD 서버에는 EMS(엔터프라이즈 모빌리티 서비스)가 포함되어 있다. 또한 클라우드 및 온프레미스에서 호스팅할 수 있는 MEAP(모바일 엔터프라이즈 애플리케이션 플랫폼)를 제공한다. 개발자들은 RAD 서버를 사용하여 사용자 지정 REST API를 공개할 수 있다. 또한 FIndDAC 데이터 액세스 라이브러리 및 구성 요소를 사용하여 엔터프라이즈 데이터베이스의 데이터에 액세스할 수 있다.

RAD 서버는 개발자에게 원격 데이터베이스 액세스, 사용자 추적, 기기 애플리케이션 관리, 사용 분석 등을 포함하는 종합적인 솔루션을 제공한다. 다른 솔루션에 비해 RAD 서버에는 사용자 지정 패키지의 통합을 지원하는 사전 구축된 애플리케이션 서버가 포함되어 있다. 이러한 사용자 지정 패키지는 데이터 집합, 비즈니스 로직 및 기타 REST 기반 리소스를 공개할 수 있다. 또한 컴포넌트들은 모바일, 웹 및 데스크톱 애플리케이션 코드에서 RAD 서버 리소스에 액세스하기 위해 사용할 수 있다.



RAD 서버 프로젝트 마법사 사용

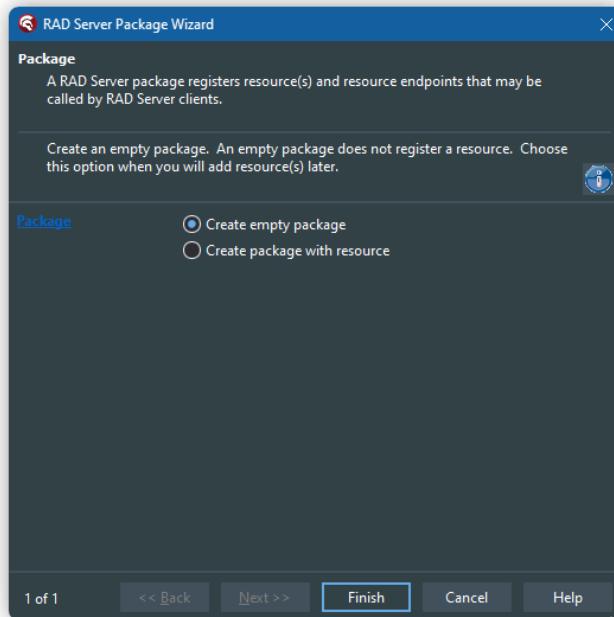
서버를 가장 빠르게 시작하는 방법은 새 프로젝트 메뉴(파일 | 새로 만들기 | 기타...)를 사용하여 델파이 또는 C++빌더용 RAD 서버 | RAD 서버 패키지 마법사를 선택하는 것이다.



델파이 및 C++용 RAD 서버 프로젝트 마법사 선택 사항

RAD 서버 패키지 프로젝트를 선택한다. 시작 프로젝트의 생성을 돋는 마법사가 나타난다. 첫 페이지에서 마법사가 RAD 서버 애플리케이션을 표시할 리소스 및 엔드포인트를 생성하는 방법을 선택한다. RAD 서버 패키지 마법사에는 두 가지 선택 사항이 제공된다.

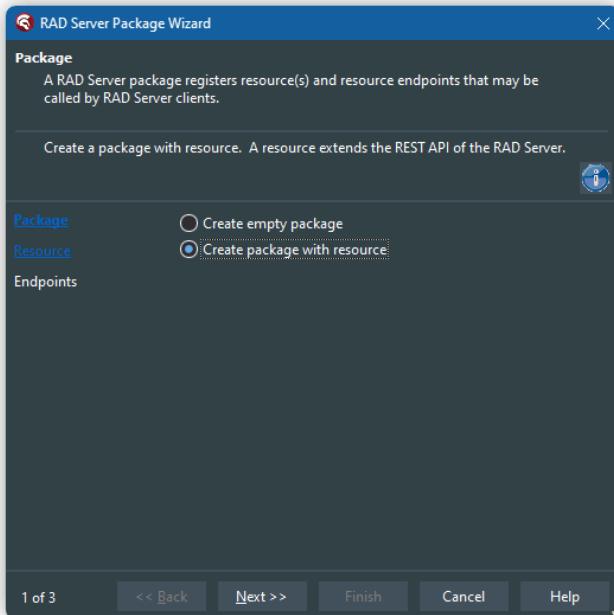
선택 1: 리소스를 등록하지 않은 빈 패키지를 만든다. 리소스를 나중에 추가하려는 경우 이 옵션을 선택한다. 이 옵션을 선택하면 시작 메인 라이브러리가 포함된 패키지 프로젝트가 생성된다.



빈 RAD 서버 패키지 만들기

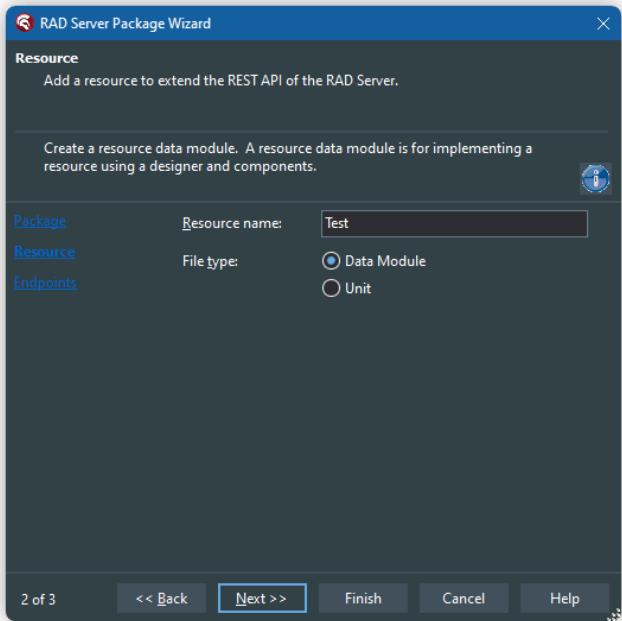
“마침” 버튼을 클릭하면 추가 개발 작업을 위한 시작 프로젝트가 생성된다. 여기에는 완성된 RAD 서버 애플리케이션을 만드는 작업이 포함된다.

선택 2: RAD 서버용 REST API를 확장하는 리소스가 포함된 패키지 만들기. 다음 버튼을 클릭하면 두 개의 추가적인 마법사 단계가 표시되며, 패키지 프로젝트, 리소스 및 엔드포인트를 만드는데 도움을 준다. 첫 번째 RAD 서버 프로젝트를 빌드하려면 이 옵션을 선택한다.



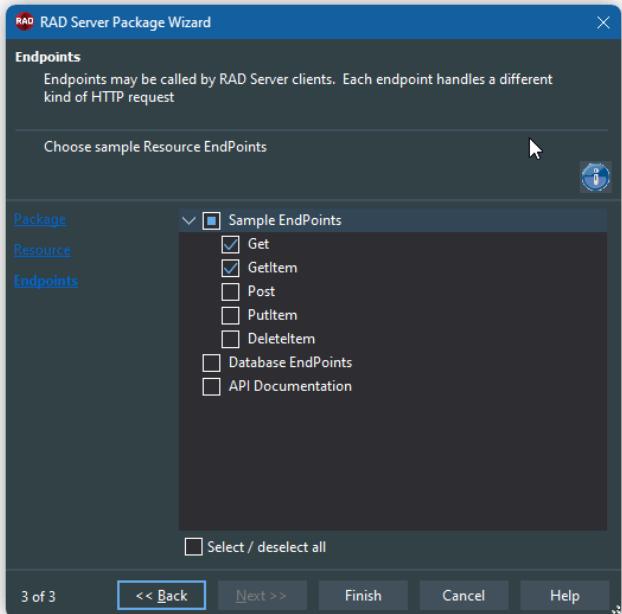
리소스 기반 RAD 서버 패키지 만들기

설치 마법사의 두 번째 페이지에서 리소스 이름을 "Test"로 설정한다. 파일 유형 라디오 버튼에는 두 가지 옵션이 표시 된다: 1) 코드에서 리소스를 구현하기 위한 단위를 만들고, 2) IDE의 디자이너, 컴포넌트 및 코드 편집기를 사용하여 리소스를 구현하기 위한 데이터 모듈을 만든다. 이 첫 번째 RAD 서버 애플리케이션에서는 데이터 모듈을 사용할 것이다.



RAD 서버 패키지 마법사 2페이지 - 리소스 이름 및 파일 유형 설정

다음 버튼을 클릭하여 시작 엔드포인트 집합을 생성한다.



RAD 서버 패키지 마법사 3페이지 - 시작 엔드포인트 선택

위 그림에서 볼 수 있는 것처럼, 마법사 세 번째 페이지에서 제안된 엔드포인트를 그대로 둔다: Get(REST GET) 및 GetItem(URL 끝에 가져올 항목을 식별하는 세그먼트가 있는 REST GET)을 선택하고 "API 문서"를 선택 취소한다. 시작 프로젝트를 만들려면 마침 버튼을 클릭한다.



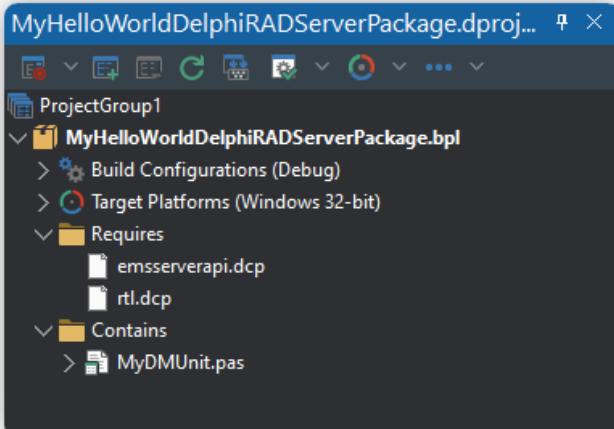
마법사에는 두 가지 추가 옵션이 있다: 데이터베이스 엔드포인트(FlatDAC)를 사용하여 데이터베이스를 엔드포인트에 연결)와 API 문서(Swagger OpenAPI)이다. 이에 관해서는 다음 장에서 더 자세히 설명하겠다.

마법사가 완료되면 IDE로 돌아온다. 가장 먼저 해야 할 일은 프로젝트를 저장하는 것이다. C++ 및 델파이 프로젝트 및 패키지의 경우 "MyDMUnit"이라는 이름을 사용한다. C++ 프로젝트 및 패키지의 경우 "MyHelloWorldCppRADServerPackage"라는 이름을 사용한다. 델파이 프로젝트 및 패키지의 경우 "MyHelloWorldDelphiRADServerPackage"라는 이름을 사용한다.

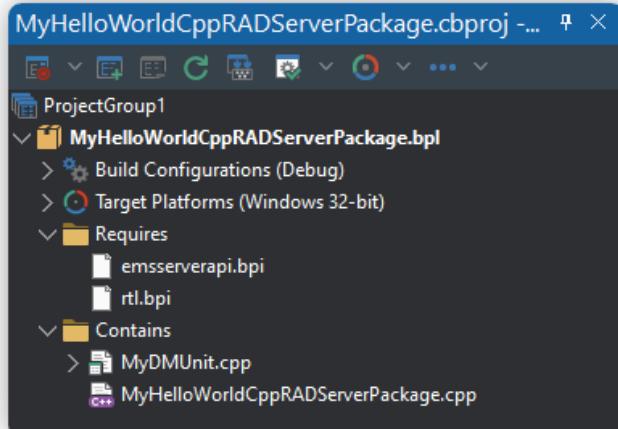
마법사 RAD 서버 프로젝트 및 소스 코드

생성된 프로젝트는 매우 로우 코드이며 각 엔드포인트에 연결된 메서드 몇 개만 있다. RAD 스튜디오는 기본 코드를 자동으로 채워 넣어주며, 우리는 이를 조정하여 좀 더 "Hello World" 같은 코드로 만들 것이다.

델파이와 C++의 프로젝트는 다음과 같다. 생성된 데이터 모듈은 컴포넌트 없이 비어 있을 것이다. 스크린 캡처를 통해 자동 생성된 샘플 코드에서 수정된 내용을 확인할 수 있다.



생성된 델파이 프로젝트



생성된 C++ 프로젝트



이 문서에 사용된 모든 소스 코드와 샘플은 [깃허브](#)에 호스팅 되어 있으며 챕터별로 나누어 있다. 더 나은 이해를 위해 전체 리퍼지토리를 다운로드하는 것을 권장한다.

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:

```

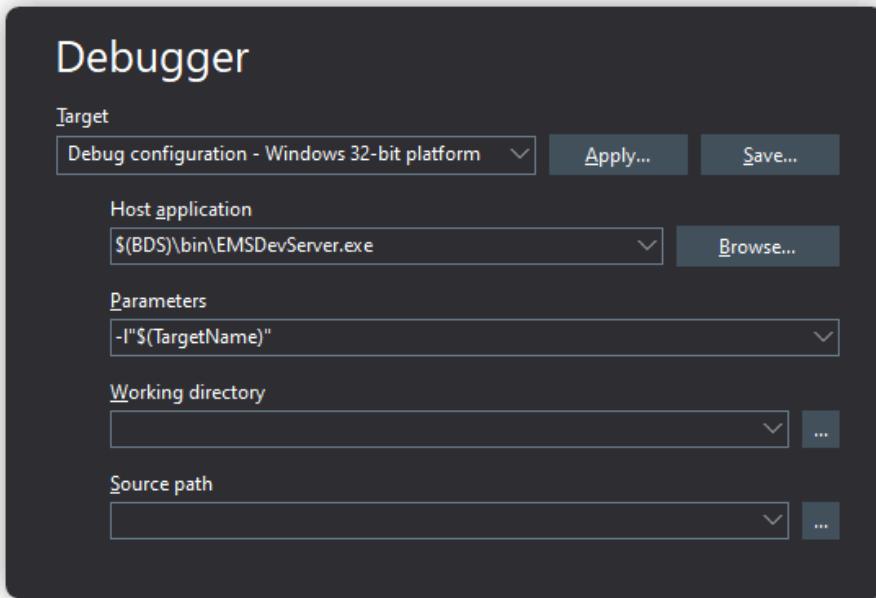
void TTestResource1::Get(TEndpointContext* Acontext,
                        TEndpointRequest* ARequest, TEndpointResponse* AResponse)
{
  AResponse->Body->SetValue(new TJSONString("Hello World"), True);

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);
}

```

첫 번째 애플리케이션에 대한 RAD 서버 구성

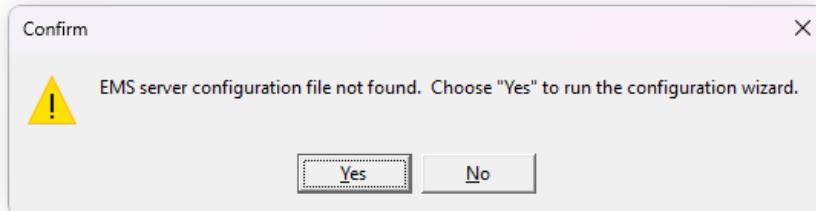
이제 마법사를 사용하여 첫 번째 RAD 서버 애플리케이션을 구축했으므로 IDE를 사용하여 애플리케이션을 컴파일하고 테스트할 수 있다. IDE는 EMSDevServer를 호스트 실행파일(EMSDriver.exe)로 사용, 로드할 패키지 파일을 파라미터로 하여 실행한다. Win32 및 Win64 개발용 EMSDevServer에는 두 가지 버전이 있다(\${BDS}\bin\EMSDriver.exe와 \${BDS}\bin64\EMSDriver.exe이다). 이는 모두 IDE에서 자동으로 구성된다.



실행 | 파라미터... 대화 상자에 EMSDevServer.exe가 호스트 애플리케이션으로 표시됨

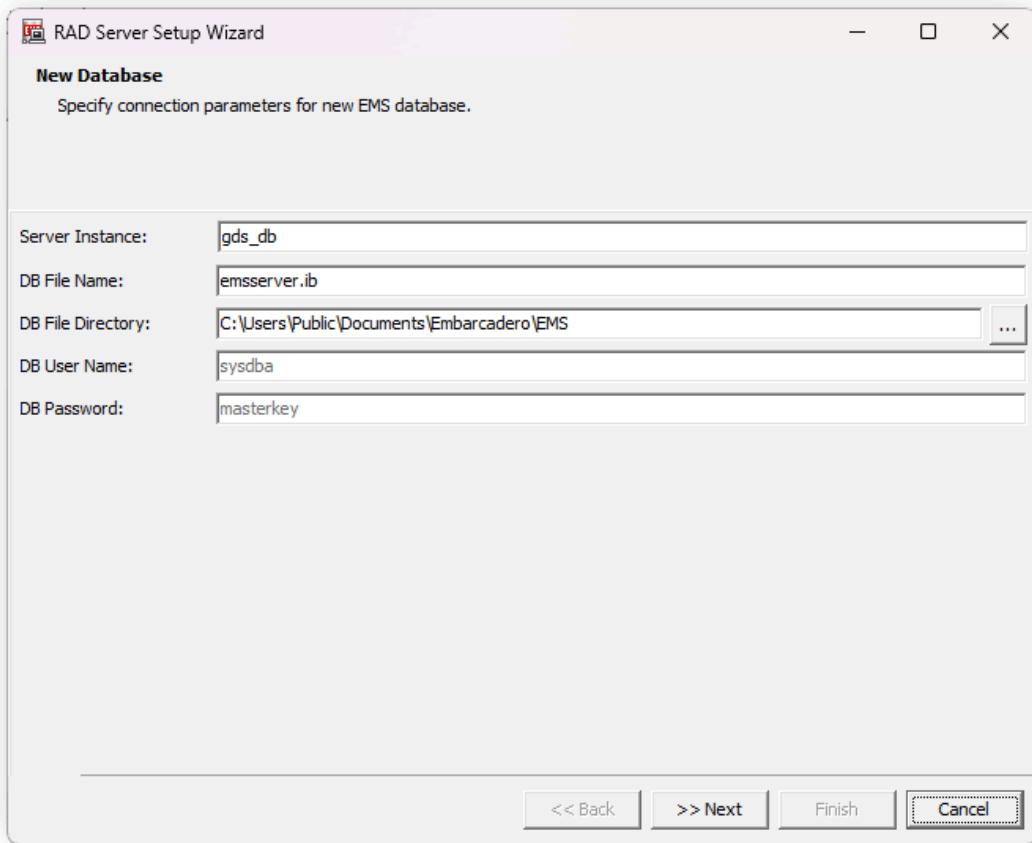
RAD 스튜디오에는 EMSDevConsole 서버를 시작하고 EMS 콘솔 서버 창을 여는 EMSDevConsole.exe도 포함되어 있다. EMS 콘솔은 RAD 서버 애플리케이션을 위한 웹 애플리케이션을 제공하여 분석을 표시하고 사용자/그룹 관리 등의 기능을 제공한다. 이 콘솔은 다음 장에서 더 자세히 설명한다.

실행 | 실행 메뉴 항목을 선택하거나 F9키를 누르면, 시작 애플리케이션을 컴파일, 링크 및 실행한다. RAD 서버 개발 서버는 기본적으로 TCP 포트 8080을 사용하여 시작된다. RAD 서버 애플리케이션을 처음 실행하는 경우 RAD 서버 구성 파일인 emsserver.ini를 찾을 수 없다는 대화 상자가 나타난다. 이 문제는 RAD 서버 레지스트리 키가 없거나 구성 파일이 존재하지 않을 때 발생한다.



구성 파일 없이 RAD 서버 구성 마법사를 시작하려고 시도할 경우 위와 같은 창이 나온다.

예 버튼을 클릭하여 RAD 서버 구성 마법사를 실행한다.



설정 마법사 1페이지 - 새 EMS 데이터베이스 연결 파라미터 지정

첫 번째 마법사 단계에서 인터베이스 서버 인스턴스를 입력한다(기본적으로 RAD 스튜디오의 개발 버전의 인터베이스 서버는 gds_db를 사용한다). 이 마법사 페이지에는 RAD 서버 데이터베이스의 이름(emsserver.ib)과 데이터베이스 및 구성 파일이 포함될 디렉토리도 포함되어 있다.

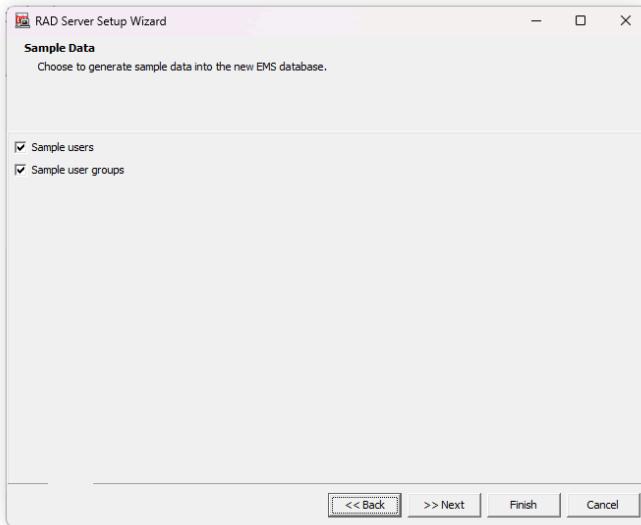


경고

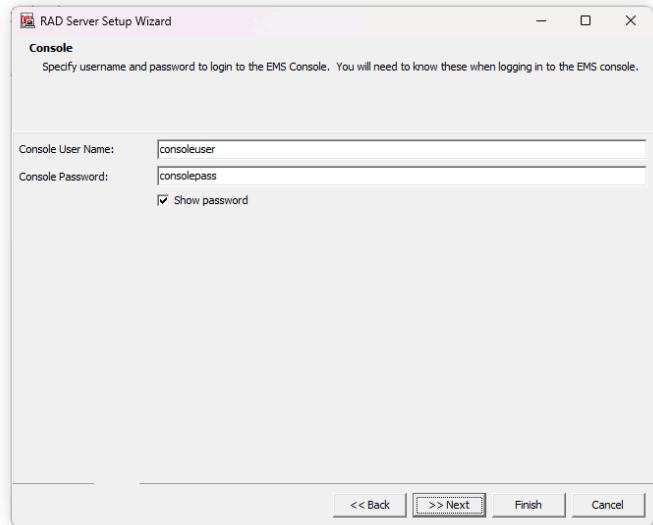
이전에 컴퓨터에 다른 서버의 인스턴스 이름을 사용하여 인터베이스를 설치한 경우 해당 이름 문자열을 입력해야 한다.

다음 버튼을 클릭하여 사용자 및 사용자 그룹에 대해 RAD 서버 데이터베이스에 샘플 데이터를 만들지 여부를 마법사에게 알린다. 개발 및 테스트를 위해 두 확인란을 모두 설정한다.

다음 버튼을 클릭하여 RAD 서버 콘솔에 로그인하기 위한 사용자 이름과 패스워드를 설정한다.

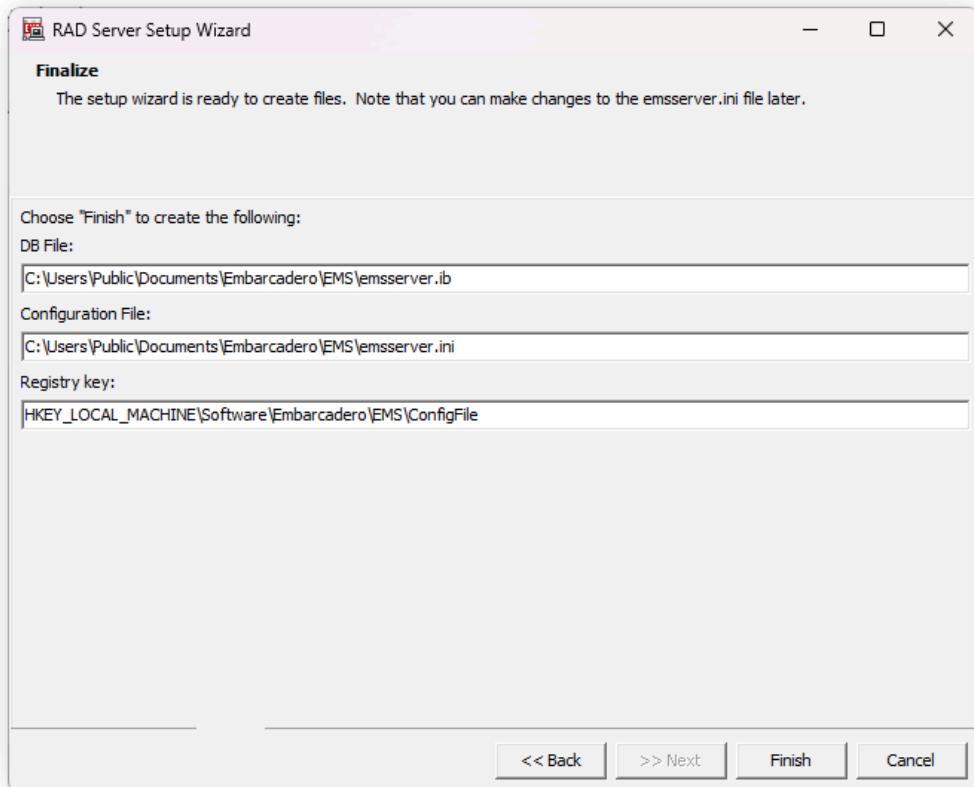


설정 마법사 2페이지 - 샘플 사용자 및 그룹



설정 마법사 3페이지 - 콘솔 사용자 이름 및 패스워드 선택

그러면 마지막으로 다음 버튼을 클릭하여 마지막 마법사 단계로 이동한다. 마법사가 RAD 서버 데이터베이스 파일, 구성 파일을 만들고 현재 로그인한 사용자에 대한 윈도우 레지스터 키를 설정할 준비가 되었다.

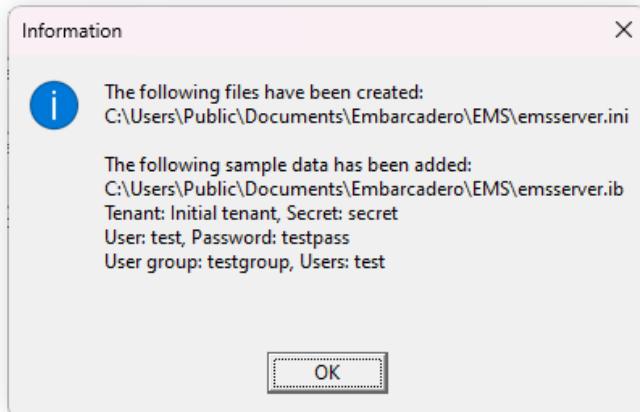


최종 RAD 서버 구성 마법사 페이지

이 페이지에는 데이터베이스 파일 경로 및 이름, 구성 경로 및 이름, 윈도우 레지스터 키가 표시된다. 언제든지 RAD 서버 구성 파일(emsserver.ini)을 변경할 수 있다. 마침 버튼을 클릭한다. 확인 대화 상자가 나타나며 구성에

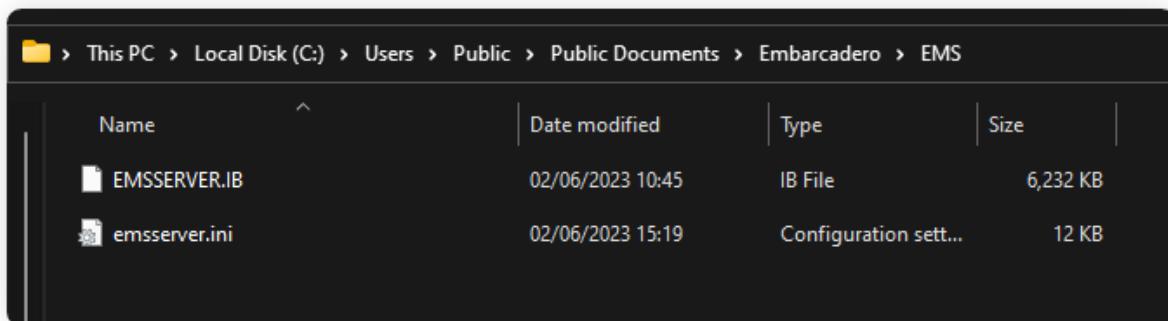
RAD 서버 라이선스가 없는 인터베이스 인스턴스가 사용된다는 알림이 표시된다. 개발 라이선스는 RAD 서버 애플리케이션을 최대 5명의 사용자로 제한한다. RAD 서버 애플리케이션을 배포할 준비가 되면 RAD 서버 및 인터베이스에 대한 배포 라이선스를 사용할 수 있다.

예 버튼을 클릭한다. 마법사에 emsserver.ini 구성 파일의 위치가 표시된다. 또한 데이터베이스에 추가된 샘플 데이터도 나열된다.



구성 마법사가 만든 RAD 서버 파일 목록

확인 버튼을 클릭한다. 이제 두 개의 파일이 C:\Users\Public\Documents\Embarcadero\EMS 디렉토리에 나타난다.



RAD 서버 구성 마법사로 만든 디스크 파일

emsserver.ini 파일은 RAD 서버의 모든 기본 파라미터가 정의되는 곳이다. 다음 장에서 다루겠지만, 파일 자체에 명시된 내용 및 상세문서를 확인해도 좋다.
일반적인 RAD 스튜디오 IDE는 관리자 권한 없이 실행된다. 따라서 윈도우 64-비트 운영체제에서 HKEY_LOCAL_MACHINE에 대한 윈도우 레지스터 항목은 HKEY_CURRENT_USER\Software\Classes\VirtualStore\MACHINE\SOFTWARE\WOW6432Node\Embarcadero\EMS로 가상화된다.



