



RAD Server (라드 서버) 기술 가이드

```
localhost:8080/rad-server/details
1  {
2    "title": "RAD Server Technical Guide",
3    "edition": 3.0,
4    "authors": [
5      {
6        "name": "Antonio Zapater",
7        "revised": ["2024-04-15", "2025-10-20"]
8      },
9      {
10       "name": "David I",
11       "revised": ["2019-04-05"]
12     }
13   ],
14   "examplesURL": "github.com/embarcadero/radserver-docs",
15   "copyright": "©2019-2025"
16 }
```

머리말

RESTful 아키텍처는 현대 “API 우선” 애플리케이션 설계를 뒷받침하는 핵심 원동력이다. 이 책은 RAD Server 프레임워크에 집중한다. RAD Server 프레임워크는 이런 플랫폼들을 개발하는 용도이며 RAD Studio (Delphi/C++Builder)에 함께 들어 있다.

RAD Server는 기능이 풍부한 백엔드 MEAP(모바일 엔터프라이즈 애플리케이션 플랫폼)이다. 데스크탑, 모바일, 웹 프론트 엔드를 어느 언어로든 개발해 연결할 수 있다. 그리고 이 책은 RAD Server 개발자를 위한 표준 참고서다.

MEAP의 장점은 사전 구축된 클라우드 또는 온프레미스 서버를 여러분이 가질 수 있다는 점이다. 그리고 그 서버 안에 내장되어 있는 많은 핵심 기능들(푸시 알림, 사용자 추적, 분석 등)을 활용할 수 있다. 그래서, 여러분은 신속하게 꽃아 넣는 방식으로 원격 데이터베이스를 연결과 기능 접근을 제공할 수 있다.

Embarcadero(엠바카데로) RAD Server 가이드는, 원래 David I가 저술했다(2019). 이 제 3판은 Antonio Zapater가 다시 작성했다 (2024, 2025). 그래서 여기에는 출시 후 시장의 요구를 지속적으로 반영하며 추가된 많은 RAD Server 기능들이 들어 있다. 이 판은 각 장마다 [종합적인 동영상 시리즈들](#)이 제공된다. 그리고, 각 시리즈의 소스코드 예시들이 깃허브에 있다. <https://github.com/embarcadero/radserver-docs>



동영상

이 책과 연결되는 모든 동영상 시리즈들은 [유튜브에서 볼 수 있다](#). 그리고 모든 예시들을 이 [깃허브 리포지토리](#)에서 다운로드할 것을 적극 권장한다.

목차

01 - RAD Server (라드 서버)란? 소개	8
RAD Server 개요	8
RAD Server 기반 애플리케이션 구축하기 -7가지 핵심 방향	10
RAD Server 애플리케이션 구축을 위한 요구 사항	11
RAD Studio IDE를 사용하려면	11
RAD Server 테스트 및 배포 라이선스	11
RAD Server 핵심 기능 요약	11
핵심 기능들	11
참고 자료	13
02- RAD 마법사를 사용해 "Hello World" 만들기	14
REST-기반 서비스를 구축하기	15
RAD Server 프로젝트 마법사를 사용하기	15
마법사 RAD Server 프로젝트 및 소스 코드	19
여러분의 첫 애플리케이션을 위해 RAD Server를 구성하기	20
여러분의 첫 애플리케이션을 테스트하기	25
참고 자료	27
03 - 여러분의 첫 CRUD 애플리케이션 만들기	28
CRUD 기능을 갖춘 REST-기반 서비스를 구축하기	28
생성된 프로젝트를 설명하기	31
프로젝트를 빌드하고 테스트하기	33
TEMSDatasetResource의 추가 기능들	34
04 - REST Debugger	37
REST Debugger란 무엇이며 어디에서 찾을 수 있는가	37
(REST Debugger를 사용해) 우리의 첫 PUT 요청을 보내기	38
REST Debugger가 제공하는 기타 기능들	40
05 - FireDAC의 일괄 이동 및 JSONWriter 사용하기	41
(메모리 스트리밍을 사용해) JSON 데이터베이스의 데이터를 반환하기	41
FireDAC(파이어닥)의 BatchMove, BatchMoveDataSetReader, BatchMoveJSONWriter 사용하기	44
참고 자료	47
06 - JSONValue, JSONWriter, JSONBuilder	48
JSON 데이터를 다루는 프레임워크들	48
JSONValue를 사용하기	49
JSON 클래스를 사용하는 예시	50
JSONWriter를 사용하기	52
JSONWriter 사용 예시	52
JSONBuilder를 사용하기	54
참고 자료	55

07 - 사용자 정의 엔드포인트 만들기.....	57
모범 사용 관행의 예시.....	57
API가 너무 수다스럽지 않도록 하기.....	58
하위-리소스들을 추가하기.....	58
중첩되는 데이터를 응답에 추가하기 (마스터/디테일).....	59
이 새 구현을 테스트하기.....	64
사용자 정의 GET, POST, PUT, DELETE 메서드를 생성하기.....	67
응답 오류들을 처리하기.....	69
참고 자료.....	69
08 - 내장된 분석정보에 접근하기.....	70
주요 특징.....	70
RAD Server Console에 접근하기.....	71
09 - RAD Server를 배포하기.....	74
RAD Server는 어디에 배포될 수 있는가.....	74
설치 프로그램을 GetIt(겟잇)에서 받아서 사용하기.....	75
RAD Server를 수작업으로 배포하기 위한 전제 조건들.....	75
Windows에 수작업으로 배포하기.....	76
InterBase Server 엔진.....	76
RAD Server 설치.....	77
웹 서버 (IIS 또는 Apache).....	80
리눅스에 수작업으로 배포하기.....	81
호환되는 배포판들.....	81
InterBase Server 엔진을 설치하기.....	81
InterBase Server를 등록하고 시작하기.....	82
InterBase를 서비스로 실행하기.....	82
RAD Server를 설치하기.....	83
Apache용으로 RAD Server를 설정하기.....	84
Docker(도커)에 배포하기.....	85
옵션 1: PA-RADServer-IB.....	86
옵션 2: PA-RADServer.....	86
RAD Studio에서 컴파일된 RAD Server 모듈들을 복사하기.....	87
EMSServer.ini 파일을 구성하기.....	88
10 - RAD Server Lite(라이트).....	89
Lite 버전이란?.....	89
RAD Server Lite 라이선스를 얻는 방법.....	90
RAD Server Lite 프로젝트를 배포하기.....	90
배포할 파일들.....	91
수작업으로 배포.....	91
배포 마법사를 사용하기.....	91

MSVS 런타임.....	92
실전 운영 데이터베이스를 생성하기.....	92
프록시 구성.....	93
리눅스용 구성.....	93
11 - 인증 및 권한부여.....	94
내장된 인증: 사용자 및 그룹 관리하기.....	94
로그인 하기.....	96
로그아웃 하기.....	97
회원가입 하기.....	98
그룹(Group)을 관리하기.....	99
내장된 권한부여(Integrated authorization).....	99
전역 자격 증명(Global Credentials).....	99
사용자 그리고 그룹에 권한부여하기.....	100
사용자 정의 인증 (Custom authentication).....	101
사용자 정의 권한부여 (Custom authorization).....	104
RAD Server Management Console(관리 콘솔).....	105
새 프로파일을 생성하기.....	106
사용자들과 그룹들을 관리하기.....	106
RSConsole 안으로 더 깊이 들어가기.....	108
12 - 여러분의 엔드포인트들에 대한 문서화와 테스트를 위해 OpenAPI (Swagger) 사용하기.....	109
OpenAPI/Swagger란 무엇인가 그리고 우리는 왜 이것을 사용하는가?.....	109
Swagger UI를 RAD Server 안에 심어 넣기(embedding).....	109
사용자 정의 문서집을 만들기.....	111
예시.....	111
EndPointRequestSummary.....	112
EndPointRequestParameter.....	113
EndPointResponseDetails.....	114
EndPointObjectsDefinitions.....	115
EMSDatasetResource에 애트리뷰트들을 정의하기.....	116
13 - 파일 관리 및 스토리지.....	119
TEMSFileResource.....	119
예시.....	120
코드에서 파일을 다루기.....	121
Content-Type HTTP 헤더들.....	122
간단한 예시.....	122
14 - EdgeModule들: RAD Server를 확장 하기.....	126
EdgeModule들을 이해하기.....	126
TEMSEdgeService 컴포넌트.....	127
컴포넌트 설정.....	128

리소스 구현.....	128
기본적인 예시를 만들기.....	129
필수 컴포넌트들을 추가하기.....	129
리소스를 생성하고 등록하기.....	130
실제 적용(Practical Application)들.....	132
1. 레거시 시스템 통합.....	132
2. 특화된 처리 서비스들.....	132
3. 크로스-플랫폼 통합.....	133
4. 하드웨어 및 IoT 통합.....	133
EdgeModule들과 InterBase의 ChangeViews를 통합하기.....	133
InterBase ChangeViews란 무엇인가?.....	133
ChangeViews라는 도전 과제.....	133
데모가 작동하는 방식.....	134
이 방식의 장점.....	135
결론.....	135
15 - RAD Server와 WebStencils(웹스텐실즈).....	136
WebStencils란 무엇인가?.....	136
RAD Server와 WebStencils를 통합하기.....	136
WebStencils Processor들을 사용하기.....	137
WebStencils Engine을 사용하기.....	138
정적 JS, CSS, 이미지 등을 처리하기.....	139
프론트엔드 소스.....	139
더 자세히 알아보기.....	139
16 - RAD Server 멀티-테넌시 지원.....	140
멀티-테넌시(Multi-Tenancy) 소개.....	140
RAD Server 멀티-테넌시의 주요 장점.....	140
RAD Server 안에 있는 테넌트 아키텍처를 이해하기.....	141
RAD Server 안에서 멀티-테넌시를 활성화하기.....	141
RAD Server Console을 사용하기.....	141
테넌트 관리.....	142
EMS Multi-Tenant Console.....	142
멀티-테넌시 구현 모델들.....	143
공유 데이터베이스에서 테넌트 필터링을 적용.....	143
테넌트별-데이터베이스.....	144
멀티-테넌트 RAD Server 애플리케이션을 개발하기.....	145
리소스 구현 안에서 테넌트 정보를 접근하기.....	145
테넌트별-필터가 적용되는 리소스를 구현하기.....	145
익명 테넌트 리소스들.....	147
클라이언트-측 구현.....	148

TEMSProvider 컴포넌트 (멀티-테넌트 RAD Studio 클라이언트 앱 안).....	149
(테넌트 안의) 사용자 및 그룹 관리를 구현하기.....	150
샘플 애플리케이션: RAD Server 멀티-테넌트 데모.....	150
결론.....	150
17 - 엔드포인트를 외부 클래스에 매핑하기.....	151
RAD Server 패키지 애플리케이션을 만들기.....	152
만든 RAD Server 애플리케이션을 테스트하기.....	156
참고 자료.....	158
18 - 리소스 인터셉터를 통해 RAD Server를 확장하기.....	159
리소스 인터셉터들을 이해하기.....	159
표준 RAD Server 기능들 만으로 부족할 때.....	160
내장된 방식을 넘어서는 멀티-테넌시.....	160
테넌트별-데이터베이스 (Database-Per-Tenant) 아키텍처를 구현하기.....	160
고급 테넌트 격리 시나리오.....	161
사용자 정의 인증과 권한 부여.....	161
엔터프라이즈 ID 통합.....	161
더 수준이 높은 로그 기록과 모니터링.....	162
광범위하게 요청을 추적하기.....	162
데이터 검증과 변환.....	163
중앙 집중식 입력 처리.....	163
성능 최적화와 캐싱(caching).....	164
지능형 응답 캐싱.....	164
요청 횟수 제한하기(Rate Limiting) 및 API 보호.....	165
수준높게 요청 횟수를 제한하기.....	165
상속을 통해 재사용하는 인터셉터 로직.....	166
기본 인터셉터(Base Interceptor)를 만들기.....	166
기본 인터셉터를 확장하기.....	169
외부 시스템들과의 통합.....	171
이벤트-주도(Event-Driven) 아키텍처 통합.....	171
최고의 실무 적용 관행 및 고려 사항.....	171
결론.....	172

01

RAD Server (라드 서버)란? 소개

지금 이 세상의 컴퓨팅 환경은 더 이상 데스크탑, 디바이스, 서버, 데이터 센터에 국한되지 않는다. 애플리케이션은 데스크탑(desktop)을 넘어 여러 디바이스들, 네트워크 단말(edge) 연결들, 사내 장비(on-premise)들, 퍼블릭 및 하이브리드 클라우드 서비스들로 확대되고 있다. RAD Server(라드 서버)와 RAD Studio(라드 스튜디오)는 회사(와 고객)이 요구하는 광범위한 컴퓨팅 니즈들과 비즈니스 요구 사항들을 충족하는 솔루션을 여러분이 구축할 수 있도록 한다.

이 문서집은 여러분이 얼마나 신속하게 서비스-기반 멀티-계층 애플리케이션을 설계/구축/디버그/배포할 수 있는지를 보여준다. RAD Server의 REST-기반 API 호스팅 엔진, 구성 요소들, 기술들을 활용하면 된다. 여러분은 RAD Studio, Delphi(델파이), C++Builder(C++빌더) 엔터프라이즈와 아키텍트 에디션 안에서 사용할 수 있다.



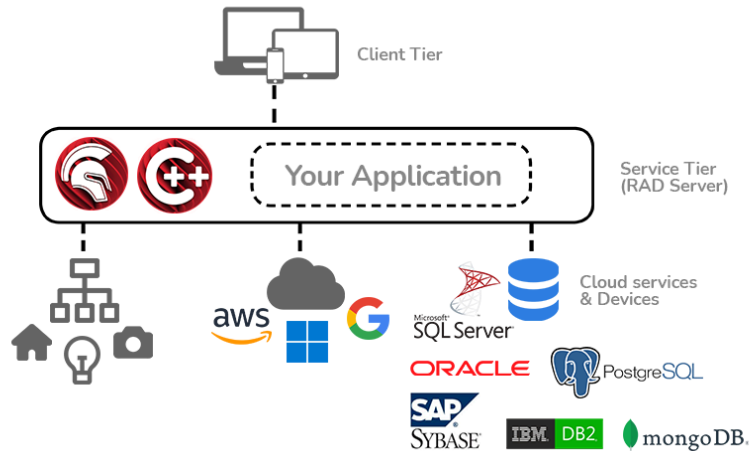
참고

이 RAD Server 문서집과 소스 코드 전반 걸쳐, EMS(엔터프라이즈 모빌리티 서비스)라는 용어가 사용된다. 원래 이름은 EMS였기 때문이다. 이제는 이름이 RAD Server다.

RAD Server 개요

Embarcadero(엠바카데로)의 RAD Server는 툰키 방식의 백엔드 솔루션이다. 여러분은 Delphi와 C++Builder를 사용해 서비스 기반 애플리케이션을 빠르게 구축하고 배포할 수 있다. RAD Server는 REST(Representational State Transfer) 프로토콜을 지원한다. 파라미터들을 JSON(또는 XML) 형식으로 전달하고 반환한다. 여러분은 RAD Server를 통해 여러분의 API들을 게시할 수 있다. 그리고 RAD Server에 연결된 사용자/장비 관리, 앱 사용성

분석을 할 수 있다. 또한, FireDAC 컴포넌트를 사용해 로컬 및 엔터프라이즈 데이터베이스 연결 등 많은 것들을 할 수 있다. RAD Server는 사용자 인증, 푸시 알림(push notification), 위치 파악, 데이터 저장소도 지원한다.



여러분의 데이터를 제공하는 REST 엔드포인트들을 개발하고 테스트한다

RAD Server 안에는 마법사, 컴포넌트, 도구들이 있다. 그것들을 활용하면, 새 미들웨어와 백-엔드 애플리케이션을 빠르게 개발할 수 있다. 또한 Delphi나 C++Builder로 만든 기존 클라이언트/서버 애플리케이션을 RAD Server 기반 애플리케이션으로 이전하고, 그것을 서버 또는 클라우드 상에서 운영하고, 엔드포인트들을 게시할 수 있다. 그 엔드포인트들은 REST 호출을 통해, 데스크탑, 모바일, 콘솔, 웹 및 기타 다양한 유형의 애플리케이션들이 사용할 수 있다. RAD Server에는 서비스 애플리케이션을 구축하는 데 필요한 모든 도구들, 컴포넌트들, 데이터베이스 연결, 인터페이스 등이 들어 있다. 여러분은 그것들을 활용해 여러분의 서비스 애플리케이션을 개발하면 된다.

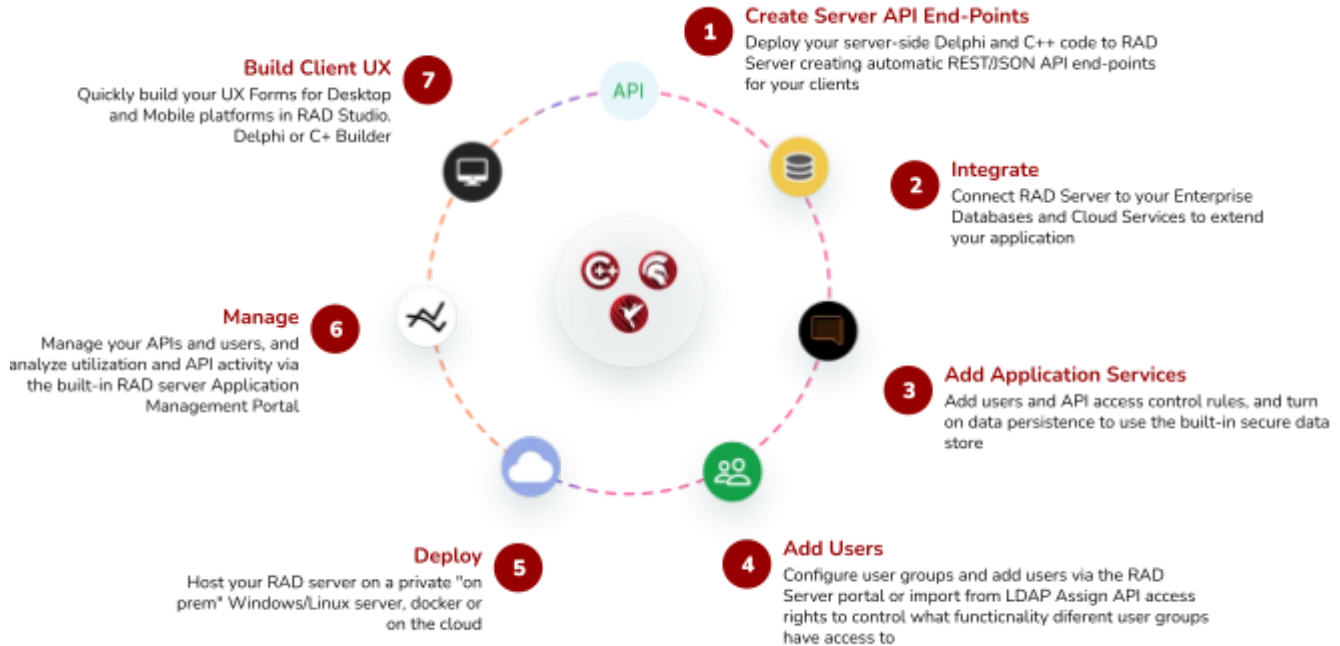
RAD Server 애플리케이션은 마이크로소프트 윈도우 IIS, Apache 웹 서버 등에 배포될 수 있다. 또한, 여러분의 Delphi 기반 서비스들은 리눅스 인텔 64비트 서버에도 배포할 수 있다. C++Builder 기반 서비스를 리눅스에서 운영하는 것에 대해서는 지켜보기 바란다. 엠바카데로 RAD Studio 블로그를 통해 계속 소식을 전하겠다.



REST 엔드포인트, 위치 추적, IoT(사물인터넷) 엣지웨어를 개발하고 테스트한다

RAD Server 기반 애플리케이션 구축하기 -7가지 핵심 방향

RAD Server 기반 애플리케이션을 구축하려면, 아래 다이어그램이 안내하는 개발자를 위한 7가지 방향과 개발 단계를 이해하면 좋다.



멀티-티어 개발이 쉽다

시작하려면, 서버 REST/JSON API 기반 엔드포인트를 만든다 (필요 시, JSON 대신 XML을 사용할 수 있다). 그런 다음, 엔드포인트를 확장한다. 데이터베이스, 클라우드 서비스, 기타 기술 등을 광범위하게 통합할 수 있다.

여러분은 더 많은 애플리케이션 엔드포인트들을 사용자에게 추가할 수 있다. 또한 API 접근 제어 규칙을 만들 수 있다. 그리고 RAD Server에 내장된 보안 데이터 저장소를 활용해 영구 데이터를 추적하는 코드를 작성할 수 있다. 그리고 사용자 그룹을 만들고 사용자를 추가할 수 있다. 콘솔 포털을 사용하면 된다. 또한 기존 사용자들을 가져오고 인증할 수 있다. LDAP-기반 API를 통하면 된다.

애플리케이션 개발과 디버깅이 끝나면, 그 RAD Server 애플리케이션을 조직 내에 있는(on-premise) 윈도우와 리눅스 서버에서 호스팅할 수 있다. 또한 클라우드 시스템으로 이전할 수도 있다. 즉, 아마존 AWS, 마이크로소프트 Azure, 구글 및 기타 클라우드 제공업체 등을 이용할 수도 있다.

애플리케이션을 실제 운영 환경에 배치한 후에는, API에 대한 접근 관리, 사용자 접근 제어, 엔드포인트 API의 활동 분석 등을 할 수 있다. 내장된 애플리케이션 관리 인터페이스를 사용하면 된다. 마지막으로, 데스크탑, 모바일, 웹, 콘솔 등 RAD Studio로 구축할 수 있는 모든 애플리케이션 유형을 만들면 된다. 웹 클라이언트 애플리케이션도 구축할 수 있다. Sencha Ext JS 컴포넌트 세트나 WebStencils(웹스텐실즈)를 사용하면 된다. 또는 다른 도구나 프로그래밍 언어를 사용해 클라이언트 애플리케이션을 만들 수도 있다. RAD Server 애플리케이션의 REST/JSON 기능을 지원하기만 하면 된다.

RAD Server 애플리케이션 구축을 위한 요구 사항

이어지는 섹션들은 RAD Server 애플리케이션을 빌드, 테스트, 배포하기 위한 제품과 기술 요구 사항을 설명한다. 별도의 언급이 없다면, “RAD Studio”, “IDE” 라는 용어는 RAD Studio, Delphi, C++Builder 제품을 의미한다.

RAD Studio IDE를 사용하려면

RAD Studio 엔터프라이즈 또는 아키텍트 에디션의 상용 라이선스가 있어야 RAD Server 애플리케이션을 구축할 수 있다. RAD Studio 엔터프라이즈 평가판은 30일 동안 개발과 테스트 용도로 사용할 수 있다. 평가판에는 운영 서버로 배포하는 기능이 없다.

RAD Server 테스트 및 배포 라이선스

무료로 30일간 제공되는 RAD Studio 평가판에는 RAD Server 사용자 5명짜리 개발 체험판이 들어 있다. RAD Server 배포 라이선스는 RAD Studio의 엔터프라이즈 및 아키텍트의 상용 에디션에 포함되어 있다. RAD Studio 엔터프라이즈에는 RAD Server용 단일-사이트 배포 라이선스가 포함되어 있다. 한편, RAD Studio 아키텍트 에디션에는 멀티-사이트 배포 라이선스가 포함되어 있다. 그런데, 업데이트 서브스크립션을 유지하는 경우에만 제공된다. RAD Studio 알렉산드리아(Alexandria) 버전부터는 엔터프라이즈와 아키텍트 모두 RAD Server Lite(라이트)를 멀티-사이트 환경에 배포할 수 있는 옵션이 제공된다.

실제 운영 환경에 애플리케이션을 배포하려면 RAD Server에 InterBase(인터베이스) 암호화된 데이터베이스가 여러분의 애플리케이션의 한 부분으로 배포해야 한다. 여러분에게 부여된 유효한 RAD Server 라이선스를 사용해야 해당 InterBase 버전을 설치할 수 있다.



InterBase를 사용하고 있는 애플리케이션을 배포하려는 경우, InterBase 인스턴스 두 개가 실행되어야 한다. 하나는 애플리케이션용이고 다른 하나는 RAD Server 용이다.

RAD Server 핵심 기능 요약

RAD Server는 REST-기반 서비스 애플리케이션을 구축하기 위해 필요한 다양한 기능들을 개발자에게 제공한다. RAD Server(구 EMS)를 처음 선보인 버전은 RAD Studio XE7이다. 그 후 계속해서 개선 사항들과 새 기능들이 추가되면서, RAD Server는 개발자의 요구 사항들을 반영하고 새로운 플랫폼/아키텍처/기법들을 지원해 오고 있다.

핵심 기능들

이것은 RAD Server의 핵심 기능 목록이다. 여러분이 서비스-기반 애플리케이션을 구축할 때 활용하고 싶을 것이다.

- **REST 엔드포인트 퍼블리싱(publishing, 게시)** - RAD Server는 턴키(turnkey, 완비된) 방식 기반이다. 애플리케이션 백엔드 API들과 서비스들의 바탕이 된다. RAD Server는 비즈니스 로직을 손쉽게 공개할 수 있는 사용하기 쉬운 API를 제공한다. Delphi 또는 C++ 코드는 API로 호스팅되고, 자동으로 REST/JSON 엔드포인트가 되어 게시될 수 있다. 그 엔드포인트들은 RAD Server가 측정하고 관리한다. 엔드포인트 게시 기능들에는 이런 것들이 있다:

- 접근 제어 - 그룹 및 사용자 수준 접근을 설정할 수 있다. 애플리케이션의 모든 API들에 대해, 인증(authentication)을 통해, 누가 어떤 API 기능을 사용할 수 있는지 제어할 수 있다. 사용자와 그룹을 직접 만들 수도 있다. 또한 여러분의 LDAP 인프라로부터 자동 가져오기를 할 수도 있다.
- API 분석 - 모든 REST API 엔드포인트 활동이 기록되고 측정된다. 그래서 강력한 통계를 통해 추적과 분석을 할 수 있다. 여러분은 사용자/API/서비스의 활동을 일/월/연간으로 분석할 수 있다. 즉, 애플리케이션이 어떻게 활용되고 있는지에 대해 통찰력을 얻을 수 있다. 활동을 필터링할 수도 있다. 그 전체 리소스의 활동을 보거나, 특정 그룹·사용자·디바이스 설치 등 원하는 기준 별로 활동 내역을 필터링할 수 있다. 또한 분석 데이터를 CSV 파일로 내보내 다른 도구에서 추가 분석을 할 수도 있다.
- 데스크톱·모바일·웹 애플리케이션 - RAD Server에 올라간 모든 C++/Delphi 코드는 REST/JSON 엔드포인트 형태로 제공된다. 따라서 다양한 플랫폼들에서 작동하는 등 어떤 종류의 클라이언트 앱에서도 쉽게 사용할 수 있다. 즉, 매우 유연하고, 미래에도 지속되는(future-proofing) 형태다.
- **통합 미들웨어** - RAD Server에는 바로 사용할 수 있는 다양한 통합 기능들이 있다. 즉, 외부에 있는 서버, 애플리케이션, 데이터베이스, 스마트 기기, 클라우드 서비스, 기타 플랫폼 등에 대한 연결이 제공된다. 통합 기능들에는 이런 것들이 있다:
 - 엔터프라이즈 데이터 - RAD Server에는 고성능 내장 연결을 널리 사용되는 모든 엔터프라이즈 RDBMS 서버들을 위해 제공한다. 데이터베이스 연결은 FireDAC 컴포넌트들과 라이브러리 세트를 사용한다. 이 기술은 매우 쉽게 다양한 소스들에 있는 데이터를 연결한다.
 - 클라우드 서비스 - RAD Server를 사용하면 REST 클라우드 서비스를 쉽게 통합할 수 있다. 즉, 구글, 아마존, 페이스북, Kinvey 등등 다양한 클라우드들, 소셜 플랫폼들, BaaS 플랫폼들과 통합할 수 있다.
- **애플리케이션 서비스** - RAD Server에는 애플리케이션에는 바로 사용할 수 있도록 내장된 서비스 모음이 포함되어 있다. 그것들을 활용하면 여러분의 애플리케이션을 강화할 수 있다. RAD Server에 들어있는 이런 핵심 기능으로는 사용자 디렉토리 서비스와 사용자 관리, 푸시 알림, 사용자 위치 추적, 내장 데이터 저장소 등이 있다. 이러한 “애플리케이션 및 디바이스” 서비스들 중 몇 가지는 다음과 같다:
 - 푸시 알림(Push Notification) - RAD Server를 사용하면, 프로그래밍 방식 또는 온디맨드 방식으로 애플리케이션 사용자들과 그들의 기기에 알림을 보낼 수 있다. RAD Server는 현재 애플의 푸시 알림 서비스(APN)와 구글 파이어베이스 클라우드 메시징(FCM)을 지원한다. 또한 여러분이 사용자 정의 코드를 작성할 수도 있다. 즉 그 외의 푸시 알림 시스템들과도 연결할 수 있다.
 - 내장된 보안 데이터저장소 - RAD Server는 InterBase(인터베이스) 서버의 암호화된 데이터 저장을 지원한다. 따라서 여러분은 내장된 API들을 사용해 JSON 데이터를 저장하고 검색할 수 있다. 이런 용도로 데이터베이스 서버를 따로 두지 않아도 된다.
 - 사용자/그룹 관리 - RAD Server API들을 사용하면, 여러분이 사용자와 사용자 그룹을 생성/관리하고, 접근 제어를 할 수 있다. RAD Server 콘솔(RSConsole.exe)을 쓰면 된다. Active Directory(LDAP)를 통합하거나 여러분이 직접 인증(authentication) 미들웨어를 개발하는 것도 가능하다.
 - 사용자 위치 정보(location)/근접성(proximity) - 여러분의 RAD Server 애플리케이션들은 RAD Studio에서 지원하는 GPS, 비콘, 비콘 펜스 기술을 활용할 수 있다. RAD Server 애플리케이션은

사용자의 이동을 추적할 수 있다. 실내외 모두 가능하며, 미리 지정해 놓은 비콘 구역에 대한 사용자 출입에 대해 또는 지정된 비콘 지점에 대한 사용자 접근에 대해 이벤트(event)를 통해 대응할 수 있다.

- 정적 파일 공급자 - URL을 폴더에 매핑하고 HTML, JS, CSS, 이미지 등 파일 콘텐츠를 반환한다. 이 기능이 특히 편하게 사용되는 경우는 소규모 배포(예: RAD Server Lite 사용) 또는 개발 환경이다.
- **API 문서집** - “애트리뷰트(attribute)”와 “내장된 Swagger OpenAPI 통합”을 사용하면, 여러분의 API에 대한 문서집을 쉽게 생성할 수 있다. 내장된 Swagger UI를 RAD Server 자체에서 사용하거나 원격 인스턴스들 안에 구성할 수 있다. 자동 생성되는 YAML 및 JSON 파일들을 사용하면 된다.
- **손쉽게 배포** - RAD Server는 개발, 배포, 운영이 쉽다. 따라서 ISV들과 OEM들에게 이상적이다. 재배포하기 좋은 솔루션을 구축할 수 있기 때문이다. 여러분은 윈도우, 리눅스, 도커(Docker)에 배포할 수 있다.

참고 자료

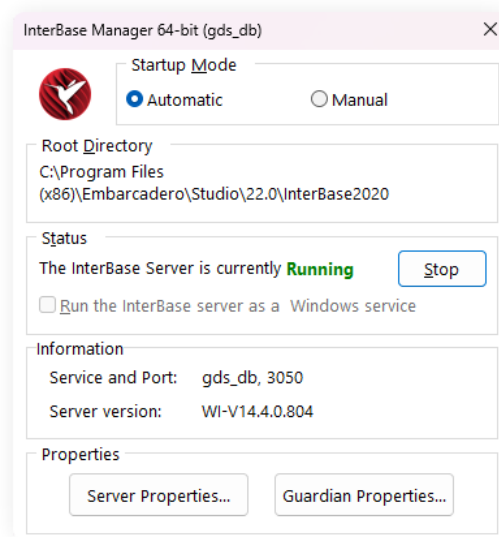
RAD Studio 설치와 RAD Server 기반 애플리케이션 배포에 대한 최신 정보는 다음 엠바카데로 온라인 링크를 참조하기 바란다.

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

02

RAD 마법사를 사용해 "Hello World" 만들기

이제 본격적으로 프로그래밍을 시작한다. 이 장에서는 RAD Server 기반 서비스 애플리케이션을 구축하는 방법을 배운다. Delphi 또는 C++Builder를 사용하면 된다. 시작하기에 앞서, InterBase 데이터베이스 서버가 실행되고 있는지를 먼저 확인해야 한다. 왜냐하면, RAD Server가 사용자 정보, 사용자 그룹, 분석, 등록된 디바이스, 버전 정보, 등록된 엡지 모듈, 푸시 알림 메시지 등을 저장하는 곳이 InterBase 데이터베이스이기 때문이다.

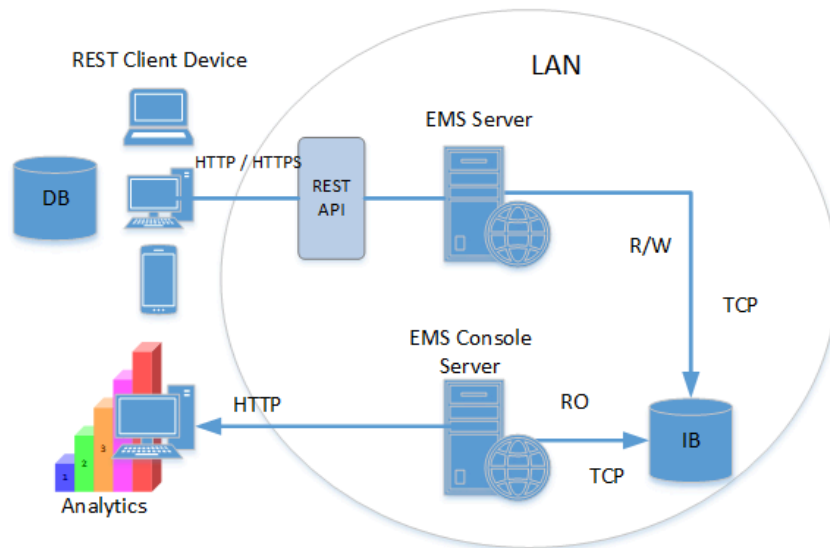


InterBase 2020 서버 관리자

REST-기반 서비스를 구축하기

RAD Server에는 EMS(엔터프라이즈 모빌리티 서비스)가 있다. 이것은 MEAP(모바일 엔터프라이즈 애플리케이션 플랫폼)이며, 클라우드 또는 내부의 자체(on-premise) 서버에서 호스팅하면 된다. 개발자들은 RAD Server를 사용해 사용자 정의 REST API를 게시할 수 있다. 그리고, 엔터프라이즈 데이터베이스의 데이터에 접근할 수 있다. FireDAC 데이터 액세스 라이브러리와 컴포넌트들을 사용하면 된다.