참고

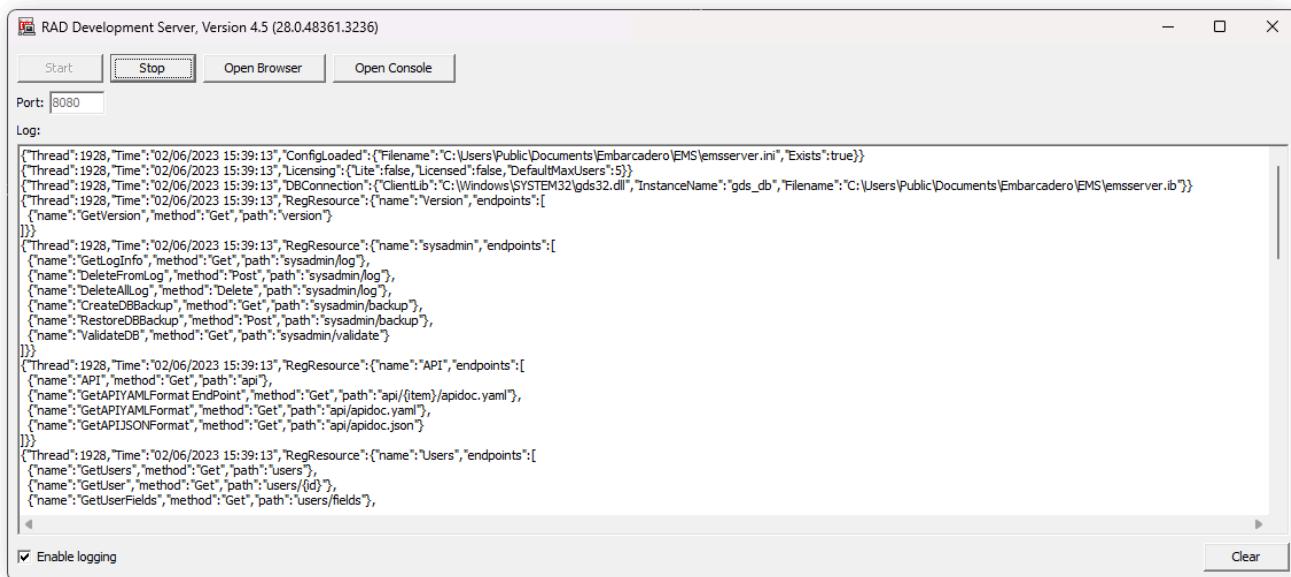
첫 번째 RAD 서버 애플리케이션 테스트

RAD 서버 구성 서버가 생성되면 RAD 서버 개발 서버가 실행을 시작한다.



EMSDevServer가 실행되는 기본 포트는 8080이다. 만일 컴퓨터의 다른 서비스가 해당 포트를 사용하는 경우 "포트" 필드에서 필요에 맞는 포트를 변경하여 기본 포트를 변경할 수 있다. 이 변경 사항을 영구적으로 적용하기 위해선 emsserver.ini 파일에서 [Server.Connection.Dev] 파라미터를 수정하면 된다.

IDE에서 실행을 누르면 RAD 서버 개발 서버가 실행되기 시작하고 로그에 패키지 애플리케이션에 대해 수행되는 작업이 표시된다. 개발 서버는 델파이와 C++에서 완전히 동일하다.



첫 번째 애플리케이션 패키지를 시작하는 RAD 서버 개발 서버

RAD 서버 개발 서버 로그에는 구성, 데이터베이스 연결, 라이선스 정보, 로드된 애플리케이션 패키지, 등록한 리소스 및 생성된 엔드포인트가 표시된다.

브라우저 열기 버튼을 클릭하면 기본 브라우저가 시작되고 GetVersion 내장 엔드포인트를 호출한 결과의 JSON이 표시된다. 이제 첫 번째 RAD 서버 REST 엔드포인트 사용해보았다!

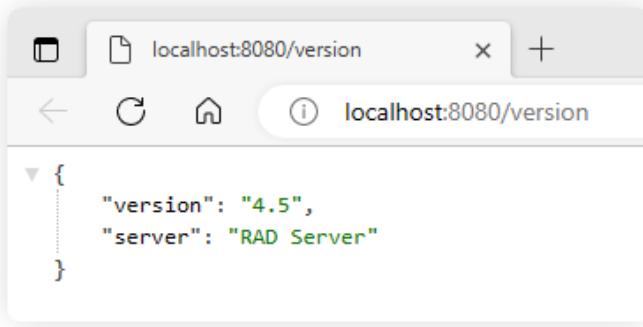


브라우저 사람이 읽을 수 있는 JSON 응답을 얻으려면 확장 프로그램인 "JSON Parser" 를 설치하면 된다. 이는 모든 주요 브라우저에서 사용이 가능하다.

마이크로소프트 엣지의 경우 "edge://flags" 아래에 "JSON 뷰어"라는 설정이 있다. 이 설정을 활성화하면 확장 프로그램을 설치하지 않고도 읽을 수 있는 JSON 응답을 얻을 수 있다.

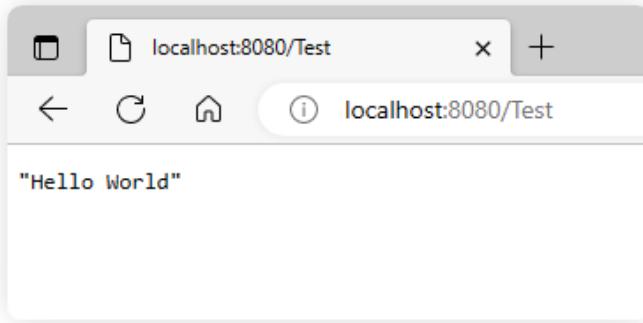
The screenshot shows the RAD Server configuration interface. A yellow header bar at the top contains the text "JSON Viewer". Below it is a white card with the following details:

- JSON Viewer**
- Allows users to view JSON files in a formatted view directly in the browser – Mac, Windows, Linux
- [#edge-json-viewer](#)
- A blue button labeled "Enabled" with a dropdown arrow.



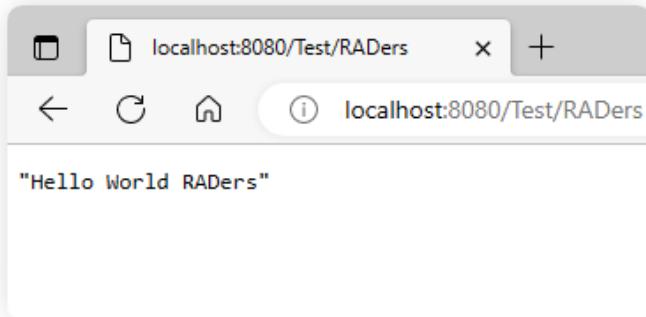
브라우저에서 버전 엔드포인트 호출의 출력을 표시하는 모습

브라우저에서 URL을 `localhost:8080/Test`로 변경하고 엔터키를 누른다. 브라우저에는 Get 엔드포인트에서 JSON 응답을 받는다.



테스트 리소스의 Get 메서드에서 JSON 출력을 표시하는 브라우저

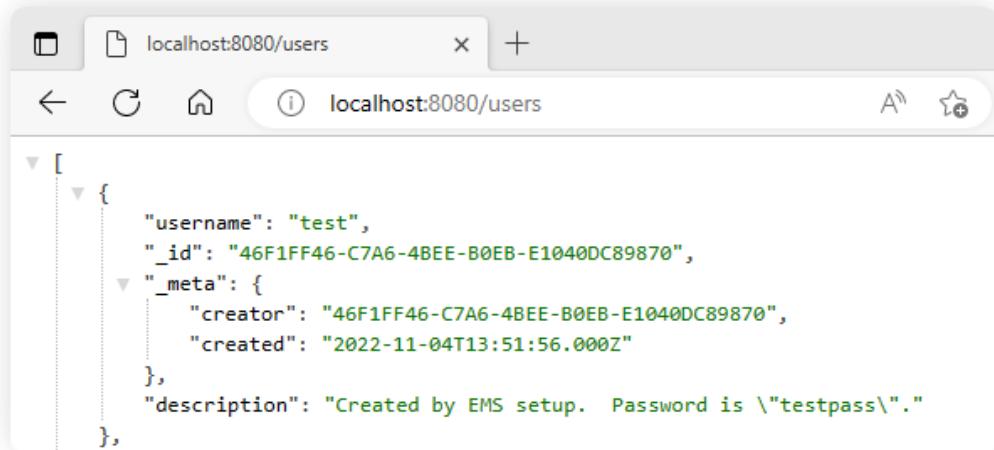
URL에 추가 항목을 전달하면 `GetItem` 엔드포인트가 호출되고 그 뒤에 있는 코드가 리소스 이름과 입력한 항목이 포함된 JSON 문자열을 반환한다.



테스트 리소스의 Get 메서드에서 JSON 출력을 표시하는 브라우저

간단한 예제에서는 JSON 문자열을 반환해도 문제는 없지만, 더 크고 복잡한 데이터 구조의 경우 큰 JSON 문자열을 반환하지 못할 수 있다. RAD 스튜디오는 JSON 객체, JSON 스트림 및 JSON 작성기를 사용하는 등 JSON 데이터를 생성하는 다른 많은 방법을 제공한다.

URL을 편집하여 "users" 리소스를 사용하여 기본 GetUsers 엔드포인트를 호출한다. 그리고 RAD 서버 구성 마법사에서 생성한 사용자에 대한 JSON을 RAD 서버 데이터스토어에 표시한다(시작할 때는 하나뿐이다).



GetUsers 엔드포인트 호출에 대한 브라우저의 JSON 응답

이제 RAD 서버 프로젝트 마법사에서 생성한 엔드포인트 중 4개를 사용했다.

참고 자료

- [이 장의 코드 샘플](#)
- [RAD 서버 엔진 \(EMS 서버\)](#)
- [RAD 스튜디오 \(EMS\) 서버 설정하기](#)
- [윈도우에서 EMS 서버 또는 EMS 콘솔 서버 구성하기](#)
- [RAD 서버 관리 API](#)

03

첫 번째 CRUD 애플리케이션 만들기



RAD 스튜디오는 바로 사용할 수 있는 여러 컴포넌트를 제공한다. 이때, CRUP API를 생성할 때 가장 유용한 것은 EMSDataSetResource이다. 이 컴포넌트를 사용하면 FIndDAC 쿼리를 연결하고 데이터 공개 및 조작이 가능하다. 이 컴포넌트는 CRUD에 필요한 모든 엔드포인트를 자동으로 생성하고 페이지 매김, 정렬 등과 같은 추가 기능을 제공한다.

EMSDataSetResource는 현재의 유닛 중 어느 것이든 생성할 수 있다. 더 간편하게는 RAD 서버 마법사를 사용하여 필요한 모든 컴포넌트를 자동으로 FDConnection에 연결할 수 있다.

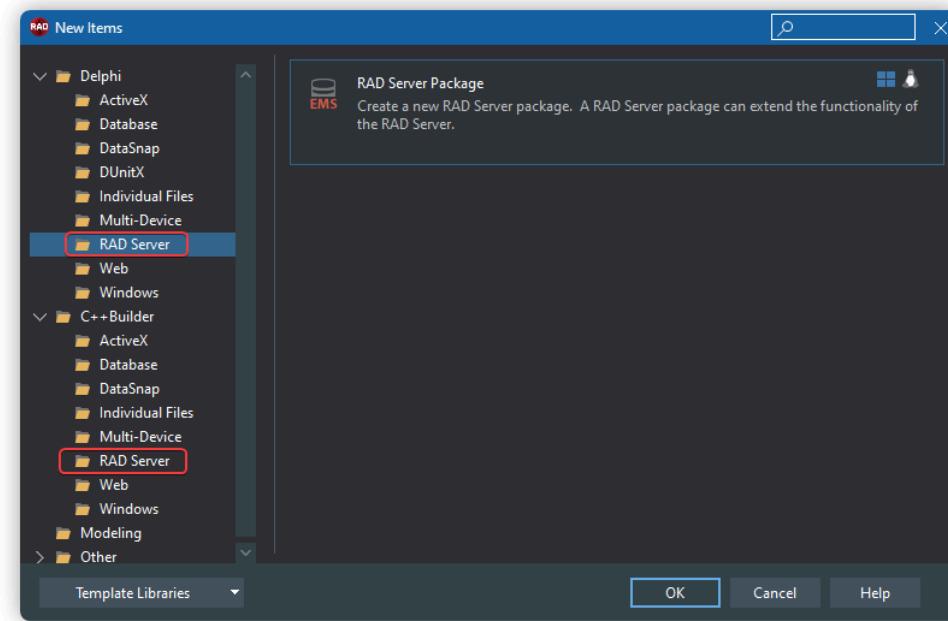


참고

이 데모에서는 직원 인터페이스 데이터베이스를 사용하지만 FireDAC와 호환되는 다른 데이터베이스를 자유롭게 사용해도 된다. RAD 서버 마법사를 사용하기 위한 유일한 요구사항은 데이터베이스 연결이 "데이터 탐색기"에서 미리 구성되어 RAD 스튜디오에서 인식할 수 있어야 한다는 것이다.

CRUD 기능을 사용하여 REST 기반 서비스 빌드

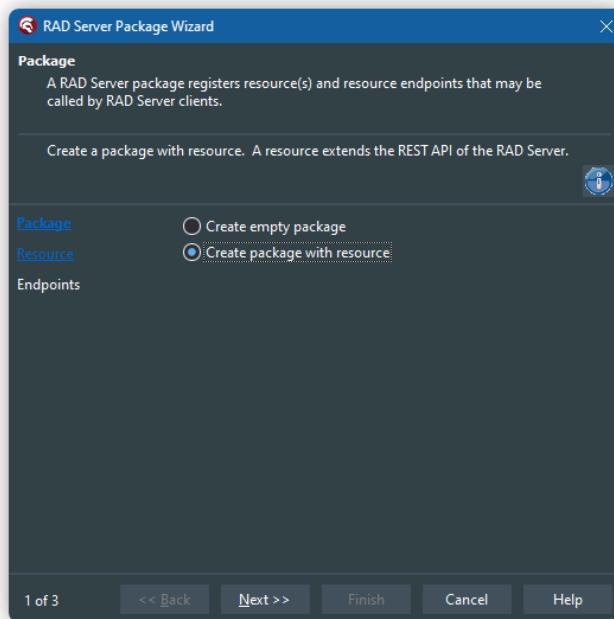
CRUD 기능을 사용하여 REST 기반 서버를 가장 빠르게 시작하는 방법은 새 프로젝트 메뉴(파일 | 새로 만들기 | 기타...)를 사용하여 델파이 또는 C++빌더용 RAD 서버 | RAD 서버 패키지 마법사를 선택하는 것이다.



델파이 및 C++용 RAD 서버 프로젝트 마법사 선택 사항

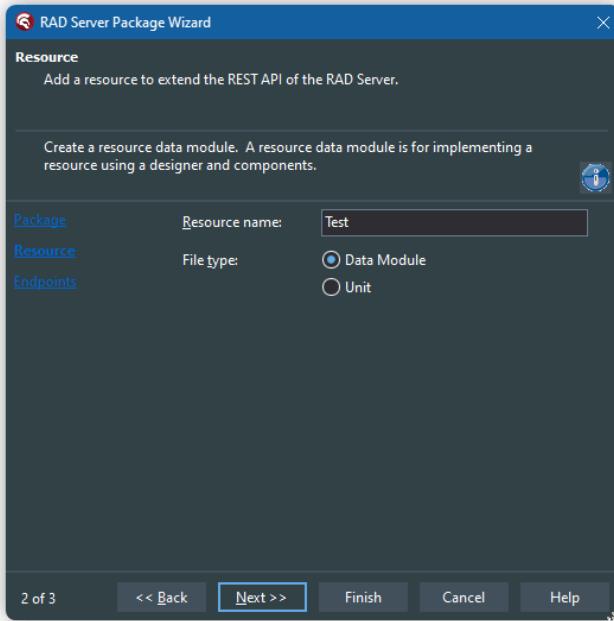
RAD 서버 패키지 프로젝트를 선택한다. 시작 프로젝트의 생성을 돋는 마법사가 나타난다. 첫 페이지에서 마법사가 RAD 서버 애플리케이션을 표시할 리소스 및 엔드포인트를 생성하는 방법을 선택한다. RAD 서버 패키지 마법사에는 두 가지 선택 사항이 제공된다.

RAD 서버용 REST API를 확장하는 리소스가 포함된 패키지 만들기. 다음 버튼을 클릭하면 두 개의 추가적인 마법사 단계가 표시되며, 패키지 프로젝트, 리소스 및 엔드포인트를 만드는데 도움을 준다. 첫 번째 RAD 서버 프로젝트를 빌드하려면 이 옵션을 선택한다.



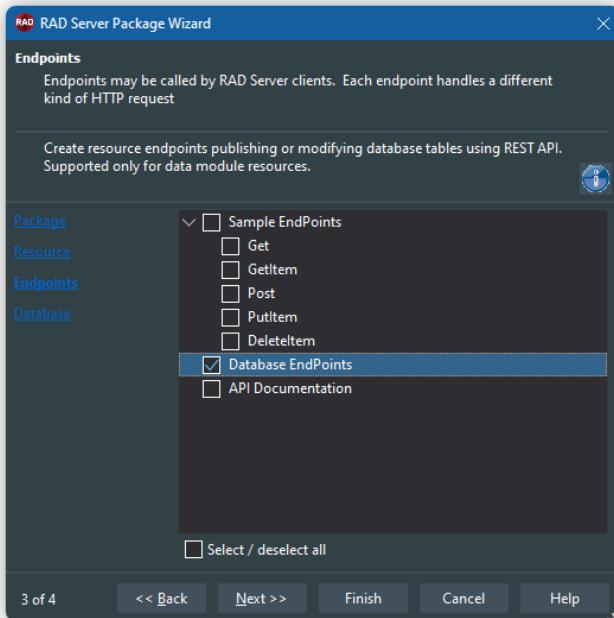
리소스 기반 RAD 서버 패키지 만들기

설치 마법사의 두 번째 페이지에서 리소스 이름을 "Test"로 설정한다. 파일 유형 라디오 버튼에는 두 가지 옵션이 표시된다: 1) 코드에서 리소스를 구현하기 위한 단위를 만들고, 2) IDE의 디자이너, 컴포넌트 및 코드 편집기를 사용하여 리소스를 구현하기 위한 데이터 모듈을 만든다. 이 첫 번째 RAD 서버 애플리케이션에서는 데이터 모듈을 사용할 것이다.



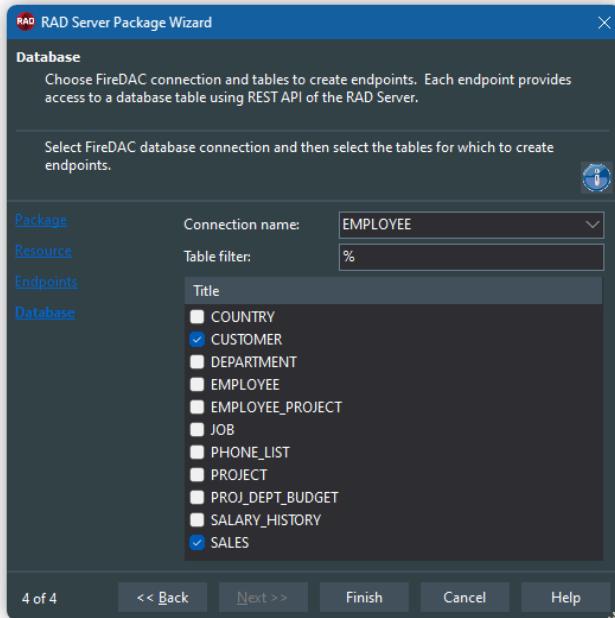
RAD 서버 패키지 마법사 2페이지 - 리소스 이름 및 파일 유형 설정

다음 버튼을 클릭하여 시작 엔드포인트 집합을 생성한다.



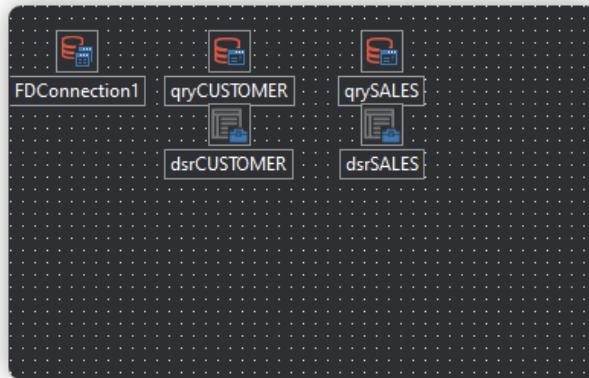
RAD 서버 패키지 마법사 3페이지 - 시작 엔드포인트 선택

마법사 세 번째 페이지에서는 이전 장과 다른 옵션을 선택한다. 이 경우 "샘플 엔드포인트"를 선택 취소하고 "데이터베이스 엔드포인트"를 선택한다. 이제 "다음"을 클릭한다.



RAD 서버 패키지 마법사 4페이지 - 데이터베이스 및 테이블 선택

프로젝트가 생성되면 FDConnection, 2개의 FDQueries 및 2개의 EMSDataSetResource가 표시된다.



마법사로 생성된 데이터 모듈

생성된 프로젝트 설명

이 데모의 장점은 이미 빌드할 수 있으며 이를 위해 자동으로 생성된 엔드포인트에 액세스할 수 있다는 것이다. 하지만 우선, 약간의 수정을 하겠다. 데이터 모듈의 코드에 액세스하여 엔드포인트의 속성을 변경한다. 이는 실제로 큰 관련은 없지만 엔드포인트를 소문자와 복수형으로 유지하는 것이 일반적으로 더 나은 관행이다.

Delphi:

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

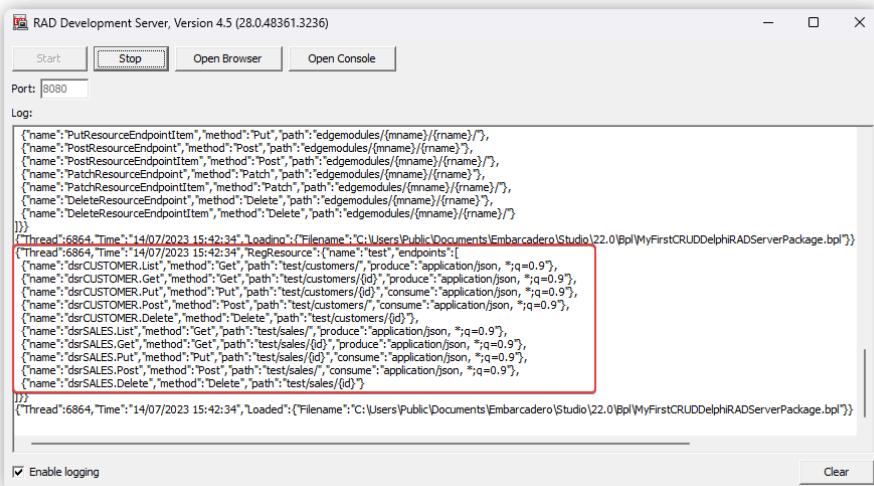
C++:

```
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());
}
```

보게 되면, ResourceSuffix 속성은 EMSDatasetResources에 링크되어있다. 즉, 해당 데이터 집합 리소스에 연결된 쿼리가 해당 엔드포인트 아래에 공개된다.

프로젝트는 이 이상 간단해질 수 없다. 지금까지 코딩한 로직은 단 한 줄도 없으며, 2개의 테이블에 연결된 완전한 기능의 CRUD 시스템을 갖추게 되었다. 프로젝트를 빌드하고 더 자세히 분석해 보겠다.

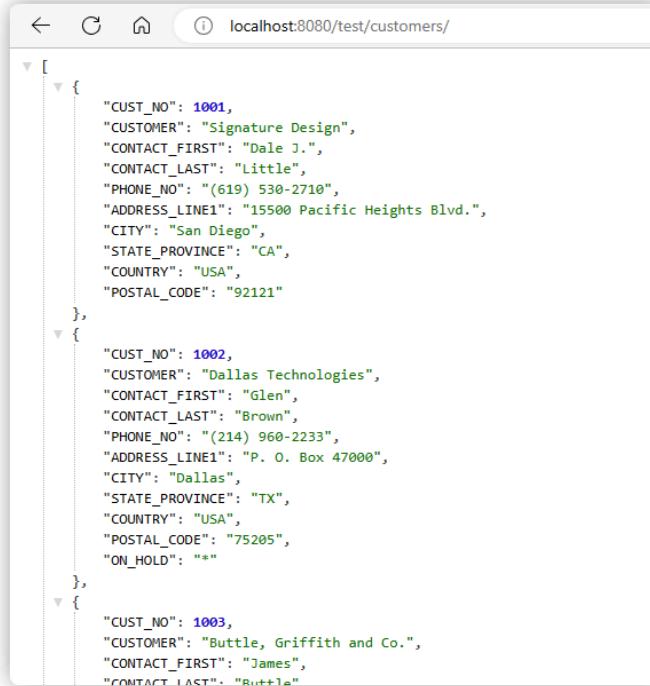
프로젝트 빌드 및 테스트



자동으로 생성된 모든 엔드포인트를 보여주는 RAD 서버 로그

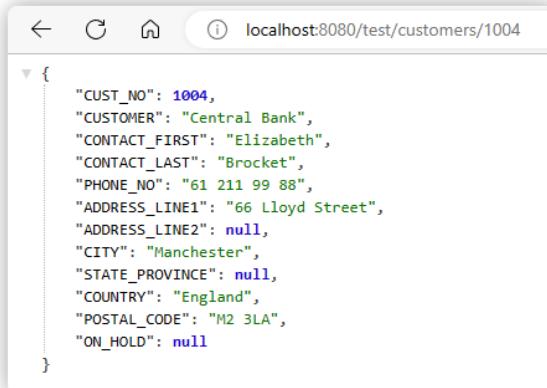
RAD 서버 로그에서 "customers" 및 "sales" 엔드포인트가 생성되었음을 확인할 수 있을 뿐만 아니라 개별 레코드를 가져오거나 삭제하거나 새 레코드를 게시하기 위해 URL에서 {id} 파라미터를 사용할 수 있다.

브라우저 URL <http://localhost:8080/test/customers/>에 액세스하면 고객 테이블의 모든 레코드가 포함된 배열이 반환된다.



RAD 서버가 반환하는 고객 배열

ID(Cust_No)를 사용하여 특정 고객에게 액세스하려면 <http://localhost:8080/test/customers/1004>에 요청을 보내면 된다. 이미 짐작했겠지만, sales 엔드포인트에 액세스하려면 <http://localhost:8080/test/sales/>를 호출하고, 다른 액세스 또한 이와 같은 방식으로 액세스할 수 있다.



```

<pre>
{
    "CUST_NO": 1004,
    "CUSTOMER": "Central Bank",
    "CONTACT_FIRST": "Elizabeth",
    "CONTACT_LAST": "Brocket",
    "PHONE_NO": "61 211 99 88",
    "ADDRESS_LINE1": "66 Lloyd Street",
    "ADDRESS_LINE2": null,
    "CITY": "Manchester",
    "STATE_PROVINCE": null,
    "COUNTRY": "England",
    "POSTAL_CODE": "M2 3LA",
    "ON_HOLD": null
}
</pre>

```

특정 고객의 ID를 사용하여 특정 고객에게 액세스하기



경고

*TEMSDatasetResource를 사용할 때는 엔드포인트 끝에 슬래시(/)를 넣는 것이 중요하다.
엔드포인트에 / 없이 액세스를 시도하면 RAD 서버가 "찾을 수 없음" 예외를 발생시킨다.*

TEMSDatasetResource의 추가 기능

지금까지 살펴본 내용은 이미 충분히 인상적이다. 지금까지 본 것으로는 클릭 몇 번으로 필요한 만큼의 데이터 집합을 공개하고 매우 빠르게 API를 개발할 수 있다. 하지만 TEMSDatasetResources는 훨씬 더 많은 기능을 기본으로 제공한다. 주요 기능을 분석해 보겠다.

AllowedActions		[List,Get,Post,Put,Delete]
List	<input checked="" type="checkbox"/>	True
Get	<input checked="" type="checkbox"/>	True
Post	<input checked="" type="checkbox"/>	True
Put	<input checked="" type="checkbox"/>	True
Delete	<input checked="" type="checkbox"/>	True
> DataSet		qryCUSTOMER
KeyFields		
> LiveBindings Designer		LiveBindings Designer
MappingMode		rmGuess
Name		dssCUSTOMER
> Options		[roEnableParams,roEnablePaging]
roEnableParams	<input checked="" type="checkbox"/>	True
roEnablePaging	<input checked="" type="checkbox"/>	True
roEnableSorting	<input checked="" type="checkbox"/>	True
roReturnNewEntityKey	<input checked="" type="checkbox"/>	True
roReturnNewEntityValue	<input type="checkbox"/>	False
roAppendOnPut	<input checked="" type="checkbox"/>	True
PageParamName		page
PageSize		50
ParamBindMode		Mixed
SortingParamPrefix		sf
Tag		0
ValueFields		

AllowedActions – 내장된 제어 기능으로, 앤드포인트에서 목록, 가져오기, 게시, 넣기 및 삭제를 허용하거나 방지하기 위한 것이다.

DataSet – 데이터 집합에 연결한다: 쿼리, 테이블 등

KeyFields – 데이터를 조회할 때 일치해야 할 데이터 집합 필드를 선택한다.

PageParamName – URL을 통해 페이지 매김을 사용할 파라미터의 이름이다.

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

SortingParamPrefix – 리스트 액션에 액세스할 때 페이로드의 페이지네이션 크기를 정의한다.

ValueFields – 데이터 setValueFields 항목에 미리 붙일 텍스트 문자열이다.

Options – 파라미터 사용, 행단위 페이징, 데이터 집합 필드 정렬 등을 활성화/비활성화하기 위한 하위 속성 설정이다.

이제 이 컴포넌트에 대해 조금 더 알게 되었으니 API에서 이러한 기능 중 일부를 사용해 보겠다. http://localhost:8080/test/customers/?sfCONTACT_LAST=A&page=1에 액세스 하면 CONTACT_LAST 필드에 따라 오름차순으로 고객의 첫 페이지가 표시된다. 만약 =D에 대해 =A값을 변경하면 응답은 내림차순으로 표시된다.

하지만 이 '정렬 기준'은 어떻게 SQL에 삽입될까? FDQuery를 열면 다음과 같은 서술이 표시된다.

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

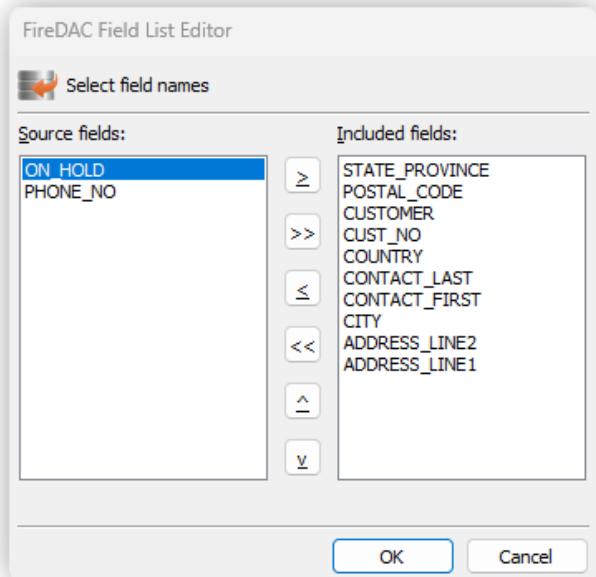
보다시피 SQL 구문은 매우 간단하지만, 이 구문으로 작동하기 위해서는 이 매크로가 핵심이다. EMSDatasetResource는 이 매크로를 사용하여 동일한 쿼리에서 페이지 매김과 순서를 혼합할 수 있다.



참고

페이지 매김이 사용 중이고 데이터 집합의 끝에 도달할 경우 RAD 서버는 해당 페이지에 다른 항목이 없음을 알리기 위해 빈 배열을 반환하기만 하면 된다.