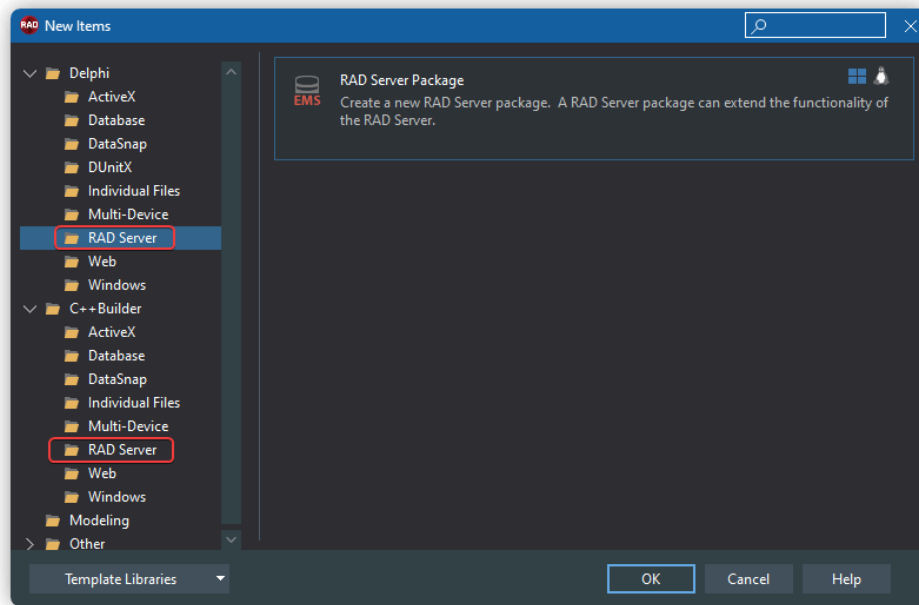
RAD Server는 개발자에게 원격 데이터베이스 접근, 사용자 추적, 기기(device) 애플리케이션 관리, 사용 분석 등을 포함해 종합적인 솔루션을 제공한다. 다른 솔루션들과 달리, RAD Server에는 미리 구축된 애플리케이션 서버가 있다. 이 서버는 사용자 정의 패키지(of custom package)들을 통합할 수 있도록 지원한다. 그 사용자 정의 패키지들은 데이터셋, 비즈니스 로직, 기타 REST-기반 리소스 등을 게시할 수 있다. 또한 RAD Server 리소스들을 모바일, 웹, 데스크탑 애플리케이션 코드 안에서 접근할 수 있도록 하는 컴포넌트들이 제공된다.



RAD Server REST API 아키텍처

RAD Server 프로젝트 마법사를 사용하기

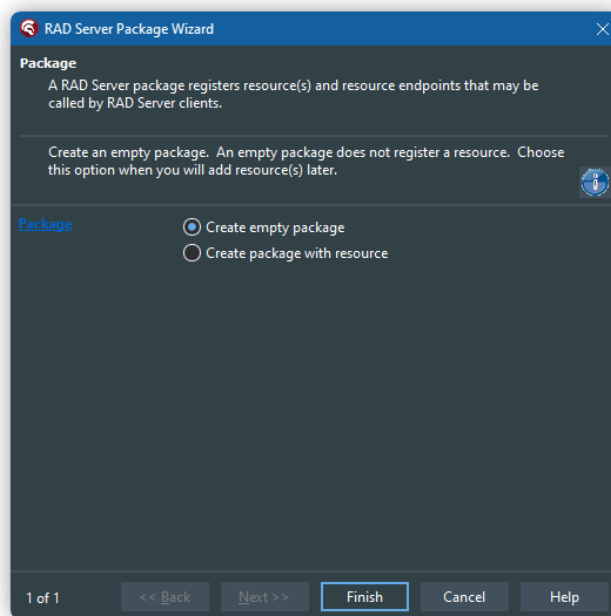
처음 시작할 때, 가장 빠른 방법은 New Projects 메뉴(File | New | Other...)를 사용하고, Delphi 또는 C++Builder에서, RAD Server | RAD Server Package 마법사를 선택하는 것이다.



RAD Server 프로젝트 마법사를 Delphi 또는 C++ 아래에서 선택한다

RAD Server Package 프로젝트를 선택하라. 그러면, 마법사가 나타난다. 이 마법사는 새로 프로젝트 만드는 것을 도와준다. 첫 페이지에서 여러분은 RAD Server 애플리케이션 안에 나타낼 리소스들과 엔드포인트들을 마법사가 어떻게 생성할 것인지를 선택한다. 두 가지 중 하나를 선택하면 된다.

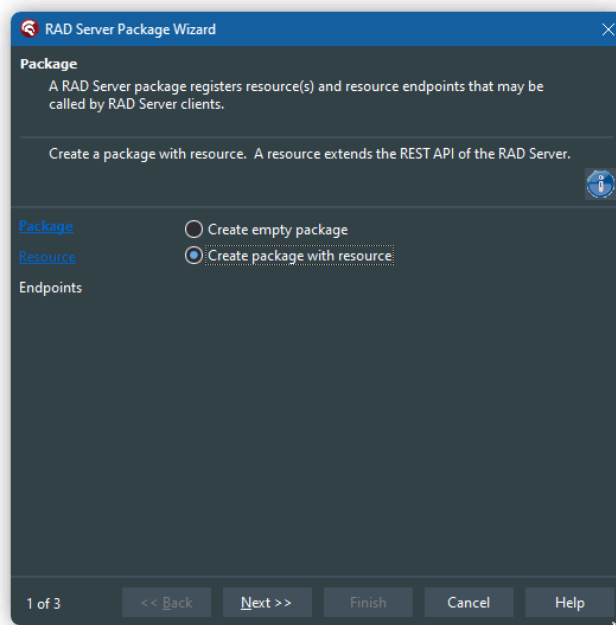
선택 1: 리소스를 등록하지 않은 빈 패키지를 만들기(**Create an empty package**). 여러분이 만약 리소스 추가를 나중에 하고 싶다면, 이 옵션을 선택하면 된다. 이 옵션은, 패키지 프로젝트를 생성한다. 그리고, 패키지 구축에 필요한 기본적인 메인 라이브러리가 포함된다.



빈 RAD Server Package를 만든다

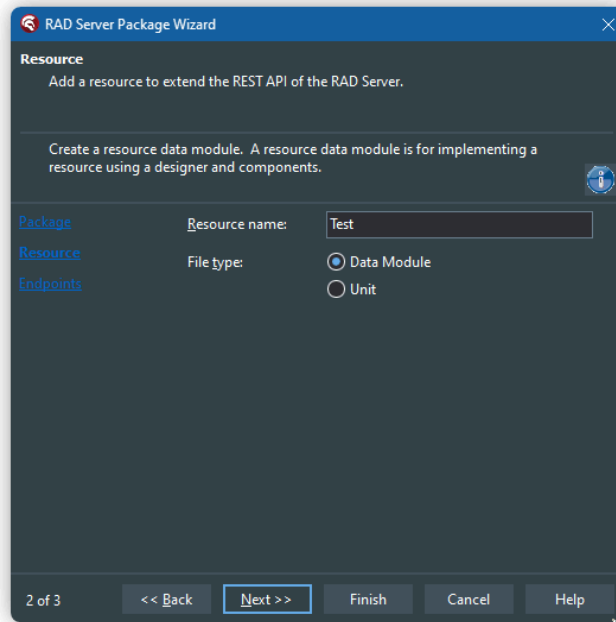
“Finish” 버튼을 클릭하면, 새 프로젝트가 생성된다. 여기에서부터 추가 개발 작업을 시작하게 된다. 그래서 여러분의 최종 RAD Server 애플리케이션을 완성해 가면 된다.

선택 2: 리소스가 포함된 패키지 만들기(**Create a package with a resource**). 리소스는 RAD Server가 더 많은 REST API를 제공하도록 확장한다. 이것을 선택하고, Next 버튼을 클릭하라. 그러면, 마법사에서 두 단계를 더 진행해 패키지 프로젝트, 리소스, 엔드포인트들을 만들게 된다. 여러분은 RAD Server 프로젝트 구축이 처음이므로 이 옵션을 선택한다.



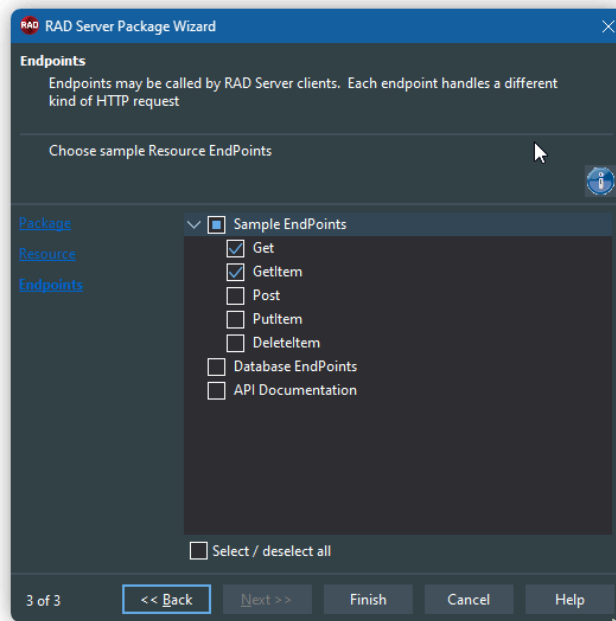
리소스 기반 RAD Server Package를 만든다

설치 마법사의 두 번째 페이지에서 리소스 이름을 “Test”라고 지정한다. File type 라디오 버튼에는 두 가지 옵션이 표시 된다: 1) **Unit**: 유닛이 하나 생성된다. 여러분은 그 유닛의 코드 안에서 직접 해당 리소스를 구현할 수 있다. 2) **Data Module**: 데이터모듈이 하나 생성된다. 여러분은 IDE의 디자이너, 컴포넌트, 코드 편집기를 사용해 해당 리소스를 구현할 수 있다. 여러분은 RAD Server 프로젝트 구축이 처음이므로, Data Module을 사용하라.



RAD Server Package 마법사 2페이지 - 리소스 이름과 파일 유형을 지정한다

Next 버튼을 클릭한다. 그러면, 새 프로젝트 안에 만들어질 엔드포인트 집합이 생긴다.



RAD Server Package 마법사 3페이지 - 새 프로젝트 안에 만들어질 엔드포인트를 선택한다

마법사 세 번째 페이지에서는 마법사가 미리 선택해 놓은 엔드포인트들을 그대로 둔다: 즉 Get(REST GET)과 GetItem(가져올 항목을 URL 맨 뒤에 적을 수 있는 부분이 포함되는 REST GET)이 선택된 그대로 둔다. 그리고 “API Documentation”의 선택은 취소한다. 위 그림처럼 선택했다면, Finish 버튼을 클릭한다. 그러면 여러분이 시작할 새 프로젝트가 만들어진다.

**참고**

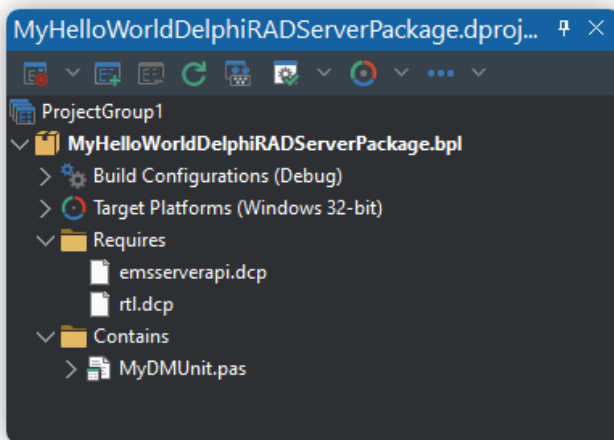
마법사에는 두 가지 옵션이 더 있다: *Database Endpoint (FireDAC를 사용해 데이터베이스를 엔드포인트에 연결하기)* 그리고 *API Documentation(Swagger OpenAPI)*다. 이에 관해서는 다음 장에서 더 자세히 설명하겠다.

마법사가 끝나면, 여러분은 개발 화면을 보게 된다. 처음 할 일은 “프로젝트 저장”이다. C++/Delphi 데이터 모듈에 “MyDMUnit”이라고 이름을 붙인다. C++ 프로젝트 검 패키지는 “MyHelloWorldCppRADServerPackage”라고 이름을 붙인다. Delphi 프로젝트 검 패키지는 “MyHelloWorldDelphiRADServerPackage”라고 이름을 붙인다.

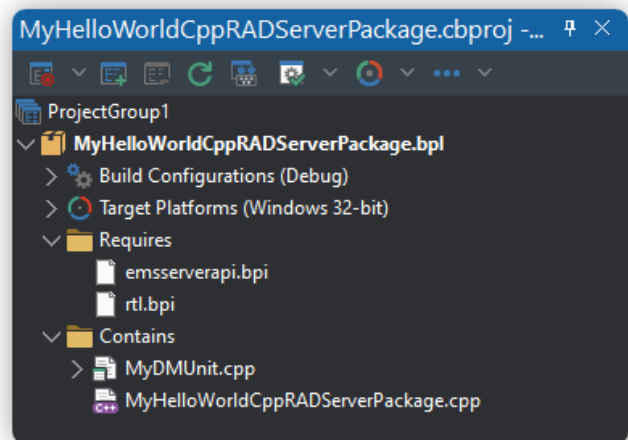
마법사 RAD Server 프로젝트 및 소스 코드

생성된 프로젝트에는 코드가 매우 적다(low-code). 그저 메서드 두세 개만 있고 각각 엔드포인트에 연결되어 있다. RAD Studio는 그것들의 기본 코드들을 자동으로 채워준다. 우리는 그 코드를 조금만 바꿔 Hello World 같은 것을 만들어 보자.

이것들은 이 프로젝트들의 모습이다. 각각 Delphi와 C++ 프로젝트다. 생성된 데이터모듈은 텅 비어 있을 것이다. 컴포넌트가 전혀 들어있지 않다. 이 화면 캡처 바로 밑에서는 소스 코드를 볼 수 있다. 자동 생성된 샘플 코드를 우리가 조금만 바꾸면 이렇게 된다. 여러분의 코드가 이렇게 되도록 편집하라.



생성된 Delphi 프로젝트



생성된 C++ 프로젝트

**참고**

이 문서에 사용된 모든 소스 코드와 샘플은 [깃허브](#)에 호스팅 되고 있다. 각 장별로 나뉘어 있다. 전체 리포지토리를 다운로드하는 것을 권장한다. 그러면 더 잘 이해할 수 있다.

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:

```

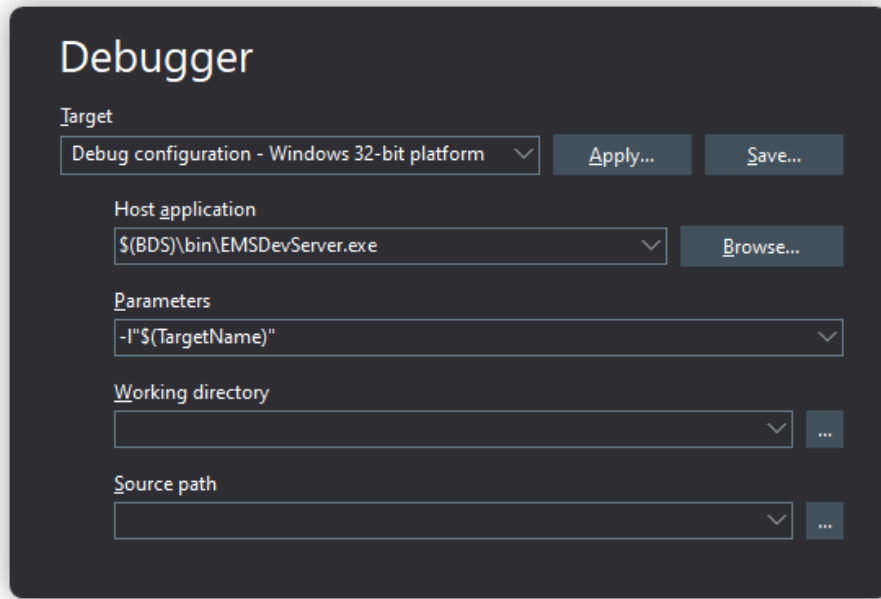
void TTestResource1::Get(TEndpointContext* Acontext,
  TEndpointRequest* ARequest, TEndpointResponse* AResponse)
{
  AResponse->Body->SetValue(new TJSONString("Hello World"), True);}

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);
}

```

여러분의 첫 애플리케이션을 위해 RAD Server를 구성하기

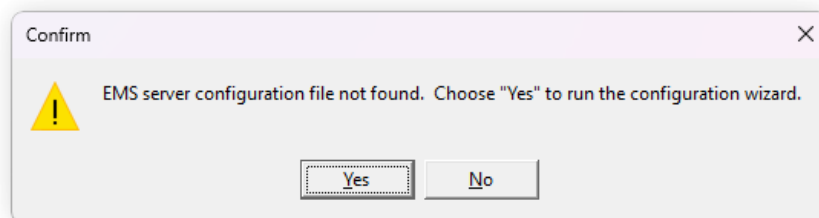
마법사를 사용해 여러분의 첫 RAD Server 애플리케이션을 구축했으니, 이제 IDE에서 애플리케이션을 컴파일하고 테스트하면 된다. IDE는 EMSDevServer를 호스트 실행 프로그램으로 사용한다(EMSDevServer.exe). 이 호스트 프로그램은 여러분이 만든 패키지 파일을 파라미터로 받아 적재한다. EMSDevServer는 개발용이다. Win32용과 Win64용, 즉 \$(BDS)\bin\EMSDevServer.exe와 \$(BDS)\bin64\EMSDevServer.exe가 들어 있다. 이 개발용 서버를 위한 구성은 IDE가 모두 자동으로 설정한다.



Run | Parameters... 대화 상자: EMSDevServer.exe가 호스트 애플리케이션으로 지정되어 있다

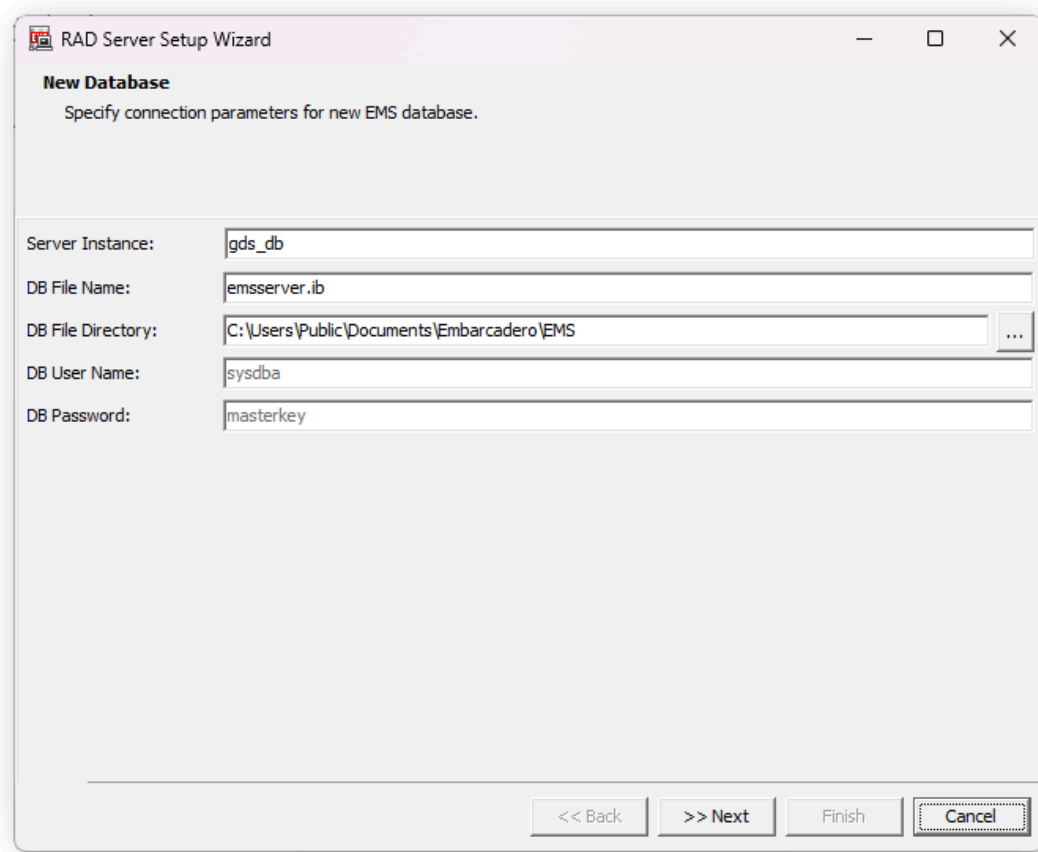
RAD Studio 안에는 EMSDevConsole.exe도 들어 있다. IDE는 이 EMSDevConsole 서버를 작동시켜 그 콘솔 서버의 창을 열어줄 것이다. 이 EMS 콘솔은 웹 애플리케이션 하나를 제공한다. 그래서 통계 표시, 사용자/그룹 관리 등 여러분의 RAD Server 애플리케이션을 위한 다양한 기능들을 제공한다. 이 콘솔에 대해서는 다음 장에서 더 자세히 설명한다.

Run | Run메뉴 항목을 선택하거나 F9키를 눌러보라. 그러면, IDE는 우리가 만든 애플리케이션을 컴파일하고, 링크하고, 실행한다. 그러면 개발용 RAD Server가 작동을 시작한다. 기본 설정에 따라 TCP 포트 8080을 사용한다. 만약 RAD Server 애플리케이션이 처음 실행된다면, RAD Server 구성 파일인 emsserver.ini를 찾을 수 없다는 대화 상자가 나타난다. 이 문제는 RAD Server 레지스트리 키가 없거나, 구성 파일이 존재하지 않을 때 발생한다.



구성 파일이 없이 RAD Server Development Server를 시작하려고 시도하는 경우

Yes 버튼을 클릭해 RAD Server 구성 마법사를 실행하라.



Setup wizard 1 페이지 - 새 EMS 데이터베이스 연결 파라미터들을 지정한다

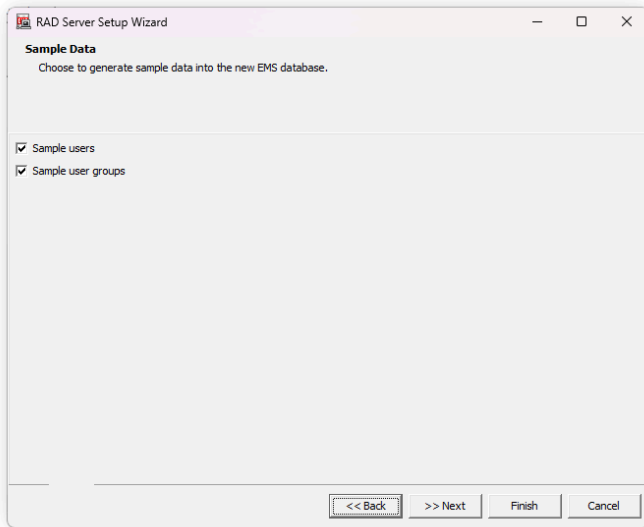
마법사 첫 단계에서, InterBase 서버 인스턴스를 입력하라 (기본 설정 상, RAD Studio가 제공하는 InterBase 서버의 개발 버전이 사용하는 서버 인스턴스는 gds_db다). 이 마법사 페이지에는 RAD Server 데이터베이스의 이름(emsserver.ib)과 디렉토리 입력 창도 있다. 그 디렉토리는 해당 데이터베이스와 구성 파일이 있는 곳이다.

**경고**

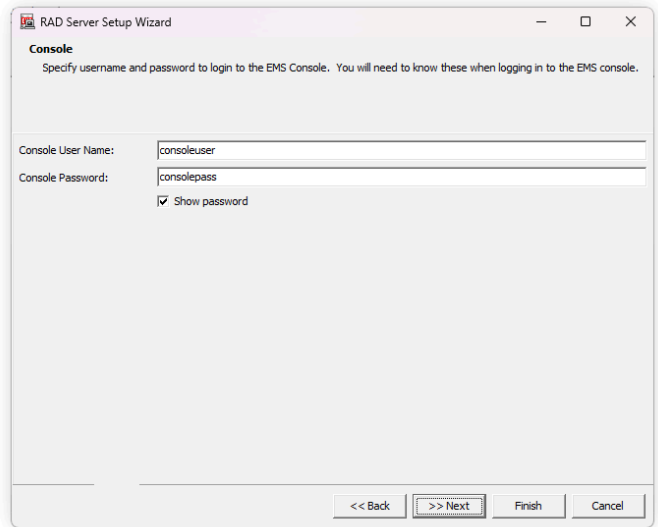
여러분이 그 컴퓨터 안에 InterBase를 설치한 적이 있고, 당시 서버의 인스턴스 이름을 다르게 지정했다면, 그 이름 문자열을 입력해야 한다.

Next 버튼을 클릭한다. 그러면, 사용자와 사용자 그룹이 담긴 샘플 데이터를 RAD Server 데이터베이스 안에 만들지 여부를 마법사에서 지정할 수 있다. 우리는 개발과 테스트가 목적이므로, 이 두 확인란을 모두 선택한다.

Next 버튼을 클릭한다. 그러면 이 개발용 RAD Server 콘솔에 로그인할 수 있는 사용자 이름과 비밀번호를 설정할 수 있다.

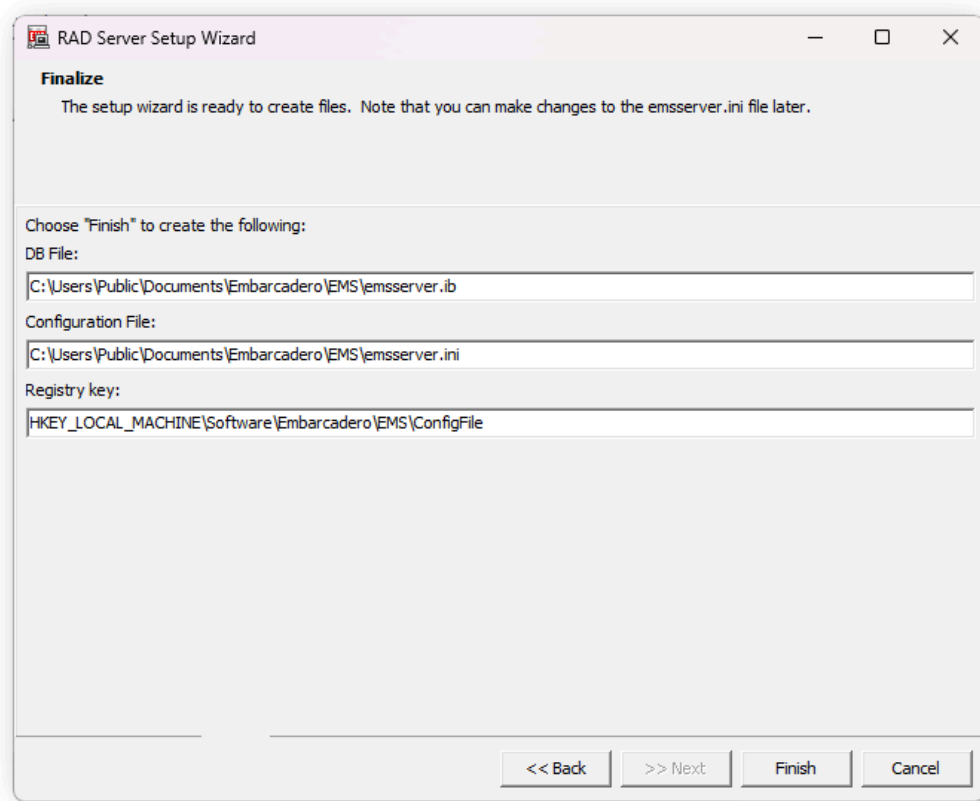


Setup wizard 2 페이지 - 샘플 사용자와 그룹



Setup wizard 3 페이지 - 콘솔 사용자 이름과 비밀번호 선택

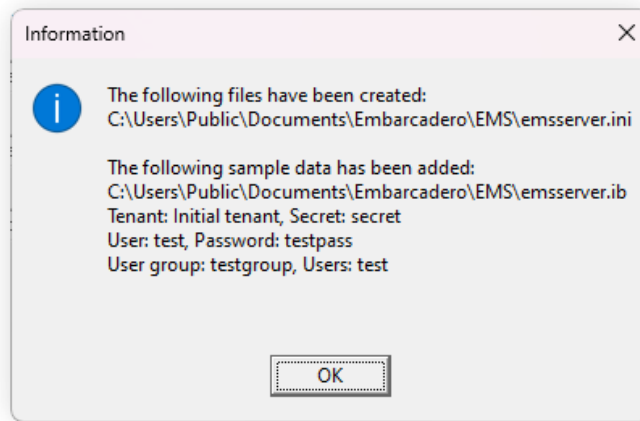
이제, 마지막으로 Next 버튼을 클릭한다. 그러면, 마법사의 마지막 단계로 이동한다. 마법사는 이제 RAD Server를 위한 데이터베이스 파일과 구성 파일을 생성하고, 윈도우 레지스터 키를 현재 로그인한 사용자를 위해 설정할 준비가 되었다.



RAD Server 구성 마법사의 마지막 페이지

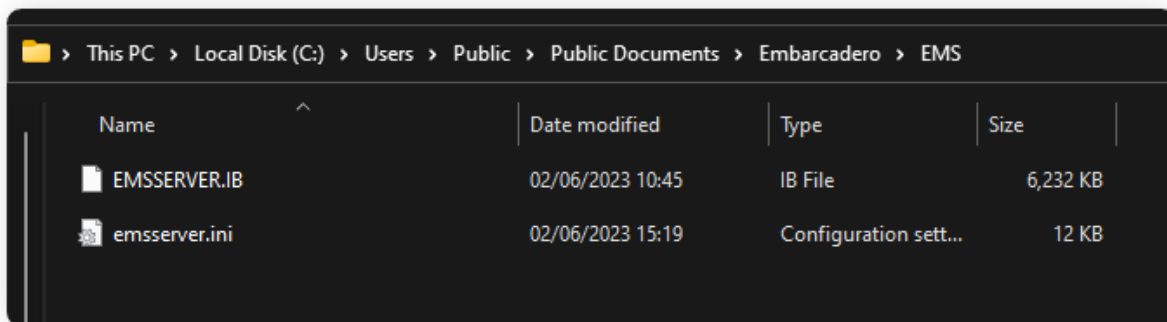
위 페이지는 데이터베이스 파일의 경로와 이름, 구성 파일의 경로와 이름, 윈도우 레지스터 키를 표시한다. 언제든지 여러분은 RAD Server 구성 파일(emsserver.ini)을 변경할 수 있다. Finish 버튼을 클릭하라. 그러면 확인 대화 상자가 메시지를 표시한다. 그래서 이 구성은 InterBase의 인스턴스를 사용하게 될 것이고 그 InterBase는 RAD Server 라이선스를 가지고 있지 않다고 다시 알려준다. 이 개발용 라이선스는 여러분의 RAD Server 애플리케이션의 사용자를 최대 5명으로 제한한다. 여러분이 RAD Server 애플리케이션을 실제 운영 용으로 배포할 준비를 모두 완료한 경우에는, RAD Server와 InterBase용 배포 라이선스를 사용할 수 있을 것이다.

Yes 버튼을 클릭하라. 마법사에는 emsserver.ini 구성 파일의 위치가 표시된다. 또한 샘플 데이터도 나열하고 그것들이 해당 데이터베이스에 추가되었다고 알려준다.



구성 마법사가 만든 RAD Server 파일 목록

OK 버튼을 클릭하라. 이제 파일 두 개가 C:\Users\Public\Documents\Embarcadero\EMS 디렉토리에 나타난다.



RAD Server 구성 마법사가 디스크에 만든 파일들



참고

emsserver.ini 파일은 RAD Server의 모든 기본 파라미터들이 정의되는 곳이다. 다음 장에서 설명한다. 하지만, 그 구성 파일을 열어서 그 안에 있는 내용과 방대한 설명을 보는 것도 좋다.

대체로, RAD Studio IDE 작동을 시작할 때 우리는 관리자 권한이 없이 실행한다. 그 경우, HKEY_LOCAL_MACHINE 레지스트리(registry) 항목은 가상화된다. 예: 윈도우 64-비트라면, HKEY_CURRENT_USER\Software\Classes\VirtualStore\MACHINE\SOFTWARE\WOW 6432Node\Embarcadero\EMS에 저장된다.

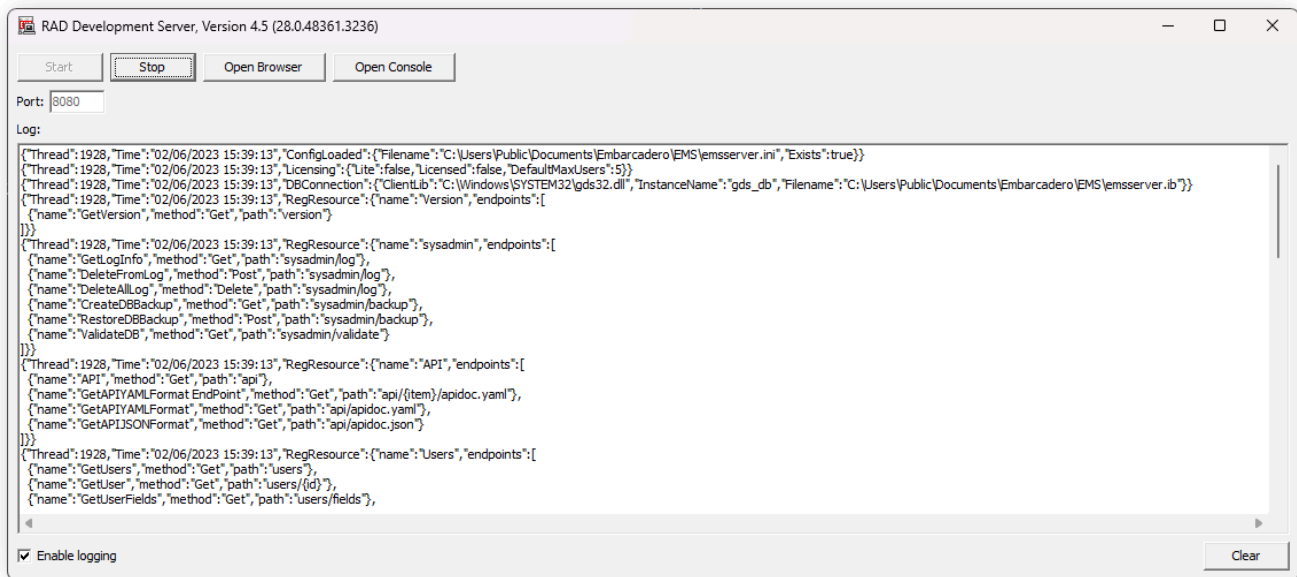
여러분의 첫 애플리케이션을 테스트하기

RAD Server 구성 서버가 생성되면, RAD Server Development(개발) Server는 실행을 시작한다.