또 다른 매우 유용한 기능은 로직에 사용할 필드를 데이터베이스에서 가져오는 옵션이지만 API에 해당 필드를 공개하고 싶지 않은 경우에 사용한다. ValueFields 속성을 사용하면 게시할 필드를 쉽게 선택할 수 있다.



API 엔드포인트에 게시할 선택된 필드

04

REST 디버거



첫 장에서 REST API 에코시스템에서 사용할 수 있는 액션에 관해 이야기했지만, 지금까지는 브라우저를 통한 GET만 사용했다. 이 장에서는 REST 디버거라는 RAD 스튜디오와 함께 제공되는 도구를 사용하여 테스트 프로세스를 간소화한다. 또한 더 빠르게 애플리케이션을 개발하는 데 도움을 주는 POST, PUT 및 DELETE 액션을 사용하는 방법을 살펴볼 것이다.



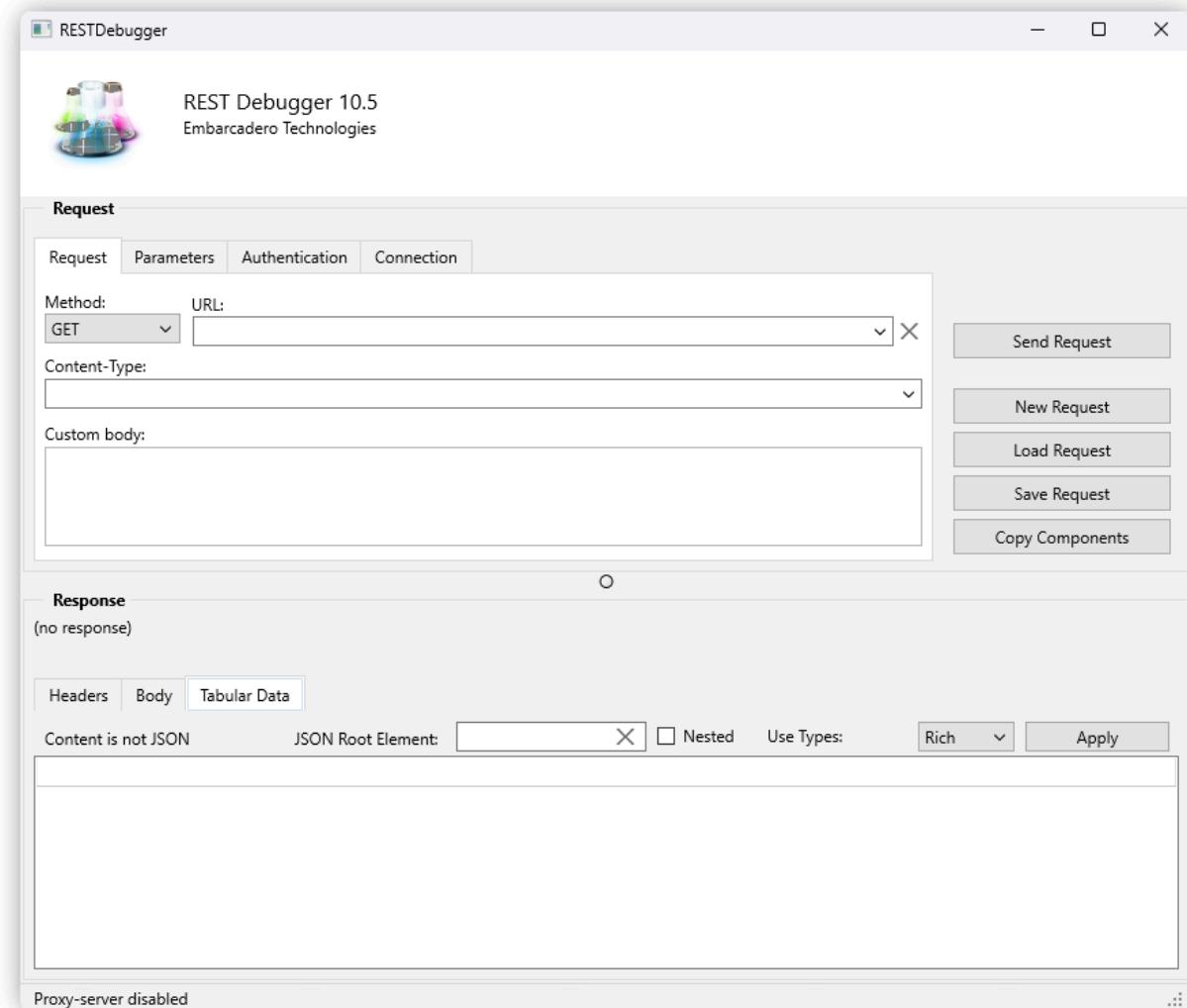
팁

REST 디버거는 RAD 서버에서만 사용할 수 있는 제품이 아니다. 타사의 REST API 서비스에 액세스하고 RAD 스튜디오와의 통합을 활용하면 개발 프로세스의 속도를 높일 수 있다.

REST 디버거란 무엇이며 어디에서 찾을 수 있는가

[REST 디버거](#)는 엠바카데로(Embarcadero)의 무료 솔루션이다. 이는 RESTful 웹 서비스를 탐색하고, 이해하고, 델파이 및 C++ 빌더 앱과 통합할 때 이것을 사용할 수 있다. 이 링크에서 무료로 다운로드 할 수 있다. 이는 개발자들이 필터링할 수 있는 JSON 블롭, 간소화된 OAuth 1.0/2.0 인증 및 구성할 수 있는 요청/리소스 파라미터와 같은 기능을 활용하여 RESTful 웹 서비스가 어떻게 작동하는지 탐색, 테스트 및 이해할 수 있도록 돋는다. 뿐만 아니라, 몇 번의 클릭만으로 REST 컴포넌트를 복사하여 프로젝트에 직접 붙여 넣을 수 있는 기능도 제공한다.

사용해 보고 싶다면 RAD 스튜디오의 **도구/REST 디버거** 메뉴에서 찾거나 [이 링크](#)를 통해 독립 실행이 가능한 버전을 무료로 다운로드할 수 있다.



REST 디버거 UI

REST 디버거로 첫 번째 PUT 요청 보내기

왼쪽의 드롭다운 메뉴에서 기본값이 GET임을 확인할 수 있다. 추가적으로 이제부터 브라우저에서는 선택할 수 없는 다른 값들도 선택할 수 있게 되었다.

3장에서 만든 것과 동일한 프로젝트를 사용하여 고객을 수정해 보겠다.



경고 3장에서 만든 RAD 서버 프로젝트를 실행은 필수이다. 그렇게 하지 않으면 API 엔드포인트를 호출할 수 없다.

고객을 수정하기 위해서는 수정하려는 고객의 ID로 고객 엔드포인트를 호출하기만 하면 된다. 또한 요청 본문에서 속성의 새 값을 지정해야 한다.

The image contains two screenshots of the REST Debugger tool. Both screenshots show a 'Request' tab selected with four sub-tabs: Request, Parameters, Authentication, and Connection.

PUT Request Screenshot:

- Method:** PUT
- URL:** http://localhost:8080
- Content-Type:** application/json
- Custom body:**

```
{
  "CUSTOMER": "Bank of Manchster",
  "PHONE_NO": "+44612119988"
}
```

Buttons on the right:

- Send Request
- New Request
- Load Request
- Save Request
- Copy Components

GET Request Screenshot:

- Resource:** test/customers/1004
- Request Parameters:** An empty list with three buttons: Add, Edit, and Delete.

Buttons on the right:

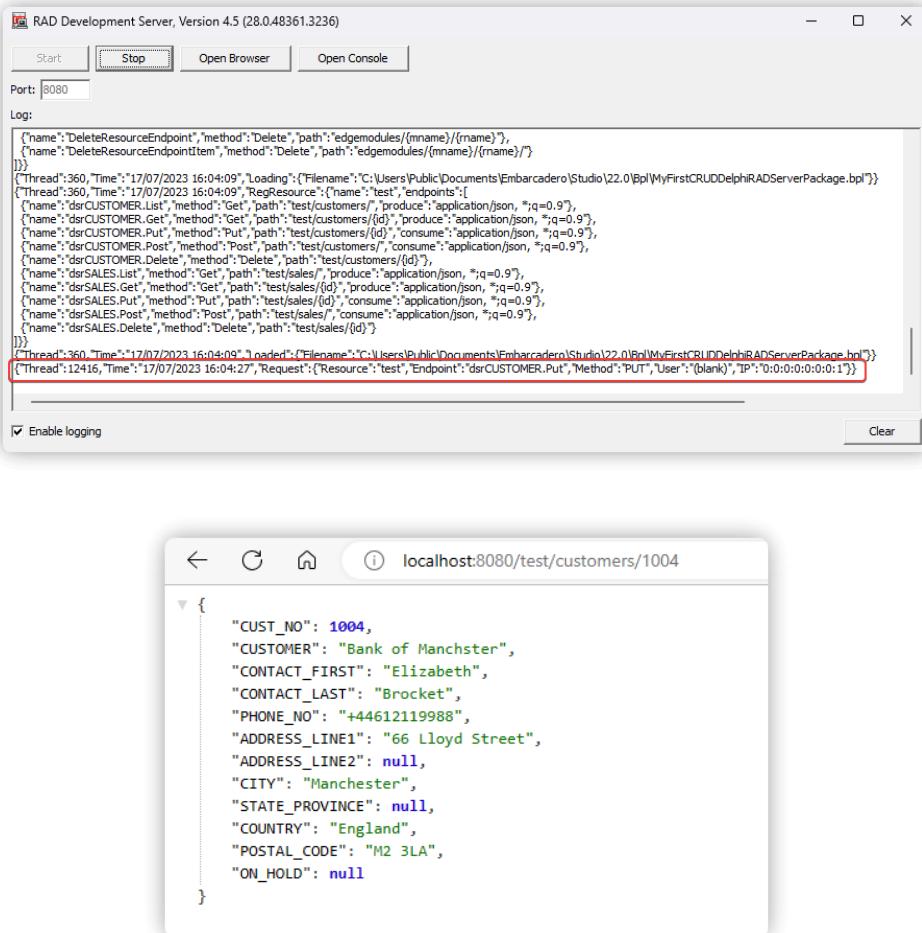
- Send Request
- New Request
- Load Request
- Save Request
- Copy Components

PUT 요청 전송에 필요한 값 정의하기

요청을 설정할 때, PUT 메서드와 URL 그리고 특정 리소스를 지정했다(이 경우에는 ID가 1004인 고객). 더불어, 업데이트하고자 하는 속성을 담은 JSON 본문도 포함했다: 고객의 이름과 전화번호다.

마지막 단계는 "요청 보내기"를 누르고 200 HTTP 응답을 받는다. 응답을 정상적으로 받았다면 이제 RAD 서버 로그에서 해당 요청이 처리된 것을 확인할 수 있다. 또한, 이 특정 고객에 대해 GET 요청을 보내면(REST 디버거 또는 브라우저에서 이 작업을 수행할 수 있다) 데이터가 성공적으로 업데이트되었음을 확인할 수 있다.

새로운 고객을 생성하려는 경우에도 절차는 동일하다. 하지만 본문에 필요한 모든 정보를 제공해야 하며 방법을 POST로 변경해야 한다.



등록된 PUT 요청 및 수정된 데이터가 포함된 RAD 서버 로그

REST 디버거에 포함된 기타 기능

RAD 서버와 크게 관련 있지는 않지만 REST 디버거가 제공하는 매우 강력한 기능을 언급할 가치는 있다. URL, 파라미터 등을 정의한 후에는 "컴포넌트 복사" 버튼을 사용하여 프로젝트에 붙여넣기만 하면 된다. 이렇게 하면 RAD 서버 또는 기타 타사 API에 액세스하기 위한 더욱 빠른 UI를 프로토타이핑할 수 있다.

이 장의 깃허브 리퍼지토리에서 이에 대한 기본 FMX 예제를 찾을 수 있다. API 액세스에 필요한 모든 컴포넌트는 "컴포넌트 복사" 버튼을 사용하여 복사하여 붙여 넣었다. 테스트하기 위해서 먼저 RAD 서버 애플리케이션을 실행한 후 FMX 애플리케이션을 실행하고, "요청 보내기"를 누르면 된다.

REST API와 관련하여 또 다른 중요한 주제는 인증이다. API를 인증해야 하는 경우 "인증" 탭에서 여러 가지 방법을 사용할 수 있으며, "파라미터" 탭의 "파라미터 추가" 버튼을 사용하여 요청에 특정 파라미터(예: 헤더에 api-key 파라미터)를 포함할 수도 있다.

05

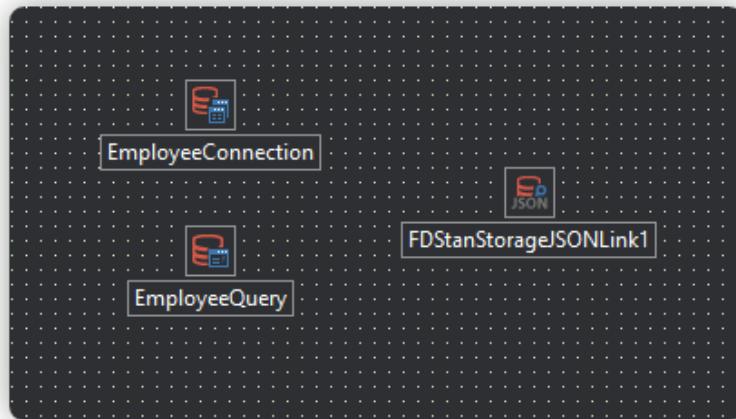
FireDAC 일괄 처리 및 JSONWriter 사용

프로젝트의 요구사항이나 어떤 기술에 얼마나 숙련됐는지에 따라 RAD 스튜디오에서 REST API를 만들 때 더 다양한 도구를 활용할 수 있다.

FireDAC 컴포넌트는 데이터베이스의 메타데이터가 포함된 스트림을 생성하고 소비하는 데 사용할 수 있으며, RAD 서버 앤드포인트 중 하나에서 응답을 위해 JSON으로 인코딩된 데이터를 사용할 수 있다. 이러한 접근 방식은 클라이언트 애플리케이션이 VCL 또는 FMX일 경우 유용하다. 메모리 테이블을 사용하면 데이터베이스 정보와 메타데이터를 자동으로 매핑할 수 있다. JavaScript와 같은 언어를 사용하는 다른 클라이언트 애플리케이션은 응답에 포함될 데이터 정보 및 데이터를 처리하는 데 문제가 있을 수 있다. 하지만, RAD 스튜디오는 JavaScript 혹은 다른 언어가 수신할 것으로 예상되는 분명한 JSON을 생성하는 방법을 제공한다.

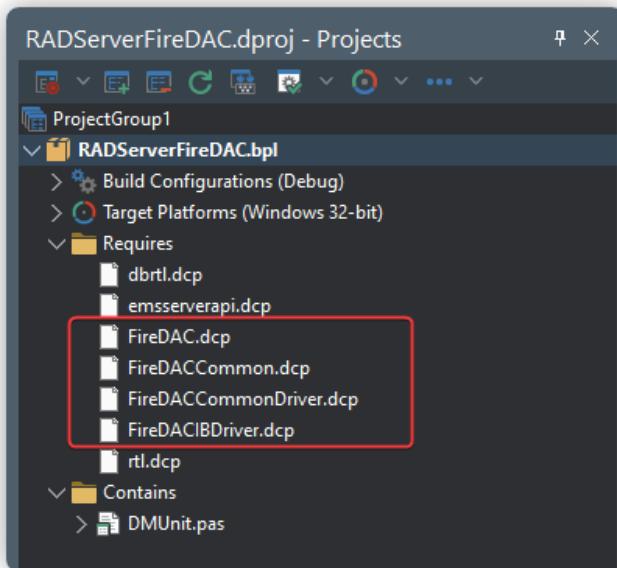
메모리 스트리밍을 사용하여 JSON 데이터베이스의 데이터 반환하기

FireDAC에는 데이터베이스 테이블 정보에 액세스하고 결과를 JSON 문자열로 생성하는 컴포넌트가 포함되어 있다. 리소스 모듈을 사용하여 RAD 서버 애플리케이션을 생성한다. FDConnection 컴포넌트를 추가하고 인터베이스 샘플 Employee.gdb 데이터베이스와 연결한다. FDQuery 컴포넌트를 추가하고 EmployeeQuery SQL 구문을 "select * from employee"로 설정한다. FDStanStorageJSONLink 컴포넌트를 추가하여 JSON을 쉽게 생성할 수 있도록 한다.

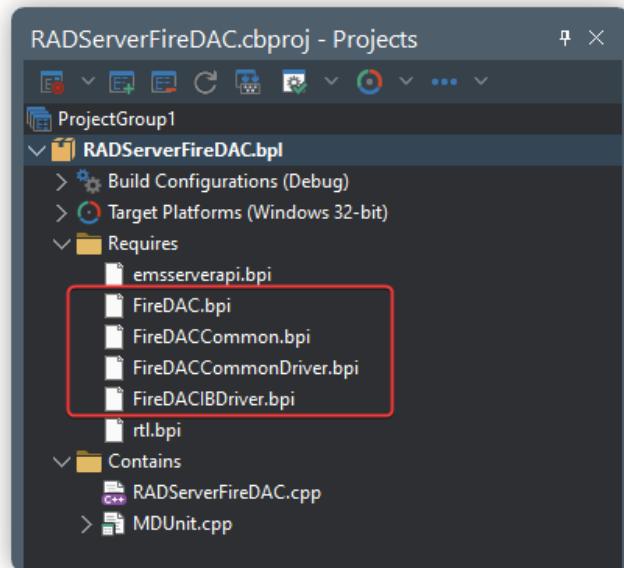


RAD 서버 프로젝트의 리소스 모듈

RAD 서버의 델파이 기반 애플리케이션을 빌드할 때, 일련의 경고와 함께 대화상자가 나타날 수 있다. 이 대화상자는 애플리케이션 패키지가 설치된 다른 패키지와 호환되도록 허용할 수 있다. 확인 버튼을 클릭하면 필요한 패키지 파일이 프로젝트의 요구 구역에 추가된다. C++빌더의 경우 패키지를 수동으로 추가할 수 있다(프로젝트 관리자 창에서 필요 노드를 마우스 오른쪽 버튼으로 클릭하고 팝업 메뉴에서 참조 추가...를 선택).



델파이 RAD 서버 FireDAC 프로젝트



C++ RAD 서버 FireDAC 프로젝트



이 파일은 각 대상 플랫폼의 C:\Program Files (x86)\Embarcadero\Studio\0\lib에서 찾을 수 있다.

다음은 메모리 스트림을 사용하여 직원 테이블 데이터와 함께 JSON 응답을 전송하는 RAD 서버 GET 메서드를 구현한다.

델파이:

```

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);
}

```

브라우저와 URL `http://localhost:8080/FireDAC`을 사용하여 데이터베이스의 직원 테이블에 대한 JSON 데이터가 포함된 응답을 가져온다. JSON에는 데이터뿐만 아니라 훨씬 더 많은 정보가 포함되어 있다. 테이블, 열, 유형 등에 대한 메타데이터 정보도 응답에 포함되어 있다.

```

{
  "FDBS": {
    "Version": 16,
    "Manager": {
      "UpdatesRegistry": true,
      "TableList": [
        {
          "class": "Table",
          "Name": "EmployeeTable",
          "SourceName": "employee",
          "SourceID": 1,
          "TabID": 0,
          "EnforceConstraints": false,
          "MinimumCapacity": 50,
          "ColumnList": [
            {
              "class": "Column",
              "Name": "EMP_NO",
              "SourceName": "EMP_NO",
              "SourceID": 1,
              "DataType": "Int16",
              "Searchable": true,
              "AllowNull": true,
              "AutoInc": true,
              "Base": true,
              "AutoIncrementSeed": -1,
              "AutoIncrementStep": -1,
              "AllowNull": true,
              "OInUpdate": true,
              "OInWhere": true,
              "OInKey": true,
              "OAfterInsChanged": true,
              "OriginTabName": "EMPLOYEE",
              "OriginColName": "EMP_NO",
              "SourceDataTypeName": "EMPNO",
              "SourceDirectory": "EMPNO"
            },
            {
              "class": "Column",
              "Name": "FIRST_NAME",
              "SourceName": "FIRST_NAME",
              "SourceID": 2
            }
          ]
        }
      ]
    }
  }
}

```

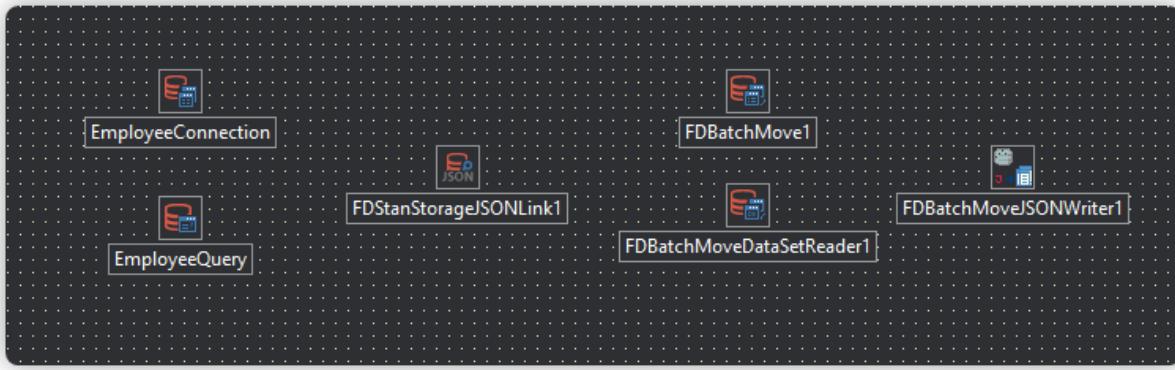
JSON 응답이 포함된 브라우저 창

이 JSON 코드는 다른 언어에서 사용하여 응답을 파싱하지 않고도 쉽게 사용할 수 있는 단순한 '키-값'쌍이 아니다. 그러나 RAD 스튜디오로 개발된 클라이언트를 사용할 경우 매우 편리하게 사용할 수 있다.

파이어닥의 BatchMove, BatchMoveDataSetReader 그리고 BatchMoveJSONWriter 사용하기

복잡한 데이터베이스의 경우 이전 장과 위에서 언급한 바와 같은 접근 방식을 사용할 경우 훨씬 더 많은 코드를 작성해야 한다. FireDAC의 FDBatchMove, FDBatchMoveDataSetReader 및 FDBatchMoveJSONWriter 컴포넌트를 활용하면 JSON 응답 생성을 크게 간소화할 수 있다.

우리가 만든 것과 동일한 프로젝트를 업그레이드하여 FDBatchMove, FDBatchMoveDataSetReader 및 FDBatchMoveJSONWriter 컴포넌트를 리소스 모듈에 추가하겠다.



FireDAC 쿼리, BatchMove, DataSetReader 및 JSONWriter가 포함된 리소스 모듈

FDBatchMoveDataSetReader의 DataSet 속성을 EmployeeQuery로 설정한다.

GetBatchMove라는 새 앤드포인트를 만든다.

델파이:

```
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();
}
```

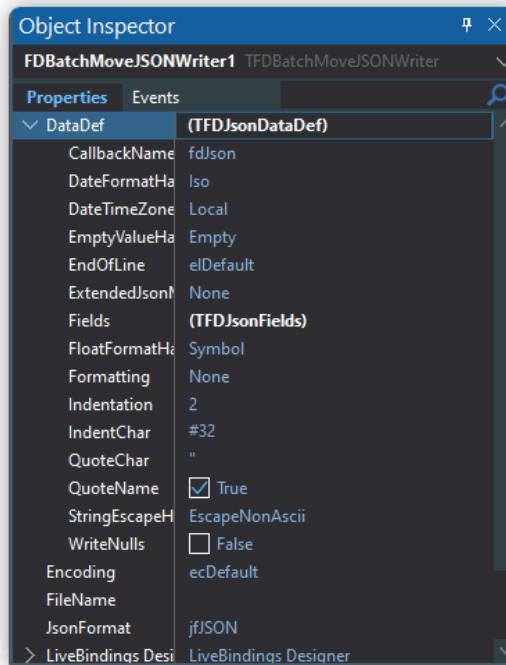
URL <http://localhost:8080/BatchMove> 을 사용하여 GET 메서드를 호출하면 JSON 데이터 결과가 반환된다:

```

[ {
    "EMP_NO": 2,
    "FIRST_NAME": "Robert",
    "LAST_NAME": "Nelson",
    "PHONE_EXT": "250",
    "HIRE_DATE": "2007-12-29T00:00:00.000Z",
    "DEPT_NO": "600",
    "JOB_CODE": "VP",
    "JOB_GRADE": 2,
    "JOB_COUNTRY": "USA",
    "SALARY": 105900,
    "FULL_NAME": "Nelson, Robert"
},
{
    "EMP_NO": 4,
    "FIRST_NAME": "Bruce",
    "LAST_NAME": "Young",
    "PHONE_EXT": "233",
    "HIRE_DATE": "2007-12-29T00:00:00.000Z",
    "DEPT_NO": "621",
    "JOB_CODE": "Eng",
    "JOB_GRADE": 2,
    "JOB_COUNTRY": "USA",
    "SALARY": 97500,
    "FULL_NAME": "Young, Bruce"
},
{
    "EMP_NO": 5,
    "FIRST_NAME": "Kim",
    "LAST_NAME": "Wong"
}
]
  
```

BatchMov를 사용한 JSON 결과 브라우저

FDBatchMoveJSONWriter는 DateFormats, 끝줄, writeNulls 등과 관련하여 JSON 결과의 형식을 지정하는 여러 옵션을 제공한다.



ObjectInspector의 BatchMoveJSONWriter DataDef 하위 프로퍼티

FDBatchMove 컴포넌트를 사용하면 소스 및 대상 열 매핑을 설정하고 현재 소스 레코드 값을 가져오는 매핑을 생성할 수 있다.

매핑 속성은 다음과 같이 채워질 수 있다:

- 디자인 또는 런타임에 수동으로 채울 수 있다. 이를 통해 사용자 지정 매핑, 변환 표현식 등을 지정할 수 있다.
- 비어 있는 경우 실행 호출 시 자동으로 채워진다. 소스 열과 대상 열의 일치 여부는 열 이름으로 수행된다. 대상 열에 대응하는 소스 열이 없는 경우 대상 열은 매핑에서 제외되며 데이터 이동 시 채워지지 않는다.

참고 자료

- [마르코 칸투 블로그: JSON에 대한 데이터 집합 매핑 - RAD 서버 웹 서비스 델파이 예제](#)
- [FireDAC.Comp.BatchMove.TFDBatchMove](#)
- [FireDAC.Comp.BatchMove.JSON.TFDBatchMoveJSONWriter](#)
- [읽기 및 쓰기 JSON 프레임워크](#)
- [FireDAC.TFDBatchMove 샘플](#)
- [RTL.JSONWriter](#)

06

JSONValue, JSONWriter 및 JSONBuilder

• •

RAD 서버는 다양한 프로그래밍 언어와 도구에서 사용할 수 있는 JSON 데이터 처리를 지원한다. JSON 문자열을 생성하고, 문자열을 응답으로 전송하고, 클라이언트 애플리케이션 코드가 반환을 처리하도록 하는 작업은 소량의 데이터에서는 괜찮다. 하지만 전체 데이터베이스나 복잡한 데이터 구조에 대한 JSON 배열 응답이 얼마나 커질지 상상이 가는가? RAD 스튜디오는 JSON 데이터 작업을 위한 세 가지 주요 프레임 워크를 제공한다. 이 장에서는 RAD 서버 애플리케이션이 호출 애플리케이션에 JSON을 반환할 수 있는 여러 가지 방법 중 몇 가지를 다룬다.

JSON 데이터 처리를 위한 프레임워크

RAD 스튜디오는 JSON 데이터를 처리하기 위한 여러 프레임워크를 제공한다. 가장 일반적인 세 가지 프레임워크는 다음과 같다:

- JSON 객체 프레임워크 - JSON 데이터를 읽고 쓸 수 있는 임시 객체를 만든다.
- 읽기 및 쓰기 JSON 프레임워크 - JSON 데이터를 직접 읽고 쓸 수 있다.
- JSONBuilder - 작성기를 사용하여 보다 유지보수가 용이한 방식으로 복잡한 구조를 생성할 수 있다.

JSON 오브젝트 프레임워크는 JSON 데이터를 파싱하거나 생성하기 위해 임시 객체를 만들어야 한다. JSON 데이터를 읽거나 쓰기 위해서는 JSON을 읽고 쓰기 전에 TJSONObject, TJSONArray, 또는 TJSONString과 같은 중간 메모리 객체를 만들어야 한다.

읽기 및 쓰기 JSON 프레임워크를 사용하면 애플리케이션이 임시 오브젝트를 만들지 않고도 스트림에 직접 JSON 데이터를 읽고 쓸 수 있다. JSON을 읽고 쓰기 위해 임시 오브젝트를 만들 필요가 없으므로 성능이 향상되고 메모리 소비가 개선된다.

JSON 빌더는 앞의 두 가지를 결합한 것이다. 코드를 더 읽기 쉽고 유지보수하기 쉽게 만들기 위해 만들어졌다. 또한 메서드를 차례로 연결할 수 있는 보다 현대적인 접근 방식을 따른다. 이 장의 데모 프로젝트에서는 이 세 가지 프레임워크를 사용하여 정확히 동일한 응답을 생성하는 3개의 서로 다른 엔드포인트를 확인할 수 있다. 프로젝트에서 더 편한 것을 자유롭게 사용하기 바란다.



각 엔드포인트에서 얻은 동일한 JSON 응답

JsonValue 사용하기

JSON 객체 프레임워크를 사용하여 JSON 문자열을 코드에서 조립하여 생성한다. **JsonValue**는 JSON 문자열, 객체, 배열, 숫자, 불리언, 참, 거짓 및 **null** 값을 정의하는 데 사용하는 모든 JSON 클래스의 조상 클래스이다. RAD 스튜디오 JSON 구현에는 다음과 같은 클래스와 메서드가 포함되어 있다.

TJSONObject - JSON 객체를 구현한다. **TJSONObject** 메서드에는 다음이 포함된다:

- **Parse** - JSON 데이터 스트림을 구문 분석하고 발견된 JSON 쌍을 **TJSONObject** 인스턴스에 저장하는 메서드다.
- **ParseJsonValue** - 바이트 배열을 구문 분석하고 데이터에서 해당 JSON 값을 생성하는 메서드다.
- **AddPair** 메서드 - JSON 객체에 새 JSON 쌍을 추가하는 메서드다.
- **GetPair** 메서드 - JSON 객체의 쌍 목록에서 지정된 인덱스 *i*의 키-값 쌍을 반환하거나, 지정된 인덱스 *i*가 범위를 벗어난 경우 **nil**을 반환한다.
- **GetPairByName** 메서드 - 지정된 **PairName** 문자열과 일치하는 키 부분을 가진 JSON 객체에서 키-값 쌍을 반환하거나, **PairName**과 일치하는 키가 없는 경우 **nil**을 반환한다.
- **SetPairs** - 이 JSON 객체에 포함된 키-값 쌍의 목록을 정의한다.
- **FindValue** - 지정된 JSON 경로에 있는 **TJsonValue** 인스턴스를 찾아서 반환한다. 그렇지 않으면 **nil**을 반환한다.
- **GetValue** - JSON 객체의 **Name** 키로 지정된 키-값 쌍에서 값 부분을 반환하거나, **Name**과 일치하는 키가 없는 경우 **nil**을 반환한다.

- Pairs - JSON 객체의 쌍 목록에서 지정된 인덱스에 있는 키-값 쌍에 액세스하거나, 지정된 인덱스가 범위를 벗어난 경우 nil을 반환한다.
- GetCount - JSON 객체의 키-값 쌍의 수를 반환한다.

TJSONArray - JSON 배열을 구현한다. JSONArray 메서드에는 다음이 포함된다:

- Add - Element 파라미터를 통해 지정된 null이 아닌 값을 현재 요소 목록에 추가한다.
- Get - JSON 배열의 지정된 인덱스에 있는 요소를 반환한다.
- Pop - JSON 배열에서 첫 번째 요소를 제거한다.
- Size - JSON 배열의 크기를 반환한다.
- ToBytes - 현재 JSON 배열의 콘텐츠를 바이트 배열로 직렬화한다.
- ToString - 현재 JSON 배열을 직렬화하여 문자열로 반환하고 결과 문자열을 반환한다.

추가적인 JSON 클래스는 다음과 같다:

- TJSONString - JSON 문자열을 구현
- TJSONNumber - JSON 숫자를 구현
- TJSONBool - JSON 부울 값을 구현
- TJSONTrue - JSON 참 값을 구현
- TJSONFalse - JSON 거짓 값을 구현
- TJSONNull - JSON null 값을 구현

JSON 클래스 사용 예제

데모 프로젝트의 다음 GetJSON 메서드는 여러 JSON 클래스를 사용하여 JSONObjects 및 JSONArray의 결과를 생성, 파싱 및 표시하는 Get 엔드포인트를 구현한다.

델파이:

```
procedure TTestResource1.GetJSON(const AContext: TEndpointContext; const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
begin
  // 몇 개의 JSON 오브젝트를 생성한다.
  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);
// 배열을 생성하고 이전 오브젝트를 할당한다.
var JSONArray := TJSONArray.Create;
JSONArray.Add(JSONRed);
JSONArray.Add(JSONBlue);
// 색상 배열을 포함할 추가 오브젝트를 생성한다.
var JSONObject := TJSONObject.Create;
JSONObject.AddPair('colors', JSONArray);
AResponse.Body.SetValue(JSONObject, True);
end;

```

C++:

```

void TTestResource1::GetJSON(TEndpointContext* AContext, TEndpointRequest* ARequest,
TEndpointResponse* AResponse)
{
    // 몇 개의 JSON 오브젝트를 생성한다.
    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);
    // 배열을 생성하고 이전 오브젝트를 할당한다.
    TJSONArray* JSONArray = new TJSONArray();
    JSONArray->Add(JSONRed);
    JSONArray->Add(JSONBlue);
    // 색상 배열을 포함할 추가 오브젝트를 생성한다.
    TJSONObject* JSONObject = new TJSONObject();
    JSONObject->AddPair("colors", JSONArray);
    AResponse->Body->SetValue(JSONObject, True);
}

```

JSONWriter 사용하기

JSONWriter를 사용하면 프로그래밍 언어 클라이언트가 사용할 데이터를 전달하는 사용자 정의 JSON을 작성하여 RAD 서버 애플리케이션 개발을 간소화할 수 있다. JSONWriter를 사용하여 JSON 객체를 시작하고 속성 이름과 값을 작성한 다음 JSON 객체를 끝낼 때까지 속성 및 값을 계속 작성한다.

JSONWriter 사용 예제

다음은 JSONWriter의 WriteStartArray, WriteStartObject, WritePropertyName, WriteValue, WriteEndObject, WriteEndArray methods 메서드를 사용하여 데이터를 반환하는 Get 엔드포인트의 구현 예시이다. AResponse 인수에는 응답에 직접 더 복잡한 구조를 생성하는 데 매우 편리한 JSONWriter가 내장되어 있다.

델파이:

```
procedure TTestResource1.GetJSONWriter(const AContext: TEndpointContext; const
AResponse: TEndpointRequest; const AResponse: TEndpointResponse);
begin
  // 모든 줄에 AResponse.Body.JSONWriter를 입력하는 것을 피하기 위해 변수에 저장
  var Writer := AResponse.Body.JSONWriter;
  // JSON 오브젝트를 시작
  Writer.WriteStartObject;
  Writer.WritePropertyName('colors');
  // JSON 배열을 시작
  Writer.WriteStartArray;
  Writer.WriteStartObject;
  Writer.WritePropertyName('name');
  Writer.writeValue('red');
  // 필요한 만큼 WritePropertyName 및 WriteValue 문을 추가
  Writer.WritePropertyName('hex');
  Writer.writeValue('#ff0000');
  Writer.WritePropertyName('default');
  Writer.writeValue(False);
  Writer.WritePropertyName('customId');
  Writer.WriteNull;
  Writer.WriteEndObject;
  // 필요한 만큼 추가 JSON 오브젝트를 작성
  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);
  // JSON 오브젝트를 종료
```

```

    Writer.WriteEndObject();
    // JSON 배열을 종료
    Writer.WriteEndArray();
    Writer.WriteEndObject();
    end;
}

```

C++:

```

void TTestResource1::GetJSONWriter(TEndpointContext* AContext, TEndpointRequest*
ARequest, TEndpointResponse* AResponse)
{
    // 모든 줄에 AResponse.Body.JSONWriter를 입력하는 것을 피하기 위해 변수에 저장
    TJsonTextWriter* Writer = AResponse->Body->JSONWriter;
    // JSON 오브젝트를 시작
    Writer->WriteStartObject();
    Writer->WritePropertyName("colors");
    // JSON 배열을 시작
    Writer->WriteStartArray();
    Writer->WriteStartObject();
    Writer->WritePropertyName("name");
    Writer->WriteValue("red");
    // 필요한 만큼 WritePropertyName 및 WriteValue 문을 추가
    Writer->WritePropertyName("hex");
    Writer->WriteValue("#ff0000");
    Writer->WritePropertyName("default");
    Writer->WriteValue(False);
    Writer->WritePropertyName("customId");
    Writer->WriteNull();
    Writer->WriteEndObject();
    // 필요한 만큼 추가 JSON 오브젝트를 작성
    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);
    // JSON 오브젝트를 종료
    Writer->WriteEndObject();
    // JSON 배열을 종료
    Writer->WriteEndArray();
    Writer->WriteEndObject();
}

```

JSONBuilder 사용

이 프레임워크는 더 빠르고 읽기 쉬운 방식으로 JSON을 빌드할 수 있는 JSONWriter 래퍼이다. 이는 매우 복잡한 JSON 구조의 경우 코드를 간소화하고 유지 보수 및 읽기를 쉽게 만들어주는 fluent 인터페이스(또는 메서드 체이닝) 접근 방식을 따른다.

동일한 프로젝트 예제에서 JSON 빌더를 사용하여 응답을 만드는 또 다른 엔드포인트를 찾을 수 있다. 코드를 살펴보자:

델파이:

```
procedure TTestResource1.GetJSONBuilder(const AContext: TEndpointContext; const
ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
begin
  var Writer := AResponse.Body.JSONWriter;
  // 응답에서 빌더로 JSONWriter를 연결
  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;
// 응답에서 빌더로 JSONWriter를 연결
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;
}
}

```

RAD 스튜디오와 함께 제공되는 fmWorkBench라는 매우 유용한 샘플 프로젝트를 찾을 수 있다.(델파이에서만 사용이 가능하다). 다음 경로 안에서 찾을 수 있다:

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

참고 자료

- [JSON](#)
- [읽기 및 쓰기 JSON 프레임워크](#)
- [JSONBuilder](#)
- [WorkBench 샘플 프로젝트](#)
- [튜토리얼: REST 클라이언트 라이브러리를 사용하여 REST 기반 웹 서비스에 액세스하기](#)

07

사용자 지정 엔드포인트 만들기

• •

지금까지 배열, 오브젝트와 같은 기본적인 JSON 구조와 매우 간단한 URI를 살펴보았다: /customers, /sales...와 같이 ST API에서 하위 리소스 URI 및 JSON 응답 내에서의 배열 또는 오브젝트를 중첩하는 것은 일반적인 모범사례이다. 이 장에서는 RAD 서버를 사용하여 이러한 종류의 구조를 수행하는 방법을 살펴본다. 또한 GET, POST, PUT 또는 DELETE 메서드를 만드는 방법에 대해 설명한다.

모범 사례의 예

API를 원하는 방식으로 구조화할 수 있지만 표준화 또는 REST API와 관련된 모범 사례에 대한 설명 문서는 수천 개가 존재한다. 결국 API를 어떻게 구성할 것인지는 사용자에게 달려 있지만, 이러한 표준의 기본을 조금 읽어보는 것이 좋다.

예를 들어, 특정 고객에게 액세스할 때 **/customers/{id}** URI를 사용한다고 배웠다. 만일 해당 특정 고객의 매출에 액세스하려면 어떻게 해야 할까? 가장 흔한 옵션은 다음과 같이 엔드포인트를 정의하는 것이다: **/customers/{id}/sales**. 이 엔드포인트는 ID로 정의된 특정 고객의 매출 주문을 반환한다.

이는 모범 사례로 간주되며 일반적으로 중첩된 리소스 또는 하위 리소스라고 한다. 이렇게 하면 서드파티나 자체 개발팀이 API를 더 쉽게 이해하는 데 도움이 되는 엔드포인트간의 계층적 관계를 정의할 수 있다.

API의 잦은 통신을 제어하는 방법

애플리케이션을 개발하다 보면 양식/웹페이지에 액세스함과 동시에 여러 엔드포인트의 데이터가 필요한 경우가 종종 발생한다. 고객의 매출 주문에 액세스 한다고 가정해 보자: 고객의 세부 정보, 주문 정보, 배송 주소, 해당 주문의 라인, 결제, 송장 등의 정보가 필요할 것이다. 이러한 요청 목록은 상당히 길어질 수 있으며 REST API의

경우 요청 비용이 많이 발생한다는 문제가 있다. 이는 서버의 요청 및 처리 비용뿐만 아니라 인터넷의 지연 시간을 고려하여 비용이 많이 발생한다는 의미이다. 각 요청이 왔다 갔다 하는 데 몇 밀리초가 걸리며, 특히 한 페이지에 액세스하기 위해 10개 이상의 요청을 보내야 하는 경우 개선의 여지가 많다. 바로 이러한 경우에 중첩된 JSON 응답이 더욱 적합하다.

한 고객의 모든 매출에 대한 요약을 단 한 번의 요청으로 RAD 서버에 요청할 수 있다고 상상해 보자. 이는 기존의 마스터-디테일 관계와 동일할 수 있지만 하나의 요청으로 모두 반환된다는 차이를 가진다. 이를 달성할 방법도 살펴보자.

하위 리소스 추가하기

하위 리소스의 경우 이전 장에서 보았던 것과 동일한 TEMSDatasetAdapter를 계속 사용할 수 있다. 애트리뷰트에서 몇 가지만 조정하면 나머지 작업은 RAD 스튜디오와 FireDAC이 자동으로 처리한다.

지금까지 사용한 것과 동일한 프로젝트(쿼리 2개: qryCUSOTMER, qrySALES)를 사용하여 qrySALES의 SQL을 다음과 같이 수정해 보겠다:

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

그리고 drsSALES EMSDatasetAdapter 위에 있는 속성을 변경해 보겠다:

델파이

```
[ResourceSuffix('customers/{CUST_NO}/sales')]
[ResourceSuffix('List', './')]
[ResourceSuffix('Get', './{PO_NUMBER}')]
```

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};
```

SQL 구문에서 특정 고객의 매출을 필터링하는 파라미터가 있는 WHERE 절을 추가했다.

속성을 확인하면 더 흥미로운 일이 벌어진다. RAD 서버는 {CUST_NO}의 값을 자동으로 FireDAC 쿼리에 삽입하고 해당 고객의 매출을 필터링한다. 2개의 키가 동일한 엔드포인트에 관련되어 있고 작동하기 위해서는 이름을 지정해야 한다. 따라서 나머지 메서드(List, Get, Post 등)를 지정해야 했다. 좋은 소식은 EMSDatasetAdapter를 사용하여 생성된 다른 엔드포인트와 마찬가지로 이러한 엔드포인트를 사용하여 특정 매출을 생성, 수정 또는 삭제할 수 있다는 것이다.

응답에 중첩된 데이터 추가하기 (마스터/디테일)

이제 첫 번째 하위 리소스 엔드포인트가 생겼으니 여러 값이 포함된 중첩된 응답을 만들어 보겠다. 이를 위해 마지막으로 몇 가지 코드를 작성해야 한다.

TTestResource1 데이터 모듈 클래스에서 하나의 게시된 메서드를 만들어 보겠다.

델파이:

```

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

// 구현은 다음과 같다

procedure TTestResource1.GetCustomerDetails(const AContext: TEndpointContext; const ARequest: TEndpointRequest;
  const AResponse: TEndpointResponse);
begin
  var lCustomerNo := ARequest.Params.Values['CUST_NO'].ToInteger;
  // SQL 삽입 공격을 방지하기 위해 CustomerNo를 연결하는 대신 파라미터를 사용한다
  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}";

// 구현은 다음과 같다

void TTestResource1::GetCustomerDetails(TEndpointContext* AContext, TEndpointRequest* ARequest, TEndpointResponse* AResponse)
{
    int lCustomerNo = ARequest->Params->Values["CUST_NO"].ToInt();
    // SQL 삽입 공격을 방지하기 위해 CustomerNo를 연결하는 대신 파라미터를 사용한다
    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();
    }
}

```

이 간단한 코드를 보게 되면 매크로를 사용하여 URL에서 가져온 특정 고객 ID의 세부 정보를 검색하고 이를 기준에 사용하던 매크로를 이용하여 URL에서 가져온 특정 고객 ID의 세부 정보를 검색, 이를 이미 사용 중인 동일한 qryALES에 파라미터로 전달하고 있음을 알 수 있다. 추가 쿼리는 필요 없다.

EMSDatasetAdapter와 같은 컴포넌트를 사용하면 로우 코드 방식으로 많은 도움이 되지만, 때로는 특정 요구 사항이 필요하고 직접 구현해야 할 때도 있다. 이전 장에서 살펴본 것처럼 JSONWriters를 사용하여 원하는 대로 응답을 사용자 정의할 수 있다.

이 예제 코드를 확인하면 모든 수신 요청에 대해 ARequest 및 AResponse 인수에 액세스하여 필요한 모든 정보에 액세스하고 자체 응답을 작성할 수 있다. 이 예제에서는 "WriteStartObject" 메서드를 사용하여 새 오브젝트를 만든 다음 TQuerySerializer 클래스의 두 가지 메서드(나중에 자세히 설명하겠다)를 사용한 다음 오브젝트를 종료한다.

코딩 환경을 훨씬 더 쉽게 만들어주는 [JSONWriters 와 JSONReaders](#)에 사용할 수 있는 여러 가지 유용한 메서드와 프로퍼티가 있다. 사용할 수 있는 모든 기능을 배우고 싶다면 문서를 참조하기를 바란다.

이제 여러분도 궁금할 것이다: **ExcludeMasterFieldFromFields**와 **SerializeMasterDetail**이 2개의 메서드는 어떤 역할을 하는가? 이 두 가지 메서드는 특정 데모 예제를 위해 코딩되었지만, 깃허브 리퍼지토리 데모에서 이를 사용할 수 있으며 당연히 여러분의 프로젝트에서도 코드를 자유롭게 사용할 수 있다. 이 메서드의 작업은 해당 메서드에 잘 문서화 되어 있지만 요약하자면 마스터-디테일 사항 관계를 JSON 오브젝트로 반환하고 디테일 쿼리를 기본 JSON 오브젝트에 JSON 배열로 삽입하는 것이다. **ExcludeMasterFieldFromFields**가 필요하지 않을 수도 있지만 중복되는 데이터를 피하기 위해 세부 정보에서 마스터 필드를 제외한다.



팁

Data.DBJson 단위를 확인하라. 여기에는 데이터 집합을 JSON으로 변환하거나 그 반대로 변환하는 데 도움이 되는 여러 클래스가 포함되어 있다. 이 예제에서는 훨씬 빠르고 세밀하게 직렬화할 수 있는 TDataSetToJSONBridge 클래스를 사용했다.

델파이:

```
// 마스터/디테일 관계의 쿼리 2개가 주어지면, Detail 쿼리가 중첩된 배열이 있는 JSON 오브젝트 1개를 반환
function TTTestResource1.SerializeMasterDetail(AMasterDataset: TFDQuery;
ADetailDataset: TFDQuery; APropertyName: string; AFields: TStringList = nil): TJSONObject;
begin
  var 1Bridge := TDataSetToJSONBridge.Create;
  try
    // 마스터 쿼리의 현재 레코드를 가져와 JSON 오브젝트로 변환
    1Bridge.Dataset := AMasterDataset;
    1Bridge.IncludeNulls := True;
```

```

// 현재 레코드만 처리하도록 지정
lBridge.Area := TJSONDataSetArea.Current;
// 마스터 레코드를 JSON 결과에 오브젝트로 추가한다.
Result := TJSONObject(lBridge.Produce);

// 내보내고자 하는 필드 목록을 전달한 경우 해당 필드를 브리지에 할당하고, 그렇지 않은 경우 기본
동작은 쿼리의 모든 필드를 내보낸다.
if Assigned(AFields) then
  lBridge.FieldNameAssign(AFields);
// 동일한 브리지가 재사용되고 있지만, 이제 디테일 데이터 집합이 할당
lBridge.Dataset := ADetailDataset;
// 이 경우 쿼리의 모든 레코드가 처리
lBridge.Area := TJSONDataSetArea.All;
// 디테일 배열을 임시 배열에 저장하여 나중에 기본 오브젝트에 추가
var lJSONArray := TJSONArray(lBridge.Produce);
// 인자로 전달된 속성 명을 가진 배열로 메인 오브젝트에 추가
Result.AddPair(APROPERTYNAME, lJSONArray);
finally
  lBridge.Free;
end;
end;

// 쿼리에 마스터 필드가 할당된 경우 해당 마스터 필드를 제외한 모든 필드로 문자열 목록을 재조정
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++:

```

// 마스터/디테일 사항 관계의 쿼리 2개가 주어지면, Detail 쿼리가 중첩된 배열이 있는 JSON 오브젝트
1개를 반환
TJSONObject* TTestResource1::SerializeMasterDetail(TFDQuery* AMasterDataset, TFDQuery*
ADetailDataset, System::UnicodeString APROPERTYNAME, TStringList* AFields)
{
  TDataSetToJSONBridge *lBridge = new TDataSetToJSONBridge;
  try {
    // 마스터 쿼리의 현재 레코드를 가져와 JSON 오브젝트로 변환
    lBridge->Dataset = AMasterDataset;
    lBridge->IncludeNulls = True;

```

```

// 현재 레코드만 처리하도록 지정
lBridge->Area = TJSONDataSetArea::Current;
TJSONObject* lJSONObject = new TJSONObject;
// 마스터 레코드를 JSON 결과에 오브젝트로 추가
lJSONObject = (TJSONObject*) lBridge->Produce();

// 내보내고자 하는 필드 목록을 전달한 경우 해당 필드를 브리지에 할당하고, 그렇지
않은 경우 기본 동작은 쿼리의 모든 필드를 내보낸다
if (AFields != NULL) {
    lBridge->FieldNames->Assign(AFields);
}
// 동일한 브리지가 재사용되고 있지만, 이제 Detail 데이터 집합이 할당
lBridge->Dataset = ADetailDataset;
// 이 경우 쿼리의 모든 레코드가 처리
lBridge->Area = TJSONDataSetArea::All;
TJSONArray* lJSONArray = new TJSONArray;
// 디테일 배열을 임시 배열에 저장하여 나중에 기본 오브젝트에 추가
lJSONArray = (TJSONArray*) lBridge->Produce();
// 인자로 전달된 속성 명을 가진 배열로 메인 오브젝트에 추가
lJSONObject->AddPair(APROPERTYNAME, lJSONArray);
return lJSONObject;
} __finally {
    lBridge->Free();
}
}

// 만일 쿼리에 마스터 필드가 할당된 경우 해당 마스터 필드를 제외한 모든 필드로 문자열 목록을 재조정
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;
}

```



실제 프로젝트에서는 이러한 메서드를 쉽게 재사용할 수 있으므로 다른 클래스/유닛에서 추상화하는 것이 더 합리적이지만, 단순함을 위해 동일한 DataModule에 유지했다.

새 구현 테스트

데모 프로젝트를 실행하고 URL <http://localhost:8080/customers/1040/sales/>에 액세스해 보겠다.

```

[{"PO_NUMBER": "V91E0210", "CUST_NO": 1004, "SALES_REP": 11, "ORDER_STATUS": "shipped", "ORDER_DATE": "20100304T000000.000", "SHIP_DATE": "20100305T000000.000", "PAID": "y", "QTY_ORDERED": 10, "TOTAL_VALUE": 5000, "DISCOUNT": 0.1, "ITEM_TYPE": "hardware", "AGED": 1}, {"PO_NUMBER": "V92E0340", "CUST_NO": 1004, "SALES_REP": 11, "ORDER_STATUS": "shipped", "ORDER_DATE": "20111016T000000.000", "SHIP_DATE": "20111017T000000.000", "DATE_NEEDED": "20111018T000000.000", "PAID": "y", "QTY_ORDERED": 7, "TOTAL_VALUE": 70000, "DISCOUNT": 0, "ITEM_TYPE": "hardware", "AGED": 1}]
  
```

하위 리소스 접근 방식을 사용하여 특정 고객의 매출에 액세스

이제 고객 1004의 매출을 필터링하고 있다. 이때 하나의 매출에 대한 액세스/수정/삭제를 수행하고 싶다면 동일한 URI를 통해 액세스할 수도 있다. 예를 들어 주문 ID(예: <http://localhost:8080/test/customers/1004/sales/V91E0210>)를 끝에 추가하기만 하면 된다.

이제 정의한 다른 엔드포인트(<http://localhost:8080/test/customers-details/1004>)에 액세스해 보겠다.

```

{
  "CUST_NO": "1004",
  "CUSTOMER": "Bank of Manchster",
  "CONTACT_FIRST": "Elizabeth",
  "CONTACT_LAST": "Brocket",
  "PHONE_NO": "+44612119988",
  "ADDRESS_LINE1": "66 Lloyd Street",
  "ADDRESS_LINE2": null,
  "CITY": "Manchester",
  "STATE_PROVINCE": null,
  "COUNTRY": "England",
  "POSTAL_CODE": "M2 3LA",
  "ON_HOLD": null,
  "SALES": [
    {
      "PO_NUMBER": "V91E0210",
      "SALES REP": 11,
      "ORDER_STATUS": "shipped",
      "ORDER_DATE": "2010-03-04T00:00:00.000Z",
      "SHIP_DATE": "2010-03-05T00:00:00.000Z",
      "DATE_NEEDED": null,
      "PAID": "y",
      "QTY_ORDERED": 10,
      "TOTAL_VALUE": 5000,
      "DISCOUNT": "0.10000001490116",
      "ITEM_TYPE": "hardware",
      "AGED": "2"
    },
    {
      "PO_NUMBER": "V92E0340",
      "SALES REP": 11,
      "ORDER_STATUS": "shipped",
      "ORDER_DATE": "2011-10-16T00:00:00.000+01:00",
      "SHIP_DATE": "2011-10-17T00:00:00.000+01:00",
      "DATE_NEEDED": "2011-10-18T00:00:00.000+01:00",
      "PAID": "y",
      "QTY_ORDERED": 7,
      "TOTAL_VALUE": 70000,
      "DISCOUNT": "0",
      "ITEM_TYPE": "hardware",
      "AGED": "1"
    }
  ]
}

```

하위 리소스(고객 및 해당 매출)로 엔드포인트에 액세스

동일한 요청에서 특정 고객의 모든 매출을 가져오기 때문에 RAD 서버를 두 번 호출하는 대신 단 한 번으로 필요한 모든 정보를 얻을 수 있다. 물론 이러한 종류의 요청은 여러 개의 중첩된 레벨 등으로 인해 더 복잡해질 수 있다.



참고

이 장과 관련된 깃허브 리퍼지토리 [데모 프로젝트](#)에는 고객 및 관련 매출 목록에 액세스할 수 있는 추가적인 엔드포인트가 있다. 특별히 하나를 필터링하지 않고 모두 가져올 수 있을 것이다. 대부분의 코드가 재사용되었기 때문에 추가 개발에서 매우 간단하게 사용할 수 있다.

사용자 정의 GET, POST, PUT, DELETE 메서드 구현

지금까지 GET 사용자 정의 메서드를 만드는 방법을 살펴보았지만 일부 시나리오에서는 POST, PUT, DELETE와 같은 다른 동사를 사용해야 할 수도 있다. 이러한 종류의 구현을 코딩하기 위해선 사용하려는 동사로 메서드 이름을 시작하면 된다. (예: **procedure PutMethodName(...)**) 이전 예제에서 보았지만, 모든 사용자 정의 메서드는 "Get"으로 시작했는데, 이를 "Post"로 변경하면 POST 메서드를 정의하게 된다. 예제를 살펴보자:

델파이:

```

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

// 구현은 다음과 같다
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;
  // 추가적인 비즈니스 로직을 추가
  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}";

// 구현은 다음과 같다.

void TTestResource1::PostCustomEndPoint(TEndpointContext* AContext, TEndpointRequest*
AResponse, TEndpointResponse* AResponse)

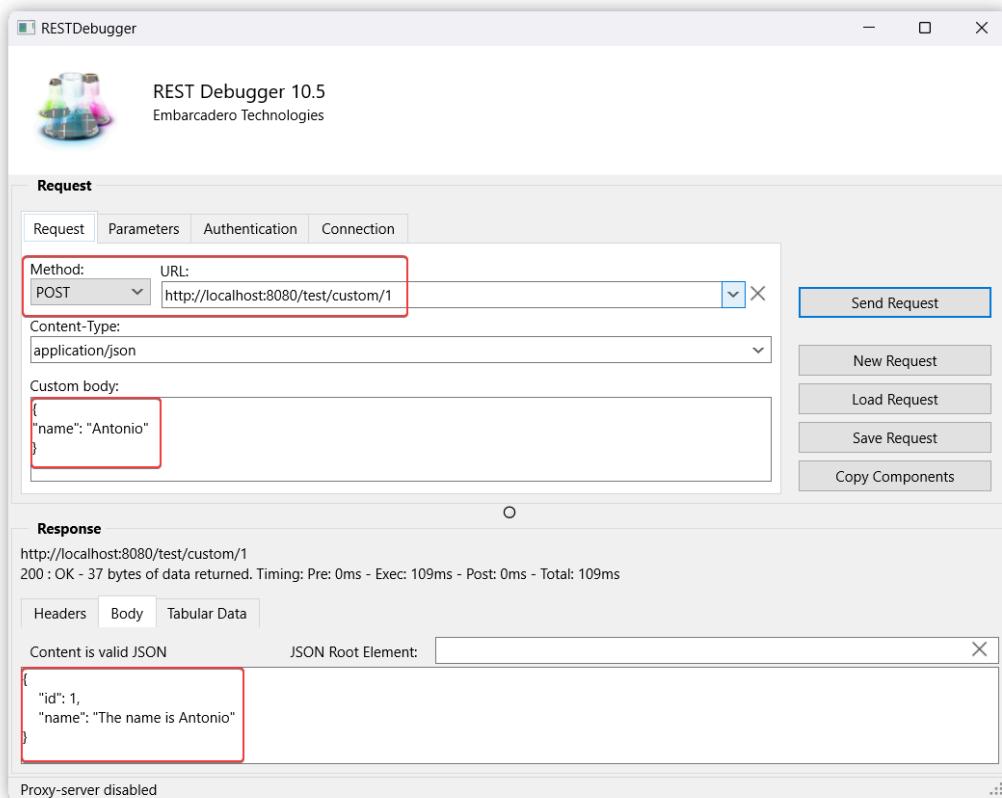
```

```

{
    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();
    // 추가적인 비즈니스 로직을 추가
    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();
}

```

이제 새로운 추가 엔드포인트 ./custom/{id}를 사용할 수 있게 되었다. REST 디버거에서 예상되는 "name" 속성을 본문에 추가하여 POST 요청을 보내면 이 값을 받게 된다.



사용자 정의 POST 메서드의 응답

응답 오류 처리하기

앞선 사용자 지정 POST 엔드 포인트 예제에서 볼 수 있듯, 예상한 모든 데이터를 얻지 못한 경우 오류가 발생하였다. RAD 서버는 가장 일반적인 오류를 AResponse 오브젝트에서 바로 발생시키고 반환하는 내장된 솔루션을 제공한다. [이 링크](#)에서 확인할 수 있는 오류에 대한 자세한 정보를 확인할 수 있다.

참고 자료

- [JSON 읽기 및 쓰기](#)
- [REST API 모범 사례](#)

08

내장된 분석도구에 액세스



RAD 서버 콘솔은 RAD 서버 엔진의 분석뿐만 아니라 여러 데이터를 표시하는 사전 구성된 웹 애플리케이션을 제공하는 서비스다. 이를 통해 RAD 서버 인스턴스의 활동을 더욱 심층적으로 파악하고 실제 데이터를 기반으로 의사 결정을 내릴 수 있다. 사용자, API 및 서비스 활동을 분석하여 애플리케이션이 어떻게 활용되고 있는지에 대한 통찰력을 얻을 수 있다.

주요 특징

RAD 서버 콘솔은 읽기 전용 모드로 데이터베이스 서버에 액세스한다.

- RAD 서버 엔진 리소스의 통계와 함께 API 호출에 대한 피드백을 제공한다: 사용자, 그룹, 설치, 모듈 및 해당 리소스.
- 콘솔을 테스트 목적으로 독립 실행형 애플리케이션으로 사용하거나 프로덕션 환경을 위해 마이크로소프트 IIS 서버에 콘솔을 설정할 수 있다.
- 참고: 리눅스에서는 마이크로소프트 IIS 서버를 사용할 수 없다. 대신 리눅스의 프로덕션 환경에서 아파치를 사용할 수 있다.
- RAD 서버 콘솔은 서버의 기능을 확장하여 새 리소스에 대한 분석을 제공한다.
- RAD 서버 콘솔은 등록된 RAD 서버 사용자에 대한 분석을 제공한다.
- 분석 데이터를 내보내고 시스템에 .csv 파일로 저장할 수 있다.