EMSDevServer가 실행되는 기본 포트는 8080이다. 만일 컴퓨터의 해당 포트를 이미 다른 서비스가 사용하고 있다면, 기본 포트를 변경하면 된다. 즉, “Port” 필드에서 여러분에게 맞는 포트를 사용하도록 지정하면 된다. 이 변경 사항을 영구적으로 적용하려면 `emsserver.ini` 파일 안에 있는 `[Server.Connection.Dev]` 파라미터를 변경하면 된다.

IDE에서 Run을 누르면, RAD Server Development(개발) Server가 실행되기 시작한다. 그리고 그 로그에는 여러분의 패키지 애플리케이션을 위해 수행되는 작업들이 표시된다. Delphi든 C++이든 개발 서버는 완전히 똑같다.



RAD Server Development(개발) Server가 첫 애플리케이션 패키지를 시작하고 있다

RAD Server Development(개발) Server 로그에는, 구성, 데이터베이스 연결, 라이선스 정보, 적재된 애플리케이션 패키지, 등록된 리소스들, 생성된 엔드포인트들이 표시된다.

Open Browser 버튼을 클릭하면 기본 브라우저가 시작되고, JSON 결과가 표시된다. 내장된 엔드포인트인 GetVersion을 호출한 결과다. 여러분의 첫 RAD Server REST 엔드포인트를 드디어 사용해 보았다!



팁

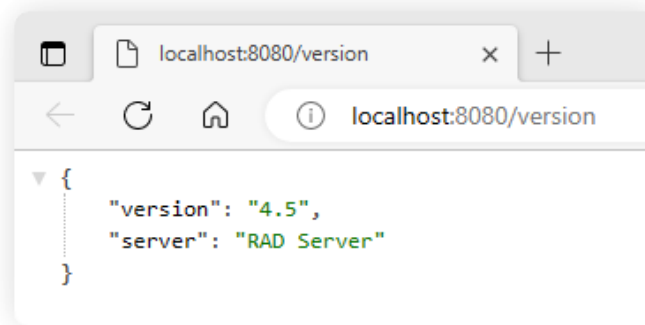
사람이 읽기 좋은 JSON 응답을 브라우저에서 얻고 싶다면, 주요 브라우저들이 모두 제공하는 "JSON Parser"라는 확장 프로그램을 설치하면 된다. **마이크로소프트 엣지**는, "edge://flags"에서 "JSON Viewer"를 활성화하면 따로 확장 프로그램을 설치할 필요가 없다. (옮긴이: 최신 버전은 이 옵션이 없음. 대신, 주소창 아래에 예쁜 인쇄(pretty print)라는 체크 박스가 있음)

● JSON Viewer

Allows users to view JSON files in a formatted view directly in the browser – Mac, Windows, Linux

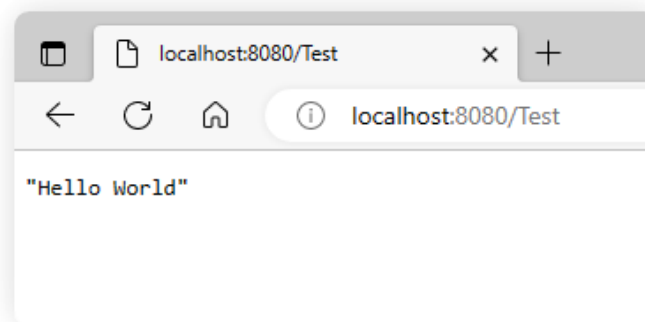
[#edge-json-viewer](#)

Enabled



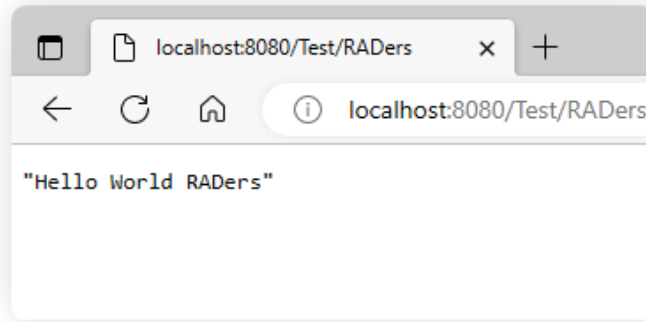
브라우저가 version 엔드포인트 호출한 결과를 표시하고 있다

브라우저에서, URL을 변경해 localhost:8080/Test라고 적고 엔터키를 눌러보라. 브라우저가 받게 되는 JSON 응답은 그것(Test)의 Get 엔드포인트가 보내준 것이다.



브라우저가 Test 리소스의 Get 메서드를 호출한 결과를 표시하고 있다

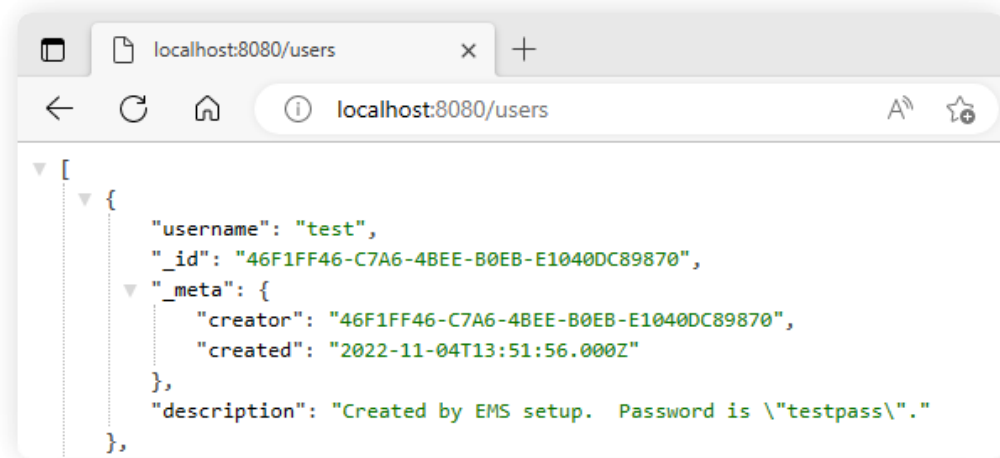
만약, URL에 항목 하나를 더 붙여 전달하면, 그것의 GetItem 엔드포인트가 호출된다. 따라서 그 GetItem 엔드포인트 뒤에 있는 코드가 JSON 문자열을 반환한다. 이 예시에서 그 결과는 “리소스 이름”에 “여러분이 URL에 적어 넣은 항목”이 합쳐진 문자열이다.



브라우저가 Test 리소스의 GetItem 메서드를 호출한 결과를 표시하고 있다

이 간단한 예시에서는 JSON 문자열 하나를 반환하는 방식이 괜찮다. 하지만, 더 크고 더 복잡한 데이터 구조를 담아야 한다면, 큰 JSON 문자열 하나를 반환하는 방식이 좋지 않을 수 있다. RAD Studio가 제공하는 JSON 데이터 생성 방식은 그 외에도 많다. JSON 오브젝트들, JSON 스트림들, JSON writer 등을 사용할 수 있다.

URL을 편집해 “users” 리소스를 사용하면, 기본인 GetUsers 엔드포인트를 호출해 JSON을 표시한다. 거기에는 RAD Server 구성 마법사가 RAD Server 데이터스토어에 만들어 놓은 사용자들이 담겨있다 (처음이므로 자동으로 만들어진 사용자 오직 1 명만 나온다).



브라우저가 GetUsers 엔드포인트를 호출하고 받은 JSON 응답을 표시하고 있다

여러분은 생성된 엔드포인트들 중 4 가지를 사용해 봤다. 모두 RAD Server 프로젝트 마법사가 생성해 준 것들이다.

참고 자료

- [이 장의 코드 샘플](#)
- [RAD Server 엔진 \(EMS 서버\)](#)
- [RAD Studio \(EMS\) 서버 설정하기](#)
- [EMS 서버 또는 EMS 콘솔 서버 구성하기 \(윈도우즈 상에서\)](#)
- [RAD Server의 관리용 AP](#)

03

여러분의 첫 CRUD 애플리케이션 만들기

RAD Studio는 바로 사용할 수 있는 여러 가지 컴포넌트들을 제공한다. 그런데, 그 중에서도, 여러분이 CRUD(생성, 조회, 변경, 삭제) API를 생성하는 경우에 가장 유용한 것은 EMSDataSetResource다. 여러분은 이 컴포넌트에 FireDAC 쿼리를 연결시킬 수 있다. 그래서 연결된 데이터를 노출할 수 있을 뿐만아니라 조작할 수도 있다. 이 컴포넌트는 CRUD에 필요한 모든 엔드포인트들을 자동으로 생성한다. 또한 페이지 나누기, 정렬하기 등등 많은 추가 기능들을 제공한다.

EMSDataSetResource는 여러분이 가진 어떤 유닛 안에 얼마든지 생성할 수 있다. 또는 더 간편하게 RAD Server 마법사를 사용해도 된다. 그러면 필요한 모든 컴포넌트들이 생성되고 FDConnection에 자동으로 연결된다.

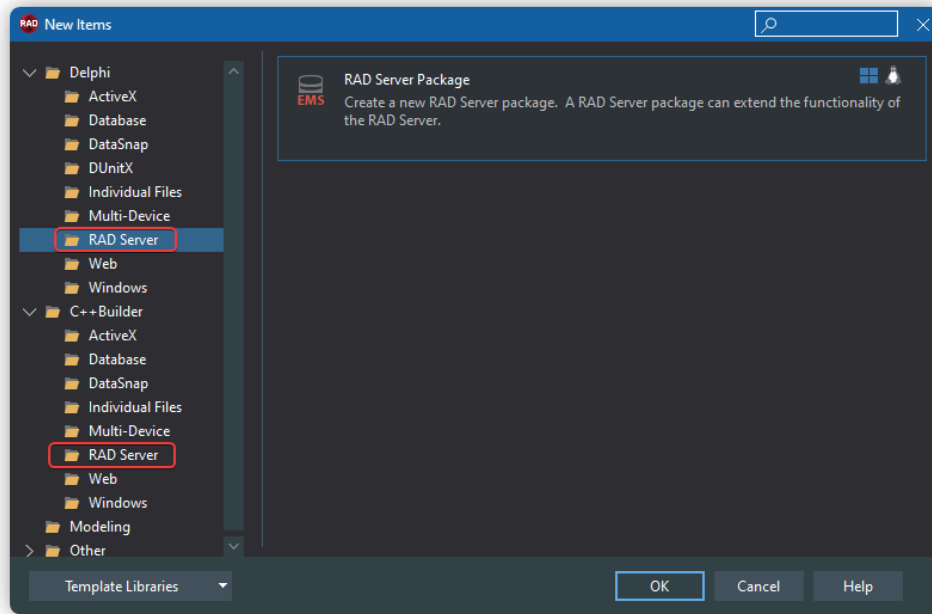


참고

이 데모에서는 InterBase 데이터베이스 파일인 Employee를 사용한다. 하지만 FireDAC이 지원하는 다른 어떤 데이터베이스도 얼마든지 사용할 수 있다. RAD Server 마법사를 사용하기 위한 유일한 요구사항은 데이터베이스 연결이 “Data Explorer” 안에 미리 구성되어 있어서, RAD Studio에서 인식할 수 있어야 한다는 것뿐이다.

CRUD 기능을 갖춘 REST-기반 서비스를 구축하기

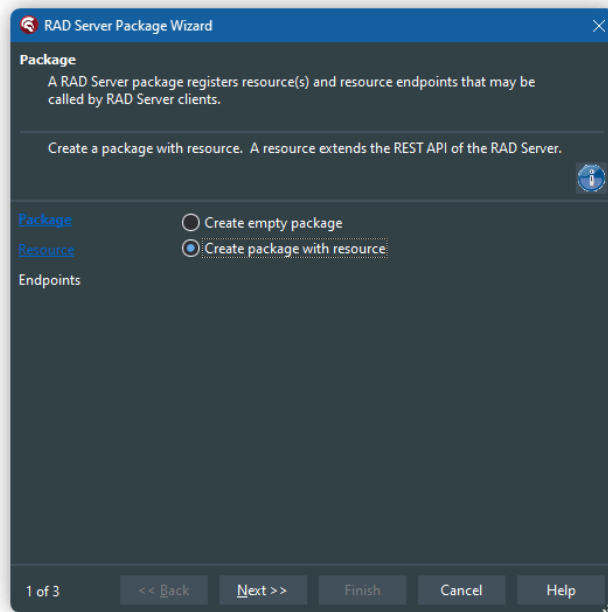
앞 장에서 우리가 했던 방식 그대로 시작하자. 처음 만들기 시작할 때, 가장 빠른 방법은 New Projects 메뉴(File | New | Other...)를 사용하고, Delphi 또는 C++Builder에서, RAD Server | RAD Server Package 마법사를 선택하는 것이다.



RAD Server 프로젝트 마법사를 Delphi 또는 C++ 아래에서 선택한다

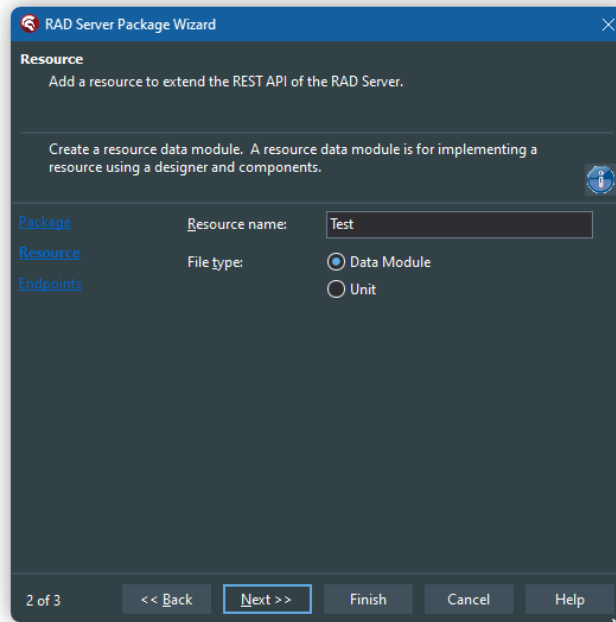
RAD Server Package 프로젝트를 선택하라. 그러면, 마법사가 나타난다. 이 마법사는 새로 프로젝트 만드는 것을 도와준다. 첫 페이지에서 여러분은 RAD Server 애플리케이션 안에 나타낼 리소스들과 엔드포인트들을 마법사가 어떻게 생성할 것인지를 선택한다. RAD Server Package 마법사가 제시하는 선택은 두 가지다.

여기에서 리소스가 포함된 패키지 만들기(**Create a package with a resource**)를 선택해 RAD Server가 제공하는 REST API를 확장하기로 한다. Next 버튼을 클릭하라. 그러면, 마법사에서 두 단계를 더 진행해 패키지 프로젝트, 리소스, 엔드포인트들을 만들게 된다. RAD Server 프로젝트 구축이 처음이므로 이 옵션을 선택한다.



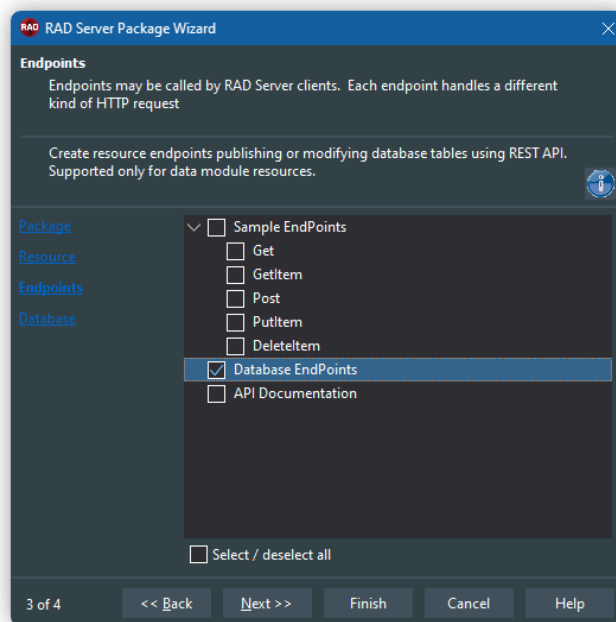
리소스 기반 RAD Server Package를 만든다

설치 마법사의 두 번째 페이지에서 리소스 이름을 “Test”라고 지정한다. File type 라디오 버튼에는 두 가지 옵션이 표시 된다: 1) **Unit**: 유닛이 하나 생성된다. 여러분은 그 유닛의 코드 안에서 직접 해당 리소스를 구현할 수 있다. 2) **Data Module**: 데이터모듈이 하나 생성된다. 여러분은 IDE의 디자이너, 컴포넌트, 코드 편집기를 사용해 해당 리소스를 구현할 수 있다. 여러분은 RAD Server 프로젝트 구축이 처음이므로, Data Module을 사용하라.



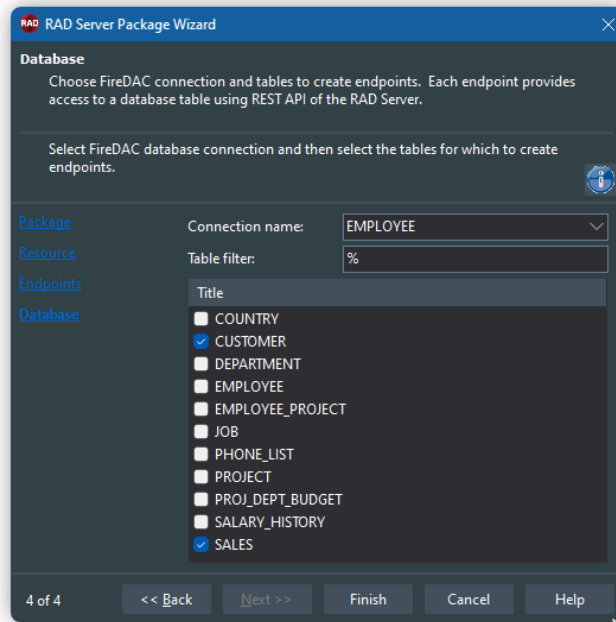
RAD Server Package 마법사 2페이지 - 리소스 이름과 파일 유형을 지정한다

Next 버튼을 클릭한다. 그러면, 새 프로젝트 안에 만들어질 엔드포인트 집합이 생긴다.



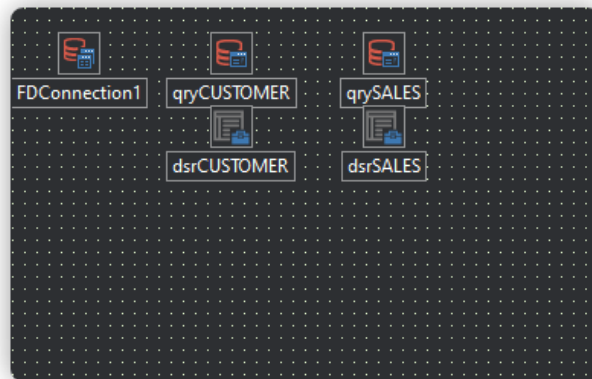
RAD Server Package 마법사 3페이지 - 새 프로젝트 안에 만들어질 엔드포인트를 선택한다

마법사의 세 번째 페이지에서 우리는 이전 장과 다른 옵션을 선택할 것이다. “Sample EndPoints”를 선택 취소하고 “Database Endpoints”를 선택한다. 이제 “Next”를 클릭하라.



RAD Server Package 마법사 4페이지 - 데이터베이스와 테이블들을 선택한다

위와 같이 선택하고 프로젝트를 만들면, 우리는 FDConnection 1개, FDQuery 2개, EMSDataSetResource 2개를 가지게 된다.



마법사가 생성한 데이터모듈의 모습이다

생성된 프로젝트를 설명하기

이 데모는 멋지다. 우리가 벌써 이것 만들 수 있고, 게다가 필요한 엔드포인트들이 자동으로 생성되기 때문에 우리가 바로 접근해서 사용할 수 있다. 하지만 먼저, 조금만 고쳐보자: 이 데이터모듈(DataModule)의 코드에 접근해, 엔드포인트들의 애트리뷰트들을 변경하자. 꼭 필요한 조치는 아니다. 하지만, 좋은 관행이므로 따르기로 하자. 엔드포인트들은 완전히 소문자로 표기한다. 그리고 복수형으로 유지한다.

Delphi:

```

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

```

C++:

```

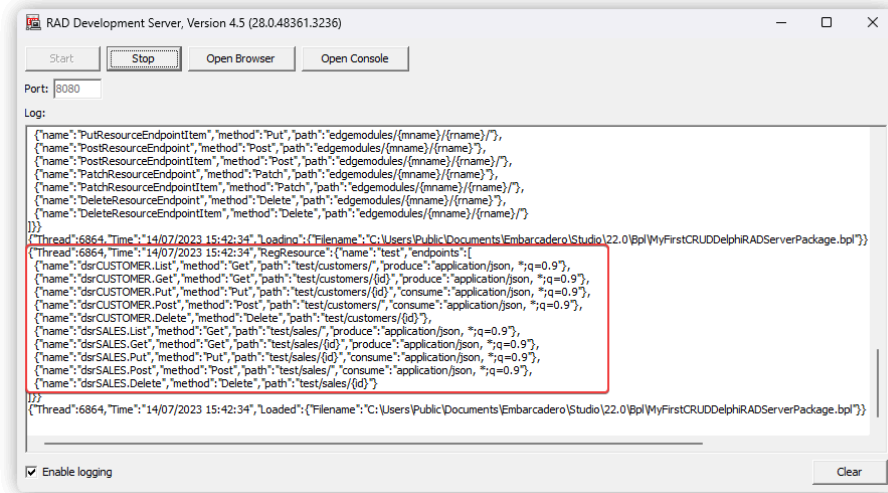
static void Register()
{
    std::unique_ptr<TEMSResourceAttributes> attributes(new
TEMSResourceAttributes());
    attributes->ResourceName = "test";
    attributes->ResourceSuffix["dsrCUSTOMER"] = "customers";
    attributes->ResourceSuffix["dsrSALES"] = "sales";
    RegisterResource(__typeinfo(TTestResource1),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

```

위 코드를 보면, ResourceSuffix 애트리뷰트들이 EMSDatasetResource들에게 연결되어있다. 그 의미는 이렇다. 쿼리(Query)가 DatasetResource에 연결되면, 그 엔드포인트에서 호출된다.

이 프로젝트는 이보다 더 간단할 수 없다. 지금까지 우리는 코드를 단 한 줄도 직접 써넣지 않았다. 그런데도 테이블 2개에 연결되어 CRUD를 완전하게 제공하는 기능을 갖춘 시스템이 만들어졌다. 이 프로젝트를 빌드하고 더 자세히 분석해보자.

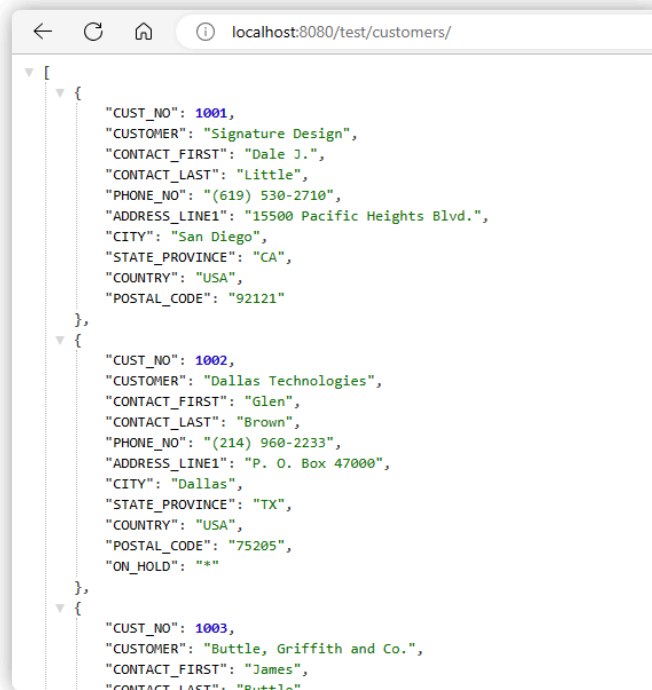
프로젝트를 빌드하고 테스트하기



RAD Server 로그가 자동 생성된 모든 엔드포인트들을 보여주고 있다

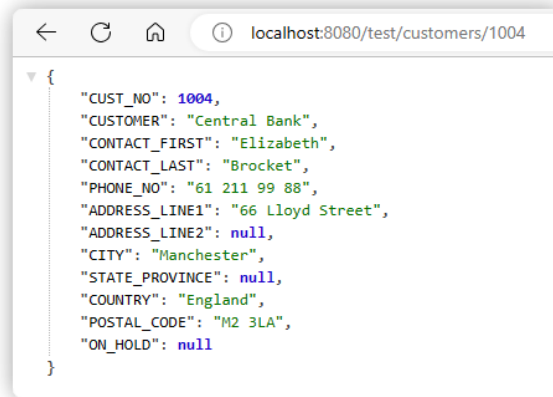
RAD Server 로그를 보자. “customers”와 “sales” 엔드포인트가 생성되었다. 뿐만 아니라 {id} 파라미터를 URI 안에서 사용할 수 있도록 제공하고 있다. 즉, URI에 이 파라미터를 명시해 개별 레코드에 대해 get/put/delete를 할 수 있고, 새 레코드를 post 할 수도 있다.

브라우저를 열고, <http://localhost:8080/test/customers/> 을 접근하면 JSON 배열이 반환된다. 그 안에는 customers 테이블의 모든 레코드가 담겨있다.



customers 배열을 RAD Server가 반환하고 있다

특정 고객 하나에 접근하려면, 고객 ID(Cust_No)를 쓰면 된다. `http://localhost:8080/test/customers/1004` 처럼 요청한다. 이미 짐작했겠지만, sales 엔드포인트에 접근하려면 `http://localhost:8080/test/sales/`를 호출하면 된다. 이런 방식으로 계속 해나가면 된다.



특정 고객 하나에 접근할 때는 ID를 사용한다



경고

TEMSDatasetResource를 사용할 때 매우 중요한 점이 있다. 엔드포인트 끝에 슬래시(/)를 붙여야 한다. 엔드포인트에 접근하면서 / 를 붙이지 않는 경우, RAD Server는 “not found” 예외를 발생시킨다.

TEMSDatasetResource의 추가 기능들

지금까지 본 것들은 이미 매우 인상적이다. 클릭을 몇 번만 하면, 얼마든지 많은 데이터셋들을 원하는만큼 게시하고 API 개발을 매우 신속하게 할 수 있다. 하지만 TEMSDatasetResources 안에는 훨씬 더 많은 기능들이 들어있다. 주요 기능들을 분석해 보자.

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	dsrCUSTOMER
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 기능들: 엔드포인트에 대한 List, Get, Post, Put, Delete를 허용하거나 방지한다.

DataSet – 데이터셋(쿼리, 테이블 등)에 연결한다

KeyFields – 데이터셋 필드 선택: 데이터를 조회할 때 일치해야 하는 필드들을 선택한다.

PageParamName – 페이지 나누기에 사용할 파라미터의 이름: URL 안에서 사용한다 (예: ?**page**=1)

PageSize – LIST 액션에 접근할 때의 페이지 나누기 크기: 한번에 받아오는 크기를 지정한다.

SortingParamPrefix – 텍스트 문자열: 이 문자열은 데이터셋의 ValueFields 항목 앞에 붙어 정렬에 사용된다.

ValueFields – 필드들을 선택: 이 필드들은 URL 쿼리 안에서 파라미터화된다. 또한 JSON 응답 안에 나타난다.

Options – 서버 프로퍼티 설정: 파라미터 사용, 행(row) 페이지 나누기, 데이터셋 필드 정렬 등등을 활성화/비활성화한다.

이 컴포넌트에 대해 이제 조금 더 알게 되었다. 그러니 우리의 API 안에 담긴 이 기능들 중 몇 가지를 사용해 보자. `http://localhost:8080/test/customers/?sfCONTACT_LAST=A&page=1`에 접근하면 customers의 첫 페이지를 얻게 되는데, CONTACT_LAST 필드를 기준으로 오름차순으로 정렬된 결과를 받는다. 이 요청에서 만약 =A를 =D라고 바꾸면, 응답은 내림차순이 된다.

하지만 이 “order by”가 어떻게 SQL 안에 삽입되는 걸까? FDQuery를 열어보면 알 수 있다. 이 문장이 들어 있다.

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

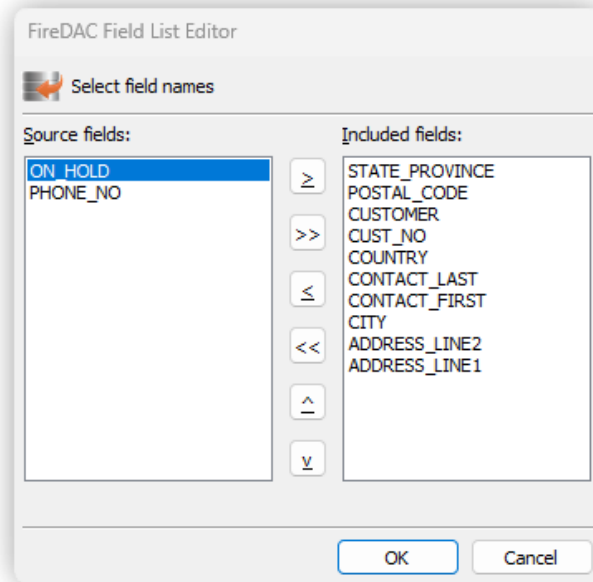
위 SQL 문장은 매우 간단하다. 하지만, 이 매크로가 핵심이 되어 작동한다. EMSDatasetResource는 이 매크로를 사용해 페이지 나누기와 정렬을 동일한 쿼리 안에서 혼합할 수 있다.



참고

페이지 나누기가 사용되고 있고 데이터셋의 끝에 도달하는 경우, RAD Server는 그저 빈 배열을 반환한다. 그래서 해당 페이지 안에 항목이 전혀 없다고 알려준다.

매우 유용한 기능 하나를 더 보자. 경우에 따라 우리의 로직 상, 데이터베이스로부터 받아오는 필드들 중에서 API가 노출하고 싶지 않은 것들이 있을 수 있다. ValueFields 프로퍼티를 사용하면, 게시할 필드들을 쉽게 선택할 수 있다.



API 엔드포인트에 게시할 필드들을 선택한다

04

REST Debugger

앞 장들에서, 우리는 REST API 에코시스템이 제공하는 액션들에 관해 보았다. 하지만 지금까지는, GET 요청만 사용해 보았다. 브라우저를 통해서였다. 여러분은 아마 POST, PUT, DELETE 액션 사용 방법이 궁금할 것이다. 이 장에서는 RAD Studio와 함께 제공되는 도구인 “REST Debugger(디버거)”를 사용한다. 이 도구는 테스트 수행 절차를 단순화할 뿐만 아니라, 여러분이 가장 빠른 방식으로 애플리케이션을 개발하도록 돕는다.



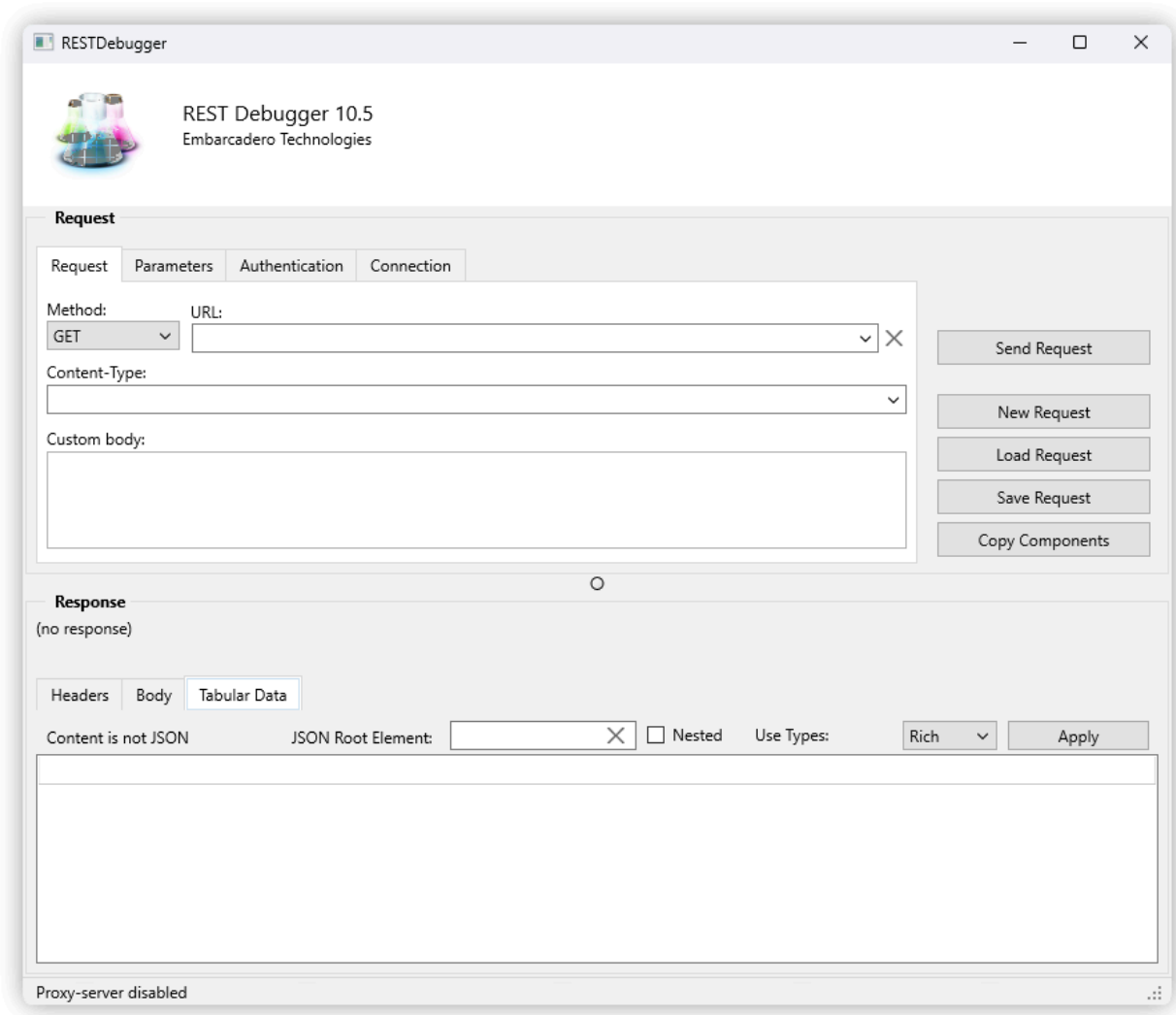
팁

REST Debugger는 RAD Server에서만 사용할 수 있는 제품이 아니다. 여러분은 REST Debugger를 사용해 써드-파티 REST API 서비스에 접근할 수 있다. 또한 개발 과정의 속도를 더 높일 수 있다. RAD Studio를 위한 통합 기능을 활용할 수 있기 때문이다.