RAD 서버 콘솔에 액세스

RAD 서버 개발 서버로 돌아가서 콘솔 열기 버튼을 클릭한다. 그러면 포트 8081에서 자동으로 실행되는 RAD 서버 개발 콘솔 서버가 시작되고 분석 콘솔 로그인 창이 있는 브라우저도 열리게 된다.

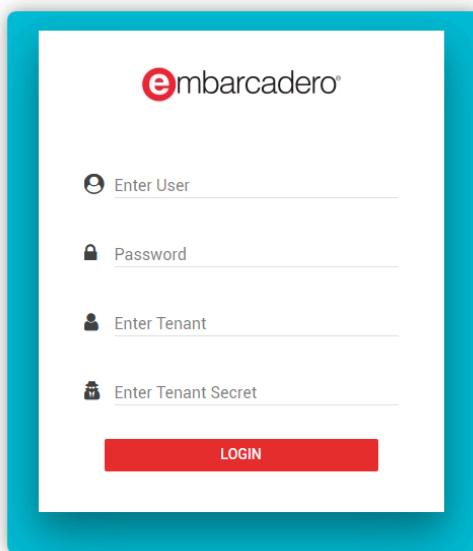


RAD 서버 개발 콘솔 서버 UI



만약 기본 포트 8081이 컴퓨터에서 사용 중이라면, 컴퓨터에서 사용할 수 있는 다른 포트로 8081을 변경하고 "시작"을 누른 후 "브라우저 열기"를 누르면 된다.

참고



RAD 서버 콘솔 사용자 로그인 화면

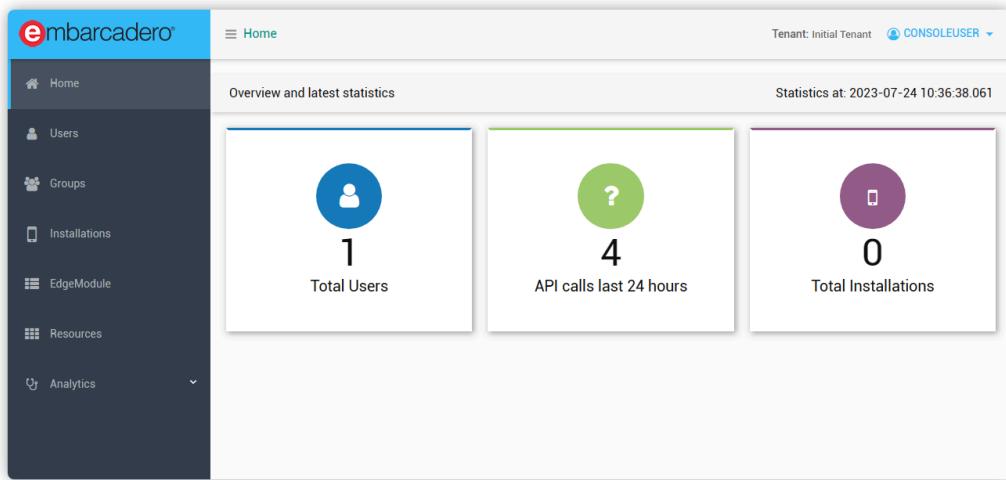
콘솔에 액세스하기 위한 RAD 서버의 기본 자격 증명 정보는 다음과 같이 제공된다(tenant 정보는 비워 두어야 한다).

user: **consoleuser**

password: **consolepass**



RAD 서버는 콘솔에 액세스할 수 있는 기본 사용자 및 암호를 제공한다. 이 자격 증명을 emsserver.ini 구성 파일에서 변경해야 한다(자세한 내용은 이 구성 파일에 대한 장을 확인하라).



RAD 서버 콘솔 홈페이지

로그인하면 왼쪽에 메뉴가 있고 오른쪽에 콘텐츠가 있는 RAD 서버 콘솔 단일페이지 JavaScript 앱의 그래픽 보기다. 메뉴는 사용자, 그룹, 디바이스 설치, 엣지 모듈, 리소스 모듈 및 분석에 대한 정보를 제공한다. 다음은 사용자 목록과 사용자 생성 시기 및 사용자 정보가 마지막으로 수정된 시기를 포함한 사용자 정보를 표시하는 화면이다.

Users Table				
userid	username	created	lastmodified	creator
3A25B7B0-1033-4B8B-A77E-EFB25B5B75CD	test	2023-07-11T17:24:30.000+01:00	2023-07-11T17:24:30.000+01:00	3A25B7B0-1033-4

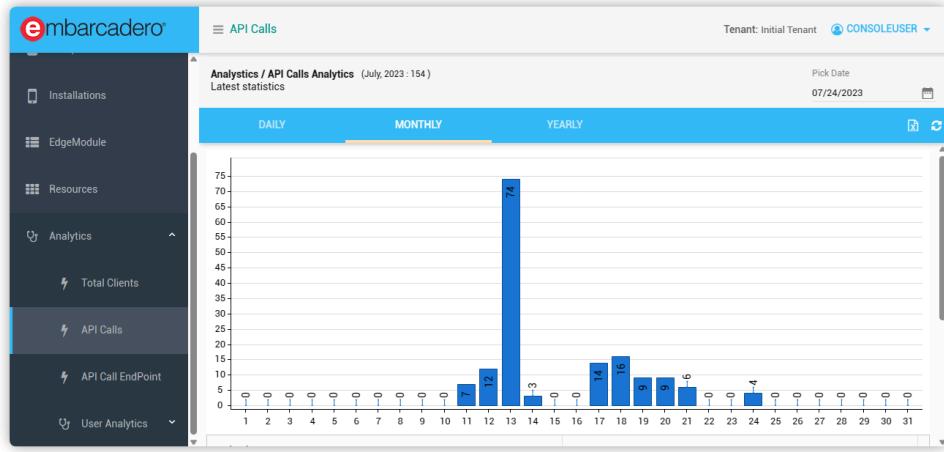
RAD 서버 콘솔 사용자 표

애널리틱스 메뉴 항목을 클릭하면 총 클라이언트, API 호출 횟수, 호출된 API 엔드포인트 등의 다양한 분석 항목을 선택할 수 있는 메뉴가 나온다. 분석은 연도, 월, 일별로 선택할 수 있다. 사용자, 그룹 등 특정 엔드포인트를 기준으로 분석을 필터링할 수도 있다. 만일 분석 결과를 외부 애플리케이션으로 추가적인 처리가 필요하다면 .csv 파일로 저장할 수도 있다.



이러한 분석은 의사 결정 과정 및 감사에 대한 훌륭한 정보를 제공한다. 서비스가 언제 가장 많이 또는 적게 사용되는지 파악하여 업데이트를 계획하거나, 어떤 엔드포인트가 거의 사용되지 않는지 확인하는 작업은 매우 가치 있는 통찰력의 예시이다.

다음 차트는 선택한 월의 API 호출 차트를 보여준다.



RAD 서버 콘솔 확장 JS API 호출 분석 페이지

09

RAD 서버 배포



지금까지는 개발 버전의 RAD 서버(EMSDevServer.exe) 및 콘솔(EMSDevConsole.exe) 애플리케이션을 사용하여 첫 번째 RAD 서버 애플리케이션을 테스트했다. 이 장에서는 프로덕션 환경에서 RAD 서버를 배포할 수 있는 여러 플랫폼에 대해 다룬다. RAD 서버 라이트에 관심이 있는 경우 다음 장으로 넘어가기 바란다.



경고

새로운 bpl 또는 dcp 리소스는 컴파일 되면 프로젝트의 "export" 폴더(일반적으로 바이너리가 있는 곳)에 포함되지 않는다. 이러한 리소스는 기본적으로 엠바카데로 스튜디오 설치 경로에 생성된다.

C:\Users\Public\Documents\Embarcadero\Studio\0\“Bpl 또는 Dcp”
Bpl 또는 Dcp 폴더 안에는 플랫폼별 폴더가 있다.

RAD 서버를 배포할 수 있는 위치

RAD 서버는 [윈도우](#), [리눅스](#) 및 [도커](#) 플랫폼과 호환된다. 개념적으로 각 플랫폼에 필요한 서비스는 동일하지만, 이 장에서 주의 깊게 살펴볼 몇 가지 차이점이 있다. 우선 유사점과 RAD 서버의 내부 작동 방식에 대해 이야기 하겠다.

겟잇의 설치 프로그램 사용

윈도우 또는 리눅스에 RAD 서버 애플리케이션을 배포하는 경우 가장 빠르게 설치하는 방법은 [겟잇](#)에서 다운로드할 수 있는 설치 프로그램을 사용하는 것이다. "RAD 서버"를 검색하면 이 두 가지를 찾을 수 있다:



겟잇의 RAD 서버 설치 프로그램

"설치"가 완료되면(다운로드에 더 가깝다) 다음 경로에서 설치 프로그램을 찾을 수 있다: C:\Users\\Documents\Embarcadero\Studio\0\CatalogRepository 경로에서 전체 폴더를 프로덕션 시스템에 복사한다. 이 폴더에는 RAD 서버 전체 설치에 필요한 모든 파일이 포함되어 있다(인터베이스 설치 프로그램 포함).

프로덕션 환경에서 설치 관리자를 실행하기 전에 설치 관리자가 모든 요구사항을 적절히 구성할 수 있도록 IIS 또는 아파치를 설치해야 한다.

설치 관리자는 설치에 필요한 다양한 옵션을 안내한다.



설치 중 인터베이스에 대한 유효한 라이선스를 제공하라는 메시지가 표시된다. EDN 계정과 RAD 서버 일련번호를 사용하여 인터베이스 인스턴스를 등록한다.

RAD 서버를 수동으로 배포하기 위한 전제 조건

이 장에서는 RAD 서버를 배포하기 위해 설치 또는 구성해야 하는 모든 부분을 이해하는 데 중점을 둔다. 설치 관리자를 사용하더라도 더 나은 디버깅 및 문제 해결을 위해 모든 요구 사항을 이해하는 것이 중요하다. 또한 RAD 서버를 최신 버전으로 업데이트해야 하는 경우 전체 애플리케이션을 다시 설치할 필요는 없으며 몇 개의 dll 및 bpls/so만 업데이트하는 것으로 충분하다.

이는 프로덕션 환경에서 작동하는 RAD 서버 설치를 위한 필수 요구 사항이다:

- InterBase Server 엔진
- RAD 서버 라이선스
- RAD 서버 설치
- 웹 서버(IIS 7+ 또는 아파치 2.4+)
- RAD 스튜디오로 컴파일된 리소스 파일
- EMServer.ini 파일 구성

선택한 플랫폼에 관계없이 배포를 위해 이러한 모든 것을 설치/구성해야 한다. 예를 들어, 윈도우에서는 마이크로소프트의 웹 서버(IIS 또는 윈도우용 Apache)를 구성해야 하며 리눅스에서는 아파치가 필요하다. RAD 서버는 실행 파일이 아니라는 점을 이해하는 것이 중요하다(라이트 버전은 제외하는데, 이는 나중에 자세히

설명한다). 리소스는 윈도우의 경우 BPL 또는 리눅스의 경우 so 라이브러리 형태로 컴파일 된다. 그렇기 때문에 이러한 리소스에 액세스하기 위해서는 웹 서버가 필요하다.

인터베이스의 경우, 데이터베이스 인스턴스는 RAD 서버 내부에서 사용해야 한다. 통계, 사용자, 역할 등 많은 정보가 저장되는데, 이 모든 정보를 저장하기 위해 자체적인 데이터베이스가 필요하다.

RAD 서버가 내부적으로 인터베이스를 사용한다고 해서 자체 데이터를 반드시 이 데이터베이스에 사용해야 한다는 의미는 아니다. FireDAC은 다양한 데이터베이스에 연결할 수 있으며 필요에 따라 원하는 데이터베이스를 선택할 수 있다.



참고

인터베이스를 선택한 데이터베이스가 동일한 컴퓨터에서 배포될 경우, 서로 다른 포트에서 실행되는 두 개의 인스턴스가 필요하다. 일반적으로 자체 데이터베이스 인스턴스에는 포트 3050을 유지하고 다른 포트에 RAD 서버 인터베이스 인스턴스를 설치한다. 예를 들어, 3051 포트를 사용할 수 있을 것이다. RAD 서버는 자체 암호화 시스템을 사용하므로 동일한 인스턴스를 사용할 수 없다.

Windows에 수동으로 배포

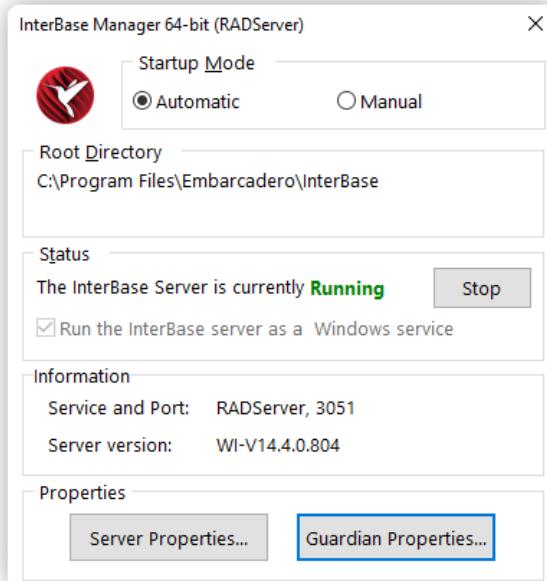
인터베이스 서버 엔진

<https://my.embarcadero.com>에서 최신 윈도우 버전 인터베이스 설치 관리자를 다운로드하여 프로덕션 시스템에 설치한다. 이전에 설치한 적이 없는 경우 [이 튜토리얼을 따라 할 수 있다](#). 여기에서 [윈도우 프로덕션 환경을 위한 RAD 서버 데이터베이스 요구 사항](#)도 확인할 수 있다.

설치에 대한 구체적인 세부 사항:

- "서버 및 클라이언트"를 선택한다
- 동일한 컴퓨터에서 여러 개의 인터베이스 인스턴스 실행을 허용한다
- 제안된 기본 포트를 3051로 변경한다
- 인스턴스 이름을 기본 gds_db 대신 RADserver로 변경한다
- 인터베이스를 등록할 때 제공받은 것과 동일한 RAD 서버 일련번호와 EDN 계정을 사용한다.

설치가 완료되면 인터베이스 서버의 RADServer 인스턴스를 시작해야 한다. 시작 | 프로그램 | 엠바카데로 인터베이스 | 64-비트 인스턴스 = RADServer | 인터베이스 서버 관리자를 선택한다. 인터베이스를 기본 값으로 실행하기 위해서는 해당 확인란을 선택한다. 컴퓨터가 시작될 때 인터베이스가 실행되도록 하기 위해서는 "자동" 라디오 버튼을 클릭한다. 그런 다음 시작 버튼을 클릭한다.



RADServer용 인터베이스 관리자 64비트



팁

인터베이스 매니저는 프로그램 목록에서 "엠바카데로 인터베이스" 아래에서 찾을 수 있거나, 설치한 경로에서 ".\bin\IBMgr.exe" 폴더 아래에서 찾을 수 있다. 인스턴스 이름을 지정하기 위해서는 ".\IBMgr.exe RADServer"와 같이 지정하라. 또 다른 옵션으로는 윈도우 검색을 사용하여 "인터베이스 매니저"를 입력하는 것이다.

RAD 서버 설치

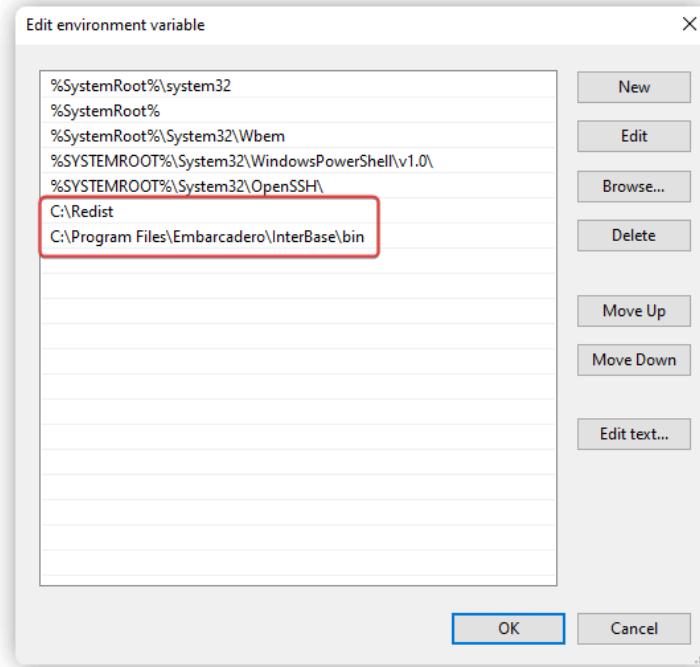
윈도우 시스템에 RAD 서버를 설치하기 위해서는 개발을 위해 컴퓨터를 구성한 방식과 매우 유사한 단계를 따른다. 이 프로세스에 필요한 대부분의 파일은 다음 폴더에서 찾을 수 있다:

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

Docwiki에는 윈도우에 RAD 서버를 설치하는 방법에 대해 매우 자세한 튜토리얼이 나와 있다. [여기에서 액세스할 수 있다](#). 그러니 여기에서는 기본적인 단계만 설명하겠다:

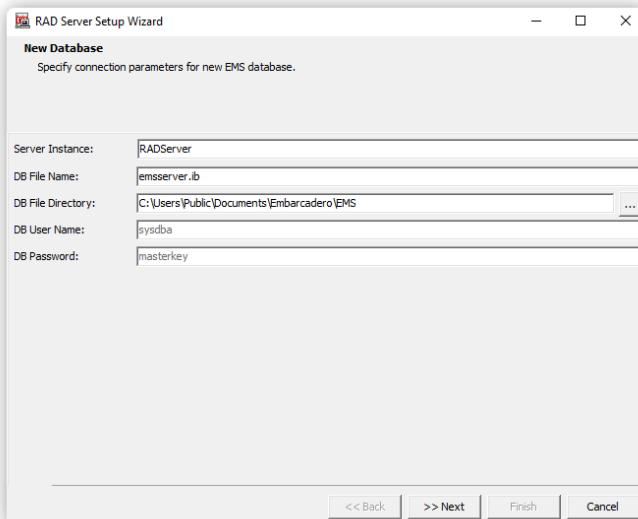
다음 단계에 따라 프로덕션 서버를 테스트할 준비를 하고 인터베이스 RAD 서버 인스턴스와 EMSDevServer.exe를 사용하여 RAD 서버 데이터베이스 및 구성 파일을 생성한다.

1. 64-비트 EMSDevServer.exe를 c:\installs\EMS 폴더에 복사한다.
2. RAD 스튜디오 Redist/win64 폴더에서 필요한 파일을 c:\Redist 폴더에 복사한다.
3. 프로덕션 서버에서 시스템 경로 환경 변수를 편집하여 c:\Redist 와 c:\Program Files\InterBase\bin 폴더를 추가한다.

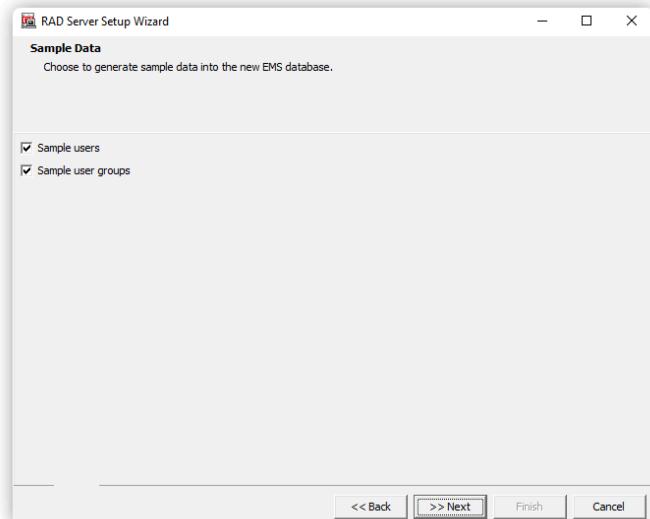


시스템 경로에 두 개의 폴더 추가

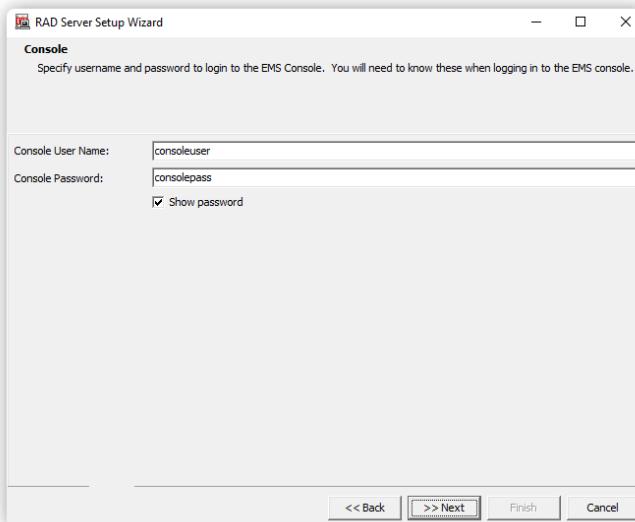
4. 개발 환경에 있는 컴퓨터의 EMS 템플릿 및 웹 리소스 파일을 C:\Program Files (x86)\Embarcadero\Studio\0\ObjRepos\en\EMS에서 생산 서버 폴더인 c:\installs\ObjRepos\EMS로 복사한다(EMSDevServer.exe는 템플릿 및 웹 리소스 파일을 EMSDevServer 폴더를 배치한 상위 폴더 아래의 ObjRepos\EMS 하위 폴더에서 찾는다).
5. 프로덕션 서버에서 RAD 서버 라이선스가 있는 인터베이스 서버가 시작되었는지 확인한다.
6. 첫 번째 RAD 서버 개발 구성에서와 같이 EMSDevServer.exe를 실행하여 프로덕션 RAD 서버 구성 파일 및 인터베이스 RAD 서버 데이터베이스를 설정한다. 다음 화면은 각 단계를 보여준다.



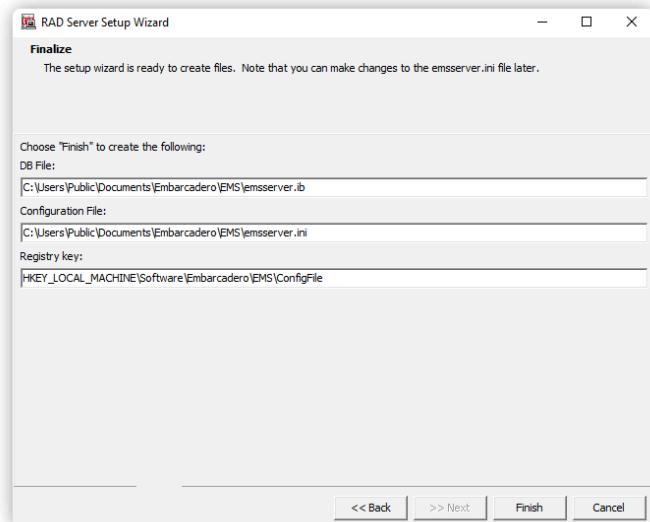
RAD 서버 설정 마법사 - RAD 서버에 대한 연결
파라미터 설정



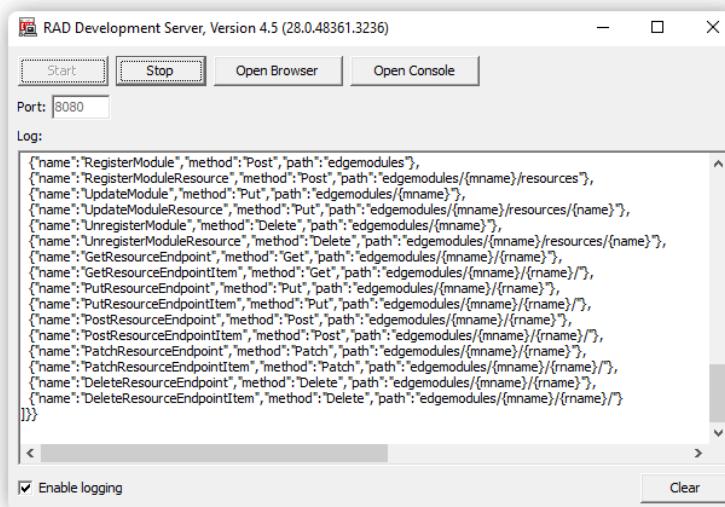
RAD 서버 설치 마법사 - 샘플 데이터 생성 선택



RAD 서버 설정 마법사 - RAD 서버에 대한 연결
파라미터 설정

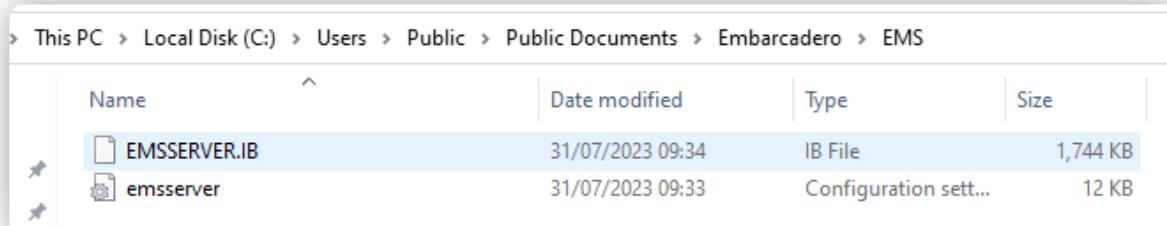


RAD 서버 설치 마법사 - 생성될 파일 및 레지스트리 키를
검토한다.



RAD 서버 개발 서버가 실행 중이다.

RAD 서버 마법사는 공용문서 폴더 아래의 기본 폴더에 두 개의 파일을 만든다.



RAD 서버 설치 마법사 - 공용문서 폴더에 생성된 두 개의 파일

웹 서버(IIS 또는 Apache)

윈도우에서 배포하는 경우 마이크로소프트에서 윈도우와 함께 제공하는 웹 서버(일명 IIS)를 사용하거나 윈도우용 아파치를 사용할 수 있다. [이 링크](#)에서 프로덕션 환경에 복사해야 하는 파일뿐만 아니라 윈도우 시스템에서 IIS 또는 아파치를 구성하는 방법에 대한 자세한 가이드를 볼 수 있다.

IIS를 선택하는 경우 마이크로소프트의 버전이 다르므로 서비스 설치 프로세스가 약간 다를 수 있다. 이 서비스를 사용해 본 경험이 없다면 [이 링크](#)에서 정보를 찾아보길 바란다.

마지막 단계는 컴파일된 리소스를 복사하는 것이다. 이 장의 끝부분에서 복사하는 방법을 확인할 수 있다.



**서버 관리자에서 "역할 또는 기능 추가" 옵션을 사용하여 웹 서버(IIS)를 추가하는 경우
"애플리케이션 개발" 섹션 아래에 "ISAPI 확장" 및 "ISAPI 필터"를 반드시 설치해야 한다.**

리눅스에서 수동으로 배포

RAD 서버 및 애플리케이션을 리눅스 서버에 배포하면 다음과 같은 옵션이 제공된다:

- 독립 실행형 RAD 서버를 만들려면 RAD 서버 설치 섹션을 참조 바란다.
- 아파치용 RAD 서버를 만들려면 아파치용 RAD 서버 설정 섹션을 참조 바란다.

호환되는 배포판

RAD 서버는 공식적으로 우분투 18 이상 및 RHEL 7 이상을 지원한다. 그렇다고 해서 록키 리눅스, 데비안 등 다른 배포판에 설치할 수 없다는 의미는 아니지만 내부 테스트에서는 항상 공식적으로 지원되는 배포판으로 수행된다.

인터베이스 서버 엔진 설치

<https://my.embarcadero.com>에서 최신 Linux용 InterBase 설치 프로그램을 다운로드할 수 있다. 압축 파일 안에 설치 프로그램이 있다. 여기에서 리눅스 [프로덕션 환경을 위한 RAD 서버 데이터베이스 요구 사항](#)도 확인할 수 있다.

다운로드한 파일의 압축을 풀고 설치 프로그램에 실행 권한을 할당한 뒤 실행한다.

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

설치에 대한 구체적인 세부 사항:

- "서버 및 클라이언트"를 선택한다
- 동일한 컴퓨터에서 여러 개의 인터베이스 인스턴스 실행을 허용한다
- 제안된 기본 포트를 3051로 변경한다
- 인스턴스 이름을 기본 gds_db 대신 RADserver로 변경한다
- 설치 폴더: /opt/interbase

**팁**

인터베이스 설치 관리자는 리눅스 설치에 데스크탑 환경이 있는지를 자동으로 감지한다. 콘솔 모드로 강제로 설치 프로그램을 실행하기 위해서는 이 인수를 사용한다.

```
sudo ./install_linux_x86_64.sh -i Console
```

**참고**

인스턴스 이름과 경로를 원하는 이름으로 정의할 수 있다. 그렇게 하는 경우 구성 프로세스 중에 해당 이름을 참조해야 한다.

인터베이스 서버 등록 및 시작

등록 마법사를 시작하려면 다음 명령을 실행한다.

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

그러면 라이선스 마법사가 시작된다. 콘솔 모드의 경우 옵션 2 "직접 등록"을 권장하며, 여기서 RAD 서버 일련번호와 EDN 계정을 지정할 수 있다. 나머지 작업은 어시스턴트가 수행하며 라이선스가 엠바카데로 서버에 연결되는지 확인한다.

라이선스가 올바르게 로드되었는지 바로 확인하려면 이전 메뉴의 "라이선스 나열"에서 옵션 1을 사용, 모든 것이 제대로 진행되었는지 확인할 수 있다.

인터베이스 인스턴스는 이미 설치되어 있고 라이선스가 적용되었어도 다시 시작 해 줄 필요가 있다. 이를 위해 다음 명령을 실행하여 인터베이스 콘솔에 들어가야 한다.

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

다른 애플리케이션과 서비스를 인터베이스 데이터베이스에 연결하는 작업을 간소화하기 위한 가장 간단한 방법은 인터베이스 라이브러리에 대한 심볼릭 링크를 만들고, 이를 /usr/lib를 가리키게 하는 것이다. 이렇게 하면 인터베이스 연결이 필요한 모든 서비스에 라이브러리를 복사할 필요가 없다.

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

인터베이스 서비스 실행

인터베이스를 서비스로 설정하여 리눅스가 시작될 때 실행되도록 할 수도 있다. 터미널 창에서 다음 명령을 사용한다.

인터베이스를 설치한 경로의 "예제" 폴더에 액세스하여 ibserverd 스크립트 파일을 설치한 서버 인스턴스용 버전으로 복사한다:

```
sudo cp ibserverd ibserverd_RADServer
```

위의 스크립트를 'sudo' 또는 'root'로 실행하여 자동 서비스 시작을 설정한다.

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

두 번째 인수는 설치 폴더, 세 번째 인수는 인스턴스 이름이다. 라이선스가 제대로 부여된 경우, 시스템을 재시작하면 서비스가 자동으로 시작된다.

다음 재부팅 또는 시작 시 인터베이스가 서비스로 시작되도록 설정되어 있는지 확인해 보자.