REST Debugger란 무엇이며 어디에서 찾을 수 있는가

[REST Debugger](#)는 Embarcadero(엠바카데로)의 무료 솔루션이다. RESTful 웹 서비스를 탐색하고, 이해하고, Delphi와 C++ 빌더 앱에 통합할 때 사용할 수 있다. 어느 RESTful 웹 서비스가 어떻게 작동하는지를 개발자가 탐색하고 테스트해 이해할 수 있도록 돕는다. 다음과 같은 기능들이 들어 있기 때문이다. JSON Blob 필터링, OAuth 1.0/2.0 인증 간소화, 요청/리소스 파라미터 구성 등이 제공된다. 뿐만 아니라, REST 컴포넌트들을 복사해 여러분의 프로젝트 안에 바로 붙여 넣을 수도 있다. 클릭 몇 번만 하면 된다.

여러분이 사용해 보고 싶다면, RAD Studio의 메뉴에서 **Tools/REST Debugger**를 선택하면 된다. 또는 [이 링크](#)를 통해 독립 실행형 버전을 무료로 다운로드할 수 있다.



REST Debugger의 UI

(REST Debugger를 사용해) 우리의 첫 PUT 요청을 보내기

왼쪽에 있는 드롭다운 메뉴를 보면, 기본 설정이 GET 명령이다. 하지만, 다른 명령들을 여러분이 선택할 수도 있다. 즉, 브라우저에서는 할 수 없는 명령들을 REST Debugger에서는 선택할 수 있다.

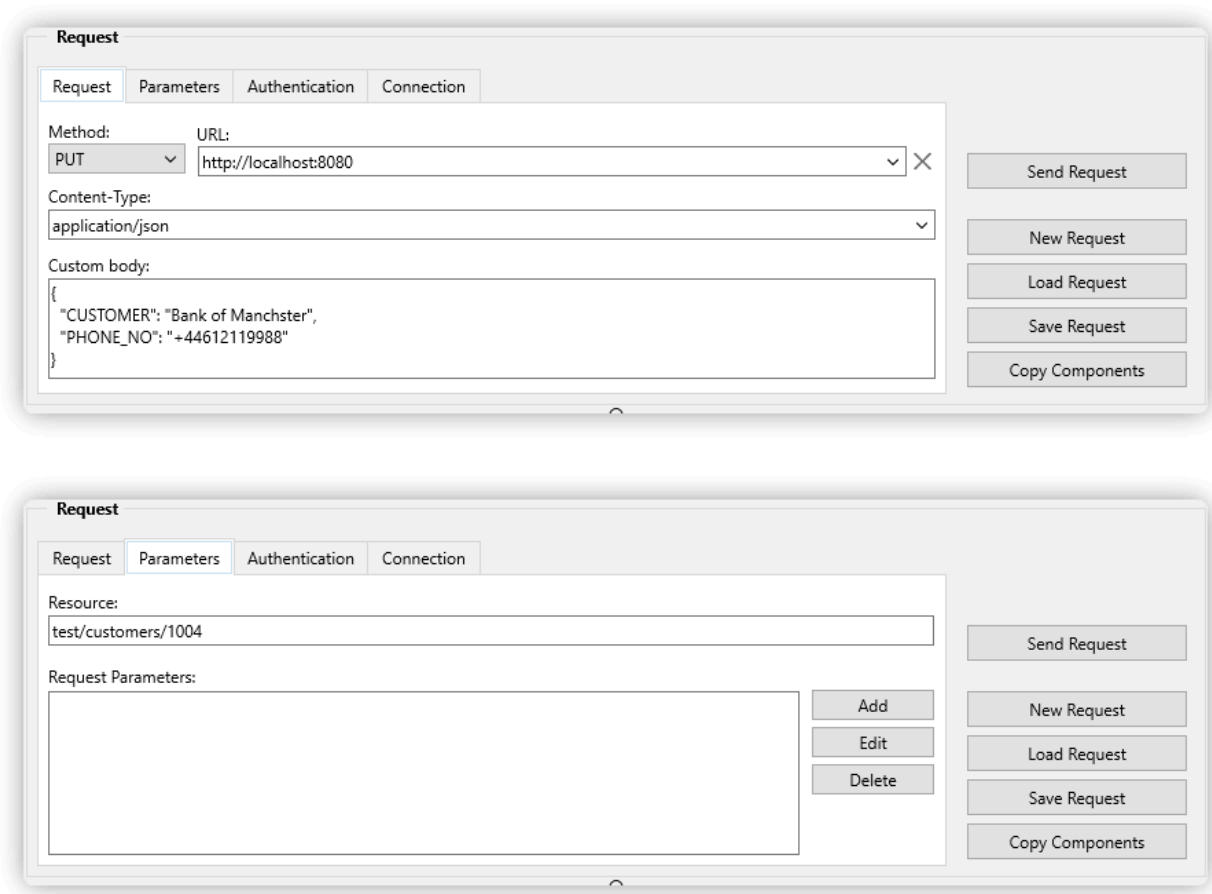
우리가 3장에서 만든 것과 동일한 프로젝트를 사용해, 이번에는 고객 한 명을 수정해 보자.



경고

먼저 3장에서 만든 RAD Server 프로젝트를 실행시켜 놓아야 한다. 그렇게 하지 않으면, 그 RAD Server가 제공하는 API 엔드포인트들을 호출하지 못한다.

고객 한 명을 수정하려면, customers 엔드포인트를 호출할 때, 그 고객의 ID를 사용해야 한다. 또한, 해당 프로퍼티들에 들어갈 새 값들을 요청 본문(body of the request) 안에 명시해야 한다.

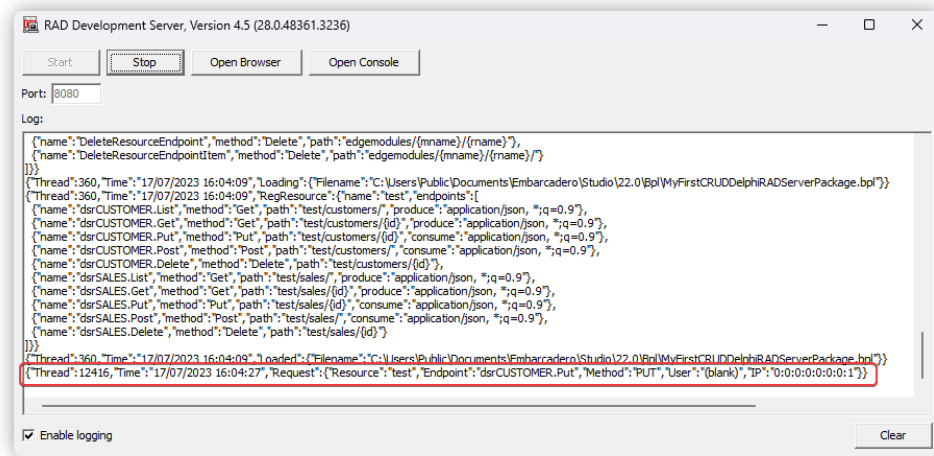


PUT 요청 전송에 필요한 값들을 정의한다

요청을 구성하기 위해 우리는 PUT 메서드와 URL(우리가 가리키고자 하는 리소스, 이 경우에는 ID가 1004인 고객)을 명시했다. 또한, JSON 본문(body)도 포함했다. 그 안에는 우리가 업데이트하려는 고객의 프로퍼티들을 넣는다. 이 경우에는 그 고객의 새 이름과 새 전화번호다.

이제 마지막 단계만 남았다. “Send Request”를 누르면 된다. 만약 200 HTTP 응답을 받는다면, 우리는 RAD Server의 로그에서 그 요청이 처리된 과정을 확인할 수 있다. 그리고 나서, 이 고객에 대해 GET 요청을 보낸다면(이 작업은 REST Debugger 또는 브라우저에서 할 수 있다) 성공적으로 업데이트된 그 고객의 데이터를 얻게 된다.

이 절차는 새 고객을 생성하려는 경우에도 똑같다. 하지만 우리는 새 고객에게 필요한 모든 정보를 본문 안에 담아 제공해야 한다. 또한, method를 변경해 POST로 지정해야 한다.



RAD Server 로그가 등록된 PUT 요청을 보여준다. 그리고 그 결과로 변경된 데이터다.

REST Debugger가 제공하는 기타 기능들

이 내용은 전적으로 RAD Server에만 해당되는 것이 아니다. 하지만, REST Debugger가 제공하는 매우 강력한 기능이므로 알아두면 좋다. URL, 파라미터 등을 정의해서 사용한 후에는, “copy components” 버튼을 사용하라. 이렇게 하면, 여러분에게 필요한 RAD Studio 컴포넌트들 모두가 클립보드 안에 담긴다. 그러므로, 여러분의 어느 프로젝트에서든지 바로 붙여넣기를 할 수 있다. 이 방식을 사용하면, 여러분이 RAD Server 또는 기타 써드-파티 API에 접근하는 UI 프로토타입을 만들 때 훨씬 더 신속하게 작업할 수 있다.

이 장에 해당하는 깃허브(GitHub) 리포지토리 안에는, 여기에 관한 기초적인 FMX 예시가 들어 있다. 그 API에 접근하기 위해 필요한 컴포넌트들이 있는데, 그것들은 모두 “copy components” 버튼을 사용해 복사해 붙여넣은 것들이다. 그 예시를 테스트해 보려면, 먼저 RAD Server 애플리케이션을 실행하고 나서, FMX 애플리케이션을 실행한다. 그리고 그 FMX 앱 안에 있는 “Send Request”를 누르면 된다.

REST API에 관한 또 다른 중요한 주제는 바로 인증(authentication)이다. 여러분이 만약 그 API에 접근하기 위해 인증을 받아야 한다면, “authentication” 탭 아래에 있는 여러 가지 방법을 사용할 수 있다. 또한 “parameters” 탭 안에 있는 “Add Parameters” 버튼을 사용해 특정 파라미터(예: 헤더에 api-key 파라미터)들을 여러분의 요청 안에 담을 수도 있다. 예를 들어 api-key 파라미터를 요청의 헤더(header)안에 넣을 수 있다.

05

FireDAC의 일괄 이동 및 JSONWriter 사용하기

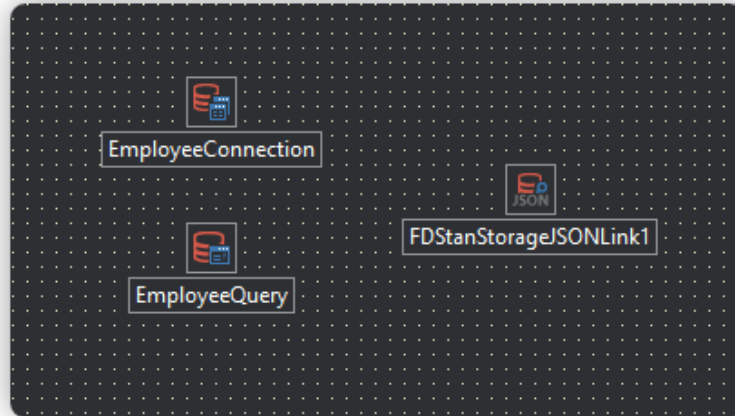
.....

프로젝트의 요구사항 또는 여러분에게 더 익숙한 기술에 무엇인가에, 여러분은 RAD Studio에서 더 많은 도구들을 활용해 여러분의 REST API를 만들 수 있다.

FireDAC(파이어닥) 컴포넌트들은 데이터베이스의 메타데이터와 데이터를 담은 스트림을 생성하고 읽어들이는데 사용할 수 있다. 여러분의 RAD Server 엔드포인트들 중 하나가 보내준 응답(response)인 JSON 안에 그것들이 인코딩 되어 있을 때에도 활용하기 좋다. 이 방식은 여러분이 만들 클라이언트 애플리케이션이 VCL 또는 FMX일 경우에 유용하다. MemoryTable들을 사용하면 자동으로 데이터베이스 정보와 메타데이터를 자동으로 매핑할 수 있다. JavaScript 등 기타 다른 언어를 사용하는 클라이언트 애플리케이션들은 응답에 담겨있는 데이터베이스 정보와 데이터를 처리하는 데 문제가 있을 수 있다. 하지만, RAD Studio는 깔끔한 JSON을 생성하는 방법을 제공한다. 그래서 JavaScript 등 다른 언어들이 기대하는 것을 수신할 수 있도록 맞춰 줄 수 있다.

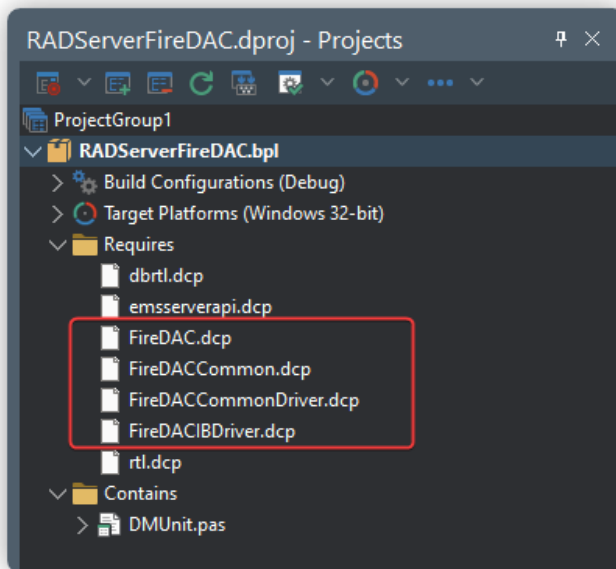
(메모리 스트리밍을 사용해) JSON 데이터베이스의 데이터를 반환하기

FireDAC에 있는 컴포넌트들 중에는 데이터베이스 테이블 정보에 접근하고 그 결과를 JSON 문자열로 생성할 수 있는 것들이 있다. RAD Server 애플리케이션을 하나 만들자. 이번에도 Create package with resource를 선택해 만들어라. FDConnection 컴포넌트를 추가하고, 그것을 InterBase 샘플 데이터베이스인 Employee.gdb에 연결하라. FDQuery 컴포넌트를 추가해 이름을 EmployeeQuery라고 지정하라. 그리고 SQL 문장을 `select * from employee` 라고 지정한다. FDStanStorageJSONLink 컴포넌트를 추가한다. 이것은 JSON 생성을 쉽게 하도록 도와준다.

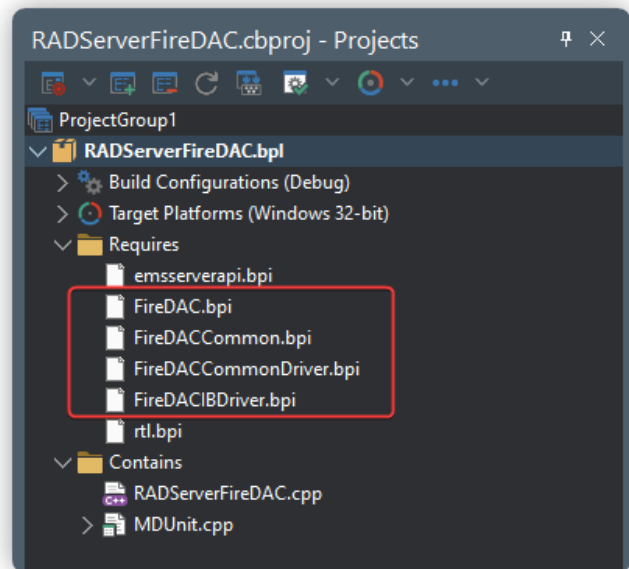


RAD Server 프로젝트의 리소스 모듈

RAD Server의 Delphi 기반 애플리케이션을 빌드할 때, 일련의 경고들이 표시되고 대화상자가 나타나면서, 이 애플리케이션 패키지가 이미 설치되어 있는 다른 패키지들과 호환되도록 유도한다. OK 버튼을 클릭하면 필요한 패키지 파일들이 프로젝트의 requires 구역에 추가된다. C++Builder의 경우 이 패키지들을 수작업으로 추가할 수 있다(Projects 창 안의 Requires 노드에서 마우스 오른쪽 버튼을 클릭하고 팝업 메뉴에서 Add Reference...를 선택).



Delphi RAD Server FireDAC 프로젝트



C++ RAD Server FireDAC 프로젝트



이 파일들은 *C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\lib* 안에서 각 목표 플랫폼 별로 찾을 수 있다.

이 RAD Server의 GET 메서드 구현은 아래와 같다. 이 구현은 메모리 스트림을 사용해 JSON 응답을 보내는데, 그 안에 employee 테이블의 데이터가 담겨서 전달된다.

Delphi:

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

C++:

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

브라우저를 사용해 URL에 *http://localhost:8080/FireDAC* 라고 적어 보내라. 이 요청을 통해 받은 응답에는 이 데이터베이스의 employee 테이블이 JSON 데이터로 담겨있을 것이다. 또한 그 JSON에는 데이터 외에도 훨씬 더 많은 정보들이 들어있다. 게다가 메타데이터 정보도 그 응답 안에 들어 있다. 즉 테이블, 컬럼들, 타입들 등에 대한 정보도 전달된다.



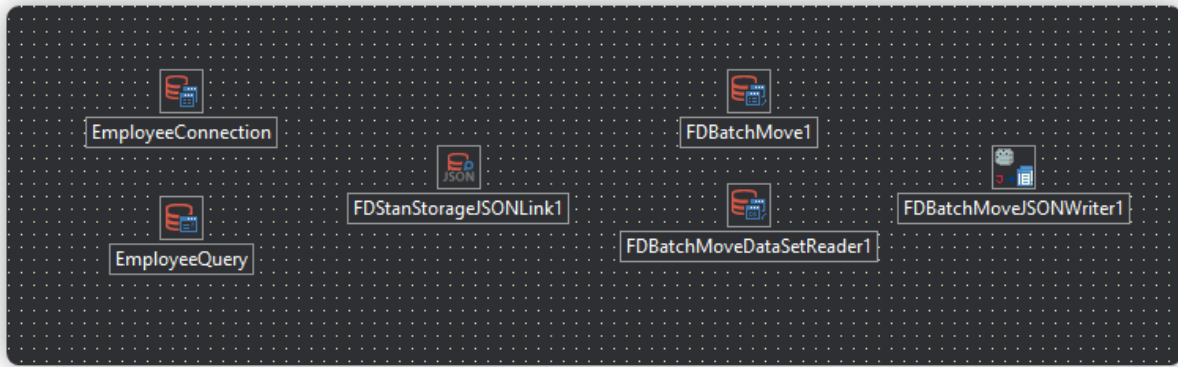
브라우저 창이 그 JSON 응답을 보여주고 있다

이것은 결코 단순한 컬럼과 값을 담은 JSON이 아니다. 따라서 다른 언어에서 이 JSON을 받아서 사용하려면 코드를 사용해 응답을 파싱(parsing)해야 한다. 그러지 않고는 사용하지 못한다. 하지만, RAD Studio를 사용해 클라이언트를 개발한다면 여러분은 매우 편리하게 이 JSON을 사용할 수 있다.

FireDAC(파이어닥)의 BatchMove, BatchMoveDataSetReader, BatchMoveJSONWriter 사용하기

복잡한 데이터베이스의 경우, 이전 장 그리고 위에서 언급한 것과 같은 방식들을 사용하면, 훨씬 더 많은 코드를 작성하게 된다. FireDAC에는 FDBatchMove, FDBatchMoveDataSetReader, FDBatchMoveJSONWriter 컴포넌트가 있다. 이것들을 사용하면, 이 JSON 응답 생성 작업을 크게 간소화된다.

이제 앞에서 만든 프로젝트를 업그레이드해보자. 리소스 모듈에 FDBatchMove, FDBatchMoveDataSetReader, FDBatchMoveJSONWriter 컴포넌트를 추가하라.



FireDAC의 Query, BatchMove, DataSetReader, JSONWriter가 들어있는 리소스 모듈

FDBatchMoveDataSetReader의 DataSet 프로퍼티를 EmployeeQuery로 지정하라.

새 엔드포인트를 하나 만들고 이름을 GetBatchMove라고 지정하자.

Delphi:

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

C++:

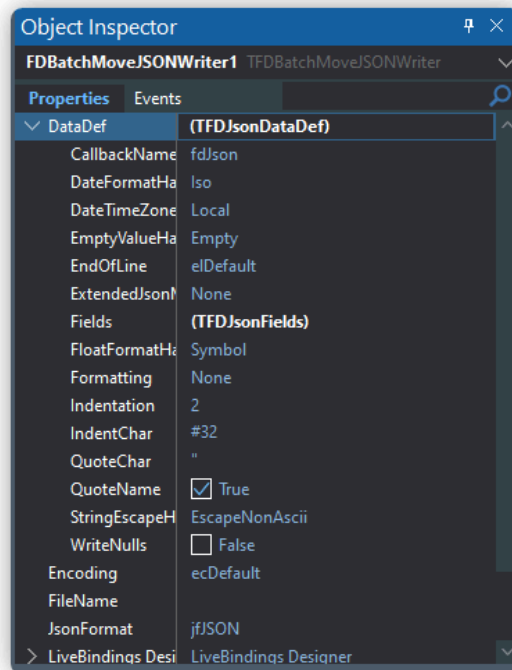
```
void TEmployeeResource1::GetBatchMove(TEndpointContext* Acontext,
    TEndpointRequest* ARequest, TEndpointResponse* AResponse)
{
    FDBatchMoveJSONWriter1->JsonWriter = AResponse->Body->JSONWriter;
    FDBatchMove1->Execute();
}
```

이 GET 메서드를 호출한다. URL을 *http://localhost:8080/BatchMove* 로 요청하면 된다. 그러면 JSON 데이터 결과가 반환된다:



브라우저가 JSON 결과를 보여준다. BatchMove를 사용한 결과다.

FDBatchMoveJSONWriter는 JSON 결과의 서식을 지정할 수 있도록 여러 옵션들을 제공한다. DateFormat들, 줄 종결, Null 쓰기 등에 관한 것들이다.



BatchMoveJSONWriter의 DataDef 아래 하위 프로퍼티들을 Object Inspector에서 지정한다

FDBatchMove 컴포넌트를 사용하면 매핑(mapping)들을 생성할 수 있다. 그래서 소스와 대상 컬럼을 짝을 지어놓고, 거기에 맞게 현재 소스 레코드의 값들을 가져올 수 있다.

이것은 Mappings 프로퍼티를 채우는 방법이다:

- 수작업으로 디자인 또는 런타임에 채우기: 여러분이 사용자 정의 매핑, 변환 표현식 등을 명시할 수 있다.
- 실행 호출 시 자동 채우기 (Mappings가 비어 있는 경우): 소스 컬럼과 대상 컬럼 매핑은 컬럼 이름을 기준으로 수행된다. 만약 대상 컬럼에 대응하는 소스 컬럼이 없는 경우, 그 대상 컬럼은 매핑에서 제외된다. 그래서 데이터 이동 시 그 컬럼은 채워지지 않는다.

참고 자료

- [마르코 칸투 블로그: JSON에 대한 데이터셋 매핑 - RAD Server 웹 서비스 Delphi 예시](#)
- [FireDAC.Comp.BatchMove.TFDBatchMove](#)
- [FireDAC.Comp.BatchMove.JSON.TFDBatchMoveJSONWriter](#)
- [읽기 및 쓰기 JSON 프레임워크](#)
- [FireDAC.TFDBatchMove 샘플](#)
- [RTL.JSONWriter](#)

06

JSONValue, JSONWriter, JSONBuilder

.....

RAD Server는 JSON 데이터 다루기를 지원해, 여러 프로그래밍 언어와 도구에서 사용할 수 있는 JSON을 만들 수 있도록 한다. JSON 문자열 생성, 그 문자열을 응답으로 전송, 반환 결과를 처리할 수 있는 클라이언트 애플리케이션 코드 정도라면 데이터가 적은 경우에 괜찮다. 하지만 전체 데이터베이스나 복잡한 데이터 구조를 담은 JSON 배열 응답은 얼마나 크겠는가? RAD Studio가 제공하는 JSON 데이터를 다루는 프레임워크는 크게 세 가지다. 이 장은 RAD Server 애플리케이션이 (호출 애플리케이션에게) 반환할 JSON을 만드는 여러 방법들 중 몇 가지를 다룬다.

JSON 데이터를 다루는 프레임워크들

RAD Studio에는 JSON 데이터를 다루는 프레임워크들이 여러 개 있다. 그 중 이 세 가지가 가장 일반적이다:

- JSON Object들 프레임워크 - 임시 오브젝트를 만들어서 JSON 데이터를 읽고 쓴다.
- Reader들과 Writer들 기반 JSON 프레임워크 - JSON 데이터를 직접 읽고 쓴다.
- JSONBuilder - Writer들을 사용한다. 복잡한 구조들을 더 유지보수하기 더 좋은 방식으로 생성한다.

JSON Object들 프레임워크는 임시 오브젝트 생성이 필요하다. 그 오브젝트를 사용해 JSON 데이터를 파싱하거나 생성한다. JSON 데이터를 읽거나 쓰려면, 매개체가 되는 메모리 오브젝트를 생성해야 한다. 즉 TJSONObject, TJSONArray, TJSONString 등을 먼저 만들고, 그것을 이용해 JSON을 읽거나 쓰는 방식이다.

Reader들과 Writer들 기반 JSON 프레임워크를 사용하면 애플리케이션이 JSON 데이터를 직접 읽거나 스트림(stream)에 쓸 수 있다. 임시 오브젝트를 만들 필요가 없다. 임시 오브젝트를 만들지 않고 JSON을 읽고 쓸 수 있기 때문에 성능 향상과 메모리 소비 개선 효과가 있다.

JSONBuilder는 앞의 두 가지를 결합한 것이다. 더 읽기 쉽고 유지보수하기 쉬운 코드를 작성할 수 있도록 하려는 목표를 위해 생겨났다. 또한, 메서드들을 이어 붙여가는 현대적인 방식(method chaining)이다.

이 장의 데모 프로젝트 안에는 3개의 서로 다른 엔드포인트들이 있다. 그런데 그것들이 만들어 내는 응답은 완전히 똑같다. 다만, 이 사용하는 프레임워크만 다르다. 여러분은 그 중에서 더 편한 것을 프로젝트에서 사용하면 된다.



```
{
  "colors": [
    {
      "name": "red",
      "hex": "#ff0000",
      "default": false,
      "customId": null
    },
    {
      "name": "blue",
      "hex": "#0000ff",
      "default": true,
      "customId": 653992
    }
  ]
}
```

똑같은 이 JSON 응답을 각 엔드포인트가 생성한다

JSONValue를 사용하기

JSON Object들 프레임워크를 사용해 JSON 문자열을 만든다. 코드 안에서 조립하는 방식이다. JSONValue는 모든 JSON 클래스들의 조상 클래스다. 즉, JSON 규격의 문자열, 오브젝트, 배열, 숫자, 불리언, 참, 거짓, null 값을 정의하는 모든 것들의 조상이다. RAD Studio JSON 구현에 들어 있는 클래스들과 메서드들은 다음과 같다.

TJSONObject - JSON 오브젝트를 구현한다. TJSONObject 안에 있는 주요 메서드들:

- **Parse** 메서드 - JSON 데이터 스트림을 파싱(parsing)한다. 만나는 JSON 쌍(pair)들을 TJSONObject 인스턴스 하나 안에 저장한다.
- **ParseJSONValue** 메서드 - 바이트(byte) 배열을 파싱해 읽어낸 데이터로 해당 JSON 값(value)을 생성한다.
- **AddPair** 메서드 - JSON 오브젝트에 새 JSON 쌍을 추가한다.
- **GetPair** 메서드 - 키(key)-값(value) 쌍(pair)을 반환한다. JSON 오브젝트의 쌍 목록 안에 I(아이) 인덱스를 명시해 얻는 방식이다. 만약 명시된 인덱스 I가 범위를 벗어난 경우에는 nil을 반환한다.
- **GetPairByName** 메서드 - 키-값 쌍을 반환한다. PairName 문자열을 명시해, JSON 오브젝트의 쌍 목록 안에 있는 키 부분들 중에서 찾아 일치하는 것을 얻는 방식이다. 만약 PairName과 일치하는 키가 없으면 nil을 반환한다.
- **SetPairs** - 이 JSON 오브젝트가 담는 키-값 쌍들의 목록을 정의한다.

- **FindValue** - 지정된 JSON 경로에서 TJSONValue 인스턴스를 찾고 반환한다. 그렇지 않으면 nil을 반환한다.
- **GetValue** - JSON 오브젝트의 Name 키로 명시된 키-값 쌍에서 값 부분을 반환한다. Name과 일치하는 키가 없는 경우 nil을 반환한다.
- **Pairs** - JSON 오브젝트의 쌍 목록에서 지정된 인덱스에 해당하는 키-값 쌍에 접근한다. 지정된 인덱스가 범위를 벗어난 경우 nil을 반환한다.
- **GetCount** - JSON 오브젝트의 키-값 쌍의 개수를 반환한다.

TJSONArray - JSON 배열을 구현한다. TJSONArray 안에 있는 주요 메서드들:

- **Add** - Element 파라미터를 통해 명시한 null이 아닌 값을 현재 요소(element) 목록에 추가한다.
- **Get** - 명시된 인덱스에 있는 요소를 JSON 배열에서 반환한다.
- **Pop** - JSON 배열에서 첫 번째 요소를 제거한다.
- **Size** - JSON 배열의 크기를 반환한다.
- **ToBytes** - 현재 JSON 배열의 콘텐츠를 직렬화해 바이트 배열을 만든다.
- **ToString** - 현재 JSON 배열을 직렬화해 문자열을 만들고, 그 결과 문자열을 반환한다.

추가적인 JSON 클래스들:

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

JSON 클래스를 사용하는 예시

데모 프로젝트 안에 있는 이 GetJSON 메서드의 구현은 다음과 같다. 이 메서드는 Get 엔드포인트를 구현한다. 몇 개의 JSON 클래스들을 사용해 JSONObject들과 TJSONArray의 결과를 생성, 파싱, 표시하는 방식이다.

Delphi:

```
procedure TTestResource1.GetJSON(const AContext: TEndpointContext; const ARequest:
TEndpointRequest; const AResponse: TEndpointResponse);
begin
```

```
// JSON 오브젝트 몇 개를 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 생성한다. 그리고 앞에서 만든 오브젝트들을 그 안에 할당한다
var JSONArray := TJSONArray.Create;
JSONArray.Add(JSONRed);
JSONArray.Add(JSONBlue);
// 오브젝트를 하나 더 추가로 생성한다. 그리고 앞에서 만든 colors 배열을 그 안에 담는다
var JSONObject := TJSONObject.Create;
JSONObject.AddPair('colors', JSONArray);
AResponse.Body.SetValue(JSONObject, True);
end;
```

C++:

```
void TTestResource1::GetJSON(TEndpointContext* AContext, TEndpointRequest* ARequest,
TEndpointResponse* AResponse)
{
    // JSON 오브젝트 몇 개를 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 생성한다. 그리고 앞에서 만든 오브젝트들을 그 안에 할당한다
    TJSONArray* JSONArray = new TJSONArray();
    JSONArray->Add(JSONRed);
    JSONArray->Add(JSONBlue);
    // 오브젝트를 하나 더 추가로 생성한다. 그리고 앞에서 만든 colors 배열을 그 안에 담는다
    TJSONObject* JSONObject = new TJSONObject();
    JSONObject->AddPair("colors", JSONArray);
}
```

```
AResponse->Body->SetValue(JSONObject, True);
}
```

JSONWriter를 사용하기

JSONWriter를 사용하면 RAD Server 애플리케이션 개발 시, 사용자 정의 JSON 만들기가 간단하다. JSON에 담긴 데이터는 프로그래밍 언어 클라이언트들이 사용할 수 있다. 먼저 JSONWriter를 사용해 여러분의 JSON 오브젝트 쓰기를 시작한다. 그리고 프로퍼티의 이름과 값을 써내려 간다. 맨 끝에는 JSON 오브젝트 쓰기를 끝낸다.

JSONWriter 사용 예시

아래 메서드도 Get 엔드포인트를 구현해 데이터를 반환한다. 이번에는 JSONWriter의 WriteStartArray, WriteStartObject, WritePropertyName, WriteValue, WriteEndObject, WriteEndArray 메서드들을 사용하는 방식이다. AResponse 파라미터는 내장된 JSONWriter를 가지고 있다. 그 JSONWriter를 사용하면, 매우 쉬운 방식으로 더욱 복잡한 구조들을 직접 해당 응답에 써내려 갈 수 있다.

Delphi:

```
procedure TTestResource1.GetJSONWriter(const AContext: TEndpointContext; const
ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
begin
    // 변수에 담는다. 그래서 모든 줄에 AResponse.Body.JSONWriter를 입력하는 것을 피한다
    var Writer := AResponse.Body.JSONWriter;
    // JSON 오브젝트를 시작한다
    Writer.WriteStartObject;
    Writer.WritePropertyName('colors');
    // JSON 배열을 시작한다
    Writer.WriteStartArray;
    Writer.WriteStartObject;
    Writer.WritePropertyName('name');
    Writer.WriteValue('red');
    // 필요한 만큼 WritePropertyName 및 WriteValue 문장을 써내려 간다
    Writer.WritePropertyName('hex');
    Writer.WriteValue('#ff0000');
    Writer.WritePropertyName('default');
    Writer.WriteValue(False);
    Writer.WritePropertyName('customId');
    Writer.WriteNull;
    Writer.WriteEndObject;
    // 필요한 만큼 추가로 JSON 오브젝트를 써내려 간다
    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 끝낸다
Writer.WriteEndObject;
// JSON 배열을 끝낸다
Writer.WriteEndArray;
Writer.WriteEndObject;
end;

```

C++:

```

void TTestResource1::GetJSONWriter(TEndpointContext* AContext, TEndpointRequest*
ARequest, TEndpointResponse* AResponse)
{
    // 변수에 담는다. 그래서 모든 줄에 AResponse.Body.JSONWriter를 입력하는 것을 피한다
    TJsonTextWriter* Writer = AResponse->Body->JSONWriter;
    // JSON 오브젝트를 시작한다
    Writer->WriteStartObject();
    Writer->WritePropertyName("colors");
    // JSON 배열을 시작한다
    Writer->WriteStartArray();
    Writer->WriteStartObject();
    Writer->WritePropertyName("name");
    Writer->WriteValue("red");
    // 필요한 만큼 WritePropertyName 및 WriteValue 문장을 써내려 간다
    Writer->WritePropertyName("hex");
    Writer->WriteValue("#ff0000");
    Writer->WritePropertyName("default");
    Writer->WriteValue(False);
    Writer->WritePropertyName("customId");
    Writer->WriteNull();
    Writer->WriteEndObject();
    // 필요한 만큼 추가로 JSON 오브젝트를 써내려 간다
    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 끝낸다
Writer->WriteEndObject();
// JSON 배열을 끝낸다
Writer->WriteEndArray();
Writer->WriteEndObject();
}

```

JSONBuilder를 사용하기

이 프레임워크는 JSONWriter를 둘러싼 래퍼(wrapper)다. 그래서 JSON 구축을 더 빠르고 더 읽기 쉬운 방식으로 한다. 이것은 fluent 인터페이스(“메서드 이어붙이기, method chaining”이라고도 한다) 방식을 따른다. 이 방식은 매우 복잡한 JSON 구조의 경우에 여러분의 코드를 간결하게 한다. 그래서 유지 보수와 읽기가 더 쉽다.

같은 프로젝트 안에서, 또 다른 엔드포인트를 찾을 수 있을것이다. 이것은 JSONBuilder를 사용해 응답을 만든다. 코드를 살펴보자:

Delphi:

```

procedure TTestResource1.GetJSONBuilder(const AContext: TEndpointContext; const
ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
begin
    var Writer := AResponse.Body.JSONWriter;
    // 응답에 담겨있는 JSONWriter를 JSONBuilder에 연결한다
    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
    end;
end;

```

```

        .EndObject;
    finally
        Builder.Free;
    end;
end;

```

C++:

```

void TTestResource1::GetJSONBuilder(TEndpointContext* AContext, TEndpointRequest*
AResponse, TEndpointResponse* AResponse)
{
    TJsonWriter* Writer = AResponse->Body->JSONWriter;
    // 응답에 담겨있는 JSONWriter를 JSONBuilder에 연결한다
    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 Studio와 함께 제공되는 매우 유용한 샘플 프로젝트가 있다. 이름은 fmWorkBench다 (Delphi에서만 사용):
 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 정의 엔드포인트 만들기

이 장까지 오기 전에, 우리는 기본적인 JSON 구조들(배열, 오브젝트,...)을 보았다. 그 URI도 꽤 단순했다: /customers, /sales.... 하지만, REST API 모범 사용 관행을 생각하면, 하위 리소스 URI들이 있는 경우 또는 JSON 응답 안에 배열/오브젝트들이 중첩되어 다른 오브젝트 안에 들어 있는 경우가 많다. 이 장은 이러한 유형의 구조들을 RAD Server를 통해 구현하는 방법에 대해 말한다. 또한 여러분의 GET, POST, PUT, DELETE 메서드를 만드는 방법을 설명한다.

모범 사용 관행의 예시

비록 여러분의 API의 구조를 여러분이 원하는 형식 만들어도 되지만, 수 천 가지 글들이 표준화 또는 REST API에 대해 모범 사용 관행들을 제시하고 있다. 결국 여러분의 API를 어떻게 구성할 것인지는 여러분에게 달려있다. 하지만 이 표준들의 기초를 조금이라도 알면 가치가 있다. 또한, 바퀴를 다시 발명하지 않으려고 노력하면 좋다.

예시: 특정 고객에게 접근할 때는, /customers/{id} URI를 사용한다고 배웠다. 그런데 만약 그 특정 고객의 매출에 접근하려면 어떻게 해야 할까? 가장 흔한 선택은 엔드포인트를 하나 더 정의하는 것이다: /customers/{id}/sales. 이 엔드포인트는 ID로 지정된 특정 고객의 매출 주문들을 반환한다.

이는 모범 사용 관행으로 간주된다. 중첩된 리소스 또는 하위-리소스라고 부르는 방식이다. 이렇게 하면 계층 관계를 여러분의 엔드포인트 사이에 정의할 수 있다. 따라서, 조직 외부 또는 내부의 개발팀이 여러분의 API를 더 쉽게 이해한다.

API가 너무 수다스럽지 않도록 하기

애플리케이션을 개발하다 보면, 어떤 양식(form)이나 웹페이지에 접속했을 때 여러 개의 엔드포인트(URL 주소들)로부터 데이터를 가져와야만 하는 상황을 만나는 경우가 많다. 고객의 판매 주문에 접근한다고 가정해 보자. 고객 상세 정보, 주문 정보, 배송 주소, 해당 주문의 품목들, 어쩌면 결제와 송장 정보까지 필요할 것이다. 요청 목록은 꽤 길어질 수 있다. 그런데, REST API의 경우에 문제가 있다. 바로 요청의 비용이 비싸다는 점이다. 여기서 비용이 비싸다는 것은 서버가 이 모든 요청을 처리하는데 드는 비용만이 아니다. 인터넷의 통신 시간을 고려하면 지연 시간(latency) 역시 그 비용에 해당된다. 요청 하나가 왕복하는 데 수 밀리초가 걸리는데, 특정 페이지 하나에 접속하는데 10개 이상의 요청을 보내야 한다면 개선할 여지가 많다. 이런 경우라면, 중첩된(nested) JSON 응답이 더 적합하다.

상상해 보자. RAD Server에 요청할 때 단 한 번만 요청한다. 그 요청은 특정한 어느 고객에 대한 요약과 모든 매출을 달라고 한다. 이는 전형적인 마스터-디테일 관계와 똑같을 수 있다. 하지만 요청 하나만으로 모두를 반환한다. 우리는 이를 달성하는 방법도 살펴본다.

하위-리소스들을 추가하기

하위-리소스를 위해, 우리는 여전히 TEMSDatasetAdapter를 사용할 수 있다. 앞 장들에서 이미 보았던 것이다. 그것의 애트리뷰트들 중 몇 가지만 살짝 바꾸면, 나머지는 RAD Studio와 FireDAC이 알아서 한다.

지금까지 사용하던 그 프로젝트를 사용하자(쿼리 2개: qryCUSOTMER, qrySALES). 이제 qrySALES의 SQL을 다음과 같이 수정하라:

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

그리고 EMSDatasetAdapter의 오브젝트인 drsSALES 위에 있는 애트리뷰트들을 변경해 아래와 같이 되도록 하라:

Delphi

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

C++:

```

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

```

우리는 방금 SQL 문장 안에 WHERE 절을 추가했다. 그래서 특정 고객의 매출을 필터링하는 파라미터를 반영했다.

이 애트리뷰트들을 살펴보면, 더 흥미로운 일이 벌어진다. RAD Server는 자동으로 {CUST_NO}에서 값을 꺼내 FireDAC 쿼리 안에 넣는다. 그래서 해당 매출만을 필터링한다. 또한, 우리는 나머지 메서드들(List, Get, Post 등)도 지정해야만 했다. 왜냐하면 이제는 동일한 엔드포인트에 관련되는 키가 2개이기 때문이다. 따라서 키의 이름을 명시해 주는 것이 꼭 필요하다. 그래야 작동한다. 좋은 소식이 있다. 우리는 여전히 이 엔드포인트들을 사용해 특정 매출을 생성, 수정, 삭제할 수 있다. 즉, 사용하는 방식은 EMSDatasetAdapter를 사용하면 생성되는 다른 모든 엔드포인트들과 마찬가지로이다.

중첩되는 데이터를 응답에 추가하기 (마스터/디테일)

이제 우리의 첫 하위-리소스 엔드포인트를 가지게 되었다. 이제 중첩된 응답을 하나 만들고 그 안에 여러 값들을 담아보자. 그러려면, 우리가 코드를 조금 작성해야 한다.

TTestResource1 데이터모듈 클래스 안에 published(퍼블리시드) 메서드를 하나 만들자.

Delphi:

```

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

// 이것의 구현은 아래와 같다

procedure TTestResource1.GetCustomerDetails(const AContext: TEndpointContext; const
  ARequest: TEndpointRequest;
  const AResponse: TEndpointResponse);
begin
  var lCustomerNo := ARequest.Params.Values['CUST_NO'].ToInteger;
  // 파라미터를 사용한다. CustomerNo를 그냥 이어붙이지 않는다. 그래서 SQL 삽입 공격을
  방지한다
  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}";

// 이것의 구현은 아래와 같다

void TTestResource1::GetCustomerDetails(TEndpointContext* AContext, TEndpointRequest*
ARequest, TEndpointResponse* AResponse)
{
  int lCustomerNo = ARequest->Params->Values["CUST_NO"].ToInt();
  // 파라미터를 사용한다. CustomerNo를 그냥 이어붙이지 않는다. 그래서 SQL 삽입 공격을
  방지한다
  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 이해해 보자. 특정 고객 ID를 이용해 세부 정보를 추출한다. 그 고객 ID는 URL에서 가져온 것이다. 그러기 위해 매크로를 사용했다. 그 다음에는, 이미 사용하고 있는 qryALES에게 얻어낸 고객 ID를 파라미터로 전달한다. 따로 쿼리를 더 만들 필요가 없다.

EMSDatasetAdapter와 같은 컴포넌트들을 사용하는 방식은 로우-코드 방식이다. 그래서 우리에게 많은 도움이 된다. 하지만, 때로는 특정 요구 사항들이 필요한 경우가 있다. 그래서 우리가 직접 코드를 구현해야 할 때가 있다. 이전 장들에서 이미 살펴봤듯이, JSONWriter들을 사용하면 우리의 응답을 원하는 대로 사용자 정의할 수 있다.

이 예시 프로젝트의 코드를 받아서 더 살펴보면 이렇게 되어 있다. 각 요청이 들어올 때마다, ARequest와 AResponse 파라미터에 담긴 값을 접근해 필요한 모든 정보에 접근한다. 그리고 우리가 보낼 응답을 구축한다. 여기에서는 “WriteStartObject” 메서드를 사용해 새 오브젝트를 만든 다음 TQuerySerializer 클래스의 두 가지 메서드(자세한 설명은 뒤에서 한다)를 사용한다. 그리고 나서 그 오브젝트를 끝낸다(end).

여러 가지 유용한 메서드와 프로퍼티가 [JSONWriter들](#) 와 [JSONReader들](#)에 있다. 그래서 코딩 환경을 훨씬 더 쉽게 해준다. 강력하게 권하건데, 문서집을 읽어서 이것들이 제공하는 모든 기능들을 배우기 바란다.

이제 여러분은 이것이 궁금할 것이다: **ExcludeMasterFieldFromFields**와 **SerializeMasterDetail** 메서드는 무엇일까? 이 두 메서드는 이 특정 데모 예시를 위해 사용되었다. 하지만, 여러분은 깃허브 리포지토리 데모들에서 이것들을 들여다 볼 수 있다. 또한, 당연히 여러분의 프로젝트에서도 이런 코드를 마음껏 사용할 수 있다. 이 메서드들이 하는 일은 해당 메서드 안에 잘 문서화 되어 있다. 하지만 요약하면, 마스터-디테일 관계를 JSON 오브젝트 하나로 변환한다. 해당 디테일 쿼리는 그 메인 JSON 오브젝트 안에 JSON 배열로 삽입된다. **ExcludeMasterFieldFromFields**는 꼭 필요한 것이 아니다. 하지만, 중복되는 데이터를 피할 수 있다. 즉, 디테일 필드들에서 해당 마스터 필드를 제외한다.



팁

Data.DBJson 유닛을 확인하라. 여기에는 여러 클래스가 포함되어 있다. 데이터셋을 JSON으로 변환하거나 그 반대로 변환하는 작업을 도와주는 것들이다. 이 예시에서, 우리는 *TDataSetToJSONBridge* 클래스를 사용했다. 그래서 훨씬 더 빠르고 세밀하게 직렬화 한다.

Delphi:

```
// 마스터/디테일 관계인 쿼리 2개가 주어지면, JSON 오브젝트 1개를 반환한다. 그 안에는 중첩된
// 배열이 있다. 그 배열은 디테일 쿼리다
function TTestResource1.SerializeMasterDetail(AMasterDataset: TFDQuery;
ADetailDataset: TFDQuery; APropertyName: string; AFields: TStringList = nil):
TJSONObject;
begin
  var lBridge := TDataSetToJSONBridge.Create;
  try
    // 마스터 쿼리의 현재 레코드를 가져와 JSON 오브젝트로 변환한다
    lBridge.Dataset := AMasterDataset;
    lBridge.IncludeNulls := True;
    // 오직 현재 레코드만 처리하면 된다고 명시한다
    lBridge.Area := TJSONDataSetArea.Current;
    // 그 마스터 레코드를 오브젝트로 추가하고 JSON 결과에 넣는다
    Result := TJSONObject(lBridge.Produce);

    // 내보내고자 하는 필드들의 목록을 전달하고 싶으면 그것들을 브리지에 할당한다. 그렇게 하지
    // 않은 경우에는 기본 동작을 수행한다. 즉, 쿼리의 모든 필드들을 내보낸다
    if Assigned(AFields) then
      lBridge.FieldNames.Assign(AFields);
    // 동일한 브리지를 재사용한다. 하지만, 이번에는 디테일 데이터셋을 할당한다
    lBridge.Dataset := ADetailDataset;
    // 이 경우 쿼리에 담겨 있는 모든 레코드들이 처리된다
    lBridge.Area := TJSONDataSetArea.All;
    // 디테일 배열을 임시 배열에 저장한다. 나중에 메인 오브젝트 안에 추가하기 위해서다
    var lJSONArray := TJSONArray(lBridge.Produce);
    // 임시 배열을 메인 오브젝트에 추가한다. 이때 인자로 전달받은 프로퍼티 이름을 반영한다
    Result.AddPair(APropertyName, lJSONArray);
  finally
    lBridge.Free;
  end;
end;

// 쿼리에 마스터 필드가 할당되어 있는 경우, TStringList를 반환한다. 그 안에는 모든 필드들이
// 담는데, 해당 마스터 필드만은 제외한다
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 관계인 쿼리 2개가 주어지면, JSON 오브젝트 1개를 반환한다. 그 안에는 중첩된
// 배열이 있다. 그 배열은 디테일 쿼리다
TJSONObject* TTestResource1::SerializeMasterDetail(TFDQuery* AMasterDataset, TFDQuery*
ADetailDataset, System::UnicodeString APropertyName, TStringList* AFields)
{
    TDataSetToJSONBridge *lBridge = new TDataSetToJSONBridge;
    try {
        // 마스터 쿼리의 현재 레코드를 가져와 JSON 오브젝트로 변환한다
        lBridge->Dataset = AMasterDataset;
        lBridge->IncludeNulls = True;
        // 오직 현재 레코드만 처리하면 된다고 명시한다
        lBridge->Area = TJSONDataSetArea::Current;
        TJSONObject* lJSONObject = new TJSONObject;
        // 그 마스터 레코드를 오브젝트로 추가하고 JSON 결과에 넣는다
        lJSONObject = (TJSONObject*) lBridge->Produce();

        // 내보내고자 하는 필드들의 목록을 전달하고 싶으면 그것들을 브리지에 할당한다.
        // 그렇게 하지 않은 경우에는 기본 동작을 수행한다. 즉, 쿼리의 모든 필드들을 내보낸다
        if (AFields != NULL) {
            lBridge->FieldNames->Assign(AFields);
        }
        // 동일한 브리지를 재사용한다. 하지만, 이번에는 디테일 데이터셋을 할당한다
        lBridge->Dataset = ADetailDataset;
        // 이 경우 쿼리에 담겨 있는 모든 레코드들이 처리된다
        lBridge->Area = TJSONDataSetArea::All;
        TJSONArray* lJSONArray = new TJSONArray;
        // 디테일 배열을 임시 배열에 저장한다. 나중에 메인 오브젝트 안에 추가하기
        // 위해서다
        lJSONArray = (TJSONArray*) lBridge->Produce();
        // 임시 배열을 메인 오브젝트에 추가한다. 인자로 전달받은 프로퍼티 이름을
        // 반영한다
        lJSONObject->AddPair(APropertyName, lJSONArray);
        return lJSONObject;
    } __finally {
        lBridge->Free();
    }
}

```

```

    }
}

// 쿼리에 마스터 필드가 할당되어 있는 경우, TStringList를 반환한다. 그 안에는 모든 필드들이
// 담는데, 해당 마스터 필드만은 제외한다
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;
}

```



참고

실제 프로젝트에서는, 이 메서드들을 추상화해 다른 클래스/유닛 안에 넣어두는 것이 더 합리적이다. 왜냐하면, 그렇게 해야 재사용하기가 더 쉽기 때문이다. 하지만, 단순하게 제시하기 위해 동일한 DataModule 안에 그냥 두었다.

이 새 구현을 테스트하기

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



특정 고객의 매출들에 접근하기. 하위-리소스 방식을 사용한다

ID가 1004인 고객의 매출들만 필터링했다. 만약, 매출 하나를 접근/변경/삭제하려면, 역시 같은 URI를 접근하면 된다. 그저 맨 끝에 주문 ID를 붙이면 된다. 예: <http://localhost:8080/test/customers/1004/sales/V91E0210>.

이번에는 우리가 정의한 다른 엔드포인트를 접근해 보자: <http://localhost:8080/test/customers-details/1004>



하위-리소스를 가진 엔드포인트 접근하기 (고객과 그 고객의 매출들)

하나의 요청 안에서 우리는 해당 고객의 모든 매출들을 받아 온다. 즉, 호출을 두 번 RAD Server에게 보내는 것이 아니라, 단 한 번으로 우리에게 필요한 정보를 모두 얻는다. 물론 이러한 유형의 요청은 훨씬 더 복잡해질 수 있다. 중첩 수준이 여러 단계이기 때문이다.



참고

깃허브 리포지토리 데모 프로젝트들 중에서 이 장과 관련된 데모에서 여러분은 또다른 엔드포인트를 찾을 수 있다. 그 엔드포인트는 모든 고객들의 목록을 접근하고 각 고객들의 모든 매출 목록을 접근한다. 특별히 고객 하나만 필터링하는 것이 아니라, 모두 가져온다. 여기에서 알아둘 점이 있다. 코드 대부분이 재사용되었다. 그래서 추가 개발 시, 매우 간단하게 구현할 수 있다.

사용자 정의 GET, POST, PUT, DELETE 메서드를 생성하기

지금까지 사용자 정의 GET 메서드를 만드는 방법을 살펴보았다. 하지만 어떤 경우에는 POST, PUT, DELETE 등 다른 동사를 사용해야 할 수도 있다. 이런 종류의 구현을 코딩하려면, 메서드 이름을 해당 동사로 시작하면 된다. 예를 들면 이렇다: “**procedure PutMethodName(..**” 여러분은 사실 이전 예시들에서도 이미 보았다. 그 사용자 정의 메서드들은 모두 “Get”으로 시작되었었다. 예를 들어, 만약 이 부분을 “Post”라고 변경하면 우리는 POST 메서드를 정의하게 된다. 예시를 보자:

Delphi:

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

// 이것의 구현은 아래와 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;
  // 추가 비즈니스 로직을 추가한다
  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}";

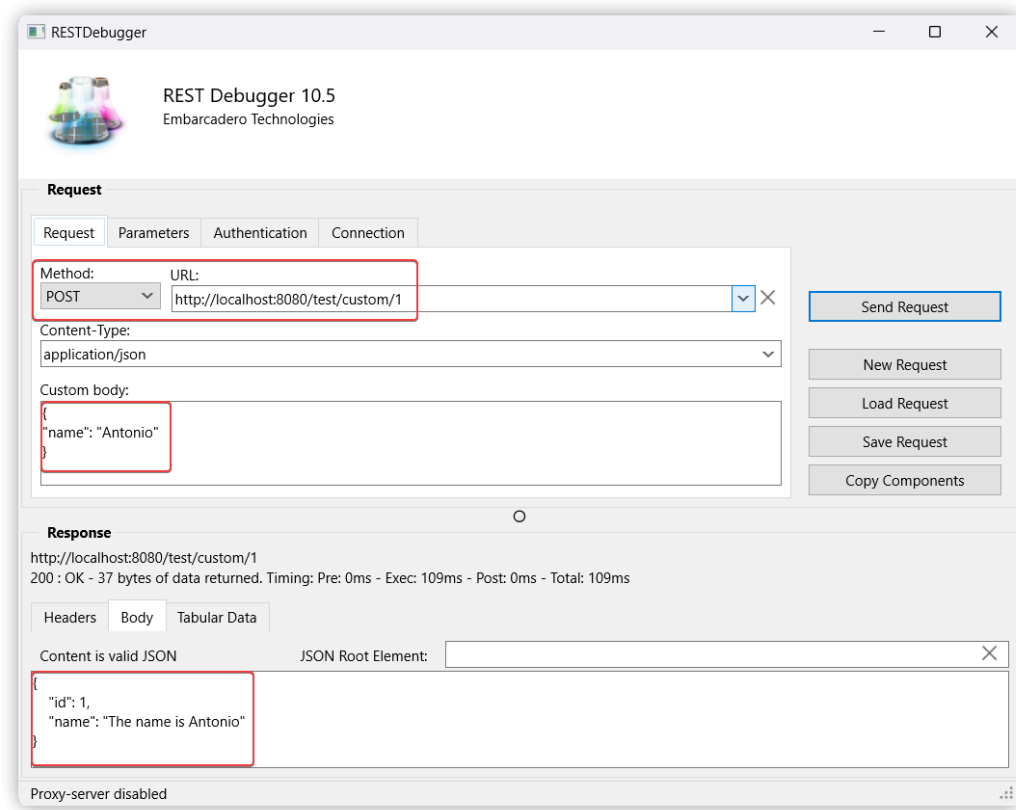
// 이것의 구현은 아래와 같다
```

```

void TTestResource1::PostCustomEndPoint(TEndPointContext* AContext, TEndPointRequest*
ARequest, TEndPointResponse* AResponse)
{
    TJsonObject *lJSON;
    System::UnicodeString lName;
    if (!ARequest->Body->TryGetObject(lJSON) && lJSON->TryGetValue("name", lName)) {
        AResponse->RaiseBadRequest("Bad Request", "Missing Data");
    }
    int lID = ARequest->Params->Values["ID"].ToInt();
    // 추가 비즈니스 로직을 추가한다
    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 우리는 새 추가 엔드포인트인 `./custom/{id}`를 제공하게 되었다. 이제 REST Debugger를 사용해 보자. POST 요청을 선택한다. 그리고 본문(body) 안에 반영할 정보를 넣는다. 이 예시에서는 “name” 프로퍼티를 추가하고 그 값에는 반영하고 싶은 이름을 명시한다. 요청을 보내면, 우리가 보낸 이름을 얻게 될 것이다.



사용자 정의 POST 메서드를 보내고 응답을 받는다

응답 오류들을 처리하기

조금 전에 만든 사용자 정의 POST 엔드 포인트 예시 코드를 본다면, 우리가 기대하는 데이터 “전체”를 얻지 못하는 경우에 오류를 일으키도록 해 놓은 것을 알 수 있다. RAD Server에는 가장 일반적인 오류들을 발생시키고 반환할 수 있는 기능이 내장되어 있다. 그래서 그런 오류들은 바로 `AResponse` 오브젝트에 반영될 수 있다. 어떤 오류들이 해당되는지에 대한 자세한 정보는 [이 링크](#)에서 확인할 수 있다.

참고 자료

- [JSON Writer들과 Reader들](#)
- [REST API 모범 사용 관행](#)

08

내장된 분석정보에 접근하기

RAD Server Console(콘솔)은 사전 구성된 웹 애플리케이션을 제공하는 서비스다. 이것은 RAD Server 엔진의 여러 데이터를 표시한다. 분석 정보도 표시한다. 그래서 RAD Server 인스턴스의 활동을 더욱 심층적으로 파악하고 실제 데이터를 기반으로 의사 결정을 할 수 있다. 사용자들, API들, 서비스들의 활동을 분석하면, 여러분의 애플리케이션이 어떻게 활용되고 있는지에 대한 통찰력을 얻을 수 있다.

주요 특징

RAD Server Console은 해당 데이터베이스 서버에 읽기 전용 모드로 접근한다.

- RAD Server 엔진 리소스들의 통계와 API 호출에 대한 피드백을 제공한다: 사용자, 그룹, 설치, 모듈 및 해당 리소스 통계와 정보
- 여러분은 이 콘솔 앱을 독립 실행형 애플리케이션으로 사용해 테스트 목적으로 활용할 수 있다. 또는 마이크로소프트 IIS 서버에 설정해 실제 운영(Production) 목적으로 사용할 수 있다.
- 알아둘 점: 마이크로소프트 IIS 서버는 Linux(리눅스)에서 사용할 수 없다. Linux 상의 실제 운영 환경에서, 여러분은 Apache (아파치)를 사용할 수 있다.
- RAD Server Console은 새 리소스들에 대한 분석 정보를 제공한다. 서버의 기능을 확장하면 된다.
- RAD Server Console은 RAD Server에 등록된 사용자들에 대한 분석을 제공한다.
- 여러분은 분석 데이터를 내보내기 할 수 있다. .csv 파일로 여러분의 시스템 안에 저장할 수 있다.

RAD Server Console에 접근하기

RAD Server Development Server로 돌아가서 Open Console 버튼을 클릭하라. 그러면 자동으로 RAD Server Development Console Server가 포트 8081에서 실행된다. 그리고, 브라우저가 열리고 분석 콘솔 로그인 창이 나타난다.

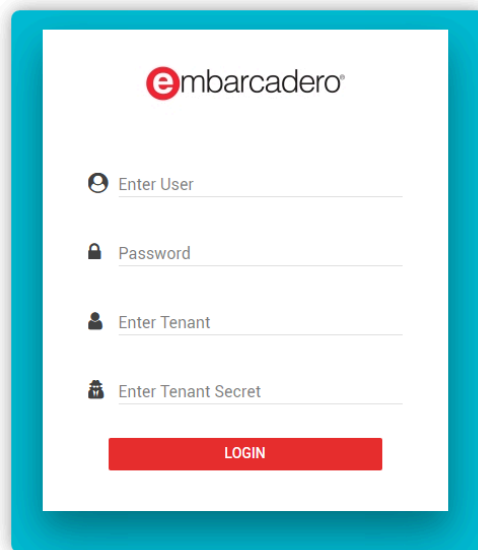


RAD Server Development Console (개발 콘솔) Server의 UI



참고

만약 기본 포트인 8081을 컴퓨터에서 사용 중이라면, 8081을 컴퓨터에서 사용 가능한 다른 포트로 변경하고 "Start"을 누른 후 "Open Browser"를 누르면 된다.

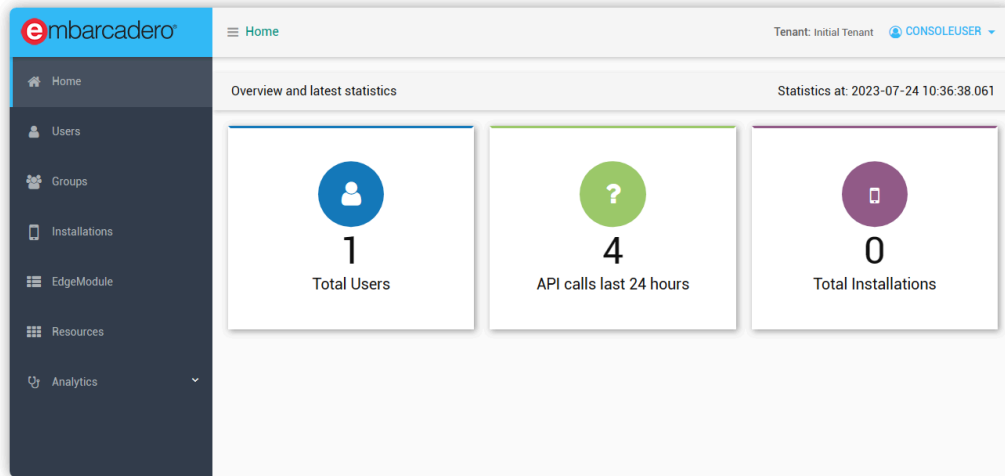


RAD Server Console의 사용자 로그인 화면

이 콘솔에 접근할 수 있도록, RAD Server에는 기본 자격 증명(credential)들이 미리 구성되어 있다. 그 인증 정보는 다음과 같다 (tenant 란은 그냥 비워 두기로 하자):

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

RAD Server는 기본으로 제공하는 사용자와 암호가 있다. 이것을 이용하면 콘솔에 접근할 수 있다. 이 자격 증명을 변경해야 한다는 점을 명심하자. `emsserver.ini` 구성 파일에서 변경하면 된다 (이 구성 파일에 대한 장에서 보다 자세한 내용을 확인할 수 있다).



RAD Server Console의 홈페이지

로그인하면 그래픽 화면이 나타난다. 그것은 RAD Server 콘솔 단일 페이지 JavaScript 앱의 화면이다. 메뉴가 왼쪽에 있고 내용은 오른쪽에 있다. 메뉴는 사용자, 그룹, 기기 설치, 엣지 모듈(EdgeModule), 리소스 모듈, 분석 등에 대한 정보를 제공한다. 아래 스크린샷은 사용자 목록과 그 사용자들의 정보를 보여주는 화면이다. 사용자별 생성 시점, 사용자 정보가 마지막으로 수정된 시점 등을 볼 수 있다.

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	3A25B7B0-1033-4B8B-A77E-EFB25B5B75CD

RAD Server Console의 사용자 표

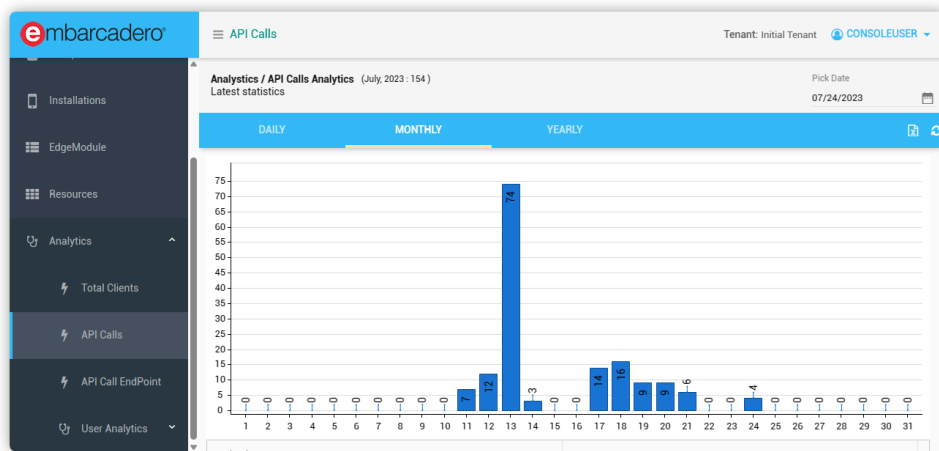
Analytics 메뉴 항목을 클릭하면 메뉴가 펼쳐진다. 그 안에서는 총 클라이언트 개수, API 호출, 호출된 API 엔드포인트, 등등을 선택할 수 있다. Analytics는 연, 월, 일 단위로 선택할 수 있다. 또한 통계 정보에 대한 필터링 기능이 있다. 그래서 사용자, 그룹 등 특정 엔드포인트들을 파악할 수 있다. 분석 결과를 .csv 파일로 저장할 수도 있다. 그래서 외부 애플리케이션에서 추가로 더 처리할 수도 있다.



팁

이 분석 통계는 의사 결정 과정 및 감사에 활용하기 좋은 정보를 제공한다. 여러분의 서비스들이 언제 더 많이 또는 더 적게 사용되는지 파악한 후에 업데이트를 작업 시간을 계획할 수 있다. 또는, 거의 사용되지 않는 엔드포인트가 무엇인지를 파악할 수 있다. 그 외에도 여러분은 매우 가치 있는 통찰력을 얻을 수 있다.

다음 차트는 선택한 월의 API 호출 통계를 보여준다.



RAD Server Console의 Ext JS가 제공하는 API 호출 분석 통계 페이지

09

RAD Server를 배포하기

지금까지, 첫 번째 RAD Server 애플리케이션을 테스트했다. 개발 버전 RAD Server (EMSDevServer.exe)와 개발 버전 콘솔 (EMSDevConsole.exe) 애플리케이션을 사용했다. 이 장은 다양한 플랫폼들을 다룬다. 여러분은 그 플랫폼들에 RAD Server를 배포해 실제 운영(production) 환경을 구성할 수 있다. 만약 RAD Server Lite (라이트)에 관심이 있다면, 다음 장으로 건너 뛰면 된다.



경고

새 bpl 또는 새 dcp 리소스가 컴파일 되었을 때, 그 결과물은 프로젝트의 “export” 폴더 (바이너리들이 일반적으로 들어가는 곳) 안에 담기지 않을 것이다. 이 리소스들은 기본적으로 여러분의 Embarcadero Studio 설치 경로 안에 생성된다:

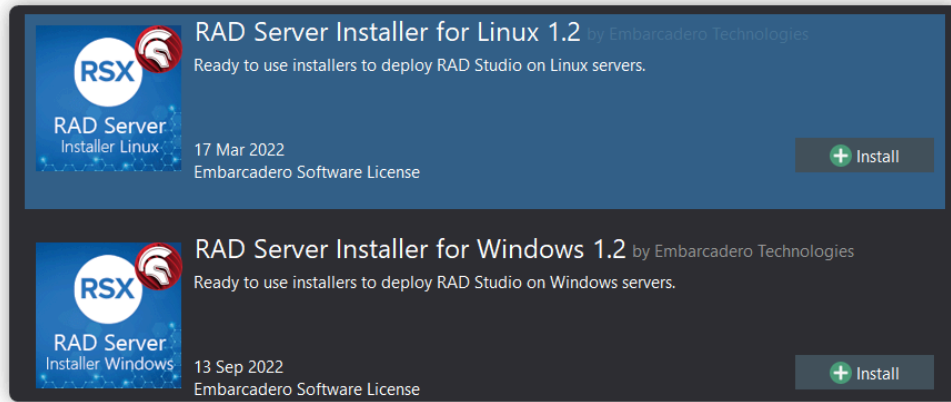
C:\Users\Public\Documents\Embarcadero\Studio\<XX>.0\“Bpl 또는 Dcp”
이 Bpl 또는 Dcp 폴더 안에는 플랫폼별로 폴더가 있다.

RAD Server는 어디에 배포될 수 있는가

RAD Server가 호환되는 플랫폼은 윈도우, 리눅스, 도커다. 개념적으로 말하자면, 각 플랫폼에 필요한 서비스들은 똑같다. 하지만, 몇 가지 차이점이 있다. 이 장에 우리는 그것들을 주의 깊게 살펴볼 것이다. 하지만, 그 전에 먼저, 유사점들 그리고 RAD Server의 내부 작동 방식에 대해 이야기 하겠다.

설치 프로그램을 GetIt(겟잇)에서 받아서 사용하기

여러분의 RAD Server 애플리케이션을 Windows 또는 Linux에 배포하는 경우, 가장 빠른 방법은 설치 프로그램을 사용하는 것이다. 그 파일은 GetIt(겟잇)에서 다운로드할 수 있다. 그저 “RAD Server”를 검색하면 이 두 개가 발견된다:



겟잇의 RAD Server 설치 프로그램

“설치”가 완료되면(이 경우에, 실제로는 설치 파일만 다운로드 된다), 설치 프로그램을 다음 경로에서 찾을 수 있다: C:\Users\<사용자명>\Documents\Embarcadero\Studio\<XX>.0\CatalogRepository. 전체 폴더를 복사해, 실제로 운영할 장비에 옮긴다. 이 폴더에는 RAD Server를 완전하게 설치하는데 필요한 모든 파일들이 포함되어 있다 (InterBase 설치 프로그램도 포함되어 있다).

설치 프로그램을 여러분의 실제 운영 환경에서 실행하기 전에, 반드시 IIS 또는 Apache를 설치해야 한다. 그래야 RAD Server 설치 프로그램이 필요한 모든 요구사항을 적절히 구성할 수 있다.

설치 프로그램은 여러분을 안내하며, 설치에 필요한 다양한 선택 사항들을 제시할 것이다.



참고

설치 중에, InterBase에 대한 유효한 라이선스를 제공하라는 메시지가 표시된다. 그러면 여러분의 EDN 계정과 RAD Server 일련번호(serial number)를 사용해 InterBase 인스턴스를 등록해야 한다.