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

인터베이스를 서비스로 실행하면 컴퓨터가 다중 사용자 모드로 실행될 때마다 인터베이스 서버가 자동으로 시작된다.

서비스를 수동으로 생성하려는 경우(또는 리눅스 배포에서 약간 다른 접근 방식을 사용하는 경우) [이 링크](#)에서 해당 설정에 대한 자세한 정보를 찾을 수 있다.



참고

인터베이스를 서비스에서 제거하기 위해서는 다음을 실행한다:

```
sudo /opt/interbase/examples/ibservice.sh -r[emove]
```

RAD 서버 설치

RAD 스튜디오가 설치된 컴퓨터의 경로에서 RAD 서버 리눅스 설치 관리자를 찾을 수 있다: C:\Program Files (x86)\Embarcadero\Studio\0\EMSServer

해당 파일을 리눅스 시스템에 복사하고, 설치 프로그램을 실행한다. 실행 권한을 부여해야 할 수도 있다.



경고

libcurl이 설치되어 있는지 확인하라. 설치하려면 배포 패키지 매니저를 사용한다. Debian 기반 시스템에서는 다음 명령어를 사용하여 libcurl4를 설치할 수 있다: apt install libcurl4

설치 셸 스크립트는 명령 파일을 실행하는 데 필요한 여러 런타임 라이브러리(.so) 파일과 함께 EMSDevServerCommand, EMSDevConsoleCommand를 포함하는 디렉토리 /usr/lib/ems를 생성한다. [이 링크](#)의 Docwiki에서 자세한 정보가 포함된 튜토리얼을 찾을 수 있다.

설치가 완료되면 설정 모드에서 EMSDevConsole을 실행한다:

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

시작을 입력하고 엔터 키를 누른다.

다음 값을 입력하여 연결 파라미터를 지정한다:

- 서버 인스턴스: 기본 인스턴스 이름 **RADServer**
- DB 파일 이름: 기본 이름 **emsserver.ib**
- DB 파일 디렉터리: **/user/lib/ems**
- DB 사용자 이름: 기본 파라미터 **sysdba**
- DB 패스워드: 기본 파라미터 **masterkey**
- 콘솔 사용자 이름: 기본값 **consoleuser**
- 콘솔 패스워드: 기본값 **consolepass**

구성 옵션이 올바른 경우 "n"을 입력한다. emsserver.ini 및 emsserver.ib 파일이 생성되고 포트 8080에서 RAD 서버가 실행을 시작한다. 구성 파일은 언제든지 수동으로 편집할 수 있다.

구성 프로세스가 완료되면 **/usr/lib/ems**에서 RADServer 데이터베이스를, **/etc/ems**에서 구성 파일을 찾을 수 있다.

이제 개발자 서버를 실행한 상태에서 올바르게 액세스할 수 있는지, 응답을 받는지 테스트해 보겠다. <http://:8080/version>에 접속한다.



브라우저에서 버전 엔드포인트 호출의 출력을 표시하는 모습

EMSDevServerCommand와 EMSDevConsoleCommand는 아파치를 사용하지 않고도 Linux RAD 서버 애플리케이션을 개발하고 테스트하는 데 사용할 수 있다. 다음 단계는 리눅스 및 아파치 프로덕션 모드에서 실행되도록 RAD 서버 및 Delphi/C++로 컴파일된 애플리케이션 모듈을 설정하고 테스트하는 것이다.



리눅스에서 RAD 서버를 배포하면서 데이터 데이터베이스로 인터페이스도 사용하려는 경우 [이 튜토리얼](#)을 따르길 바란다.

아파치용 RAD 서버 설정

인터베이스 iSQL 명령(/opt/interbase/bin 디렉터리에서)을 사용하여 RAD 서버가 emsserver.ib 데이터베이스 파일에 연결할 수 있는지 확인한다.

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

아파치 RAD 서버(libmod_emsserver.so) 및 아파치 RAD 서버 콘솔(libmod_emsconsole.so) 모듈을 불러오도록 아파치 HTTP 서버를 구성한다. 아파치의 구성은 사용 중인 리눅스 배포판에 관계없이 매우 유사하지만 RHEL과 Debian 기반 배포판에는 몇 가지 차이점이 있다는 점에 유의하라.



모듈을 불러오고 위치 태그를 정의하는 권장 방법을 확인하기 위해서는 사용 중인 리눅스 배포판의 설명서를 확인하라.

참고

다음 줄을 추가하여 RAD 서버 아파치 서버 모듈(libmod_emsserver.so)과 RAD 서버 아파치 콘솔 모듈(libmod_emsconsole.so)을 불러온다.

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

위치 태그를 추가하여 지정된 URL에 대한 액세스 제어 규칙을 지정할 수 있는 컨테이너를 만든다.

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

RAD 서버가 올바르게 실행되고 있는지 테스트하려면 브라우저를 사용하여 <http://radserver/version>에 액세스하는 RAD 서버 버전 번호를 불러온다.

마지막 단계는 컴파일된 리소스를 복사하는 것이다. 이 장의 끝부분에서 복사하는 방법을 확인할 수 있다.

도커에 배포

도커에서 RAD 서버를 배포하는 것은 윈도우 및 리눅스를 사용하는 방식보다 훨씬 간단하다. 이 플랫폼에서 사용할 수 있는 다양한 이미지가 엠바카데로의 dockerhub에 있다.

RAD 서버와 관련된 2개의 이미지를 찾을 수 있다: 이 두 이미지의 유일한 차이점은 하나는 컨테이너 내부에서 실행되는 인터베이스 서버 엔진이 있고 다른 하나는 RAD 서버를 실행하는 데 필요한 인터베이스 서버가 다른 곳에서 호스팅된다고 가정한다는 것이다.



팁

인터넷베이스 서버는 도커와도 호환되며 엠바카데로는 이를 컨테이너화할 수 있는 이미지를 제공한다. 다음은 [DockerHub에 대한 링크](#)다.

이러한 도커 이미지는 완전한 오픈 소스로 빌드되며 GitHub에서 공개적으로 사용할 수 있다. 이는 하나의 접근 방식일 뿐이지만 도커에 매우 익숙하다면 이를 템플릿으로 사용하여 특정 요구 사항에 맞게 자유롭게 조정할 수 있다.

이러한 이미지를 배포하고 사용자 지정하는 방법에 대한 자세한 정보는 아래의 DockerHub 및 GitHub 링크에서 확인할 수 있다.

옵션 1: PA-RADServer-IB

이 이미지는 "모든 기능이 포함된" 이미지라고 할 수 있다. 인터베이스가 내장되어 있으면 작업이 훨씬 쉬워지지만, 이 컨테이너를 처음 실행할 때는 분리 모드로 실행할 수 없다는 점에 유의하자. RAD 서버 라이선스와 몇 가지 추가 세부 사항을 구성하기 위해 첫 번째 마법사를 실행해야 한다. 모든 설정이 완료되면 분리 모드로 실행할 수 있다.

명심해야 할 또 다른 사항은 애플리케이션을 확장할 계획이 없고 모든 것을 한 곳에 두고 작업하고 싶은 경우 이 컨테이너는 매우 편리할 것이다. 하지만 향후 애플리케이션을 확장하려는 경우 가장 좋은 방법은 RAD 서버를 인터베이스 서버에서 분리하여 별도의 컨테이너/머신에 설치하는 것이다.

[도커허브 링크](#)

[깃허브링크](#)

이 이미지는 다음이 포함된다:

- 인터베이스 서버 엔진
- PAServer
- RADServer 필수 파일
- 아파치 사전 구성

옵션 2: PA-RADServer

이 컨테이너는 유효한 RAD 서버 라이선스가 설치된 인터베이스 서버에 연결하지 않으면 작동하지 않는다. 애플리케이션을 확장하고 동일한 인터베이스 서버에 연결된 여러 인스턴스를 배포하려는 경우 이상적인 컨테이너이다.

[도커허브 링크](#)

[깃허브 링크](#)

이 이미지는 다음이 포함된다:

- PAServer

- RADServer 필수 파일
- 아파치 사전 구성



비교적 간단한 환경의 경우, PAServer를 사용하면 컨테이너를 다시 생성할 필요 없이 리소스 업데이트를 컨테이너에 바로 업로드 할 수 있다는 점을 기억하자.

도커에 RAD 서버를 배포하는 방법에 대한 자세한 내용은 [다음 링크](#)에서 확인할 수 있다.

RAD 스튜디오로 컴파일된 RAD 서버 모듈 복사

모듈 또는 필요한 추가 라이브러리를 프로덕션 머신에 배포하는 프로세스는 선택한 운영체제와 관계없이 거의 동일하다. 자체 리소스의 경우 .bpl/.so 파일만 프로덕션 시스템에 복사하면 된다.

RAD 서버 애플리케이션 패키지 파일은 프로젝트 설정에 따라 폴더에 컴파일된다. 기본 델파이 패키지 출력 및 C++ 최종 출력 디렉터리는 다음과 같다:

- 델파이의 경우:
 - 32-비트 윈도우 - C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\Bpl
 - 64-비트 윈도우 - C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\Bpl\Win64
 - 리눅스 - C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\Bpl\Linux64
- C++의 경우 모든 RAD 서버 애플리케이션 패키지 파일은 .\\$(Platform)\\$(Config) 폴더에 컴파일된다.

필요한 RAD 서버 애플리케이션 및 런타임 DLL 파일을 프로덕션 서버에 배포하는 방법에는 여러가지가 있다. 세 가지 일반적인 전송 방법은 다음과 같다:

- 패키지 파일을 RAD 서버가 설치된 프로덕션 서버 경로로 복사한다.
- 파일을 프로덕션 서버로 FTP 전송
- "프로젝트 | 배포" 메뉴 항목이 있는 플랫폼 도우미(PAServer)를 사용하여 IDE가 파일을 프로덕션 서버로 이동하도록 한다. 이 스크린샷은 윈도우 64의 예를 보여준다.

Deployment MyHelloWorldDelphiRADServerPackage						
Release configuration - Windows 64-bit platform						
Local Path	Local Name	Type	Configurat...	Platforms	Remote Path	Remote Name
<input checked="" type="checkbox"/> \$(BDS)\Redist\Win64\	rtl280.bpl	DependencyP...	Release	[Win64]	\	rtl280.bpl
<input checked="" type="checkbox"/> ..\..\..\..\Public\Do...	MyHelloWorldDelphiRA...	ProjectOutput	Release	[Win64]	\	MyHelloWorldDelphiRA...
<input checked="" type="checkbox"/> \$(BDS)\Redist\Win64\	emsserverapi280.bpl	DependencyP...	Release	[Win64]	\	emsserverapi280.bpl

PAServer가 전송할 수 있는 프로젝트 | 배포 파일

컴파일된 RAD 서버 확장 패키지 파일(예: 1장 MyHelloWorldDelphiRADServerPackage의 프로젝트)을 프로덕션 RADServer 폴더에 복사한다.



PAServer를 사용하는 경우 파일이 배포되는 기본 경로는 프로덕션 시스템에서 paserver.config 파일을 편집하여 변경할 수 있다. 파일 쓰기에 필요한 경로에 따라 PAServer를 실행할 때 더 높은 권한이 필요할 수 있다

EMSServer.ini 파일 구성

이제 프로덕션 폴더에 새 리소스를 추가했으므로 EMSServer.ini 파일에 사용 가능한 새 리소스가 있음을 지정해야 한다.

emsserver.ini 파일을 편집하여 [Server.Packages] 섹션 아래에 각 RAD 서버 확장 패키지를 추가한다.

원도우

```
[Server.Packages]
;# 이 섹션은 확장 패키지를 위한 섹션이다.
;# 확장 패키지는 사용자 지정 리소스 엔드포인트를 등록하는 데 사용된다.
;c:\mypackages\basicextensions.bpl=mypackage description
;c:\inetpub\wwwroot\RADServer\MyFirstDelphiRADServerPackage.bpl=First Windows Test Demo
```

리눅스

```
[Server.Packages]
;# 이 섹션은 확장 패키지를 위한 섹션이다.
;# 확장 패키지는 사용자 지정 리소스 엔드포인트를 등록하는 데 사용된다.
;c:\mypackages\basicextensions.bpl=mypackage description
/usr/lib/ems/bplMyFirstDelphiRADServerPackage.so=First Linux Test Demo
```

도커

이미 실행 중인 인스턴스의 emsserver.ini 파일을 구성하려면 `./config.sh` 스크립트를 실행한다. 스크립트가 자동으로 아파치를 재시작한다.

10

RAD 서버 라이트



Lite 버전이란?

RAD 서버에는 인터베이스 기반의 백엔드 데이터베이스가 필요하며, 일반적으로 IIS 또는 아파치용 웹 서버 DLL 모듈로 배포된다. 이런 이유로, 표준 배포 시 다음이 필요하다:

- 웹 서버 및 RAD 서버 모듈의 구성
- RAD 서버 배포 및 구성
- 특수한 목적의 RAD 서버 라이선스(사용자가 대상 장치에 등록해야 활성화할 수 있는 라이선스)가 있는 인터베이스 설치

개발용으로는 성능이 제한적이지만 훨씬 쉽게 배포할 수 있다. 또한 디버거에서 실행할 수 있는 기능(RAD 서버 모듈 코드를 디버깅할 수 있다)을 제공하는 Indy HTTP 서버를 기반으로 하는 독립형 버전의 RAD 서버를 오랫동안 제공해 왔다. 개발 버전의 경우 배포를 위해 디자인되지 않았으며, 배포용 라이선스가 없다. 생성할 수 있는 사용자 수에 제한이 있으며 로컬 인터베이스 개발자 에디션(해당 라이선스는 RAD 스튜디오 라이선스의 일부이다)으로 작업할 수 있다.

RAD 서버 라이트(**RSLite**)는 많은 처리량이 필요하지 않은 테스트 서버 및 시나리오를 위한 더 간단한 배포 모델을 제공하며, 정식 서버 대신 인터베이스임베디드 데이터베이스 엔진인 IBToGo를 사용하고 이를 간소화된 라이선스 부여 모델과 결합하여 제공한다.

RSLite는 솔루션과 함께 배포할 수 있는 라이선스 슬립 파일(배포할 컴퓨터에 등록할 필요 없음)과 함께 개발 에디션의 동일한 바이너리(RAD Studio와 함께 제공됨)를 IBToGo 바이너리와 함께 사용한다. RSLite는 내장형 데이터베이스와 Indy HTTP 서버 구성요소를 사용하기 때문에 일반적인 완전한 RAD 서버 설치처럼 초당 동일한 수의 요청을 제공할 수 없다. 또한 여러 RAD 서버 프론트 엔드로 확장할 수 없다.

RSLite가 사용하는 기본 아키텍처는 확장성이 훨씬 제한적이지만, 대부분의 간단한 배포 시나리오에서는 충분할 것으로 기대된다. 물론 처리량은 RAD 서버 모듈이 실행하는 특정 코드에도 의존한다.



팁

퍼블릭 시스템에 배포하는 경우, RSLite HTTP 서버를 직접 노출하지 말고 프록시 구성으로 통해 액세스할 수 있도록 하여 들어오는 HTTPS 호출에 대한 보안 컨텍스트를 제공하고 이를 RSLite로 전달하는 웹 서버(예: 아파치 또는 IIS)가 계속 존재하도록 하는 것이 좋다.

RAD 서버 라이트 라이선스 취득 방법

RAD 스튜디오 11(델파이 11 및 C++빌더 11 포함)의 모든 엔터프라이즈 또는 아키텍트 라이선스로 라이선스를 교환할 수 있다. [이 페이지](#)를 방문하여 제공된 지침을 따르기 바란다.



참고

등록 키와 EDN 계정이 필요하다

여기서는 RSLite용 라이선스 키만 받는 것이 아니라 설치와 함께 배포할 수 있는 슬립 파일(.TXT 파일에 저장된 라이선스)도 받을 수 있다. 이 라이선스는 설치 횟수에는 제한이 없지만 동일한 컴퓨터에서 두 개의 인스턴스를 실행할 수는 없다.



참고

라이선스 파일은 등록 사이트의 일반적인 정보와 달리 특정 하위 폴더에 배치해야 한다.

RAD 서버 라이트 프로젝트 배포

프로젝트를 배포하기 전엔 라이선스를 받기 위해서는 두 가지 고려 사항이 있다:

- 첫째, RSLite, 필요한 런타임 패키지 및 IBToGo 배포를 사용하여 배포 구성을 만들어야 한다([다음 단계는 이 링크에서 볼 수 있다](#)).
- 둘째, IBToGo 라이선스와 호환되는 제대로 된 프로덕션용 데이터베이스 파일을 생성해야 한다. RAD 서버 개발자 에디션에서 생성한 로컬 데이터베이스는 호환되지 않는다.

배포할 파일들

수동으로 배포

실제로 RSLite 솔루션을 배포하는 데 필요한 파일은 애플리케이션 패키지 및 해당 종속성 이외에도 다음과 같은 파일들이 있다:

1. 개발자 에디션과 동일한 RSLite 실행 파일: RAD 스튜디오 bin 폴더(또는 유사한 64-비트 버전)에서 사용할 수 있는 **EMSDevServer.exe**
2. 최소한의 설치에 필요한 RAD 스튜디오 런타임 패키지(여기에 나열되어 있으며 RAD 스튜디오 win32 또는 win64 redistrib 폴더에서 사용 가능) 및 RAD 서버 모듈의 코드에 필요한 기타 런타임 패키지를 포함한 필수 RAD 스튜디오 런타임 패키지:
 - bindengine<XX>0.bpl
 - dbrtl<XX>0.bpl
 - emsclientfiredac<XX>0.bpl
 - emsserverapi<XX>0.bpl
 - FireDAC<XX>0.bpl
 - FireDACCCommon<XX>0.bpl
 - FireDACCCommonDriver<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. 퍼블릭 문서 인터베이스 재배포 폴더(예: C:\Users\Public\Documents\Embarcadero\Interbase redistrib\InterBase2020)의 하위 폴더 win32_togo 또는 win64_togo - 리눅스의 경우 적절한 인터베이스 폴더에서 **libtogo.so** 파일을 찾을 수 있다.
4. 위에서 얻은 라이선스 파일을 인터베이스/라이선스 폴더에 추가한다(IBToGo 리디스트 구성의 일부).

배포 마법사 사용

다음 단계에 따라 배포 마법사의 RSLite 기능을 사용하여 파일을 배포한다:

1. RSLite 기능을 추가한다.
2. 다음으로 IBToGo 기능을 추가한다.
3. **iblite** 등록 파일의 선택을 취소한다.
4. "RAD 서버 라이트 라이선스를 얻는 방법" 섹션에서 배포할 파일을 추가한다: **rsLite** 활성화 파일을 생성하고 대상을 "인터베이스/라이선스"로 설정한다.
5. 다음으로, "프로덕션 데이터베이스 만들기" 섹션에서 얻은 파일을 추가하여 ./에 **emsserver.ini**를 배포한다.
6. 마지막으로, "프로덕션 데이터베이스 만들기" 섹션에서 얻은 파일을 추가하여 ./에 **emsserver.ib**를 배포한다.

MSVS 런타임

대상 윈도우 시스템에서 IBToGo를 실행하려면(IBToGo를 사용하는 RSLite를 실행하려면) 비주얼 C++ 2013 런타임 라이브러리가 설치되어 있어야 한다. RAD 스튜디오가 설치된 개발자 컴퓨터에는 이미 설치되어 있을 가능성이 높다. 그러나 일반적으로 배포 대상이 되는 머신에서는 [マイクロソフト](#)에서 다운로드한 후 설치해야 할 수도 있다.

프로덕션 데이터베이스 만들기

이 구성을 사용하기 위해서는 **EMSDevServer.exe** 애플리케이션을 실행하여 RSLite를 시작할 수 있다. 대상 시스템에 인터베이스 클라이언트가 있는 경우 우선순위가 높은 클라이언트를 선택하며, 인터베이스 클라이언트가 RAD 스튜디오와 함께 제공되는 개발자 버전인 경우 표준 RAD 서버 개발자 구성은 제외한 모든 기능이 작동한다.

RAD 서버가 시작될 때 로그의 처음 몇 줄을 보면 이를 파악할 수 있다. "RSLite" 구성의 경우 처음 몇 줄은 다음과 같이 표시된다:

```
{"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"}}
```

코드에 "Lite"가 false로 설정되어 있는 경우, 일반적으로 C:\Windows\SysWOW64에 있는 **gds32.dll**(또는 64-비트 버전) 인터베이스 클라이언트 라이브러리의 로드를 수동으로 비활성화해야 한다(인터베이스 클라이언트 라이브러리를 찾을 수 없는 경우 로컬 **ibToGo.dll**이 로드된다).

이제 (적절한 구성으로) RSLite를 시작했는데 **emsserver.ini** 파일과 **emsserver.ib** 데이터베이스 파일이 없는 경우 이를 만들라는 메시지가 표시된다. 이 작업을 수행하려면 RSLite가 RAD 스튜디오의 오브젝트 리퍼지토리 폴더(제품 폴더 아래의 ObjRepos)에서構성을 찾아야 한다. 이 작업을 더 쉽게 수행하는 방법은 프로그램 파일(x86)\Embarcadero\Studio\0\ObjRepos\en\ems 아래의 파일을 **emsdevserver.exe**의 상대 경로인 "..\ObjRepos/EMS" 폴더에 복사하는 것이다. 즉, 프로젝트 배포 디렉터리인 RSLite 설치 폴더와 동일한 레벨의 디렉터리에 ObjRepos 폴더가 있어야 한다.



참고

이 폴더는 RSLite를 배포할 때마다 요구되지는 않는다. 프로덕션 데이터베이스를 생성할 때 한 번만 생성한다면 나중에 대상 컴퓨터에 그대로 복사할 수 있다. 실제로 개발 환경에서 생성된 데이터베이스는 RSLite 배포와 호환되지 않는다.

마법사가 배포 폴더에 **emsserver.ini** 파일과 **emsserver.ib** 데이터베이스 파일을 생성할 수 있도록 RSLite 배포와 동일한 폴더를 대상 폴더로 지정하는 것이 좋다. 이제 RSLite, 이러한 구성 파일, 런타임 패키지 및 라이선스를 포함한 IBToGo가 있는 전체 폴더를 가져오면 대상 윈도우 컴퓨터에 배포하는데 필요한 모든 준비가 완료된다.

프록시 구성

보안 및 암호화 측면에서의 제한으로 인해 RSLite를 공개 웹 애플리케이션으로 직접 공개하는 것은 권장하지 않는다. 전용 서비스와 함께 프록시 계층을 사용하거나 널리 사용되는 웹 서비스 중 하나를 프론트 엔드로 사용하는 것이 좋다. 예를 들어 아파치에서 가상 호스트를 구성하고 HTTPS를 활성화한 다음, 다음과 같은 구성으로 트래픽을 RSLite 인스턴스로 리디렉션 한다:

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

리눅스의 경우

리눅스의 경우, 위와 비슷한 단계를 수행하면 모든 것이 예상한 대로 작동한다. 혹은 전체 RAD 서버를 설치한 다음 IBToGo를 설치에 추가하는 것도 고려할 수 있다:

- RAD 설치 폴더에 있는 **ems_install.sh**를 사용하여 RAD 서버를 설치한다. 이곳을 참조하라.
- 인터베이스 "redist" 폴더에서 리눅스의 EMS 폴더(/usr/lib/ems)로 IBToGo 파일을 복사한다.
- EMSDevServerCommand를 실행하고 마법사를 따라 EMS 데이터베이스 및 구성 파일을 생성한다.



적절한 권한을 위해 `sudo` 명령어를 통해 애플리케이션을 실행해야 할 수 있다.

참고

11

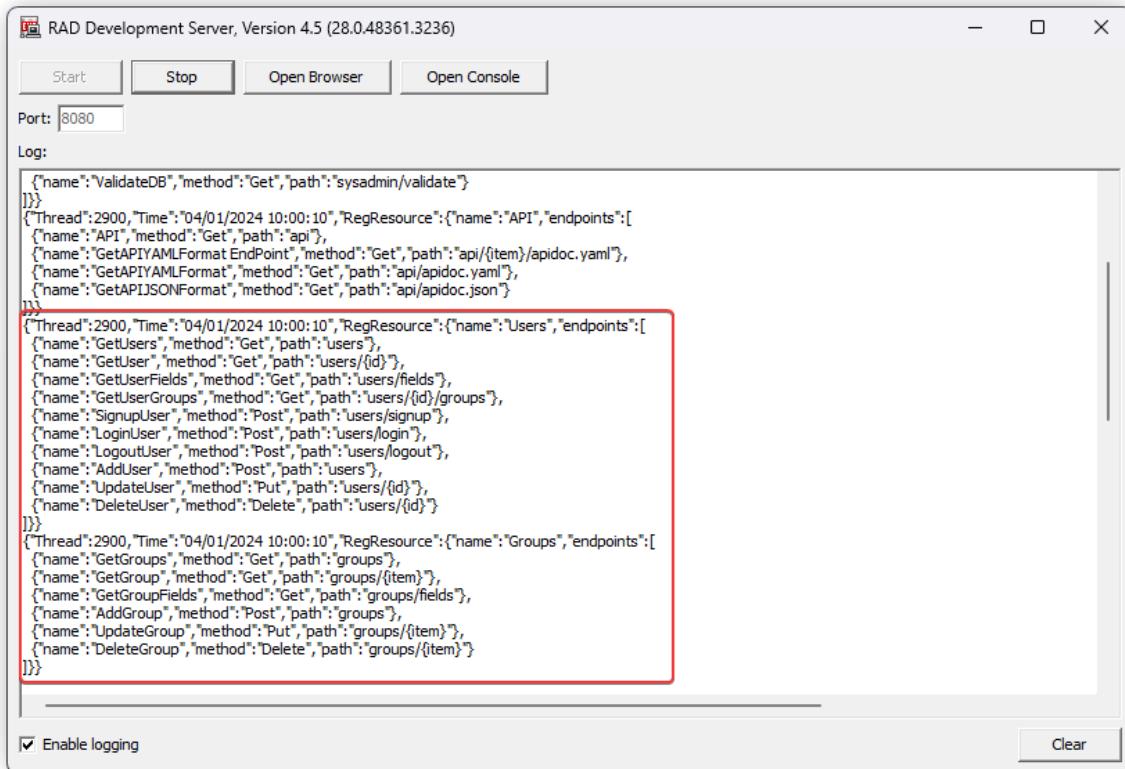
인증 및 권한부여



RAD 서버 안에는 사용자에 대한 인증(authentication)과 권한부여(authorization) 기능이 들어있다. 관리에 필요한 여러 가지 엔드포인트들이 제공된다. 그 엔드포인트들은 요청(request)를 통해 접근할 수 있다. 뿐만 아니라, RAD 스튜디오를 사용해 개발된 RAD 서버 위에서 작동하는 앱들 또는 RAD 서버 리소스들 안에서는 프로그램적으로도 접근할 수 있다. 프로그램적으로 접근할 수 있다는 점은 훌륭한 장점이다. RAD 서버에서 제공하는 인증 서비스가 여러분의 자체 인증 시스템을 사용하도록 덮어쓰는 것도 가능하기 때문이다.

내장된 인증: 사용자 및 그룹 관리하기

여러분의 개발 장비 안에서 RAD 서버 인스턴스를 실행하면, 창이 하나 나타난다. 그 안에는 로그가 나타나는데, 거기에서 우리는 몇 가지 중요한 엔드포인트들을 볼 수 있다:



RAD 서버가 생성하는 디폴트 엔드포인트들 중에는 사용자 및 그룹 관리도 들어있다

인증과 관련된 리소스는 이 두 가지다: [Users](#)와 [Groups](#). RAD 서버를 사용하면, 사용자(user)들을 정의할 수 있다. 뿐만 아니라, 그 사용자들을 특정 그룹(group)에 할당할 수 있다. 그래서 우리는 사용자에게 역할(role)을 지정할 수 있고, 특정 엔드포인트, 리소스에 대한 접근을 세부적으로 허용 또는 금지할 수 있다.

users/ 엔드포인트에 접근해보자. 그 요청과 결과는 아래의 스크린샷에서 볼 수 있다.

```

[{"username": "test", "_id": "3A25B7B0-1033-4B8B-A77E-EFB25B5B75CD", "_meta": {"creator": "3A25B7B0-1033-4B8B-A77E-EFB25B5B75CD", "created": "2023-07-11T16:24:30.000Z"}, "description": "Created by EMS setup. Password is \"testpass\"."}, {"username": "azapater", "_id": "B7F80A21-E0E0-4CD3-A6BA-D27484E25F7D", "_meta": {"creator": "B7F80A21-E0E0-4CD3-A6BA-D27484E25F7D", "updated": "2023-07-11T16:24:30.000Z"}]

```

RAD 서버 안에 생성되어 있는 사용자들의 목록에 접근하기

RAD 서버가 배열 하나를 반환해주었다. 그 배열 안에는 RAD 서버의 데이터베이스 안에 생성되어 있는 모든 사용자들이 있다. 위 스크린샷을 보면, RAD 서버에는 test라는 사용자가 "test pass"라는 패스워드를 가지고 있음을 알 수 있다(이 test 사용자와 그 패스워드는 우리가 설치 마법사를 진행하는 동안 자동으로 생성된다).



모든 리소스들에 대한 보다 자세한 사항을 보고 싶다면, 그 리소스들이 모두 들어 있는 도움말을 참고할 수 있다. 그 도움말 파일의 형식은 OpenAPI json/yaml apidoc이다.

note



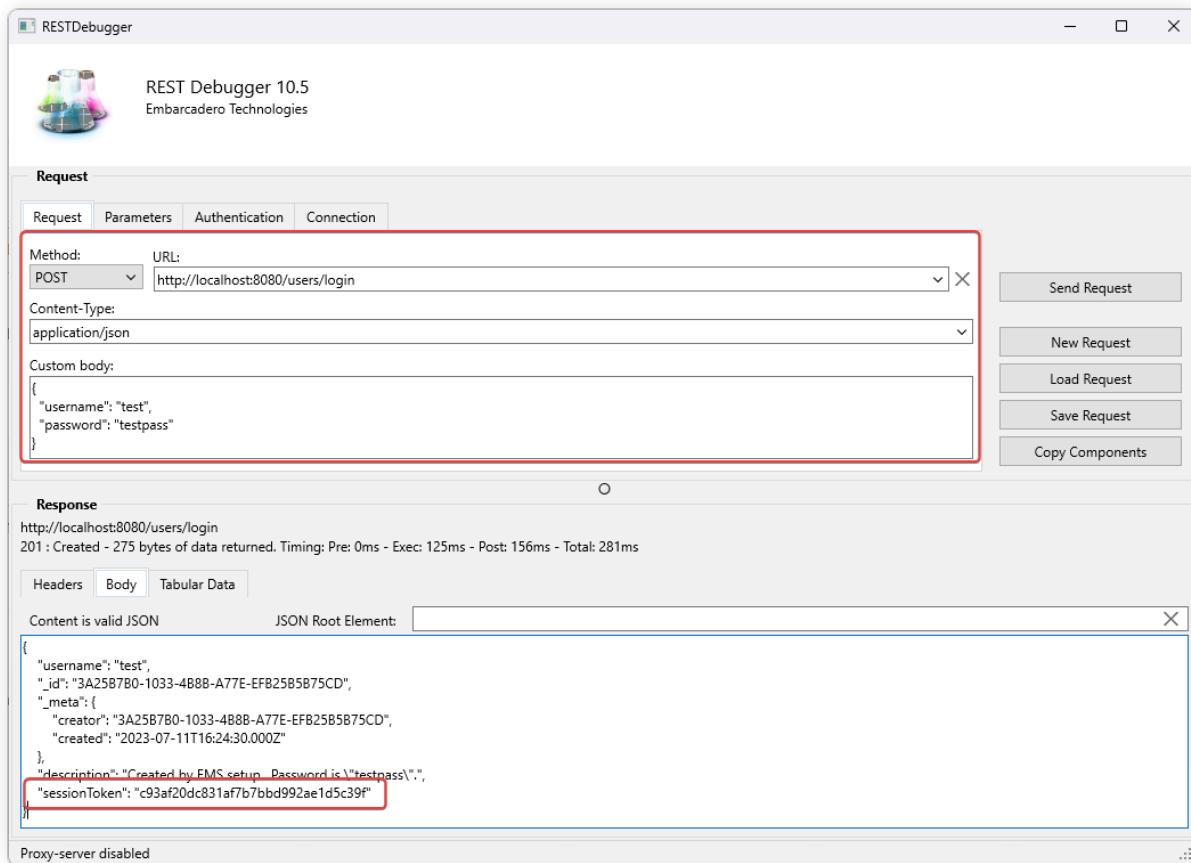
Users와 Groups 리소스들은 표준 CRUD 관행을 빠른다. 따라서, 새 레코드를 생성하고 싶다면, 이 엔드포인트로 POST 요청을 보내면 된다. 레코드를 변경하고 싶다면, PUT 요청을 (Body 안에 새 값을 담아서) 보내면 된다. 다른 것들도 같은 방식으로 하면 된다.

tip

로그인 하기

RAD 서버 안에 사용자가 생성되면, 그 사용자는 RAD 서버에 로그인할 수 있다. 로그인에 성공하면, 토큰 하나를 받는다. 그 다음 요청부터는, 그 토큰을 사용해 자신의 신원을 증명하면 된다.

자동으로 생성된 인증 관련 엔드포인트들에 대해 봤는데, 로그인도 방식은 같다. users/login 엔드포인트에게 POST 요청을 보내면 된다. 단, username과 password를 Body에 넣어서 보내야 한다.



users/login 엔드포인트에 POST로 접근해서 세션 토큰 하나를 획득한다

일단, 세션 토큰을 확보했다면, 그 다음 요청부터는 그 값을 헤더 안에 반드시 담아야 한다. 그래야 RAD 서버가 그 사용자의 신원을 정확히 식별할 수 있다.

X-Embarcadero-Session-Token=<값>

로그아웃 하기

디폴트 설정인 경우, 세션 토큰의 유효기간은 무한하다. 하지만, 이는 보안 면에서 아주 좋은 설정이라고 할 수는 없다. EMSSErver.ini 파일 안에서, 다음 설정들을 사용하라. 그러면, 토큰 유효일자를 세밀하게 지정할 수 있다. 그런 구성을 위해 사용되는 파라미터들은 아래와 같다:

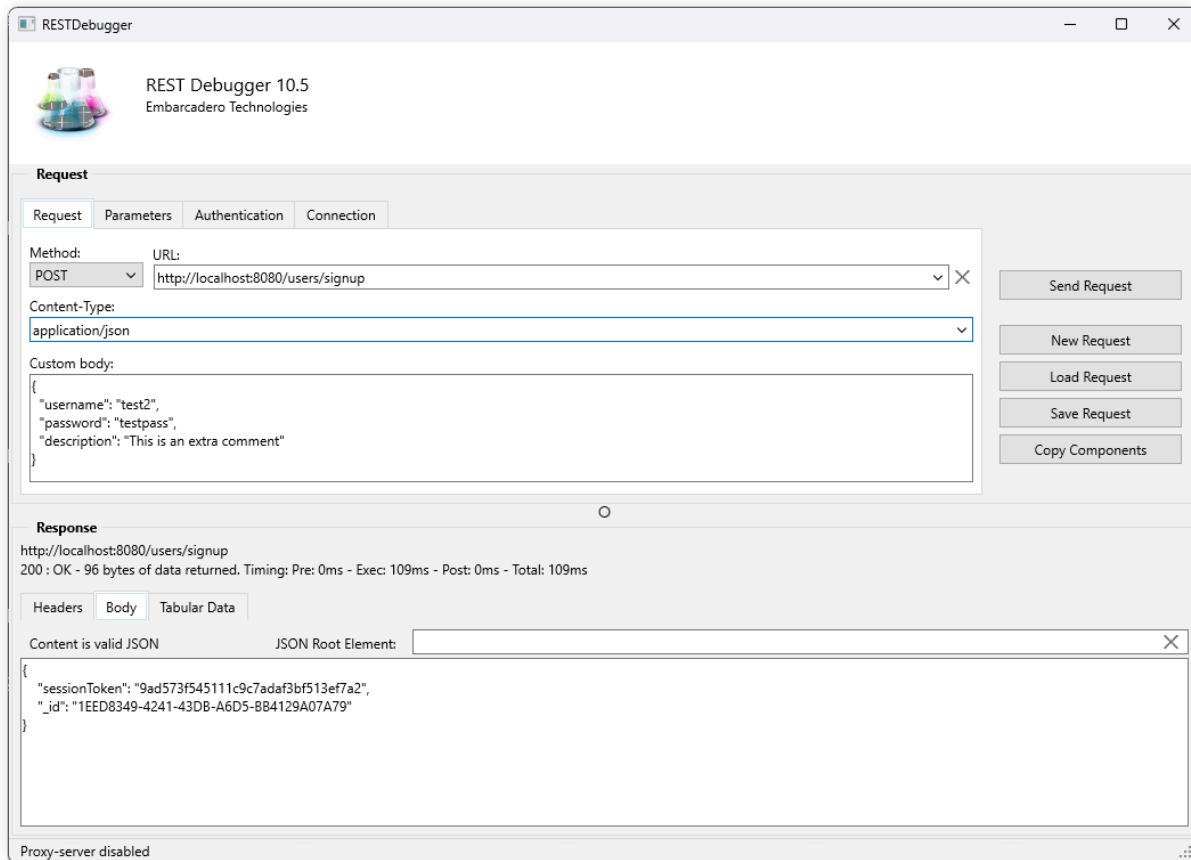
```
SessionInactivityTimeout=
;# Set SessionInactivityTimeout=60 이라는 형식으로 설정, 이 세션의 활동이 없어도 유효하도록
유지할 최대 시간(초)이 지정된다.
;# 이 명령은 세션 토큰에 반영된다. 디폴트는 0 (시간 만료 없음)이다.
SessionLiveTimeout=
;# Set SessionLiveTimeout=60 이라는 형식으로 설정, 이 세션이 살아있는 최대 시간(초)이 지정된다.
;# 이 명령은 세션 토큰에 반영된다. 디폴트는 0 (시간 만료 없음)이다.
```

두 경우 모두, Timeout을 초 단위로 명시해야 한다. 두 Timeout 중 어느 하나라도 초과되면, 그 토큰은 자동으로 비활성화된다.

로그아웃을 강제하고 싶거나, 활성 상태인 토큰을 강제로 비활성화하고 싶다면, users/logout 엔드포인트로 요청을 보내면 된다 (단, 헤더 안에 해당 사용자의 토큰을 담아주어야 한다).

회원가입 하기

여러분이 /users/signup 으로 요청을 보내면, 그 사용자는 RAD 서버 안에 등록된다. 그와 동시에, 우리는 그 사용자의 ID와 그 사용자용 세션 토큰을 획득하게 된다.



POST 방식인 users/singup 엔드포인트에 접근해서 새 사용자를 생성하기

위 예시에서, 중요하게 눈여겨 봐야하는 것이 있다. 요청(request) 안에 사용자 정의 필드를 얼마든지 많이 추가해서 보낼 수 있다. 요청의 Body 안에 넣으면 된다 (예를 들어, 위에서는 'description' 필드를 추가했다).

Username은 고유하다. 따라서 이미 존재하는 username을 signup 요청 안에 넣어 보내면, RAD 서버는 응답에 409 에러를 담아 반환한다.



tip 회원가입(sign-up)을 외부에 공개하고 싶지 않다면? users.signup 엔드포인트에 대한 접근을 제한 하라. 즉 그 엔드포인트에 대해 오직 권한이 있는 사용자와 그룹만 접근을 허용하라. 역할을 정의하는 구역에 적어놓으면 된다. 자세한 사항은 [authorization section](#) 참조.

그룹(Group) 관리하기

그룹(group)을 관리하는 것도 가능하다. 내장된 엔드포인트들을 사용하면 된다. 이 리소스는 표준 CRUD 관행(convention)을 따른다.

사용자 한 명은 여러 그룹에 속할 수 있다. 이와 마찬가지로, 그룹 하나는 여러 사용자들을 가질 수 있다.

중요: 사용자를 그룹에 할당(assign)하려면, POST 요청을 /users/groups/{id}로 보내면 된다. 이 때 배열 하나를 함께 전달해야 한다. 그 배열 안에는 해당 그룹에 속할 모든 구성원들을 담는다.

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



tip 비록 여러분이 역할 관리를 그룹 수준에 정의하겠다는 계획이 없더라도, 사용자들을 그룹들에 할당하는 것을 권장한다. 그렇게 하면, 분석(analytics)이 보다 쓸모있어 진다. 왜냐하면, 그룹을 기준으로 하는 상세한 정보를 보여주는 것이 가능하기 때문이다.

내장된 권한부여(Integrated authorization)

전역 자격 증명(Global Credentials)

전역 자격 증명 (global credential)을 다수의 보안 계층용으로 정의하는 것이 가능하다. RAD 서버 엔진의 구성 파일(emsserver.ini 파일) 안에는 이와 관련된 파라미터 세 개가 있다. 그것들은 [Server.Keys] 구역 안에 있다:

MasterSecret

이것은 여러분이 그 RAD 서버의 데이터베이스 안에 저장된 모든 데이터에 완전하게 접근할 수 있는 권한을 제공한다.

관리자 작업용으로, 이 RAD 서버 MasterSecret 키를 사용하라. 그 RAD 서버의 엔진 (EMS Server) 안에 있는 RAD 서버 리소스 전체에 접근할 때 사용하면 좋다.

AppSecret

RAD Server Client Application으로부터 접근이 허용된 엔드포인트들에 대한 접근 권한을 제공한다.

ApplicationID

Application ID의 용도는 RAD 서버-기반 클라이언트로부터 온 요청(request)에 대한 식별이다. 이를 통해, 오직 애플리케이션의 ID가 유효한 요청만 RAD 서버가 처리하도록 할 수 있다. 이 경우, 유효하지 않는 ID를 담고 있는 요청들은 모두 거부된다. 이 식별자(ApplicationID)는 여러분이 서로 다른 RAD 서버 인스턴스들을 가지고 있을 때 그것들을 차별화하는데 사용될 수도 있다.

보다 자세한 정보는 [docwiki](#)에 있다.

사용자들 그리고 그룹 권한부여

RAD 서버의 엔드포인트들의 보안을 지키는 가장 쉬운 길은 EMSServer.ini 파일을 사용하는 것이다. 해당 모든 규칙은 반드시 [Server.Authorization] 구역 아래에 정의되어야 한다. 아래는 ini 파일 안에 디플트로 들어 있는 내용이다. 좋은 예들이 있으니 살펴보자:

```
[Server.Authorization]
;# 이 구역은 리소스들과 엔드포인트들에 대한 권한부여(Authorization) 요구 사항을 설정하는 곳이다.
;# 권한부여(Authorization)는 내장된 리소스(예; Users)들 그리고 사용자 정의 리소스들을 대상으로
설정될 수 있다.
;# 알아둘 점: MasterSecret 인증(authentication)이 사용되면, 여기의 모든 요구사항들이 무시된다.
;# 리소스(Resource)에 대한 설정은 그 리소스 안에 포함되는 모든 엔드포인트들에게 반영된다.
;# 엔드포인트 설정들은 해당 리소스의 설정들을 덮어쓴다	override).
;# 디폴트에 의하면, 모든 리소스는 public(공개)다.
;# 설정들은 JSON으로 명시된다.
;# JSON 애트리뷰트(attribute)들
;# {"public": true} - 어떤 클라이언트에게든 접근 권한을 부여한다
;# {"public": false} - 허용된 클라이언트가 접근할 수 있다. 이는 그 user나 group에 달려있다.
따라서 요청 안에는 사용자 자격증명 (sessionid)가 반드시 담겨서 전달되어야 한다
;# {"users": ["username1", "username2"]} - 사용자에게 권한을 부여(username 기준).
;# {"users": ["userid1", "userid2"]} - 사용자에게 권한을 부여(userid 기준).
;# {"users": ["*"]} - 어떤 사용자든 권한을 부여.
;# {"groups": ["groupname1", "groupname2"]} - user group에 속한 사용자에게 권한을 부여.
;# {"groups": ["*"]} - 어떤 user group에는 속해있지만 하면, 그 사용자에게 권한을 부여.
;#
;# 예시
;#
;# "Users" 리소스 안에 있는 모든 메서드들을 비공개(private)로 한다. 그런데, LoginUser와
SignupUser 엔드포인트만은 예외로 설정한다
;Users={"public": false}
;Users.LoginUser={"public": true}
;Users.SignupUser={"public": true}
;#
;# 사용자 정의 리소스인 "Resource1" 안의 모든 메서드들은 group1에 속한 사용자만 사용할 수 있다
;Resource1={"groups": ["group1"]}
```

```

;#
;# 사용자 정의 리소스인 "Resource2" 안의 모든 메서드들을 오직 MasterSecret 인증을 사용하는
경우에만 사용할 수 있다
;Resource2={"public": false}
;#
;# 사용자와 그룹을 만드는 생성자들에게만 적용되는 특별한 규칙.
;# 사용자를 만드는 생성자는 아래의 엔드포인트들에 대한 권한을 자동으로 가지게 된다:
;#   Users.GetUser, Users.UpdateUser, Users.DeleteUser
;# 그룹을 만드는 생성자는 아래의 엔드포인트들에 대한 권한을 자동으로 가지게 된다:
;#   Groups.GetGroup, Groups.UpdateGroup, Groups.DeleteGroup

```

위와 같이, 이 파일에는 스스로를 매우 잘 설명하는 주석이 들어있다. 하지만, 몇 가지 사례를 더 살펴 보자.

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

sales 그룹에 속한 모든 구성원들은 'Orders' 리소스로부터 나오는 모든 엔드포인트들에 접근할 수 있다.

```

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

```

위 예는, 'Users' 리소스 전체에 대한 외부 접근을 제한하고 있다. 하지만, 'LoginUser'와 'SignupUser' 엔드포인트만 쿠션에서 외부 접근을 허용한다. 이것은 매우 유용한 방식이다. 리소스 수준에서 모든 접근을 제한하되 꼭 필요한 엔드포인트 몇 개만 예외를 허용할 때 사용하면 좋다. 'public'은 키워드다. 인증 토큰(auth token)을 담고 있지 않은 요청도 받아준다는 뜻이다.

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

위 예는, 유효한 토큰이 함께 전달된 모든 요청(request)들이라면, 'Customers' 리소스에 접근할 수 있다. 그 구성원의 역할은 무엇이든 상관없다. 하지만, 반드시 RAD 서버에 사용자로 등록되어 있어야만 한다.

사용자 정의 인증 (Custom authentication)

만약, 여러분이 이미 인증 서비스 즉, ActiveDirectory/LDAP 또는 기타 써드-파티 서비스 등을 구현했다면, 그 서비스를 RAD 서버에 통합할 수 있다. Samples 디렉토리 안에는 두 가지 예시가 있다. 바로, 사용자 정의 로그인과 AD와의 기본적인 통합 (RAD 서버가 윈도우 장비 위에서 작동되고 있는 경우)에 대한 예이다.

사용자 정의 인증은 다음과 같은 가정이 바탕이다. 우리는 제공받은 자격 증명이 올바른지를 우리가 이미 구현해 사용하고 있는 별도의 인증 서비스를 통해 검증하겠다는 것이다. 일단 RAD 서버가 아닌 곳에서 검증한 사용자라면, 그 사용자는 반드시 RAD 서버 안에 내장되어 있는 권한 관리 안에서 인증되도록 처리해야 한다 (만약 RAD 서버에 처음 연결한 사용자라면, RAD 서버 안에 그 사용자를 생성해야 한다). 그리고 나서, 그 사용자를 위한 RAD Server 토큰을 반환한다.

RAD 서버의 내부 API를 코드에서 접근하려면, 해당 API의 새 인스턴스를 만들고 해당 Context를 명시해서 사용하는 것이 좋다. 이 예를 보자:

Delphi

```
// 사용자 정의 방식으로 EMS 로그인 처리
procedure TCustomLogonResource.PostLogin(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  lEMSAPI: TEMSInternalAPI;
  lResponse: IEMSResourceResponseContent;
  lValue: TJSONValue;
  lUserName: string;
  lPassword: string;
begin
  // EMS API를 생성 (프로세스 안에서)
  lEMSAPI := TEMSInternalAPI.Create(AContext);
  try
    // 자격증명을 꺼내기 (요청으로부터)
    if not (ARequest.Body.TryGetValue(lValue) and
      lValue.TryGetValue<string>(TEMSInternalAPI.TJSONNames.UserName, lUserName) and
      lValue.TryGetValue<string>(TEMSInternalAPI.TJSONNames.Password, lPassword)) then
      AResponse.RaiseBadRequest(' ', '자격증명을 찾지 못했습니다');

    var lExternalUserGUID := ValidateExternalCredentials(lUserName, lPassword);

    if not lEMSAPI.QueryUserName(lUserName) then
    begin
      // 사용자를 추가 (전달받은 자격 증명에 해당하는 사용자가 없는 경우에만 해당됨)
      // Users/Signup 엔드포인트가 실제로 수행하는 내부 메서드를 호출 (프로세스 안에서)
      var lUserFields := TJSONObject.Create;
      lUserFields.AddPair('ExternalUserGUID', lExternalUserGUID);
      lUserFields.AddPair('comment', '이 사용자 추가: RAD 서버 CustomLoginUser가 생성함');
      lResponse := lEMSAPI.SignupUser(lUserName, GenerateHashedPassword(lUserName),
        lUserFields);
      end
    else
      // Users/Login 엔드포인트가 실제로 수행하는 내부 메서드를 호출 (프로세스 안에서)
      lResponse := lEMSAPI.LoginUser(lUserName, GenerateHashedPassword(lUserName));
    if lResponse.TryGetValue(lValue) then
      AResponse.Body.SetValue(lValue, False);
```

```

finally
    LEMSAPI.Free;
end;
end;

```

C++

```

void TCustomLoginResource::PostLogin(TEndpointContext* AContext, TEndpointRequest* ARequest, TEndpointResponse* AResponse)
{
    // EMS API를 생성 (프로세스 안에서)
    std::unique_ptr<TEMSInternalAPI> lEMSAPI(new TEMSInternalAPI(AContext));
    // 자격증명을 꺼내기 (요청으로부터)
    TJSONObject * lValue;
    String lUserName;
    String lPassword;

    if(!!(AResponse->Body->TryGetObject(lValue) &&
        (lValue->GetValue(TEMSInternalAPI_TJSONNames_UserName) != NULL) &&
        (lValue->GetValue(TEMSInternalAPI_TJSONNames_Password) != NULL)))
        AResponse->RaiseBadRequest("", "자격증명을 찾지 못했습니다");

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

    String lExternalUserGUID = ValidateExternalCredentials(lUserName, lPassword);

    _di_IEMSRessourceResponseContent lResponse;
    if (!lEMSAPI->QueryUserName(lUserName)) {
        // 사용자를 추가 (전달받은 자격 증명에 해당하는 사용자가 없는 경우에만 해당됨)
        // Users/Signup 엔드포인트가 실제로 수행하는 내부 메서드를 호출 (프로세스 안에서)
        std::unique_ptr<TJSONObject> lUserFields;
        lUserFields->AddPair("ExternalUserGUID", lExternalUserGUID);
        lUserFields->AddPair("comment", "이 사용자 추가:  
CustomResource.CustomLoginUser가 생성함");
        lResponse = lEMSAPI->SignupUser(lUserName,
GenerateHashedPassword(lUserName), lUserFields.get());
    } else
        // Users/Login 엔드포인트가 실제로 수행하는 내부 메서드를 호출 (프로세스 안에서)
        lResponse = lEMSAPI->LoginUser(lUserName,
GenerateHashedPassword(lUserName));
    if(lResponse->TryGetObject(lValue)) {
        AResponse->Body->SetValue(lValue, false);
    }
}

```

```
}
```

요청이 권한이 있는 사용자로부터 온 것임을 RAD 서버가 그 알 수 있도록 하는 방법은 간단하다. 그 사용자를 signup 또는 login으로 전달하면 된다. 그 앤드포인트들은 처리 과정 중에 그 요청 안에 포함된 자격 증명이 유효한지를 우리의 외부 인증 서비스를 통해 검증하는 과정이 이 예시에 구현되어 있다.

이미 눈치챘을 것이다. 우리는 RAD 서버 사용자를 생성할 때 RAD서버 안에 그 사용자가 외부 인증 서비스에서 사용하는 패스워드를 그대로 넣지 않는다. 만약 그렇게 한다면, 그 사용자가 외부 인증 서비스에서 자신의 패스워드를 변경하는 경우에는 RAD 서버 안에 있는 패스워드 역시 다시 업데이트 해야 하는 문제가 생긴다. 그렇게 되면, 항상 최신 정보를 일치켜야 한다는 불필요하게 복잡한 층을 하나 더 만드는 결과를 낳는다. 이 예에서는 그 사용자명에 해쉬를 반영한다. 그러면, 외부 인증 서비스에 전적으로 의존하면서도 RAD 서버가 가진 보안성이 높은 매커니즘을 계속 유지할 수 있다. 이 예시 프로젝트를 살펴보면, 보다 상세한 구현을 알 수 있을 것이다.



note

사용자 정의 로그인 그리고 AD 통합 예시는 RAD 스튜디오의 Samples 폴더 안에 있다:

C:\Users\Public\Documents\Embarcadero\Studio\XX.0\Samples\Object

Pascal\Database\EMS

C:\Users\Public\Documents\Embarcadero\Studio\23.0\Samples\CPP\Database\EMS

RAD 서버 안에는 반드시 각 사용자를 개별적으로 생성해야 할까? 기술적으로는 그렇지 않다. 하지만, 매우 권장한다. 주요 이유는 분석(analytics)과 로그 기록(logging)을 혜택을 받기 위해서다. 만약 우리가 RAD 서버 안에 각 사용자를 개별적으로 생성한다면, 내장된 분석 기능을 활용할 수 있고, 각 사용자들에 대한 세부적인 데이터를 개별적으로 얻을 수 있다. 기술적으로는 RAD 서버의 “일반” 사용자 하나를 만들고, 그 사용자를 재사용해서 로그인을 하고 각 토큰을 제공하는 것이 가능하다. 하지만, 그렇게 하면 분석 기능의 유용성이 낮아진다는 점을 충분히 이해해야 한다.

위 예시는, 내부 RAD 서버 API를 사용하는 방법을 보여주었다. 이 예에서 회원 등록과 로그인을 프로그램적으로 처리하는 방식을 통해 그 방식을 이해했을 것이다. 하지만, 이는 이 API들의 능력을 활용하는 여러 가지 방식 중 겨우 두 가지에 불과하다. 그룹을 생성하기, 사용자들을 그룹에 할당하기 등등 거의 모든 것들을 이 API를 통해 구현할 수 있다. 즉, 프로그램적으로 이 모든 것들을 구현하는 것이 가능하다.



tip

RAD 서버의 내부 API는 모두 EMS.Services 유닛 안에 있다. 그 코드를 살펴보거나 또는 [도움말](#)에서 해당 메서드들을 보면 된다.

사용자 정의 권한부여 (Custom authorization)

또 다른 선택을 보자. 프로그램적으로 이 앤드포인트들의 보안을 지키는 것이 있다. 우리는 각 앤드포인트에 특정 규칙을 정의할 수 있다. 그래서 사용자 그리고/또는 그룹의 접근을 허용하거나 제한할 수 있다.

요청(request)과 관련되는 각 메서드에는 AContext라는 파라미터(argument)가 있다. 우리는 이것을 사용해, 그 요청에 할당되어 있는 사용자 또는 그룹을 확인할 수 있다.

Delphi

```
//이 요청에 연계된 userName을 반환
AContext.User.UserName
//이 요청에 연계된 userID를 반환
AContext.User.UserID
//그 사용자가 속한 그룹들을 반환
AContext.User.Groups
```

C++

```
//이 요청에 연계된 userName을 반환
AContext->User->UserName
//이 요청에 연계된 userID를 반환
AContext->User->UserId
//그 사용자가 속한 그룹들을 반환
AContext->User->Groups
```

인증 정보 검증을 수행한 다음에는, 그 요청이 접근하려는 메서드에 대해 접근을 제한하고 싶을 수도 있다. 방법은 간단하다. unauthorized error를 반환하도록 구현하면 된다. 이 때는 AResponse 파라미터를 사용한다.

Delphi

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

C++

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

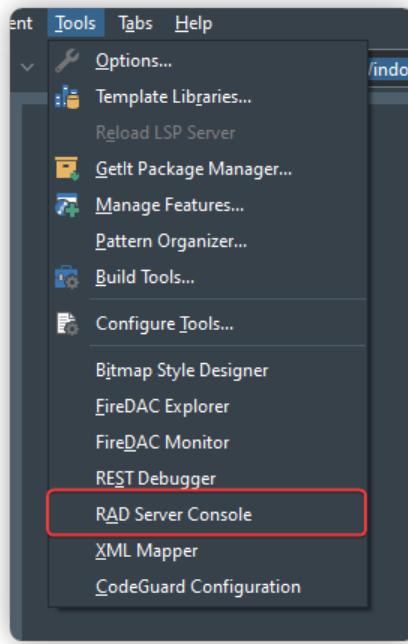
RAD 서버 관리 콘솔

RAD 서버 콘솔 (줄여서, RSConsole)는 이미 몇 년 전부터 있던 것이다, 하지만, 좀 숨겨져 있다. 그 경로는 여기다:

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

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

RAD Studio 12 아테네 이후부터는, 이 도구가 “Tools” 메뉴 아래에 추가되었다.



RAD 서버 콘솔에 접근하도록 하는 메뉴 항목

새 프로파일 생성하기

이 콘솔을 사용하면, 로컬에 있는 개발자 환경 또는 원격에 있는 운영 서버에 연결할 수 있다. 그리고 여러 개의 프로파일들을 미리 정의해 놓고, 필요할 때 해당 서버로 연결할 수 있다.

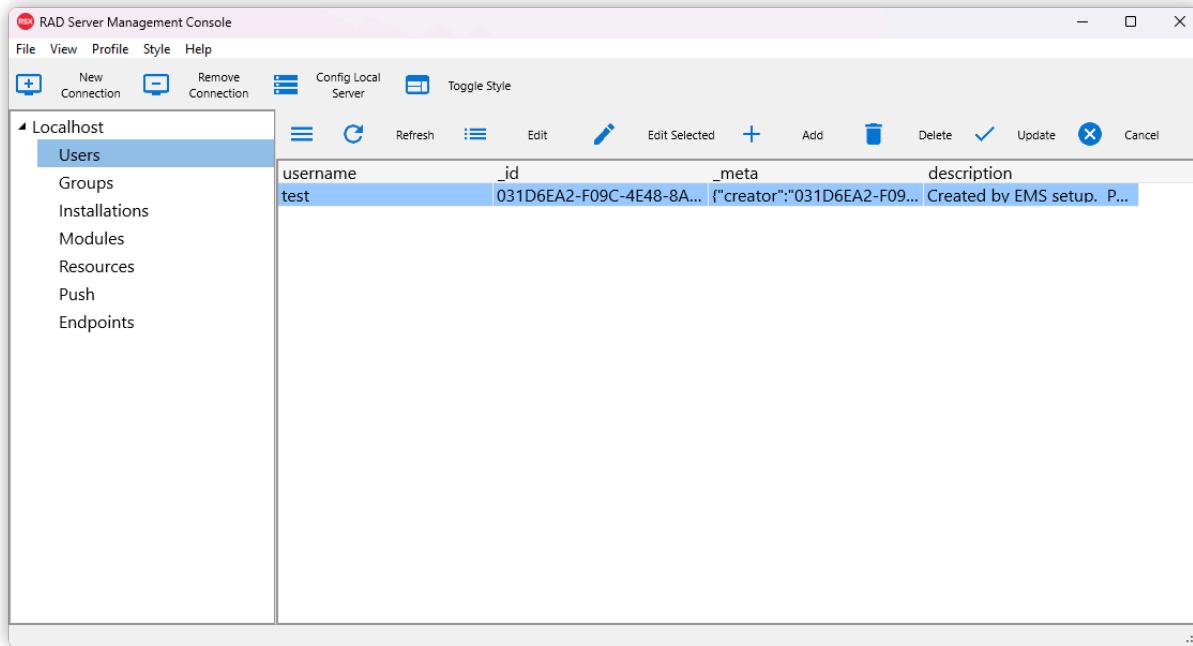
새 profile/connection을 생성하는 방법은 간단하다. “New Connection”을 누르면 된다. 또는 그 콘솔의 메뉴에 있는 “Profile/New Profile...”를 사용하면 된다. 그 구성은 꽤 명료하기 때문에 쉽다. 그 호스트에 연결하기 위한 기본 세부사항만 있으면 된다.



RAD 서버 인스턴스에 연결할 수 없는 경우는 다음과 같다. RAD 서버 개발 환경이라면, 실행(Run)하지 않은 경우, 운영 환경이라면, Windows에서 EMSDevServer.exe 또는 Linux에서 EMSDevServerCommand가 실행되고 있지 않고 있는 경우.