RAD Server를 수작업으로 배포하기 위한 전제 조건들

이 장은 RAD Server를 배포하기 위해, 여러분이 설치 또는 구성해야 하는 모든 부분을 이해하는 데 집중한다. 비록 여러분이 설치 프로그램을 사용하는 경우라 하더라도, 모든 요구 사항을 이해하는 것이 중요하다. 그래야 디버깅과 문제 해결을 더 잘 할 수 있다. 또한, RAD Server를 더 새 버전으로 업데이트해야 하는 경우, 전체 애플리케이션을 다시 설치할 필요가 없다. 그저 몇 개의 dll 및 bpls/so만 업데이트하는 것으로 충분하다.

실제 운영 환경에서 작동하는 RAD Server 설치를 위한 필수 요구 사항은 아래와 같다:

- InterBase Server 엔진
- RAD Server 라이선스

- RAD Server 설치
- 웹 서버(IIS 7+ 또는 Apache 2.4+)
- (RAD Studio를 사용해 컴파일된) 리소스 파일들
- EMSServer.ini 파일 구성하기

여러분이 배포하기로 선택한 플랫폼과 관계없이, 이것들을 모두 설치/구성해야 한다. 예를 들어, 윈도우에서는 Microsoft의 웹 서버인 IIS 또는 윈도우용 Apache를 구성해야 한다. 그리고 Linux에서는 Apache가 필요하다. 꼭 이해해야 하는 중요한 점이 있다. RAD Server는 실행 파일이 아니다 (단, Lite 버전이라면 실행 파일이다, 더 자세한 설명은 뒤에서 한다). 리소스들은 컴파일 된 결과물들이다. 그 형태는 윈도우의 경우 BPL, 리눅스의 경우 so 라이브러리다. 그래서 웹 서버가 필요하다. 웹 서버를 통해서 이 리소스들에게 접근하기 때문이다.

InterBase 즉, RAD Server가 자체 내장 기능을 위해 사용하는 데이터베이스 인스턴스도 필요하다. RAD Server는 많은 정보들을 저장한다 (통계 정보, 사용자, 역할 등). 그러니, 이 모든 정보를 담는 자체 데이터베이스가 필요하다.

RAD Server가 내부용으로 InterBase를 사용한다는 사실 때문에 오해하지 않도록 분명히 할 점이 있다. 여러분의 데이터가 저장되는 데이터베이스는 InterBase가 아니어도 상관없다. FireDAC은 다양한 데이터베이스에 연결할 수 있다. 그러니 여러분은 여러분의 필요에 맞게 각자 원하는 데이터베이스를 선택하면 된다.



참고

만약 여러분의 데이터까지도 InterBase를 쓰기로 선택했고, 그 InterBase 데이터베이스도 동일한 컴퓨터에 배포한다면, 그 두 개의 인스턴스는 각각 서로 다른 포트에서 실행되어야 한다. 일반적인 관행을 따르면, 여러분의 데이터베이스 인스턴스는 포트 3050을 유지하고, RAD Server InterBase 인스턴스를 다른 포트에 설치한다(예: 3051 포트). 같은 인스턴스를 사용할 수는 없다. 왜냐하면 RAD Server는 자신만의 암호화 시스템을 사용하기 때문이다.

Windows에 수작업으로 배포하기

InterBase Server 엔진

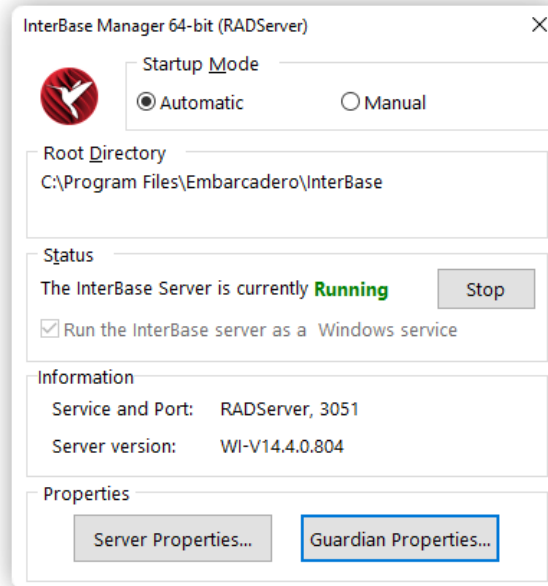
최신 윈도우용 InterBase 설치 프로그램을 <https://my.embarcadero.com> 에서 다운로드한다. 그리고 여러분의 실제 운영 시스템에 설치한다. 이전에 설치한 적이 없다면, [이 튜토리얼을 따르면 된다](#). 여기에서 [윈도우 실제 운영 환경을 위한 RAD Server 데이터베이스 요구 사항](#)도 확인할 수 있다.

설치를 위한 구체적인 세부 사항:

- “Server and Client”를 선택한다
- 여러 개의 InterBase 인스턴스가 동일한 컴퓨터에서 실행될 수 있도록 허용한다
- 제안된 기본 포트를 3051로 변경한다
- 인스턴스 이름을 RADserver로 변경한다 (기본 설정된 gds_db를 그냥 사용하지 않도록 변경한다)
- InterBase를 등록하려면, 여러분이 제공받은 RAD Server 일련번호와 똑같은 번호 그리고 여러분의 EDN 계정을 사용한다.

설치가 끝나면, 여러분은 InterBase 서버의 RADServer 인스턴스(앞에서 이 이름을 지정했다면)를 시작해야 한다. 시작 | 프로그램 | Embarcadero InterBase | 64-bit instance = RADServer | InterBase Server Manager를

선택한다. InterBase를 서비스로 실행(기본 설정임)하고 싶다면, 해당 확인란을 선택한다. 만약 컴퓨터가 시작될 때 InterBase가 실행되게 하고 싶다면, “Automatic” 라디오 버튼을 클릭한다. 그런 다음 Start 버튼을 클릭한다.



InterBase Manager 64비트 (RADServer용)



팁

InterBase Manager는, 컴퓨터의 프로그램 목록의 “Embarcadero InterBase” 아래에서 찾을 수 있다. 또는 설치한 경로 아래 “.bin\IBMgr.exe”폴더 아래에서 찾을 수 있다. 그리고 나서 여러분이 해당 인스턴스 이름을 명시하면 된다. 예: “.IBMgr.exe RADServer”. 또 다른 방법은 윈도우 검색 창에 “InterBase Server Manager”를 입력하는 것이다.

RAD Server 설치

RAD Server를 윈도우 시스템에 설치하기 위해서는, 우리가 개발용 컴퓨터를 구성했던 방식과 매우 비슷한 단계를 따라야 한다. 이 과정에 필요한 대부분의 파일들은 다음 폴더에서 찾을 수 있다:

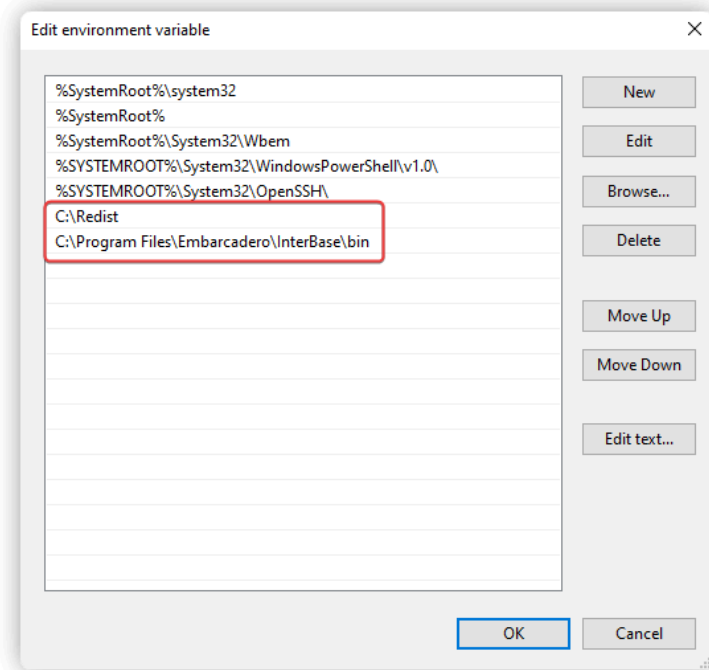
- C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\bin64
- C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\redist\win64

윈도우에 RAD Server를 설치하는 방법에 대해 매우 자세한 튜토리얼이 Docwiki에는 있다. [여기에서 접근할 수 있다](#). 그렇긴 하지만, 기본적인 단계들을 지금 설명하겠다:

다음 단계들을 따라 실전 운영 서버를 테스트할 준비를 하라. 그리고 우리가 만든 InterBase RAD Server 인스턴스와 EMSDevServer.exe를 사용해 RAD Server 데이터베이스를 생성하고 구성 파일을 생성하라.

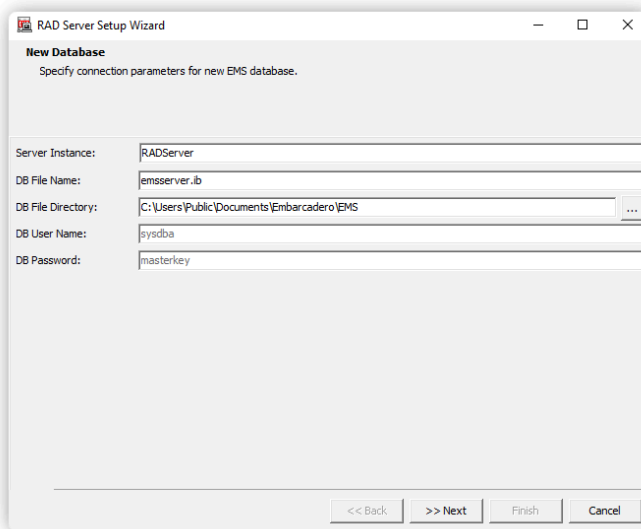
1. 64-비트 EMSDevServer.exe를 복사해 실전 운영 서버의 c:\installs\EMS 폴더 안에 넣는다.

- 필요한 파일들을 RAD Studio의 Redist/win64 폴더에서 복사해 실전 운영 서버의 c:\Redist 폴더 안에 넣는다.
- 실전 운영 서버에서 시스템 경로 환경 변수를 편집해, c:\Redist 와 c:\Program Files\InterBase\bin 폴더를 환경 변수에 추가한다.

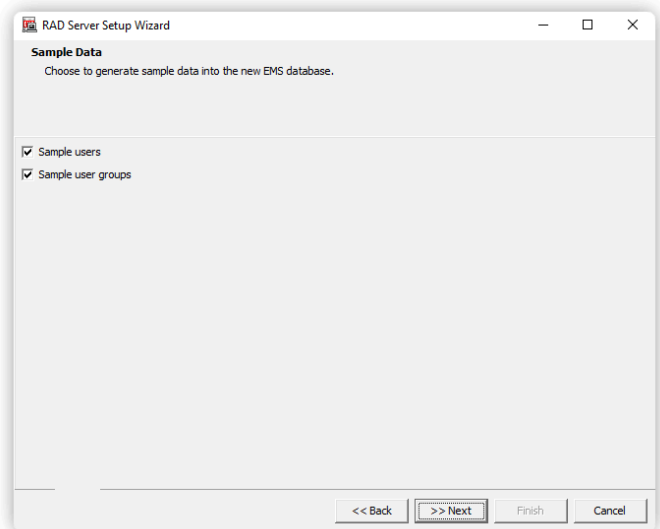


여러분의 시스템 경로에 두 개의 폴더를 추가한다

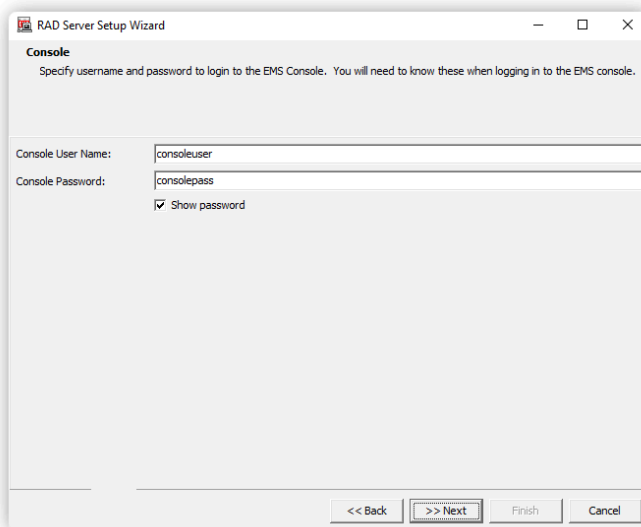
- 개발 컴퓨터에 있는 EMS 템플릿과 웹 리소스 파일들을 복사한다. 그 파일들은 C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\ObjRepos\en\EMS 안에 있다. 이것들을 복사해 실전 운영 서버의 폴더인 c:\installs\ObjRepos\EMS 안에 넣으면 된다 (EMSDevServer.EXE는 해당 템플릿과 웹 리소스 파일들을 찾을 때, 여러분이 놓아둔 EMSDevServer 폴더의 상위 폴더 아래의 ObjRepos\EMS 하위-폴더 안에서 찾는다).
- 실전 운영 서버에서 InterBase 서버가 RAD Server 라이선스를 가지고 시작되었는지 확인한다.
- EMSDevServer.exe를 실행한다 (여러분의 첫 RAD Server 개발 구성에서 했던 것과 똑같이 하면 된다). 그래서 실전 운영용으로 RAD Server 구성 파일과 InterBase인 RAD Server 데이터베이스를 설정한다. 다음 화면은 각 단계를 보여준다.



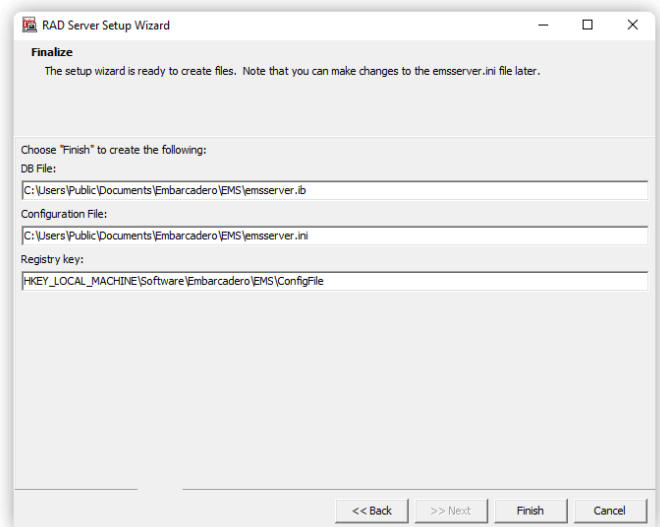
RAD Server Setup Wizard - RAD Server를 위한
연결 파라미터를 지정한다



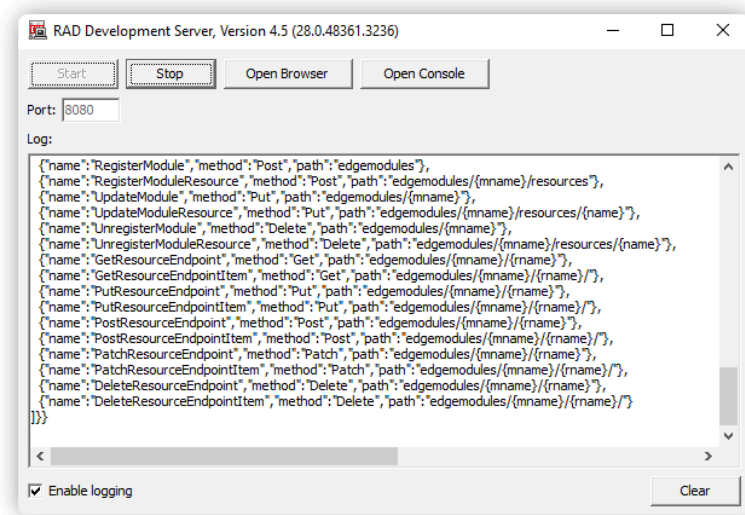
RAD Server Setup Wizard - 생성할 샘플 데이터를
선택한다



RAD Server Setup Wizard - RAD Server를 위한
사용자명과 비밀번호를 지정한다

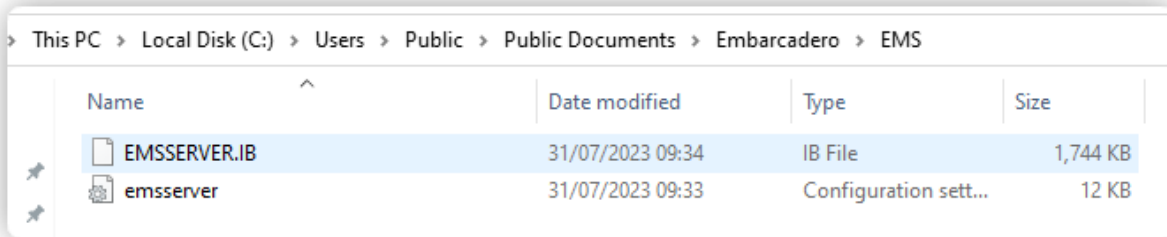


RAD Server Setup Wizard - 생성될 파일들과
레지스트리 키를 검토한다



RAD Server Development(개발) Server가 실행 중이다.

이 RAD Server 마법사는 공용문서 폴더 아래에 있는 기본 폴더 안에 두 개의 파일을 만든다.



RAD Server Setup Wizard - 파일 두 개를 공용문서 폴더 안에 생성한다

웹 서버 (IIS 또는 Apache)

윈도우에 배포하는 경우, 마이크로소프트가 윈도우와 함께 제공하는 웹 서버(일명 IIS)를 사용할 수 있다. 또는 윈도우용 Apache를 사용할 수도 있다. [이 링크](#)에 있는 상세한 안내에는 여러분이 실제 운영 환경에 복사해야 하는 파일들 뿐만 아니라 IIS 또는 Apache를 윈도우 장비 위에 구성하는 방법이 설명되어 있다.

만약 IIS를 선택한다면, 마이크로소프트에서 제공하는 버전들이 많으므로, 해당 서비스를 설치하는 절차가 조금씩 다를 수 있다. 이 서비스를 예전에 경험한 적이 없다면 [이 링크](#)에서 정보를 찾아보면 된다.

마지막 단계다. 우리가 만든 컴파일된 리소스들을 복사한다. 이 장의 맨 끝으로 가면, 어떻게 하는 지 알 수 있다.



팁

Server Manager에서 “add role or feature” 옵션을 사용해 웹 서버(IIS)를 추가하는 경우, 반드시 “ISAPI Extensions”와 “ISAPI Filters”를 “Application Development” 섹션 아래에 설치해야 한다.

리눅스에 수작업으로 배포하기

RAD Server와 애플리케이션을 리눅스 서버에 배포할 때는 다음과 같은 중 하나를 선택할 수 있다:

- 독립 실행형 RAD Server를 만들려면, RAD Server 설치 섹션을 참조하면 된다.
- Apache용 RAD Server를 만들려면, Apache용 RAD Server 설정 섹션을 참조하면 된다.

호환되는 배포판들

RAD Server는 공식적으로 Ubuntu(우분투) 18 이상과 RHEL 7 이상을 지원한다. 그렇다고 해서 RockyLinux, Debian 등 다른 배포판에 설치할 수 없다는 뜻은 아니다. 하지만, 내부 테스트는 항상 공식 지원되는 배포판만으로 수행된다.

InterBase Server 엔진을 설치하기

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

다운로드한 압축 파일을 풀고, 이 설치 프로그램에 실행 권한을 할당한다. 그리고 나서 실행한다.

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

설치에 대한 구체적인 세부 사항:

- “Server and Client”를 선택한다
- 동일한 컴퓨터에서 InterBase 인스턴스 여러 개가 실행되도록 허용한다
- 제안된 기본 포트를 3051로 변경한다
- 인스턴스 이름을 RADServer로 변경한다 (기본 이름은 gds_db 였을 것이다)
- 설치 폴더: /opt/interbase



팁

InterBase 설치 프로그램은 리눅스 설치에 데스크탑 환경이 있는지 없는지를 자동으로 감지한다. 설치 프로그램을 실행을 강제로 콘솔 모드로 하고 싶다면 아래 인자를 사용한다:

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



참고

인스턴스 이름과 경로를 원하는 이름으로 정의할 수 있다. 그러는 경우, 명심할 점이 있다. 구성을 진행할 때, 여러분이 정의한 이름을 참조하도록 해야 한다.

InterBase Server를 등록하고 시작하기

등록 마법사를 시작하려면 다음 명령을 실행한다:

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

그러면 라이선스 마법사가 시작된다. 콘솔 모드인 경우, 옵션 2 “Direct register”를 권장한다. 이 옵션은 여러분이 직접 RAD Server 일련번호와 EDN 계정을 명시할 수 있다. 나머지 작업은 이 마법사가 알아서 수행한다. 여러분이 명시한 라이선스도 알아서 엠바카데로 서버에 연결해 확인해 준다.

라이선스가 올바르게 적재된 상태인지 지금 확인하고 싶다면, 이전 메뉴에서 옵션 1 “List license”를 사용하면 된다. 그러면 모든 것이 제대로 진행되었는지를 확인할 수 있다.

InterBase 인스턴스는 이미 설치되어 있다. 라이선스도 적용된 상태다. 하지만, 아직 실행을 시작한 상태는 아니다. 우리가 시작을 시켜줘야 한다. 그러려면, InterBase 콘솔에 들어가야 한다. 이 명령을 사용하면 된다.

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

다른 애플리케이션들과 서비스들을 InterBase 데이터베이스에 연결하는 작업을 간소화하기 위한 가장 간단한 방법이 있다. InterBase 라이브러리로 가는 심볼릭 링크(symbolic link)를 하나 만들고, 그것이 /usr/lib를 가리키도록 하는 것이다. 그러면 InterBase 연결이 필요한 모든 서비스들 안에 이 lib를 복사해 넣지 않아도 된다.

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

InterBase를 서비스로 실행하기

InterBase를 서비스로 설정할 수도 있다. 그러면, 리눅스가 시작될 때 실행된다. 터미널 창에서 다음 명령을 사용한다.

InterBase를 설치한 경로의 “examples” 폴더에 접근해 ibserverd 스크립트 파일을 복사해 여러분이 설치한 서버 인스턴스용 버전을 만든다:

```
sudo cp ibserverd ibserverd_RADServer
```

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

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

두 번째 인자는 설치 폴더, 세 번째 인자는 인스턴스 이름이다. 이제, 시스템을 재시작하면 서비스가 자동으로 시작된다. 단, 라이선스가 제대로 적용된 경우에 그렇다.

다음 재부팅 또는 시작 시, InterBase가 서비스로 시작되도록 잘 설정되어 있는지 확인해 보자.

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

InterBase를 서비스로 실행하면, 컴퓨터가 다중 사용자 모드로 실행될 때마다 그 InterBase 서버는 항상 자동으로 시작된다.

서비스를 수작업으로 생성하기를 선호한다면 (또는 여러분의 Linux 배포가 조금 다른 방식을 사용한다면), 그 설정 방식에 대한 자세한 정보를 [이 링크](#)에서 찾을 수 있다.



참고

InterBase를 서비스에서 제거하기 위해서는 다음을 실행한다:

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

RAD Server를 설치하기

RAD Studio가 설치된 컴퓨터의 경로에는, RAD Server 리눅스 설치 프로그램이 들어 있다:

C:\Program Files (x86)\Embarcadero\Studio\<XX>.0\EMSServer

그 파일들을 복사해 Linux 시스템에 넣고, 그 설치 프로그램을 실행한다. 실행 권한을 부여해야 할 수 있다.



경고

libcurl이 설치되어 있는지 확인하라. 그것을 설치하려면 배포 패키지 매니저를 사용한다. 예:

Debian 기반인 경우: apt install libcurl4

설치 셸 스크립트는 /usr/lib/ems 폴더를 만든다. EMSDevServerCommand, EMSDevConsoleCommand, 그리고 이 명령 파일들을 실행하기 위해 필요한 여러 런타임 라이브러리(.so) 파일들이 그 디렉토리 안에 들어간다. Docwiki에 튜토리얼이 있다. [이 링크](#)로 가면 자세한 정보를 볼 수 있다.

일단 설치가 완료되면, EMSDevConsole을 설정 모드에서 실행하라:

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

start를 입력하고 엔터 키를 누른다.

연결 파라미터들을 명시한다. 다음 값을 입력하면 된다:

- Server instance: 기본 인스턴스 이름은 **RADServer**
- DB File Name: 기본 이름은 **emsserver.ib**

- DB File Directory: **/user/lib/ems**
- DB User Name: 기본 파라미터는 **sysdba**
- DB Password: 기본 파라미터 **masterkey**
- Console User Name: 기본값은 **consoleuser**
- Console Password: 기본값은 **consolepass**

구성 옵션이 올바른 경우 “n”을 입력한다. emsserver.ini와 emsserver.ib 파일이 생성된다. 그리고 RAD Server가 실행하기 시작한다. 사용 포트는 8080이다. 구성 파일은 언제든지 수작업으로 편집할 수 있다.

구성 절차가 완료되면, RADServer 데이터베이스가 **/usr/lib/ems** 안에, 구성 파일이 **/etc/ems** 안에 생긴다.

DevServer가 실행되면 그대로 둔다. 이제 테스트를 하자. 접근을 잘 하고, 응답을 잘 받는지 브라우저에서 접속해 보자: <http://<리눅스장비IP>:8080/version>



브라우저에서 version 엔드포인트를 호출한 출력 결과를 볼 수 있다

EMSDevServerCommand와 EMSDevConsoleCommand는 Linux RAD Server 애플리케이션을 개발하고 테스트하는 데 사용할 수 있다. 이 경우, Apache를 사용하지 않아도 된다. 이제 Linux와 Apache 상에서 실전 운영 모드로 RAD Server와 Delphi/C++로 컴파일된 애플리케이션 모듈들을 설정하고 테스트할 차례다.



팁

리눅스에 RAD Server를 배포하면서, 여러분의 데이터를 담은 데이터베이스도 역시 InterBase를 사용하고 싶다면, [이 튜토리얼](#)을 따르면 된다.

Apache용으로 RAD Server를 설정하기

InterBase의 iSQL 명령(/opt/interbase/bin 디렉토리 안에 있음)을 사용해 RAD Server가 emsserver.ib 데이터베이스 파일에 연결할 수 있는지를 확인하라.

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


Apache HTTP 서버를 구성해, Apache RAD Server(libmod_emsserver.so)과 Apache RAD Server Console(libmod_emsconsole.so) 모듈을 적재한다. Apache의 구성은 여러분이 무슨 리눅스 배포판을 사용하든 매우 유사하다. 하지만 RHEL과 Debian 기반 배포판 사이에는 몇 가지 차이점이 있다는 점에 유의하라.



참고

여러분이 사용 중인 리눅스 배포판의 문서집을 확인하라. 그래서 모듈을 불러오거나 위치 태그를 정의할 때 사용할 권장 방법을 확인하라.

다음 줄을 추가한다. 그래서 RAD Server Apache 서버 모듈(libmod_emsserver.so)과 RAD Server Apache 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 태그들을 추가한다. 그래서 컨테이너를 만든다. 주어진 URL에 대한 접근 제어 규칙들을 지정하는 곳이다.

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

여러분의 RAD Server가 올바르게 실행되고 있는지 테스트하자. 브라우저를 사용하여 RAD Server 버전 번호를 불러와 보면 된다. 접근할 주소: <http://<리눅스장비IP>/radserver/version>

이제 마지막 단계 즉 우리가 컴파일한 리소스들을 복사하는 단계다. 이 장의 맨 끝에서 그 방법을 확인할 수 있다.

Docker(도커)에 배포하기

RAD Server를 Docker(도커)에 배포하기는 윈도우나 리눅스를 사용하는 것보다 훨씬 간단하다. Embarcadero는 Docker용 다양한 이미지들을 dockerhub에 제공하고 있다.

여러분은 RAD Server와 관련된 2개의 이미지를 찾을 수 있을 것이다: 이 두 이미지 사이에 차이점은 오직 하나다. 즉, InterBase 서버 엔진이 컨테이너 내부에서 실행되는 이미지가 있고, 또 다른 하나는 RAD Server를 실행하는 데 필요한 그 InterBase 서버를 여러분이 다른 곳에서 호스팅한다고 가정하고 작성된 이미지다.



팁

InterBase 서버는 Docker와도 호환된다. 그리고 엠바카데로는 이를 컨테이너화할 수 있도록 이미지를 제공한다. [DockerHub에 대한 링크](#)를 참고.

이 도커 이미지들은 완전히 오픈 소스로 빌드되었다. 그리고 GitHub에서 공개적으로 제공한다. 이것은 불과 한가지 접근 방식에 지나지 않는다. 만약 여러분이 Docker를 편하게 사용할 줄 안다면, 이 이미지들을 템플릿으로 사용해 여러분의 고유한 요구 사항들에 맞추어 자유롭게 조정하기 바란다.

이 이미지들을 배포하고 사용자 정의하는 방법에 대한 정보는 매우 많다. 아래의 DockerHub 및 GitHub 링크 참조.

옵션 1: PA-RADServer-IB

“모든 기능이 포함된” 이미지라고 할 수 있다. InterBase가 내장되어 있으므로 작업이 훨씬 더 쉽다. 하지만, 이 컨테이너를 처음 실행할 때 여러분은 분리 모드로 실행할 수 없다는 점을 유의하라. 첫 번째 마법사를 실행해 RAD Server 라이선스와 추가 세부 사항들을 구성해야 한다. 모든 설정을 완료한 후에는, 분리 모드로 실행할 수 있다.

명심해야 할 게 또 하나 있다. 만약 애플리케이션을 확장할 계획이 없고 모든 것을 한 곳에 두고 작업하고 싶은 경우라면 이 컨테이너가 매우 편리하다. 하지만 향후 애플리케이션을 확장하려는 경우에 가장 좋은 방법은 RAD Server를 InterBase 서버에서 분리해 별도의 컨테이너들/머신들에 설치하는 것이다.

[DockerHub\(도커허브\) 링크](#)

[GitHub\(깃허브\) 링크](#)

이 이미지 안에는 들어 있는 것들:

- InterBase 서버 엔진
- PAServer
- RADServer 필수 파일들
- Apache 사전 구성

옵션 2: PA-RADServer

이 컨테이너는 유효한 RAD Server 라이선스가 설치된 InterBase 서버에 연결해야만 작동한다. 그렇지 않으면 작동하지 않는다. 이 컨테이너가 이상적인 경우는 바로 여러분이 애플리케이션을 확장해 여러 인스턴스를 배포하고 그것들을 동일한 InterBase 서버에 연결하고 싶을 때이다.

[DockerHub\(도커허브\) 링크](#)

[GitHub\(깃허브\) 링크](#)

이 이미지 안에는 들어 있는 것들:

- PAServer
- RADServer 필수 파일
- Apache 사전 구성



팁

비교적 간단한 환경인 경우, PAServer를 사용하면 여러분의 리소스 업데이트를 이 컨테이너에 바로 업로드 할 수 있다. 그래서 컨테이너를 다시 생성할 필요 없다.

[다음 링크](#)에 접속하면, Docker에 RAD Server를 배포하는 방법에 대한 자세한 내용을 확인할 수 있다.

RAD Studio에서 컴파일된 RAD Server 모듈들을 복사하기

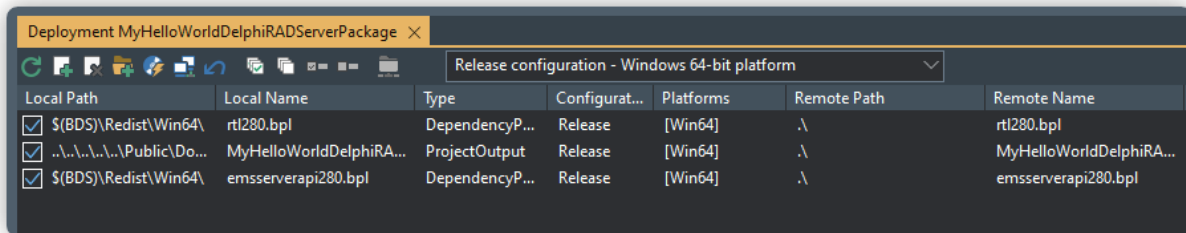
모듈 또는 필요한 추가 라이브러리들을 여러분의 실전 운영 장비에 배포하는 절차는 여러분이 어느 운영체제를 선택하든 관계없이 거의 똑같다. 자체 리소스들인 경우, .bpl/.so 파일들을 실전 운영 시스템으로 복사하기만 하면 된다.

RAD Server 애플리케이션 패키지 파일들은 프로젝트 설정에서 지정한 폴더 안에 컴파일된다. Delphi 패키지 출력 및 C++ 최종 출력의 기본 디렉토리는 다음과 같다:

- Delphi의 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 Server 애플리케이션 패키지 파일은 .\\$(Platform)\\$(Config) 폴더에 컴파일된다.

필요한 RAD Server 애플리케이션과 런타임 DLL 파일들을 실전 운영 서버에 배포하는 방법은 여러가지다. 가장 흔한 세 가지 전송 방법은 다음과 같다:

- 패키지 파일들을 복사해 RAD Server가 설치된 실전 운영 서버 경로에 넣는다
- FTP를 통해 그 파일들을 실전 운영 서버로 전송한다
- Platform Assistant (PAServer)를 사용한다. 이때 “Project | Deployment” 메뉴 항목을 이용해 IDE가 그 파일들을 실전 운영 서버로 옮기도록 지정한다. 이 스크린샷은 윈도우 64의 예를 보여준다.



Project | Deployment 파일들을 PAServer가 전송할 수 있다

컴파일된 RAD Server 확장 패키지 파일들(예: 1장 MyHelloWorldDelphiRADServerPackage의 프로젝트)을 복사해 실전 운영 RADServer 폴더에 넣는다.



팁

PAServer를 사용하는 경우, 파일이 배포되는 기본 경로를 변경할 수 있다. 실전 운영 시스템에서 `paserver.config` 파일을 편집하면 된다. PAServer를 실행할 때 더 높은 권한이 필요할 수 있다. 이는 파일들을 쓰기(write)할 경로에 의해 달라진다.

EMSServer.ini 파일을 구성하기

새 리소스를 실전 운영 폴더에 추가하기까지를 완료했다. 이제 새 리소스를 사용할 수 있다는 것을 EMSServer.ini 파일 안에 명시해야 한다.

emsserver.ini 파일을 편집하면 된다. 각 RAD Server 확장 패키지들을 [Server.Packages] 구역 아래에 추가한다.

Windows

```
[Server.Packages]
;# 이 구역은 확장 패키지들을 위한 구역이다.
;# 확장 패키지들은 사용자 정의 리소스 엔드포인트들을 등록하는 데 사용된다.
;c:\mypackages\basicextensions.bpl=mypackage description
c:\inetpub\wwwroot\RADServer\MyFirstDelphiRADServerPackage.bpl=First Windows Test Demo
```

Linux

```
[Server.Packages]
;# 이 구역은 확장 패키지들을 위한 구역이다.
;# 확장 패키지들은 사용자 정의 리소스 엔드포인트들을 등록하는 데 사용된다.
;c:\mypackages\basicextensions.bpl=mypackage description
/usr/lib/ems/bplMyFirstDelphiRADServerPackage.so=First Linux Test Demo
```

Docker

이미 실행 중인 인스턴스의 emsserver.ini 파일을 구성하려면 `./config.sh` 스크립트를 실행한다. 이 스크립트는 Apache를 자동으로 재시작한다.