사용자들과 그룹들을 다루기

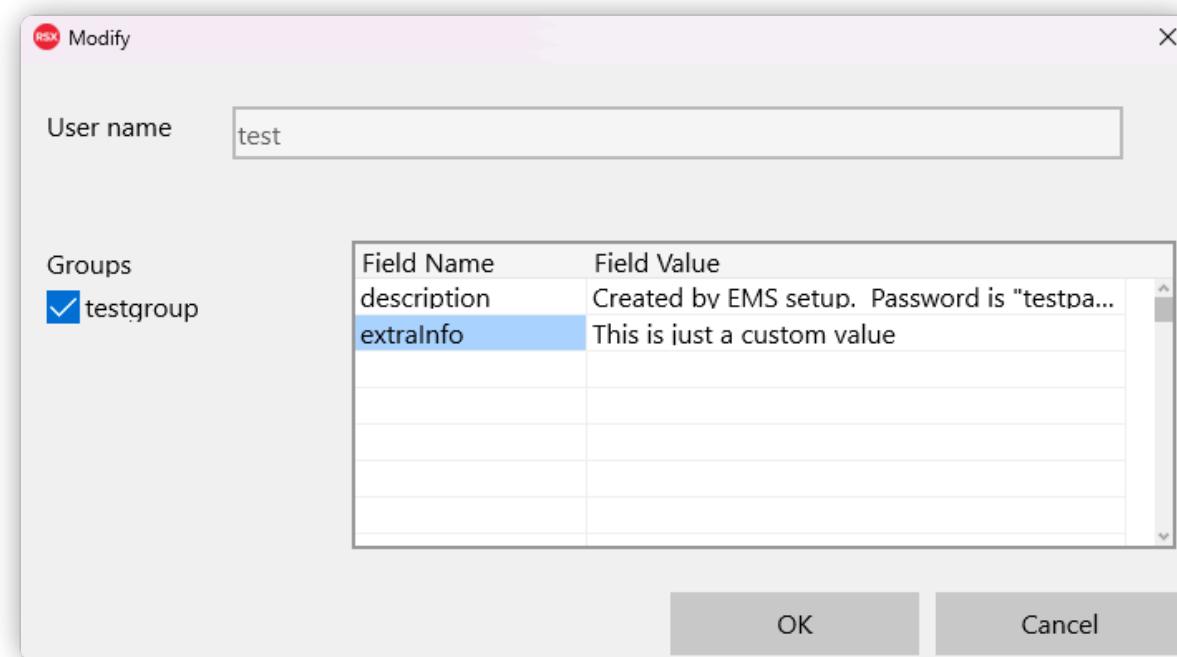
원하는 RAD 서버 인스턴스에 연결이 되면, 사용할 수 있는 리소스들의 전체 목록이 왼쪽에 나타난다. 아래 그림을 보고, “Users”와 “Groups”에 대해 이야기를 해보자.



생성된 사용자들의 목록을 RAD 서버 콘솔을 이용해 접근할 수 있다

사용자 또는 그룹을 생성, 편집, 삭제하는 것은 간단하다. 왼쪽에서 해당 구역을 클릭하면, 툴바가 나타난다. 그 안에는 여러 가지 옵션들이 있다: Edit, Add, Delete 등등.

예를 들어, “modify user” 창을 보자 (이 창은 “create user” 창과 모습이 거의 똑같다):



“Modify User” 창



사용자를 생성할 때는 password 필드가 보인다. 하지만, 사용자를 편집할 때는 그 필드에 접근하지 못한다. 사용자의 패스워드를 바꾸는 건 간단하다. 사용자 정의 필드들을 넣는 구역 안에서 “password” 필드를 추가하고, 그 값에 새 패스워드를 넣으면 된다.

새 그룹을 생성하는 절차 역시 매우 간단하다. 왼쪽 구역에서 “groups”를 클릭한다. 그리고 나서, “Add”를 선택한다. 창이 나타나면 그룹을 생성한다. 그 창에서는 그 그룹에 속할 구성원들을 할당하는 것도 가능하다.

RSConsole 안으로 더 깊이 들어가기

RAD 서버 콘솔은 RAD 서버에 연결하는 외부 클라이언트의 좋은 예이다. 따라서, RAD 서버 콘솔이 동작이 어떻게 구현되었는지 관심이 있을 수 있을텐데, 이 FMX 앱, 즉 RAD 서버 콘솔의 소스 코드 전체가 RAD 스튜디오와 함께 들어 있다. 그 경로는 여기다:

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

비록, 꽤 복잡한 애플리케이션이지만, 원격에서 RAD 서버에 내장된 API들에 접근하는 모든 가능성들 그리고 각 액션(action)을 통해 전송되는 요청(request)들의 유형들을 볼 수 있는 완벽한 곳이다.

12

여러분의 엔드포인트들에 대한 문서화와 테스트를 위해 **OpenAPI (Swagger) 사용하기**

.....

OpenAPI/Swagger란 무엇인가 그리고 우리는 왜 이것을 사용하는가?

예전에는 스웨거(Swagger) 규격이라고 알려졌던 훌륭한 규격의 현재 이름은 [OpenAPI](#)다. 이것은 여러분의 API를 설명하는 도움말 문서를 동적으로 생성한다. 그 결과물은 JSON/YAML 형식이다. 그리고, 이것을 사용하면, 개발자들은 자신들이 구축한 API의 스웨거(Swagger) 인스턴스를 생성할 수 있다. Swagger의 생태계는 API 문서화 도구들 중에서 가장 크다. Swagger가 활성화되어 있는 API들이 제공하는 것은 상호작용하는 도움말 문서, 클라이언트 SDK 생성, 탐색 능력 등이다.

RAD 스튜디오를 사용하면, Swagger의 규격들을 직접 활용하는 문서를 RAD 서버가 제공하도록 할 수 있다. 개발자는 애트리뷰트를 사용하면 된다. 그 애트리뷰트로는, Summary, Parameters, Details/Responses 등이 있다다.



note

델파이는 애트리뷰트를 제공하지만, C++ 언어는 애트리뷰트를 지원하지 않는다. 아마 C++빌더를 사용하는 개발자라면 이미 알고 있을 것이다. 하지만, 우리가 이미 앞에 있는 장들을 통해 봤듯이, C++빌더에는 우회해서 실현하는 방법이 있다. 이 기능도 마찬가지다. 여러분은 C++ 언어로도 여러분의 API들을 문서화할 수 있다.

Swagger UI를 RAD 서버 안에 심어 넣기(embedding)

RAD 서버는 정적인 파일들을 제공할 수도 있다. 이 능력을 활용하면, RAD 서버에서 직접 Swagger UI를 호스팅할 수 있다. 이는 여러분의 웹사이트에 있는 프론트엔드를 호스팅하기 위한 정적인 파일들을 RAD 서버가 서비스하는 것과 그 방식이 같다.

방법은 간단하다. EMServer.ini 파일에 접근해서, 키 중에 [Server.PublicPaths]를 찾는다. 그 아래를 보면, Swagger UI 경로를 가리키는 줄이 있다. 그 줄의 주석을 끈다. 그리고, 올바른 경로를 명시한다. 그러면 된다.

앞의 장들에서도 언급했지만, 이 ini 파일은 여러분의 개발 장비 안에서 찾을 수 있다. 경로는 아래와 같다:
C:\Users\Public\Documents\Embarcadero\EMS\emsserver.ini

```
[Server.PublicPaths]
...
;# 아래 엔트리는 swagger-ui의 최상위(root) 경로를 명시하고 있다
Path3={"path": "swagger-ui", "directory": "C:\\swagger-ui\\", "default": "index.html",
"extensions": ["css", "html", "js", "map", "png"], "charset": "utf-8"}
```

이렇게 정의했다면, RAD 서버를 시작하라. 그리고 URL/swagger-ui/를 접속하라. 예:

<http://localhost:8080/swagger-ui/>

The screenshot shows the Swagger UI interface for the RAD Server API Documentation. The title bar indicates the URL is `localhost:8080/swagger-ui/`. The main content area is titled "RAD Server API Documentation 1.0.0". It includes a "Schemes" dropdown set to "HTTP". Below it, there's a section for "Api Doc" containing several API endpoints listed as buttons:

- GET /api Get API EndPoints
- GET /api/apidoc.json Get JSON
- GET /api/apidoc.yaml Get YAML
- GET /api/{item}/apidoc.yaml Get API Endpoint

Swagger 메인 페이지의 모습. 위에서는 apidoc.json 정의를 사용하고 있다

위 화면에서 스웨거 UI 참조가 자동으로 참조하고 있는 규격은 JSON이다. 우리는 이 요청을 /api/apidoc.yaml로 바꿀 수도 있다. 그러면, yaml 규격이 나타날 것이다.

또한, RAD 서버 안에 내장되어 있는 앤드포인트들은 그 모두가 이미 완전하게 문서화되어 있다. 즉, 해당 규격 파일들이 이미 만들어져 있다.



Swagger UI 파일들은 RAD 스튜디오와 함께 제공된다. 그 경로는: "C:\Program Files (x86)\Embarcadero\Studio\XX.0\ObjRepos\en\EMS\swagger-ui"다. [공식 리포지토리](#)로 가도 이 파일들을 다운로드 받을 수 있다.

사용자 지정 도움말 문서를 만들기

예시

RAD 스튜디오에는 훌륭한 예시가 함께 제공된다. 그것을 보면, 문서화와 관련된 모든 가능성들 그리고 사용할 수 있는 모든 애트리뷰트들에 대해 자세히 알 수 있다. 이 예시는 델파이와 C++ 모두에서 제공되며, 경로는 다음과 같다:

Delphi:

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

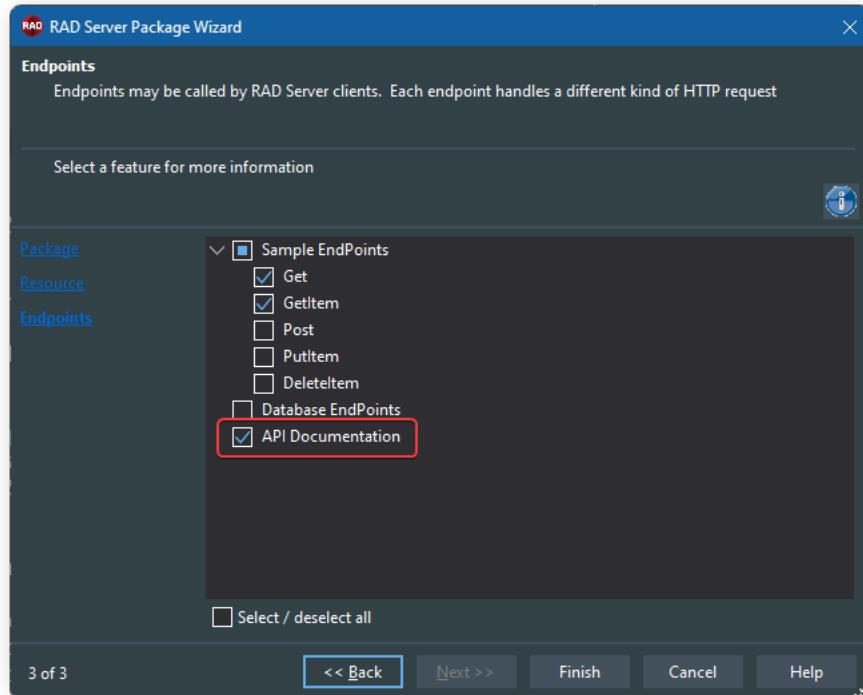
C++:

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



이 장에서 제공하는 예시 프로젝트는 RAD 스튜디오와 함께 제공되는 예시를 업데이트한 버전이다. RAD 스튜디오에 새로 추가된 여러줄-문자열 리터럴 기능을 활용하는 코드로 업데이트 했기 때문에, 이 프로젝트는 12 아테네 또는 그 이상의 버전에서만 작동한다.

또다른 선택이 있다. 여러분은 새 RAD 서버를 생성할 때, 해당 마법사 안에 있는 API Documentation 체크박스를 선택할 수도 있다. 그러면, 자동으로 문서화와 관련된 기본 애트리뷰트들이 채워질 것이다.



마법사 안에서 API Documentation 자동 생성 옵션을 선택하면 된다

EndPointRequestSummary

메서드에 대해 설명한다:

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

- **Tags:** tag를 정의한다.
- **Summary:** 메서드 제목.
- **Description:** 메서드 설명.
- **Produces:** API가 생산할 수 있는 MIME 유형. 이것은 모든 API들에 전역적으로 반영된다. 하지만, 특정 API 호출 시 이것을 오버라이딩 override 할 수도 있다. 그 값은 반드시 Mime Types 아래에 명시되어야 한다.
- **Consume:** API가 소비할 수 있는 MIME 유형. 이것은 모든 API들에 전역적으로 반영된다. 하지만, 특정 API 호출 시 이것을 오버라이딩 override 할 수도 있다. 그 값은 반드시 Mime Types 아래에 명시되어야 한다.

EndPoint의 GET 하나에 대한 설명을 선언하는 예시:

Delphi

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

C++

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

EndPointRequestParameter

요청 하나 안에서 사용되는 파라미터들에 대한 설명.

파라미터의 이름과 그 파라미터가 들어가는 위치를 조합해 정의하면, 고유한 파라미터 하나가 정의된다.

파라미터 유형에는 다섯 가지가 있다: Path, Query, Header, Body, Form.

```
[EndPointRequestParameter('ParamIn', 'Name', 'Description', 'Required', 'ParamType',
'ItemFormat', 'ItemType', 'Schema', 'Reference')]
```

- **ParamIn:** 그 파라미터의 위치: Path, Query, Header, Body, Form.
- **Name:** 그 파라미터의 이름. 파라미터 이름은 대소문자를 가린다.
 - 그 파라미터의 위치가 'Body'인 경우, 그 이름은 반드시 'body'라고 적어야만 한다.
 - 그 파라미터의 위치가 'Path'인 경우, 그 이름은 반드시 Paths 오브젝트 안에 있는 path 필드에 연결된 해당 경로 조각과 일치해야 한다.
 - 그 외의 경우에는, 그 이름이 ParamIn과 일치한다.
- **Description:** 그 파라미터에 대한 간략한 설명. 여기에는 사용 예시가 포함될 수 있다. GFM 문장 규칙을 사용해 리치 텍스트(rich text)를 표현할 수 있다.
- **Required:** 그 파라미터의 필수 여부. 그 파라미터가 'Path' 안에 위치한다면 필수다. 따라서, 그 경우에는 이 값은 반드시 True여야 한다, 그 이외의 경우에는 선택 사항이며, 그 디폴트는 False다.

- **ParamType:** 그 파라미터의 타입. ‘Body’가 아니라 다른 값이라면, 그 값은 반드시 다음 값들 중 하나여야 한다: ‘spArray’, ‘spBoolean’, ‘spInteger’, ‘spNumber’, ‘spNull’, ‘spObject’, ‘spString’, ‘spFile’. 만약 ParamType이 ‘spFile’이라면, consume MIME 유형은 반드시 “multipart/form-data”와 “application/x-www-form-urlencoded” 중 하나여야 하며, 그 파라미터는 반드시 “form-data” 안에 있어야 한다. ‘Body’ 값의 경우, JSONSchema와 Reference가 꼭 필요하다.
- **ItemFormat:** ParamType을 위한 확장 형식: ‘None’, ‘Int32’, ‘Int64’, ‘Float’, ‘Double’, ‘Byte’, ‘Date’, ‘DateTime’, ‘Password’.
- **ItemType:** ParamType이 ‘array’인 경우에 필요하다. 그 배열 안에 있는 항목의 타입을 명시한다.
- **Schema:** 서버로 보내지는 Primitive의 Schema Definition. 그 body의 요청 구조에 대한 정의. 만약, ParamType이 ‘Array’ 또는 ‘Object’인 경우, 스키마를 JSON 그리고/또는 YAML 형식으로 정의할 수 있다.
- **Reference:** 서버로 보내지는 Primitive의 Schema Definition. 그 body의 요청 구조에 대한 정의. 만약, ParamType이 ‘Array’ 또는 ‘Object’인 경우, 스키마를 정의할 수 있다. 예: '#/definitions/pet'

파라미터 정의 예시:

Delphi

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

C++

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

EndPointResponseDetails

요청(request)의 응답(response)에 대한 설명.

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

- **Code:** 그 응답의 코드.

- **Description:** 그 응답 코드에 대한 설명.
- **PrimitiveType:** 반환되는 [Primitive](#) 타입. Swagger Specification 안에 있는 Primitive 데이터 타입들은 JSON-Schema Draft 4가 지원하는 타입들을 기반으로 한다. JSON Schema는 JSON 값을 위한 프리미티브 타입들으로 일곱 가지를 정의하고 있다: 'spArray', 'spBoolean', 'spInteger', 'spNumber', 'spNull', 'spObject', 'spString'. [JSON Schema primitive types](#) 참조. 추가적인 프리미티브 타입인 'spFile'은 해당 Parameter Object 그리고 해당 Response Object에서 사용한다. 이는 그 파라미터 타입 또는 그 응답이 파일이 될 수 있도록 해준다.
- **PrimitiveFormat:** 반환되는 Primitive의 형식: 'None', 'Int32', 'Int64', 'Float', 'Double', 'Byte', 'Date', 'DateTime', 'Password'.
- **Schema:** 서버로 보내지는 Primitive의 Schema Definition. 그 body의 요청 구조에 대한 정의. 만약, ParamType이 'Array' 또는 'Object'인 경우, 스키마를 JSON 그리고/또는 YAML 형식으로 정의할 수 있다
- **Reference:** 서버로 보내지는 Primitive의 Schema Definition. 그 body의 요청 구조에 대한 정의. 만약, ParamType이 'Array' 또는 'Object'인 경우, 스키마를 정의할 수 있다. 예: '#/definitions/pet'

응답 정의 예시:

Delphi:

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

C++:

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

EndPointObjectsDefinitions

오브젝트 정의를 JSON 그리고/또는 JSON 형식으로 정의하는 것이 가능하다. 예시를 보자:

Delphi

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

C++

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

하지만, 이 Definition들을 어떻게 명시할까? 길이 때문에, 짧은 델파이 예시를 적어 놓았다. 완전한 예시는, RAD 스튜디오와 함께 제공되는 Sample 안에 들어 있는 예시를 확인하기 바란다.

Delphi

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



tip 명심할 점이 있다. YAML 파일을 정의할 때는 들여쓰기에 주의하는게 너무나 중요하다. YAML에는 오브젝트 구조 형식이 따로 없기 때문에, 들여쓰기가 그 구조의 계층을 결정한다. 따라서 여러줄 문자열을 사용한다면, 닫기 위해 사용하는 따옴표 세 개의 위치를 확인하는 것이 중요하다. 참고로, JSON은 중괄호를 사용해 구조를 정의하므로 이런 문제가 없다.

EMSDatasetResource에 애트리뷰트들을 정의하기

EMSDatasetResource는 훌륭한 컴포넌트다. 이것은 표준 CRUD 앤드포인트를 매우 빠르게 만들 수 있도록 도와준다. 하지만, 우리가 정의하고 싶은 애트리뷰트들을 처음부터 각 앤드포인트에 반영할 수도록 접근을 허용하지는 않는다. 겉으로 드러나지는 않지만, 이 컴포넌트가 힘든 일들의 대부분을 수행하기 때문이다.

디폴트로, 정의되는 애트리뷰트는 EndPointRequestSummary 뿐이다. RAD 서버는 일반적인 CRUD 정의를 하는데, 이때 주된 키로 사용되는 것은 {id}다. 또한 그 앤드포인트들은 테스트가 제공되지 않는다. 이것을 교정하려면, 우리는 주된 키 이름을 정의할 뿐만 아니라 각 액션(action)과 그것의 사용자 지정 상세를 정의하면 된다.

Delphi

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

C++

```
std::unique_ptr<EndPointRequestParameterAttribute>
  ResponseParameter(new EndPointRequestParameterAttribute(
    TAPIDocParameter::TParameterIn::Path,
    "CUST_NO", // Param name
    "Customer number", // desc
    true, // required
    TAPIDoc::TPrimitiveType::spInteger,
    TAPIDoc::TPrimitiveFormat::Int64,
    TAPIDoc::TPrimitiveType::spInteger,
    "", // Schema
    "")); // Reference
attributes->AddRequestParameter("dsrCUSTOMER.Get", ResponseParameter.get());
```

이 장에 해당하는 GitHub 리포지토리로 가면, 우리가 3 장에서 사용했던 프로젝트를 찾을 수 있을 것이다. 하지만, 사용되었던 EMSDatasetResource 두 개가 커스터마이징 되었는데, 그 관행(convention)들에 대한 보다 상세한 정보는 README.md 파일 안에 있다. 또한 그 유닛들 안에 주석으로도 설명되어 있다.



tip

YAML과 JSON 스펙을 제공하려면, 각각 정의를 해야 한다. 쉬운 방법이 있다. 일단 모든 정의를 YAML로 작성하라 (사람이 읽기에 더 좋기 때문이다). 그리고 나서 온라인 변환기 또는 GPT 봇 같은 것을 사용해, 여러분을 위해 그에 대응하는 JSON 정의를 자동으로 생성하라.

13

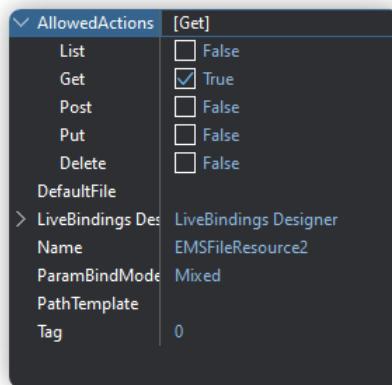
파일 관리 및 스토리지



이 장에서, 우리는 RAD 서버가 파일을 관리하고 인프라 스토리지에 접근하는 여러가지 방법들을 분석한다. 또한 파일을 생성하거나 소비할 수 있는 엔드포인트 각각에게 파일의 유형들을 전역적으로(globally) 명시하는 방법 또는 각 엔드포인트별로 세분화된 방식으로 파일 유형을 지정하는 방법을 살펴본다.

TEMSFileResource

TEMSFileResource 컴포넌트는 TEMSDatasetResource와 매우 비슷한 방식으로 작동한다. 덕분에, 우리는 훨씬 편하게 작업할 수 있다. 이 컴포넌트는 파일 관리를 위한 비즈니스 로직 대부분을 추상화한다. 그리고 필요한 모든 코드를 대신해 애트리뷰트(attribute)만 지정하면 되도록 한다.



TEMSFileResource의 주요 프로퍼티들

AllowedActions: 디폴트에서는, 오직 GET(얻기)만 활성화된다. 하지만, 우리는 이 컴포넌트에게 허용되는 액션들을 직접 지정할 수 있다. 그 액션들에는 파일들의 목록을 얻기, 파일을 업로드, 업데이트, 삭제하기 등이 있다.

DefaultFile: 만약, GET 파일 요청을 받았는데, 요청하는 파일이 무엇인지가 파라미터에 담겨있지 않은 경우, DefaultFile에 지정된 파일을 반환한다.

PathTemplate: 이것은 단순하면서도 강력한 프로퍼티다. 여기에 지정되는 가장 기본적인 예를 넣어 설명하겠다. c:\temp\{id} 값을 지정했다고 가정하자. 이 값은 절대 경로다. 그 경로에는 이 컴포넌트가 다루는 파일들이 저장된다. 그리고, {id}는 와일드카드다. 이것은 컴포넌트의 애트리뷰트 안에 명시된 {id}와 매칭된다. 따라서, 우리는 요청의 {id} 파라미터에 파일 이름을 넣어서 전달할 수 있다. 이처럼, 프로퍼티 안에서 중괄호를 사용하면, 보다 복잡한 경로를 만들 수 있다. 예를 들어, 와일드 카드를 사용해 하위폴더, 파일 확장자 등을 가리키도록 할 수 있다. 예시: c:\uploads\{folder}\{file}.{extension}. 이 예시에서 파라미터는 3개다. 이 경우, 우리는 이 컴포넌트의 애트리뷰트 안에 folder, file, extension을 명시해야 한다. 예시를 보자.



note

대체로, PathTemplate에 지정하는 값은 환경 (debug 또는 release)에 따라 다르게 지정하는 경우가 많다. 따라서 그 값이 환경에 따라 변할 수 있도록 작성하기를 권장한다. 컴파일 지시어를 사용하면 된다.

예시

ResourceSuffix 엔트리 3 개를 데이터 모듈 안에 있는 EMSFileResource 정의 앞에 명시한다.

Delphi

```
TFilesResource1 = class(TDataModule)
[ResourceSuffix('list', './')]
[ResourceSuffix('get', './{id}')]  

[ResourceSuffix('post', './')]
EMSFileResource1: TEMSFileResource;
```

C++

```
static void Register()
{
    std::unique_ptr<TEMSResourceAttributes> attributes(new  

TEMSResourceAttributes());
    attributes->ResourceName = "test";
    attributes->ResourceSuffix["PostUpload"] = "./upload";
    attributes->ResourceSuffix["EMSFileResource1"] = "./fileResource";
    attributes->ResourceSuffix["EMSFileResource1.List"] = "./";
    attributes->ResourceSuffix["EMSFileResource1.Get"] = "./{id}";
    attributes->ResourceSuffix["EMSFileResource1.Post"] = "./";
```

```
    RegisterResource(__typeinfo(TDataResource1), attributes.release());
}
```

위 예에서, 우리는 액션 세 개를 정의했다: list, get, post이다. 하지만, put, delete 등도 같은 방식으로 정의할 수 있다. test/fileResource/ 엔드포인트는 이제 “PathTemplate” 프로퍼티에 해당하는 모든 필드들을 나열해(list) 제공한다. 특정 파일 하나를 접근하고 싶다면, {id} 와일드 카드를 사용하면 된다. 또한 새 파일을 업로드 할 수도 있다. 알아둘 중요한 점이 있다. 이 컴포넌트를 사용해 파일을 업로드하려면 반드시 해당 파일의 바이너리 결과물을 body 안에 담아서 전달해야 한다. 이 컴포넌트로는, 멀티-파티 폼즈(multi-part forms)를 사용해 여러 개의 파일을 업로드 하는 것이 가능하지 않다 (이것을 실현하는 방법에 대한 예는 뒤에 나온다).



C++에서 TEMSFileResource를 사용하려면, emsserverresource.bpi 라이브러리를 “requires” 구역 안에 추가해야 한다. 그 파일이 있는 위치는 다음과 같다:
C:\Program Files (x86)\Embarcadero\Studio\XX.0\lib\{Platform}\release\emsserverresource.bpi

코드에서 파일을 다루기

조금 더 복잡한 상황이라서, TEMSFileResource 만으로는 충분하지 한다면, 코드에서 파일을 접근하는 것도 가능하다. 각 요청에는 ARequest 파라미터가 있다. 우리는 이것을 사용해 body 안에 있는 파일들에 접근할 수 있다. 이것들은 반드시 멀티-파티 폼(multi-part form)으로 보내진 것이어야 한다. 코드를 보자:

Delphi

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

C++

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

위 예시에서, 우리는 body의 part 0에 접근한다. (물론, body 안에 part들 즉 파일들이 여러개 있는 경우도 있다) 그리고 나서, 그 파일을 주어진 경로 안에 저장한다. 파일을 저장하기 위해서는 TFile 클래스를 사용한다. body 안에 있는 해당 스트림을 읽어서, 그 TFile 변수 안에 넣는다. 정말로 이렇게 간단하다. 리포지토리 안에 있는 예시의 코드는 이것보다 조금 더 복잡하다. 거기에서는 반복(loop)을 사용해, body 안에 있는 모든 part마다 파일을 읽어서 처리한다. 즉, 파일 여러 개를 하나의 엔드포인트에게 업로드하는 예시가 들어 있다.

Content-Type HTTP 헤더들

RAD 서버가 새 EndPoint(엔드포인트) 애트리뷰트들을 통해 Content-Type과 Accept 기반 맵핑을 지원한다. 덕분에, 리소스 맵핑이 더 좋아졌다. URL에만 의존하는게 아니기 때문이다. 즉, 여러분은 URL과 HTTP 동사(verb)들 다 같은 곳에 두 개의 다른 메서드를 맵핑할 수 있다. 그리고, 요청에 맞춰, 다른 데이터 타입을 반환할 수 있다.

새로 추가된 EndPoint 애트리뷰트는 이 두가지다:

- **EndpointProduce:** 여기에는 이 엔드포인트가 GET 메서드에 대한 응답으로 생성할 수 있는 MIME 타입 / 파일 확장자를 명시한다. 이 엔드포인트가 수행할 메서드를 무엇인지는 HTTP request header 안에 있는 Accept에 의해 결정된다.
- **EndpointConsume:** 여기에는 이 엔드포인트가 PUT, POST, PATCH를 통해 소비할 수 있는 MIME 타입 / 파일 확장자를 명시한다. 이 엔드포인트가 수행할 메서드를 무엇인지는 HTTP request header 안에 있는 Content-Type에 의해 결정된다.

지금부터, 우리는 RAD 서버 애플리케이션 리소스들이 EndpointProduce 애트리뷰트를 어떻게 사용하는지를 볼 것이다. 그래서, REST Get 요청들이 왔을 때 그 요청이 무엇을 원하는지에 따라, 실제로 수행될 메서드가 달라진다. 이 예에서 사용하는 타입은, image/jpeg과 application/xml MIME 타입이다.

간단한 예시

프로젝트 패키지 마법사를 사용해 RAD 서버 프로젝트 하나를 시작한다. 유형에서 Data Module File을 선택하고, Resource 이름을 AcceptTypes라고 지정한다. 모든 표준 EndPoints들의 선택을 해제하고, Finish 버튼을 클릭한다.

새 폴더를 하나 만든다. 경로는 c:\temp\page 가 되도록 한다. 그리고, 그 경로 안에, 아무 jpeg 이미지를 넣고 그 이름을 content.jpeg 라고 지정한다. 이어서, 아무 텍스트 파일을 넣고 그 이름을 content.txt 라고 지정한다.

Delphi

여러분의 리소스 클래스의 published 구역 안에, 메서드 두 개를 선언한다. 그리고, 각 메서드마다 애트리뷰트로 ResourceSuffix와 EndpointProduce를 아래와 같이 추가한다.

```
[ResourceName('AcceptTypes')]
TAcceptTypesResource1 = class(TDataModule)
published
  [ResourceSuffix('*')]
  [EndpointProduce('image/jpeg')]
  procedure GetImage(const AContext: TEndpointContext;
```

```

const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
[ResourceSuffix ('*')]
[EndpointProduce ('application/xml')]
procedure GetText(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
end;

```

implementation 구역 안에, 아래 코드를 추가한다. GetImage와 GetText 메서드가 실행할 코드다.

```

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

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

```

C++

ServerUnit.h

```

// EMS Resource Modules
//-----
#ifndef ServerUnitH
#define ServerUnitH
//-----
#include <System.Classes.hpp>
#include <System.SysUtils.hpp>
#include <EMS.Services.hpp>
#include <EMS.ResourceAPI.hpp>

```

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

ServerUnit.cpp

아래 코드를 추가한다. GetImage와 GetText 메서드가 실행할 코드다. Register 함수 안에는, ResourceSuffix들과 EndpointProduce 애트리뷰트들을 추가한다. 각각 GetImage와 GetText 엔드포인트를 위한 것들이다.

```
//-
#pragma hdrstop

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

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

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

}

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

#pragma startup Register 32
```

RAD 서버 프로젝트를 저장하고 빌드한다. 이제 이 content.jpeg와 content.txt에 접근이 된다. 무슨 파일에 접근하게 되는지는 요청의 header 안에 들어 있는 Content-Type에 의해 결정된다.