10

RAD Server Lite(라이트)

Lite 버전이란?

RAD Server는 백엔드 데이터베이스가 필요하다. 그 데이터베이스는 InterBase 기반이며, 주로 웹 서버 DLL 모듈로 배포된다. IIS용과 Apache용이 일반적이다. 이런 이유때문에, 표준 배포 시 다음 사항들이 필요하다:

- 웹 서버(IIS 또는 Apache)를 준비하고, 그 웹 서버에 RAD Server의 모듈(DLL 등)을 설정한다
- RAD Server를 배포하고 구성한다
- InterBase를 설치한다. 이때 특별히 제공된 RAD Server 라이선스를 사용해 등록한다 (이 라이선스는 사용자가 대상 장비에 등록해야 활성화된다)

개발용으로, 우리는 오래 전부터 독립-실행형 RAD Server 버전을 제공해 왔다. 이 버전은 Indy HTTP 서버를 기반으로 한다. 그래서, 성능이 제한적이다. 하지만 배포가 훨씬 더 쉽고, 디버거 하에서 실행할 수 있다 (따라서 여러분은 RAD Server 모듈들의 코드를 디버깅할 수 있다). 이 개발용 버전은 배포를 위한 것이 아니며, 배포용 라이선스를 제공하지도 않는다. 이 버전의 경우, 만들 수 있는 사용자 수가 제한된다. 또한 로컬 InterBase Developer(개발자) 에디션을 활용한다 (이 버전을 위한 라이선스는 RAD Studio 라이선스 안에 들어 있다).

RAD Server Lite(**RSLite**)는 배포 모델이 더 간단하다. 그래서 테스트 서버 그리고 처리량이 크게 필요하지 않은 경우에 사용하기 좋다. 그리고 RSLite는 InterBase 임베디드 데이터베이스 엔진 즉 IBToGo를 사용하도록 되어 있다. 모든 기능을 갖춘 InterBase 서버가 아니므로, RSLite의 단순화된 라이선스 부여 모델과 잘 맞는다.

RSLite는 RAD Server Development(개발) 에디션에 있는 것과 똑같은 바이너리(RAD Studio와 함께 제공됨)를 사용한다. 그리고 IBToGo 바이너리와 라이선스 슬립 파일을 사용하므로 여러분의 솔루션과 함께 배포하면 된다. (배포 대상 컴퓨터에서 등록할 필요가 없다). RSLite는 임베디드 데이터베이스를 사용하고, Indy HTTP Server 컴포넌트를 사용한다. 그렇기 때문에, 완전한 정식 RAD Server 설치 시 처리할 수 있는 초당 요청을 감당하지는 못한다. 게다가 RAD Server 여러 개를 앞단에 두는 수평 확장을 할 수 없다.

RSLite가 기반으로 사용하는 아키텍처는 확장성이 훨씬 더 제한적이다. 하지만, 간단한 배포 수준의 규모라면 충분하다고 보고 있다. 명심할 점이 있다. 처리량을 결정하는 요인에는 RAD Server 모듈이 실행하는 코드도 해당된다.



공개되는 시스템에 배포하는 경우, RSLite HTTP 서버를 직접 노출하지 말 것을 권장한다. 그렇게 하지 말고, 프록시 구성을 통해 접근하도록 하면, 여러분은 앞단에 여전히 웹 서버(예: Apache 또는 IIS)를 가지고 있게 되고, 그것들은 들어오는 HTTPS 호출들에 대한 보안 컨텍스트를 제공한다. 그리고 그 요청들을 RSLite로 전달한다.

RAD Server Lite 라이선스를 얻는 방법

RAD Studio (Delphi 및 C++Builder 포함)의 엔터프라이즈 또는 아키텍트 라이선스가 있으면, RAD Server Lite 라이선스를 받을 수 있다. [이 페이지](#)를 방문해 안내를 따르면 된다.



등록 키와 EDN 계정이 필요하다

이 절차를 따르면, RSLite용 라이선스 키를 직접 받는 것이 아니라 슬립 파일(.TXT 파일 안에 저장된 라이선스)를 받게 된다. 따라서 여러분이 설치할 때 함께 배포할 수 있다. 이 라이선스는 설치 횟수에 제한이 없다. 하지만, 여러분은 동일한 컴퓨터에서 두 개의 RAD Server Lite인스턴스를 실행할 수는 없다.



이 라이선스 파일은 특정 하위 폴더 안에 배치해야 한다. 라이선스 확보 사이트에 있는 일반적인 정보만 읽으면 오해할 수도 있으니 주의하자

RAD Server Lite 프로젝트를 배포하기

일단 여러분의 프로젝트를 배포하기 전에 해당 라이선스를 확보했으니, 이제 두 가지 서로 다른 고려 사항이 있다:

- 첫째, 여러분은 배포 구성을 만들어야 한다. 거기에는 RSLite, 필요한 런타임 패키지들, IBToGo 배포가 들어가야 한다 ([그 단계들은 여기에 있다](#)).

- 둘째, 여러분은 데이터베이스 파일을 실전 운영용으로 만들어야 한다. 그리고 IBToGo 라이선스와 호환되는 것이어야 한다. RAD Server Developer(개발자) 에디션에서 만든 로컬 데이터베이스는 호환되지 않는다.

배포할 파일들

수작업으로 배포

실제로 배포할 때, RSLite 솔루션 하나를 배포하는 데 필요한 파일은 다음과 같다 (여기에 더해, 여러분이 만든 애플리케이션 패키지들과 그것들에 종속되는 것들을 배포하면 된다):

1. RSLite 실행 파일(개발자 에디션과 똑같은 것임): **EMSDevServer.exe** 파일, 이것은 RAD Studio의 bin 폴더(또는 해당 64-비트 버전 폴더) 안에 있다.
2. 필수 RAD Studio 런타임 패키지들 (최소 설치에 필요한 것들은 아래에 나열되어 있다. 이것들은 RAD Studio의 redist 폴더 아래 win32 또는 win64 안에 있다) 또한 여기에 더해 여러분의 RAD Server 모듈들 안에 있는 코드들이 사용하는 기타 런타임 패키지들을 배포하면 된다:
 - bindengine<XX>0.bpl
 - dbRTL<XX>0.bpl
 - emsclientfiredac<XX>0.bpl
 - emsserverapi<XX>0.bpl
 - FireDAC<XX>0.bpl
 - FireDACCommon<XX>0.bpl
 - FireDACCommonDriver<XX>0.bpl
 - FireDACIBDriver<XX>0.bpl
 - rtl<XX>0.bpl
 - vcl<XX>0.bpl
 - vclDb<XX>0.bpl
 - vclFireDAC<XX>0.bpl
 - vclimg<XX>0.bpl
 - vclwinx<XX>0.bpl
 - vclx<XX>0.bpl
 - Xmlrtl<XX>0.bpl
3. InterBase ToGo 배포 파일들: 공용문서 아래 InterBase redist 폴더(예: C:\Users\Public Documents\Embarcadero\Interbase redist\InterBase2020)의 하위 폴더인 win32_togo 또는 win64_togo 안에 있다 - Linux의 경우 **libtogo.so** 파일은 알맞은 InterBase redist 폴더에 있다.
4. 위에서 얻은 라이선스 파일을 interbase/license 폴더에 추가한다(IBToGo redist 구성의 일부다).

배포 마법사를 사용하기

Deployment Wizard의 RSLite 기능을 사용해 여러분의 파일들을 배포한다. 다음 단계를 따르면 된다:

1. RSLite 기능을 추가한다.
2. 그런 다음, IBToGo 기능을 추가한다.

3. **iblite** 등록 파일을 “선택 취소”한다.
4. How to get a RAD Server Lite License 구역 안에서, 배포할 파일을 추가한다: **rsLite** 활성화 파일을 생성하고 대상을 "interbase/license"로 설정한다.
5. 그런 다음, Creating the Production Database 구역에서 얻은 파일을 추가한다. 그래서 **emsserver.ini**이 ./에 배포되도록 한다.
6. 마지막으로, Creating the Production Database 구역에서 얻은 파일을 추가한다. 그래서 **emsserver.ib**이 ./에 배포되도록 한다.

MSVS 런타임

IBToGo를 실행하려면(그래서 IBToGo를 사용하는 RSLite가 실행될 수 있게 하려면), RSLite를 실행할 윈도우 컴퓨터에 비주얼 C++ 2013 런타임 라이브러리가 설치되어 있어야 한다. RAD Studio가 설치된 개발자 컴퓨터에는, 이미 설치되어 있을 가능성이 높다. 그러나 일반적인 배포 대상 컴퓨터인 경우라면, 여러분이 설치할 필요가 있다. [마이크로소프트](#)에서 다운로드해 설치하면 된다.

실전 운영 데이터베이스를 생성하기

이렇게 구성을 해 놓았으면, 여러분은 RSLite를 시작할 수 있다. **EMSDevServer.exe** 애플리케이션을 실행하면 된다. 알아둘 점이 있다. 대상 시스템에 InterBase 클라이언트가 있다면, 그것이 더 높은 우선 순위가 된다. 그리고 그 InterBase 클라이언트가 Developer(개발자) 에디션이라면, 즉 RAD Studio와 함께 제공된 것이라면, 모든 것이 순조롭게 진행된다. 하지만 Lite가 아니라 표준 RAD Server Developer(개발자) 구성으로 작동한다.

그것을 알려면, RAD Server가 시작될 때 남기는 로그의 첫 몇 줄을 보면 된다. “RSLite” 구성이라면, 이렇다:

```
{ "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" } }
```

위 코드에 “Lite”가 false로 되어 있다면, 여러분은 수작업으로 **gds32.dll**(또는 해당 64-비트 버전) InterBase 클라이언트 라이브러리가 적재되지 않도록 해야 한다. 그 라이브러리는 일반적으로 C:\Windows\SysWOW64 안에 있다 (만약 이 InterBase 클라이언트 라이브러리를 찾지 못하게 되면, 로컬에 있는 **ibToGo.dll**이 적재된다).

이제, RSLite를 (이 알맞은 구성으로) 시작한다. 만약 **emsserver.ini** 파일이 없고, **emsserver.ib** 데이터베이스 파일이 없다면, 하나 만들라고 메시지가 나온다. 그러려면, RSLite는 반드시 해당 구성 정보를 RAD Studio의 오브젝트 리포지토리(ObjRepos) 폴더 안에서 찾아야 한다. 이를 보다 쉽게 하려면, 프로그램 파일 (x86)\Embarcadero\Studio\<XX>.0\ObjRepos\en\ems에 있는 파일들을 복사해 **emsdevserver.exe**를 기준으로 상대 경로인 "../ObjRepos/EMS" 폴더에 넣으면 된다. 즉, ObjRepos 폴더는 RSLite 설치 폴더, 즉 프로젝트 배포 디렉토리와 동일한 수준에 있어야 한다.

**참고**

이 폴더는 RSLite를 배포할 때마다 필요한 것은 아니다. 오직 한 번만 실전 운영 데이터베이스를 생성하면 된다. 그 후에는 대상 컴퓨터에 그대로 복사하면 된다. 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

RSLite를 공개 웹 애플리케이션으로 직접 노출하는 것은 권장하지 않는다. RSLite는 보안 및 암호화 측면에서 제한이 있기 때문이다. 따라서, 프록시 계층을 사용하는 것을 권장한다. 그래서 전용 서비스 즉 널리 사용되는 웹 서비스들 중 하나를 앞 단에 놓고 사용하는 것이 좋다. 예를 들어, Apache 안에 가상 호스트를 구성하고 HTTPS를 활성화한 다음, 들어오는 트래픽을 리디렉트 해 RSLite 인스턴스로 보내도록 한다. 그 구성은 다음과 같다:

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

리눅스용 구성

리눅스의 경우, 위와 비슷한 단계들을 따르면 모든 것이 기대한 대로 작동한다. 다른 대안도 있다. 여러분은 완전한 정식 RAD Server를 설치하고, 그 다음에 IBToGo를 설치에 추가해도 된다:

- RAD Server를 설치한다. **ems_install.sh**를 사용하면 된다. 그 파일은 RAD 설치 폴더 안에 있다. [여기를 참조](#)
- IBToGo 파일을 InterBase의 “redist” 폴더에서 복사한다. 그리고 리눅스의 EMS 폴더(/usr/lib/ems)에 넣는다.
- EMSDevServerCommand를 실행한다. 그리고 마법사를 따라 EMS 데이터베이스와 구성 파일을 만든다.

**참고**

애플리케이션을 실행하려면 `sudo` 를 통해야 할 수도 있다. 알맞은 권한이 필요할 수 있기 때문이다.

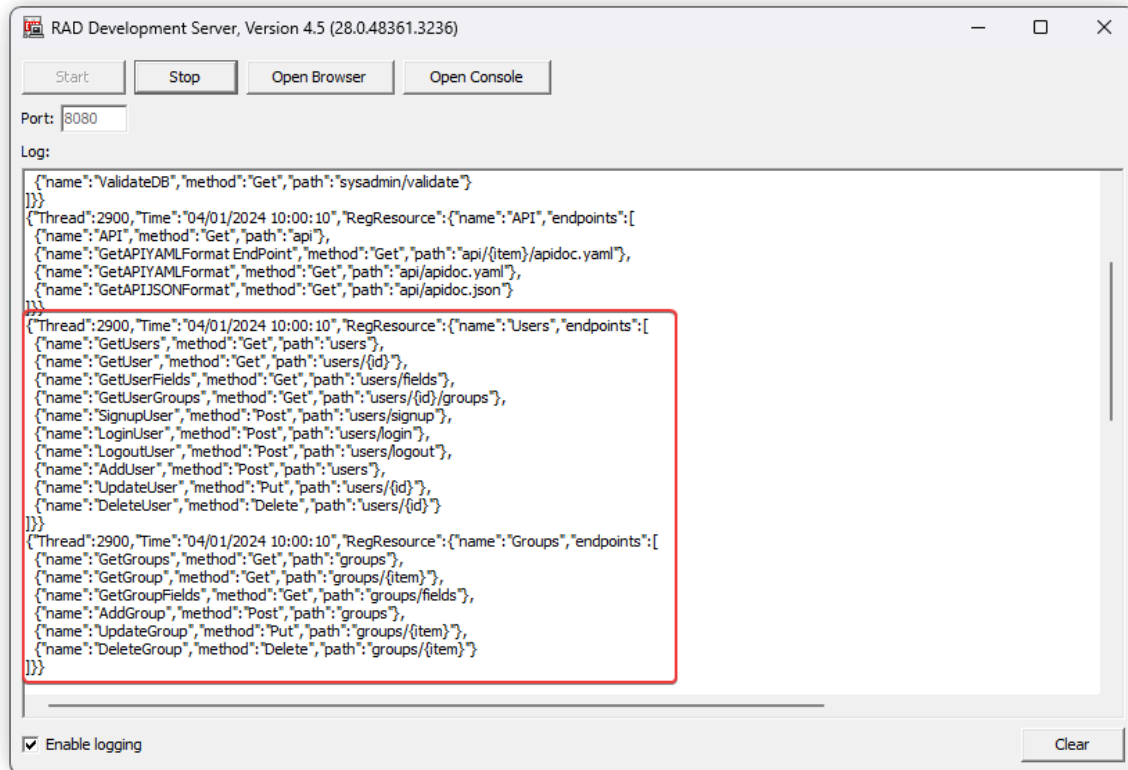
11

인증 및 권한부여

RAD Server 안에는 사용자 인증(authentication)과 권한부여(authorization) 기능이 들어있다. 또한 이를 관리할 수 있는 여러 엔드포인트들이 제공된다. 관리를 위해 접근할 때는 그 엔드포인트들에 요청(request)을 보내는 방식을 사용할 수도 있고, 또한 RAD Studio를 사용해 개발된 RAD Server 기반 앱들 즉 RAD Server 리소스들에서 프로그램 방식으로 접근할 수도 있다. 프로그램 방식으로 접근할 수 있다는 점은 훌륭한 장점이다. 예를 들어, 여러분의 자체 인증 시스템에 통합하기를 원한다면, RAD Server에서 제공하는 인증 서비스를 덮어쓸 수도 있다.

내장된 인증: 사용자 및 그룹 관리하기

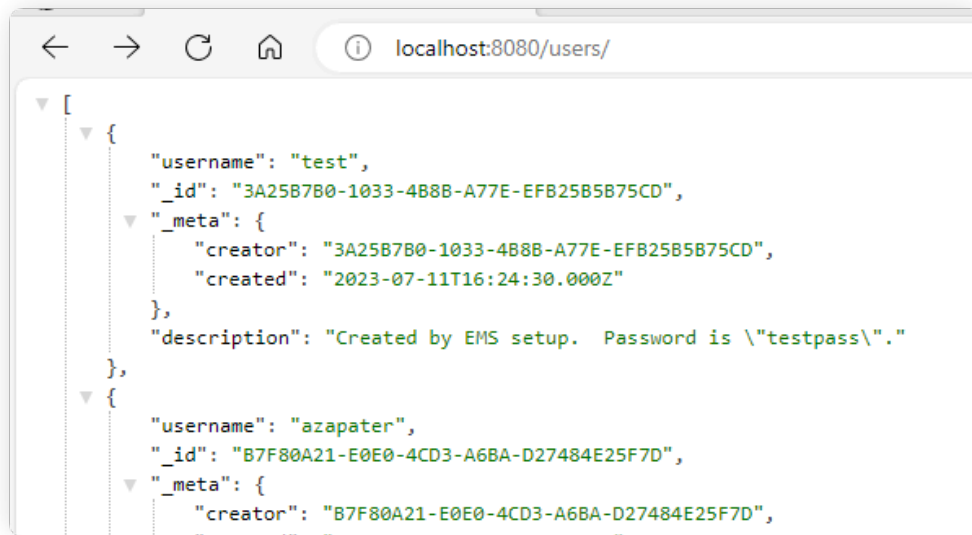
RAD Server 인스턴스를 여러분의 개발 장비 안에서 실행한 후에, 우리는 그 로그를 분석한다면, 몇 가지 중요한 엔드포인트들을 볼 수 있다:



RAD Server가 생성하는 기본(default) 엔드포인트들에서 사용자와 그룹을 관리한다

인증과 관련된 리소스는 이 두 가지다: [Users](#)와 [Groups](#). RAD Server를 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 엔드포인트들 그리고/또는 리소스들에 대한 접근을 세부적으로 허용 또는 금지할 수 있다.

users/ 엔드포인트에 접근해보자. 그리고 결과를 보자. 아래와 같다.



RAD Server 안에 생성되어 있는 사용자들의 목록에 접근하기

RAD Server가 배열 하나를 반환해주었다. 그 배열 안에는 RAD Server의 데이터베이스 안에 생성되어 있는 모든 사용자들이 있다. 위 스크린샷을 보면, RAD Server에는 test 라는 사용자가 "test pass"라는 비밀번호를 가지고 있음을 알 수 있다(이 test 사용자와 그 비밀번호는 우리가 설치 마법사를 진행하는 동안 자동으로 생성된다).



참고

모든 리소스들에 대해 보다 자세히 보고 싶다면, 그 문서집을 참조할 수 있다. OpenAPI json/yaml apidoc 파일 규격으로 정의되어 있다.

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



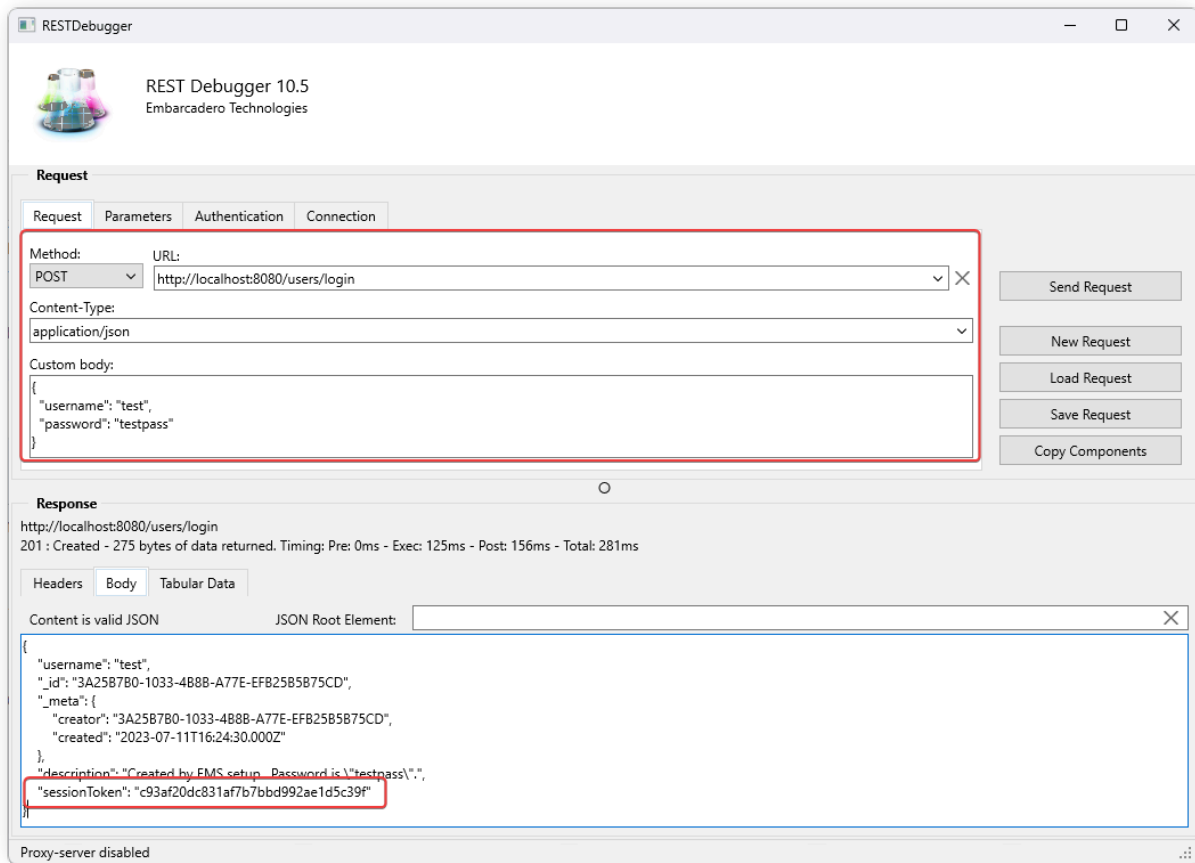
팁

Users와 Groups 안에 사용자 정의 필드들을 추가하는 것도 가능하다. 방법은 간단하다. 무슨 필드이든 body(본문) 안에 그것을 추가해 보내면 된다. 그러면, 추가 필드로 저장된다.

로그인 하기

RAD Server 안에 사용자가 생성되면, 그 사용자는 RAD Server에 로그인할 수 있다. 로그인에 성공하면, 토큰 하나를 받는다. 그 다음 요청부터는, 그 토큰을 사용해 자신의 신원을 증명한다.

자동으로 생성되는 인증 관련 엔드포인트들을 사용하는 방식을 봤는데, 로그인도 방식이 같다. users/login 엔드포인트에게 POST 요청을 보내면 된다. 단, username과 password를 body 안에 넣어서 보내야 한다.



POST 방식으로 users/login 엔드포인트에 접근한다. 그러면 세션 토큰 하나를 획득한다

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

X-Embarcadero-Session-Token=<값>

로그아웃 하기

세션 토큰의 유효기간은 무한한 것이 기본 설정이다. 하지만, 이는 보안 면에서, 가장 좋은 설정이라고 할 수 없다. 다음 설정들을 EMSServer.ini 파일 안에서 사용할 수 있다. 그래서, 토큰의 유효일시를 세밀하게 지정할 수 있다. 몇 가지 파라미터들로 그 구성을 지정할 수 있다:

SessionInactivityTimeout=

;**# Set SessionInactivityTimeout=60** 으로 설정, 그러면 이 세션의 활동이 없어도 유효하게 유지되는 최대 시간이 초 단위로 지정됨.

;**#** 이 명령은 세션 토큰에 반영된다. 기본값은 0 (시간 만료 없음).

SessionLiveTimeout=

;**# Set SessionLiveTimeout=60** 으로 설정, 그러면 이 세션이 살아있는 최대 시간이 초 단위로 지정됨.

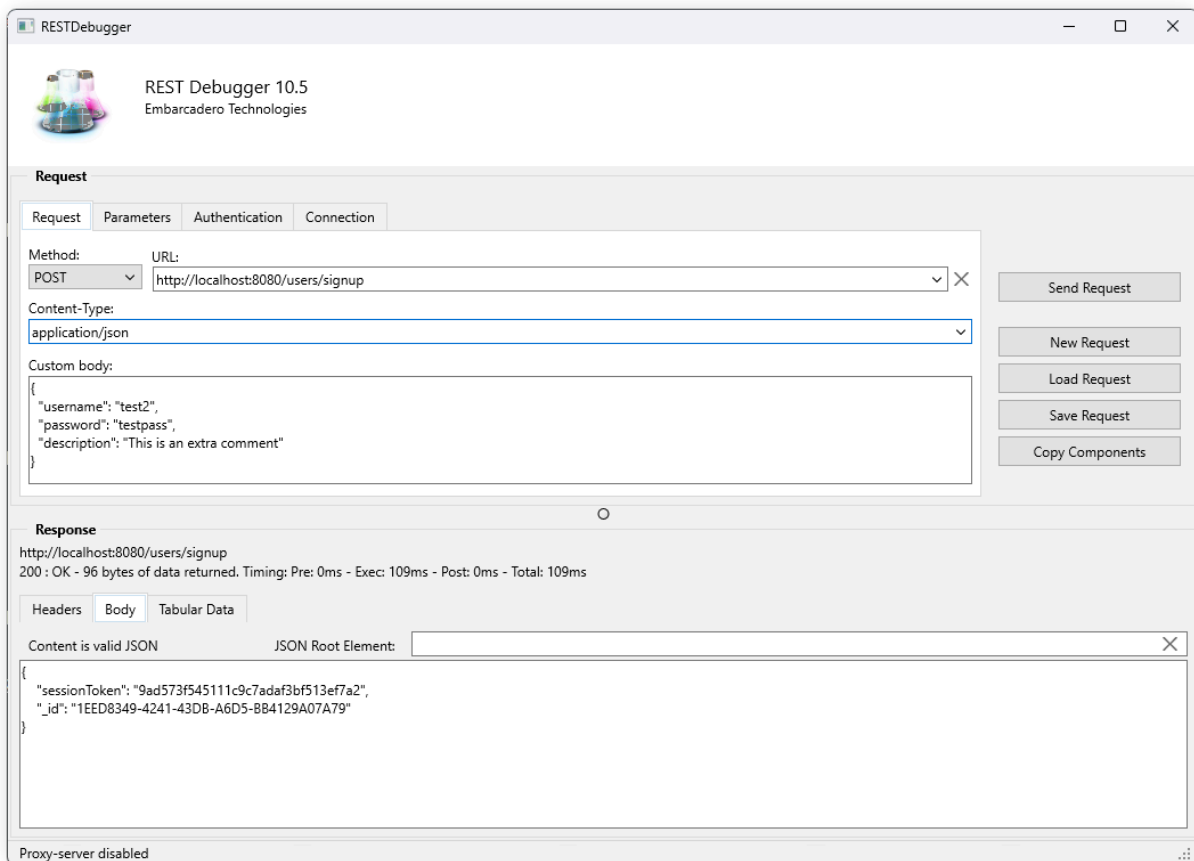
;**#** 이 명령은 세션 토큰에 반영된다. 기본값은 0 (시간 만료 없음).

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, 해당 사용자의 토큰을 헤더 안에 담아야 한다).

회원가입 하기

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



POST 방식으로 users/signup 엔드포인트에 접근한다. 그러면 새 사용자가 생성된다

위 예시에서, 중요하게 눈여겨 볼 것이 있다. 요청(request) 안에는 사용자 정의 필드를 얼마든지 많이 추가해서 보낼 수 있다. 요청의 body 안에 넣으면 된다 (예를 들어, 위 화면에서는 'description' 필드를 추가하고 있다).

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



팁

공개적으로 회원가입(sign-up)을 받고 싶지 않다면? `users.signup` 엔드포인트에 대한 접근을 제한하는 것을 잊지 말자. 즉 그 엔드포인트에는 오직 권한이 있는 사용자와 그룹만 접근을 허용하도록 `roles(역할)` 구역에 지정하면 된다. 자세한 사항은 [authorization 구역](#) 참조.

그룹(Group)을 관리하기

그룹(group)을 관리하는 것도 가능하다. 이것 역시 내장된 엔드포인트들을 사용하면 된다. 이 리소스 역시 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"
  ]
}
```



팁

비록 여러분이 역할 관리를 그룹 수준에 정의할 계획이 없어도, 사용자들을 그룹들에 할당하는 것을 권장한다. 그렇게 하면, 분석(analytics)이 보다 쓸모있다. 왜냐하면, 그룹을 기준으로 상세한 정보를 보여주는 것이 가능하기 때문이다.

내장된 권한부여(Integrated authorization)

전역 자격 증명(Global Credentials)

전역 자격 증명(global credential)을 정의해 여러 가지 보안 계층을 활용하는 것이 가능하다. RAD Server 엔진의 구성 파일(emsserver.ini 파일) 안에는 이와 관련된 파라미터가 세 개 있다. 그것들은 `[Server.Keys]` 구역 안에 있다:

MasterSecret

여러분이 그 RAD Server의 데이터베이스 안에 저장된 모든 데이터에 완전하게 접근할 수 있도록 권한을 제공한다.

이 RAD Server MasterSecret 키를 관리자 작업에 사용하라. 이 키를 사용하면, 그 RAD Server의 엔진 (EMS Server) 안에 있는 모든 RAD Server 리소스들을 접근할 수 있다.

AppSecret

여러분이 허용된 해당 엔드포인트들에게 RAD Server Client Application이 접근할 수 있도록 권한을 제공한다.

ApplicationID

Application ID를 사용하면 RAD Server-기반 클라이언트들로부터 온 요청(request)을 식별할 수 있다. 따라서 오직 유효한 애플리케이션 ID를 가진 요청만 처리된다. 즉 유효하지 않은 요청들은 모두 거부된다. 이 식별자 (ApplicationID)는 여러분이 운영하는 서로 다른 여러 RAD Server 인스턴스들을 차별화 하는데 사용될 수도 있다.

보다 자세한 정보는 [docwiki](#)에 있다.

사용자 그리고 그룹에 권한부여하기

가장 쉽게 RAD Server의 엔드포인트들의 보안을 지키는 길은 EMSServer.ini 파일을 사용하는 것이다. 모든 규칙들은 반드시 [Server.Authorization] 구역 아래에 정의되어야 한다. 아래 예시를 보자. ini 파일 안에 기본으로 들어 있는 내용이다:

```
[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).
;# 기본값에 의하면, 모든 리소스들은 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 group에든 속해있기만 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

```

이 파일은 스스로를 매우 잘 설명하는 주석이 들어있다. 하지만, 몇 가지 예시를 더 살펴 보자.

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

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

```

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

```

위 예에서, 우리는 'Users' 리소스 전체에 대해 외부 접근을 제한하고 있다. 하지만, 'LoginUser'와 'SignupUser' 엔드 포인트만 꼭 짚어서 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 역할이 무엇이든 상관없다. 하지만, 반드시 RAD Server에 사용자로 등록되어 있어야만 한다.

사용자 정의 인증 (Custom authentication)

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

우리가 가장 먼저 고려해야 할 핵심 전제는, 사용자가 입력한 자격 증명이 올바른지 여부를 우리의 인증 서비스를 통해 확인해야 한다는 점이다. 일단 검증이 되었다면, RAD Server의 내장 인증 시스템에서도 그 사용자를 인증해야 한다(처음 접속한 사용자인 경우에는 RAD Server 안에 사용자 계정을 생성하는 것까지도 한다). 그리고 나서, RAD Server 토큰을 반환한다.

RAD Server의 내부 API를 우리의 코드에서 접근하려면, 해당 API의 인스턴스 하나를 새로 만들면 된다. 인스턴스를 생성할 때는 해당 Context를 사용하도록 명시해야 한다. 예시를 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 경우에)
        // Users/Signup 엔드포인트 호출이 실제로 수행하는 내부 메서드를 호출 (프로세스 안에서)
        var lUserFields := TJSONObject.Create;
        lUserFields.AddPair('ExternalUserGUID', lExternalUserGUID);
        lUserFields.AddPair('comment', '이 사용자 추가: RAD Server CustomLoginUser가
        생성함');
```

```

    lResponse := lEMSAPI.SignupUser(lUserName, GenerateHashedPassword(lUserName),
lUserFields);
end
else
    // Users/Login 엔드포인트 호출이 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;
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(!(ARequest->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_IEMSResourceResponseContent 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);
        }
    }
}

```

RAD Server가 ‘이 요청은 인증된 사용자로부터 온 것’임을 알 수 있도록, 외부 인증 서비스가 자격 증명을 검증해주면 우리는 그저 RAD Server에서 해당 사용자를 가입(sign up)시키거나 로그인(login)시키면 된다.

그런데 눈치챘을텐데, RAD Server는 사용자 생성 시 외부 서비스의 비밀번호를 그대로 사용하지 않는다. 왜냐하면 사용자가 외부 인증 서비스에서 비밀번호를 변경했을 때, RAD Server의 비밀번호도 함께 업데이트해야 유지해야 하는 방식은 관리하기가 복잡하기 때문이다. 따라서 이렇게 사용자 이름을 해시해서 RAD Server의 비밀번호로 사용하는 방식은 오로지 외부 인증 서비스만 믿고 인증을 처리하면서도 RAD Server 내부에서는 사용자 계정을 안전한 방식으로 유지할 수 있다. 자세한 구현은 해당 샘플 프로젝트를 참고하면 된다.



참고

사용자 정의 로그인과 AD 통합 예시들은 RAD Studio의 Samples 폴더 안에 있다:

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

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

RAD Server 안에 모든 사용자를 반드시 개별적으로 생성해야만 할까? 기술적으로는 꼭 그럴 필요가 없다. 하지만, 강력히 권장한다. 주요 이유는 분석(analytics)과 로그 기록(logging) 때문이다. RAD Server 안에 각 사용자가 개별적으로 있으면, 내장되어 있는 분석 기능을 활용할 수 있다. 또한 각 사용자별 세밀한 분석 데이터를 개별적으로 얻을 수 있다. 기술적으로는 RAD Server에 “일반” 사용자라는 계정만 하나 만들고, 그 사용자를 재사용해 모든 로그인 처리를 하고, 토큰 제공도 그 계정으로 제공하는 방식이 가능하다. 하지만, 그렇게 하면, 분석 기능의 활용도가 크게 떨어진다는 점을 분명히 알아야 한다.

위 예시는, 내부 RAD Server API를 사용해 프로그램 방식으로 사용자를 자동으로 생성하거나 로그인시키는 방법을 보여준다. 하지만 그 두 가지는 그저 API들의 능력을 활용하는 여러 가지 방식 중 일부에 불과하다. 그룹 생성하기, 사용자를 그룹에 할당하기...등등 우리가 앞에서 말했던 REST API를 통해 할 수 있는 거의 모든 것들을 우리는 프로그램 방식으로 할 수 있다.



팁

RAD Server의 내부 API 전체는 EMS.Services 유닛 안에 있다. 그 코드 또는 [문서집](#)을 살펴보기 바란다. 제공되는 모든 메서드들을 볼 수 있다.

사용자 정의 권한부여 (Custom authorization)

또 다른 선택이 있다. 이 엔드포인트들에 대한 보안을 프로그램 방식으로 지킬 수 있다. 여러분은 각 엔드포인트에 특정 규칙들을 정의할 수 있다. 그래서 사용자 그리고/또는 그룹의 접근을 허용하거나 제한할 수 있다.

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

Delphi

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

C++

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

필수 인증 정보 검증을 마치고 난 후에, 만약 그 요청이 접근하려는 메서드에 대해 접근을 제한하고 싶다면, 방법은 간단하다. unauthorized 오류를 반환하도록 구현하면 된다. 이 때는 AResponse 파라미터를 사용한다.

Delphi

```
AResponse.RaiseUnauthorized('허가받지 않은 접근', '');
```

C++

```
AResponse->RaiseUnauthorized("허가받지 않은 접근", "");
```

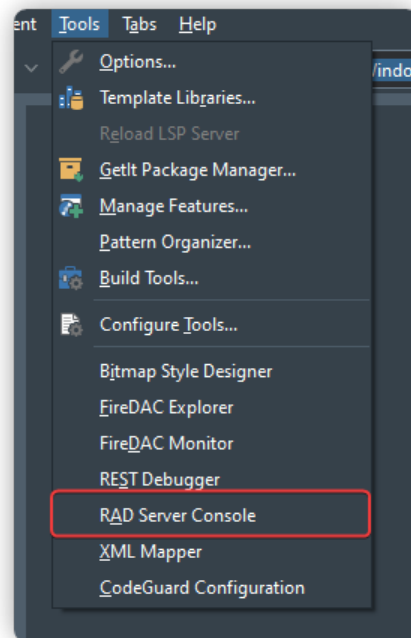
RAD Server Management Console(관리 콘솔)

RAD Server Console(줄여서, RSConsole)은 이미 오래 전부터 있었다, 하지만, 그 경로가 좀 숨겨져 있었다:

32 bits: C:\Program Files (x86)\Embarcadero\Studio\XX.0\bin

64 bits: C:\Program Files (x86)\Embarcadero\Studio\XX.0\bin64

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



메뉴 항목을 통해 RAD Server Console(콘솔)에 접근하기

새 프로파일을 생성하기

이 콘솔을 사용하면, 로컬에 있는 개발자 환경에 연결하는 것은 물론이고 원격에 있는 실제 운영 서버에도 연결할 수 있다. 콘솔에는 여러 프로파일들을 미리 정의해 놓을 수 있다. 따라서 필요할 때 해당 서버로 바로 연결할 수 있다.

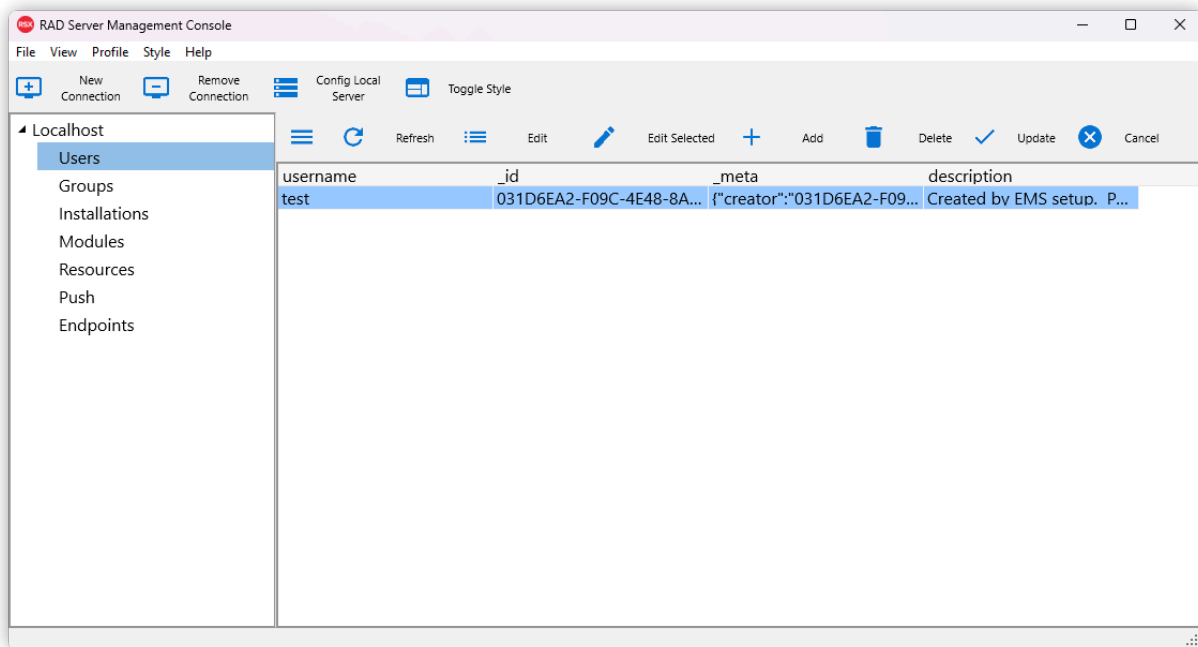
새 profile/connection을 생성하는 방법은 간단하다. “New Connection”을 누르면 된다. 또는 콘솔의 메뉴에서 “Profile/New Profile...”를 사용하면 된다. 그 구성은 꽤 명료하다. 그저 연결할 호스트에 대한 기본 세부사항만 있으면 된다.



RAD Server 인스턴스에 연결할 수 없는 경우는 다음과 같다. RAD Server Development가 실행되고 있지 않는 경우, 또는 운영 환경이라면, Windows에서는 EMSDevServer.exe가 Linux에서는 EMSDevServerCommand가 실행되고 있지 않는 경우다.

사용자들과 그룹들을 관리하기

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



생성된 사용자들의 목록을 RAD Server 콘솔을 이용해 접근한다

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 name: test

Groups: ☒ testgroup

Field Name	Field Value
description	Created by EMS setup. Password is "testpa..."
extraInfo	This is just a custom value

OK Cancel

“Modify User” 창



사용자를 생성할 때는 *password* 필드가 보인다. 하지만, 사용자를 편집할 때는 그 필드에 접근하지 못한다. 사용자의 비밀번호를 바꾸는 건 간단하다. 사용자 정의 필드들을 넣는 구역 안에서 “password” 필드를 추가하고, 그것의 값에 새 비밀번호를 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 Server 콘솔은 실제로 RAD Server에 연결하는 클라이언트다. 따라서, 여러분은 RAD Server 콘솔의 동작이 어떻게 구현되었는지에 대해 관심이 있을 것이다. 이것은 FMX 앱이다. 그리고 이 RAD Server 콘솔의 소스 코드 전체는 RAD Studio와 함께 제공된다. 그 경로는 여기서:

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

비록, 꽤 복잡한 애플리케이션이지만, RAD Server에 내장된 API들을 원격에서 접근하는 모든 가능성들 그리고 사용자가 어떤 동작(action)을 할 때 어떤 종류의 요청이 서버로 전송되는지를 볼 수 있는 완벽한 곳이다.

12

여러분의 엔드포인트들에 대한 문서화와 테스트를 위해 OpenAPI (Swagger) 사용하기

.....

OpenAPI/Swagger란 무엇인가 그리고 우리는 왜 이것을 사용하는가?

예전에 Swagger(스웨거)라고 알려졌던 규격이 있다. 현재 이름은 [OpenAPI](#)다. 이것은 여러분의 API를 설명하는 도움말 문서집을 동적으로 생성한다. 그 결과물은 JSON/YAML 형식이다. 그리고, 이것을 사용하면, 개발자들은 자신들이 구축한 API를 Swagger 문서 형태로 자동 생성할 수 있다. Swagger의 생태계는 API 문서화 도구들 중에서 가장 크다. Swagger 기능이 활성화되어 있는 API들은 상호작용하는 도움말 문서집, 클라이언트 SDK 자동 생성, API 탐색 능력 등을 제공한다.

RAD Studio를 사용하면, 개발자들이 RAD Server를 문서화할 수 있다. 그 문서집은 Swagger 규격을 활용한다. 그러기 위해, 애트리뷰트를 사용한다. 예를 들어 Summary, Parameters, Details/Responses 등이 있다.



참고

아마 C++Builder를 사용하는 개발자라면 이미 알겠지만, 델파이와 달리, C++ 언어는 애트리뷰트를 지원하지 않는다. 하지만, 이미 앞에 있는 장들에서 봤듯이, 이 기능을 C++Builder이 가질 수 있도록 우회해 실현하는 방법이 있다. 이 방식을 사용하면, C++ 언어로도 작성한 API들 역시 여러분이 문서화할 수 있다.

Swagger UI를 RAD Server 안에 심어 넣기(embedding)

RAD Server는 정적인 파일들을 제공할 수 있다 (예: 웹사이트의 앞단을 호스팅할 수 있다). 그런데, 이 능력은 RAD Server 안에서 Swagger UI를 직접 호스팅하는데도 활용되고 있다.

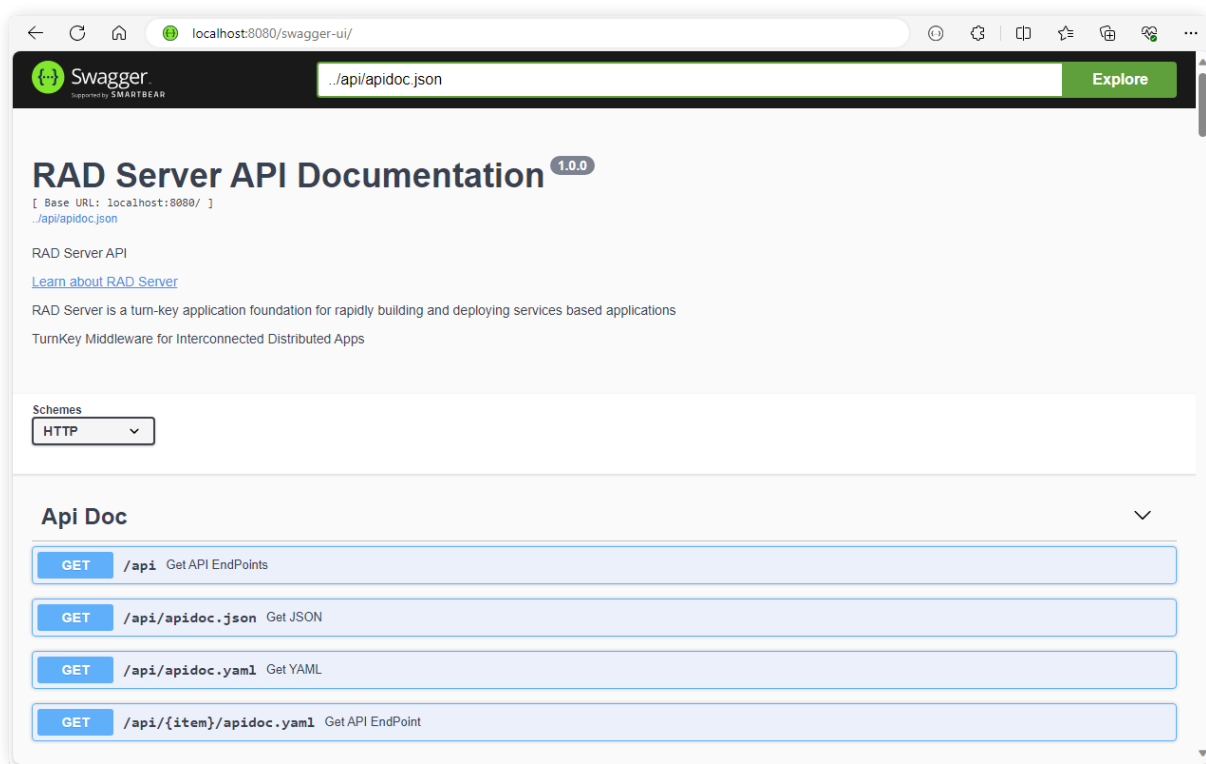
그 방법은 간단하다. EMSServer.ini 파일에 접근해서, 키 중에 [Server.PublicPaths]를 찾는다. 그 아래를 보면, Swagger UI 경로를 가리키는 줄이 있다. 그 줄의 주석을 푼다. 그리고, 올바른 경로를 명시한다. 그러면 된다.

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"}
```

일단 이렇게 정의하고 나면, RAD Server를 시작하라. 그리고 [URL]/swagger-ui/를 접속하라. 예:

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



Swagger 메인 페이지의 모습. apidoc.json 정의를 사용하고 있다

위 화면에서 우리는 Swagger UI가 자동으로 JSON 규격을 가리키고 있는 것을 볼 수 있다. 우리는 위 요청을 변경해 /api/apidoc.yaml 을 호출할 수도 있다. 그러면, yaml 규격이 나타날 것이다.

또한, RAD Server 안에 내장되어 있는 모든 엔드포인트들은 이미 완전하게 문서화 되어 있다. 즉, 해당 규격 파일들이 이미 만들어져들어 있다.



팁

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

사용자 정의 문서집을 만들기

예시

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

Delphi:

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

C++:

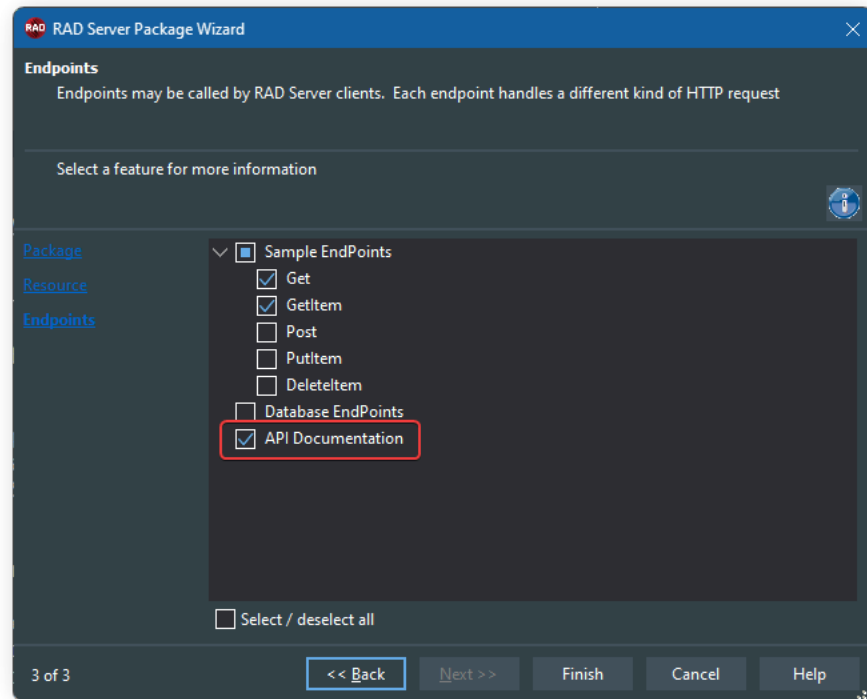
C:\Users\Public\Documents\Embarcadero\Studio\XX.0\Samples\CPP\Database\EMS\APIDocAttributes



참고

이 장에서 제공하는 예시 프로젝트는 RAD Studio와 함께 제공되는 예시를 업데이트한 버전이다. 새로 추가된 여러줄-문자열 리터럴 기능을 활용하기 때문에, 이 프로젝트는 12 아테네 또는 그 이상의 버전에서만 작동한다.

또 다른 선택이 있다. 여러분은 새 RAD Server를 생성할 때, 해당 마법사 안에 있는 API Documentation 체크박스를 선택할 수 있다. 그러면, 문서화와 관련된 기본 애트리뷰트들이 자동으로 생성되어 채워질 것이다.



마법사 안에서 API Documentation 자동 생성 옵션을 선택하면 된다

EndPointRequestSummary

메서드 설명(description):

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

- **Tags:** tag를 정의한다.
- **Summary:** 메서드 제목.
- **Description:** 메서드 설명.
- **Produces:** API가 생산할 수 있는 MIME 타입. 이것은 모든 API들에 전역적으로 반영된다. 하지만, 특정 API를 호출하면서 이것을 덮어쓰기(override, 오버라이드) 할 수도 있다. 그 값은 반드시 Mime Types 아래에 명시되어야 한다.
- **Consume:** API가 소비할 수 있는 MIME 타입. 이것은 모든 API들에 전역적으로 반영된다. 하지만, 특정 API를 호출하면서 이것을 덮어쓰기(override, 오버라이드) 할 수도 있다. 그 값은 반드시 Mime Types 아래에 명시되어야 한다.

EndPoint의 GET 메서드에게 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.

교유한 하나의 파라미터를 정의하는 기준은 “이름과 위치” 조합이다.

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 경로의 조각(segment)와 일치해야 한다.
 - 그 외의 경우에는, 그 이름과 ParamIn는 일치한다.
- **Description:** 그 파라미터에 대한 간략한 설명. 여기에는 사용 예시들이 포함될 수 있다. GFM 문장 규칙을 사용해 리치 텍스트(rich text)를 표현할 수 있다.
- **Required:** 그 파라미터의 필수 여부. 그 파라미터의 위치가 ‘Path’인 경우에는 이 파라미터는 필수다. 그리고 그 값은 반드시 True여야 한다, 그 이외의 경우에는 선택 사항이며, 그 기본값은 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 타입은 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 그 Primitive(원시 데이터)의 Schema Definition. 그 body 요청의 구조에 대한 정의다. 만약, ParamType이 ‘Array’나 ‘Object’인 경우, 스키마를 JSON 그리고/또는 YAML 형식으로 정의할 수 있다.
- **Reference:** 서버로 전달되는 Primitive의 Schema Definition. 그 body 요청의 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',
'AResponse')]
```

- **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 Primitive(원시 데이터) 타입들로 일곱 가지를 정의하고 있다: 'spArray', 'spBoolean', 'spInteger', 'spNumber', 'spNull', 'spObject', 'spString'. [JSON Schema primitive types](#) 참조. 추가적인 Primitive 데이터 타입인 'spFile'은 해당 Parameter Object 그리고 해당 Response Object에서 사용한다. 이는 그 파라미터 타입 또는 그 응답은 파일이라고 지정한다.
- **PrimitiveFormat:** 반환되는 Primitive의 형식: 'None', 'Int32', 'Int64', 'Float', 'Double', 'Byte', 'Date', 'DateTime', 'Password'.
- **Schema:** 서버로 보내지는 Primitive의 Schema Definition. 그 body 요청의 구조에 대한 정의. 만약, ParamType이 'Array'나 '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(response)에 대한 더 자세한 설명은 [DocWiki의 이 링크](#)에서 확인할 수 있다

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 YAML 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 Delphi 예시를 적어 놓았다. 완전한 예시는, RAD Studio와 함께 제공되는 Sample 프로젝트들 안에 들어 있는 예시를 확인하기 바란다.

Delphi

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



팁

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

EMSDatasetResource에 애트리뷰트들을 정의하기

EMSDatasetResource는 훌륭한 컴포넌트다. 이것은 표준 CRUD 엔드포인트를 매우 빠르게 만들 수 있도록 도와준다. 하지만, 이 컴포넌트는 힘든 일들 대부분을 뒤에서 처리하는 방식이므로, 맨 처음부터 개발자가 각 엔드포인트를 직접 접근하고 원하는 애트리뷰트들을 붙일 수 있는 구조가 아니다.

디폴트로, 정의되는 애트리뷰트는 EndPointRequestSummary 뿐이다. RAD Server는 일반적인 CRUD 정의를 하는데, {id}를 프라이머리 키로 사용한다. 또한 그 엔드포인트들은 테스트를 할 수 없다. 이를 해소하려면, 우리가 프라이머리 키 이름을 정의할 뿐만 아니라 각 액션(action)과 그것의 사용자 정의 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 동일한 프로젝트가 있다. 다만, 이번 장에서 사용된 EMSDatasetResource 두 개에 맞도록 커스터마이징 되어 있다. 이 관행(convention)들에 대한 보다 상세한 정보는 README.md 파일 안에 있다. 또한 그 유닛들 안에도 주석으로 설명되어 있다.



팁

YAML과 JSON 스펙을 제공하려면, 각각 따로 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, 그에 대응하는 JSON 정의를 여러분을 위해 자동 생성하도록 하라.

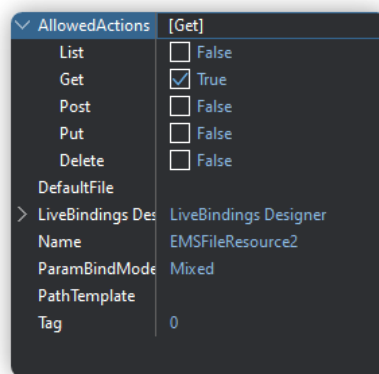
13

파일 관리 및 스토리지

이 장에서, 우리는 RAD Server의 기능들 중 파일을 관리하고 인프라 스토리지에 접근하는 여러가지 방법들을 분석한다. 또한, 엔드포인트마다 생성하거나 소비할 수 있는 파일의 유형을 어떻게 명시하는지 그 방법을 알아본다. 전역적으로(globally) 명시할 수도 있고 또는 각 엔드포인트별로 세분화된 방식으로 지정할 수도 있다.

TEMSFileResource

TEMSFileResource 컴포넌트는 TEMSDatasetResource와 동작 방식이 매우 비슷하다. 그래서 훨씬 편하다. 이 컴포넌트는 파일 관리를 위한 비즈니스 로직 대부분을 추상화한다. 필요한 모든 코드는 애트리뷰트에서 하면 된다.



TEMSFileResource의 주요 프로퍼티들

AllowedActions: 기본 설정으로는, 오직 GET(얻기)만 활성화된다. 하지만, 허용되는 액션들을 우리가 직접 지정할 수 있다. 그래서 파일 목록 얻기(List), 업로드(Post), 업데이트(Put), 삭제(Delete)를 할 수 있다.

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

PathTemplate: 단순하고 강력한 프로퍼티다. 가장 기본적인 예로 `c:\temp\{id}` 값을 지정했다고 가정하자. 이 값은 절대 경로다. 이 경로에는 이 컴포넌트가 다루는 파일들이 저장된다. `{id}`는 와일드카드다. 우리는 이 `{id}`를 애트리뷰트 안에서 파라미터처럼 사용할 수 있다. 즉, 요청의 안에 있는 파일 이름을 `{id}`를 통해 전달 받을 수 있다. 이 프로퍼티 안에서는 중괄호를 사용할 수 있다. 그래서, 보다 복잡한 경로를 만들 수 있다. 예를 들어, 이런 와일드카드를 사용해 하위폴더 또는 심지어 파일 확장자를 정의할 수 있다. 예: `c:\uploads\{folder}\{file}.{extension}`. 여기에서 파라미터는 3개다. 이 경우, 우리는 애트리뷰트 안에 folder, file, extension을 명시해야 한다. 예를 보자.



참고

대체로, `PathTemplate`에 들어가는 값은 환경 (`debug` 또는 `release`)에 따라 달라야 하는 경우가 많다. 따라서 환경에 따라 그 값이 달라지도록 작성하는 것을 권장한다. 컴파일 지시어를 사용하면 된다.

예시

델파이라면, `ResourceSuffix` 엔트리 3 개를 `EMSFileResource` 정의 앞에 명시한다 (해당 데이터 모듈 안에)

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, post다. 그리고 put, delete도 이런 식으로 정의하면 된다. test/fileResource/ 엔드포인트는 이제 “PathTemplate” 프로퍼티에서 해당하는 모든 필드들을 나열한다(list). 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).



경고

TEMSFileResource를 C++에서 사용하려면, emsserverresource.bpi 라이브러리를 “requires” 구역 안에 추가해야 한다. 그 파일은 이 곳에서 찾을 수 있다:

C:\Program Files (x86)\Embarcadero\Studio\XX.0\lib\{Platform}\release\emsserverresource.bpi

코드에서 파일을 다루기

조금 더 복잡한 상황이라서, TEMSFileResource 로는 충분하지 않다면, 파일을 코드에서 접근하는 것도 가능하다. 각 요청에는 ARequest 파라미터가 있다. 이것을 사용하면 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(ARequest.Body.Parts[0].GetStream, 0);
```

C++

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

위 예시에서, 우리는 body의 part 0 에 접근한다 (물론, 여러 part/파일들이 body 안에 있을 수 있다). 그리고 나서, 주어진 경로 안에 그 파일을 저장한다. 그 방법은 TFile 클래스를 사용해 body 안에 있는 해당 스트림을 읽고,

그 TFile 변수 안에 넣는다. 정말로 이렇게 간단하다. 이 리포지토리 예시 안에서는 이것보다 조금 더 복잡한 상황을 볼 수 있다. 거기에서는 반복(loop)을 사용해, body 안에 있는 모든 part에서 각 파일을 읽어서 모두 처리한다. 그래서 파일 여러 개를 하나의 엔드포인트로 보내 업로드한다.

Content-Type HTTP 헤더들

RAD Server는 이제 EndPoint 애트리뷰트들을 통해 Content-Type과 Accept 기반 매핑을 지원한다. 덕분에, 리소스 매핑이 더 좋아졌다. URL에만 의존하지 않기 때문이다. 즉, 여러분은 두 개의 다른 메서드를 동일한 “URL과 HTTP 동사(verb)” 조합에 매핑할 수 있다. 그러면서도, 요청에 따라 그에 맞는 다른 데이터 타입을 반환할 수 있다.

새로 추가된 EndPoint 애트리뷰트는 이 두가지다:

- EndpointProduce: 여기에는 MIME 타입 / 파일 확장자를 명시한다. 엔드포인트는 여기에 명시된 대로 GET 메서드에 대해 응답을 생성(produce)할 수 있다. 이 엔드포인트가 수행할 메서드가 무엇인지는 HTTP request header 안에 있는 Accept에 의해 결정된다.
- EndpointConsume: 여기에는 MIME 타입 / 파일 확장자를 명시한다. 엔드포인트는 여기에 명시된 대로 PUT, POST, PATCH 메서드에 대해 소비(consume)할 수 있다. 이 엔드포인트가 수행할 메서드가 무엇인지는 HTTP request header 안에 있는 Content-Type에 의해 결정된다.

이 장은 RAD Server 애플리케이션 리소스들이 어떻게 EndpointProduce 애트리뷰트를 사용하는 지, 그래서 여러 가지 REST Get 요청들을 image/jpeg과 application/xml MIME 타입을 기반으로 어떻게 처리하는 지를 보겠다.

간단한 예시

RAD Server 프로젝트를 새로 시작한다. 프로젝트 패키지 마법사를 사용하면 된다. File type에서 Data Module을 선택하고, Resource 이름을 AcceptTypes라고 정한다. 모든 표준 EndPoint 선택을 풀고, Finish 버튼을 누른다.

c:\temp\page 안에 새 폴더를 하나 만든다. 그리고 그 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 애트리뷰트인 ResourceSuffix와 EndpointProduce를 추가한다. 아래와 같다:

```
[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 엔드포인트가 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 GetImage와 GetText 엔드포인트를 추가한다. Register 함수 안에 애트리뷰트를 추가한다. 즉 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 Server 프로젝트를 저장하고 빌드한다. 이제 이 content.jpeg와 content.txt에 접근할 수 있다. 무슨 파일에 접근하게 되는지는 요청의 header 안에 들어 있는 Content-Type에 의해 결정된다.

14

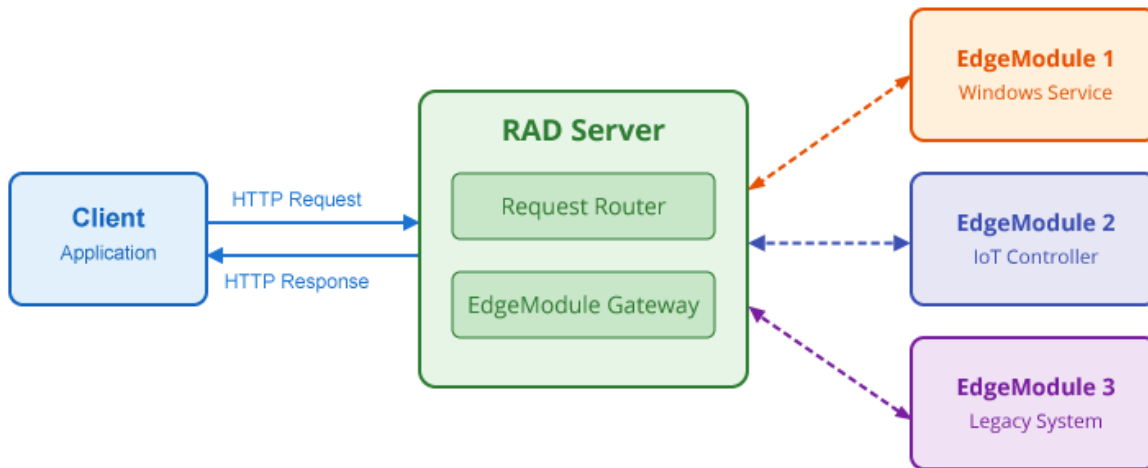
EdgeModule들: RAD Server를 확장하기

.....

RAD Server는 견고한 프레임워크를 제공해, 확장할 수 있는 서버 애플리케이션을 구축할 수 있도록 한다. 하지만, 현실에서는, 시스템들이 다양한 서비스들/애플리케이션들과 통합되어야 하는 경우가 많다. EdgeModule (엣지모듈)들은 외부에 있는 애플리케이션/서비스/디바이스를 여러분의 RAD Server 인스턴스에 연결할 수 있는 강력한 기능이다. 여러분은 이것을 사용해, RAD Server의 기능을 확장할 수 있다. 그러면서도, 깔끔하고 잘 조직된 아키텍처를 그대로 유지할 수 있다.

EdgeModule들을 이해하기

EdgeModule은 독립적인 서비스다. 즉 여러분의 메인 RAD Server 인스턴스와 별개로 실행된다. 그런데, RAD Server에 연결되어 있고, RAD Server를 통해 접근 할 수 있다. RAD Server는 EdgeModule에 대한 요청을 받으면, 그 요청을 적절한 서비스에게 전달한다. 그리고 그 결과를 클라이언트에 반환한다. 이 아키텍처 덕분에 클라이언트들은 자연스럽게(Seamless) 서비스를 사용할 수 있다 (서버 호출만 한 번 하면 된다). 한편, 개발자는 시스템의 여러 부분들을 각각 독립적으로 구축하고 유지보수할 수 있다. 심지어 여러 장비에 걸쳐 분산할 수도 있다.



이 EdgeModule 아키텍처는 클라이언트의 요청이 RAD Server를 거쳐 각 EdgeModule로 전달되고 다시 돌아오는 과정을 보여준다.

이 방식의 주요 장점은 다음과 같다. 즉 각 EdgeModule은 이런 것들을 할 수 있다:

- 실행 중인 RAD Server 인스턴스에 손쉽게 기능을 추가/삭제 할 수 있다. 여기에는 서버 자체에서 직접 연결하기 어려운 기기들(예: 블루투스, 시리얼 포트 등)도 해당된다.
- 지원되는 모든 플랫폼(Windows, Linux, macOS)에 배포될 수 있다.
- 업데이트 또는 변경을 하는 경우, 메인 RAD Server 인스턴스에 영향을 주지 않는다.
- 각각 독립적으로 확장(Scaling)할 수 있다. 그래서 각자 필요에 맞게 확장할 수 있다.
- 어떤 언어나 프레임워크로도 작성할 수 있다.



참고

RAD 스튜디오가 아닌 언어로 개발된 모듈을 통합하려면, EdgeModule들을 등록할 수 있도록 제공되는 API 엔드포인트를 통해 통합을 진행해야 한다. 뿐만 아니라 이 EdgeModule 기술의 규격들을 준수해야 한다. 제공되는 모든 엔드포인트들은 이 [DocWiki 링크](#)에서 확인할 수 있다.

TEMSEdgeService 컴포넌트

EdgeModule 기능의 핵심은 TEMSEdgeService 컴포넌트다. 이 컴포넌트는 (EMS Provider에 연결되어) 여러분의 서비스들과 RAD Server 사이의 모든 소통을 처리한다. 즉, 등록(registration), 요청 처리(request handling), 수명 주기 이벤트(lifecycle event)들을 관리한다.

TEMSEdgeService를 여러분의 애플리케이션에 추가하면 다음 작업들이 자동으로 수행된다:

- 여러분의 서비스를 RAD Server에 등록한다
- 여러분의 엔드포인트들에게 접근할 때 RAD Server URL 구조를 사용할 수 있도록 한다
- 들어오는 요청(request)들을 다룬다. 그리고 그 요청들을 여러분의 리소스 구현들에게 보낸다(routing)

- 엣지 서비스의 생명 주기를 관리한다

EdgeModule들을 사용하는 주요 이점 중 하나는 메인 RAD Server 인스턴스를 다시 빌드하지 않고도 기능을 수정할 수 있다는 점이다. 즉 메인 인스턴스에서 별도의 작업을 수행하지 않고도 EdgeModule들을 업데이트하거나 수정할 수 있다. EdgeModule을 다시 등록하는 것만으로, 변경된 새 구현이 자동으로 배포된다.

컴포넌트 설정

TEMSEdgeService 컴포넌트는 몇 가지 중요한 프로퍼티들을 제공한다. 그것들은 이 컴포넌트의 동작을 제어한다:

- **ModuleName**: 여러분의 EdgeModule을 위한 고유 식별자
- **ModuleVersion**: 버전 문자열 - 업데이트들을 추적하고 관리 할 수 있다
- **ListenerProtocol**: 통신 프로토콜(http/https)
- **ListenerService**: 호스트와 포트 - 이곳에 EdgeModule이 노출된다.(옮긴이: 원문에는 ListeningService)
- **Active**: 그 모듈의 실행 여부를 제어한다
- **AutoActivate**: Provider가 설정되었을 때, 자동으로 활성화하기
- **AutoRegister**: 활성화 시 RAD Server에 자동으로 등록하기
- **AutoUnRegister**: 비활성화 시 자동으로 등록 취소하기

그 컴포넌트는 이벤트들도 제공한다. 그래서 여러분이 그것의 다양한 생명 주기 단계를 가로채 들어갈 수 있게 한다:

- **OnRegistering**: 호출되는 시점 - 모듈 등록 전
- **OnRegistered**: 호출되는 시점 - 모듈 등록 성공 후
- **OnUnregistering**: 호출되는 시점 - 등록 취소 전
- **OnUnregistered**: 호출되는 시점 - 등록 취소 성공 후



팁

ModuleName은 EdgeModule의 핵심 프로퍼티다. 엣지 서비스를 RAD Server에 등록할 때 이 식별자가 사용된다. 따라서, 좋은 이름 짓기 규칙을 지키는 것이 매우 중요하다. 그래야 중복을 방지할 수 있다.

리소스 구현

EdgeModule들의 리소스 구현 패턴은 RAD Server 리소스 구현 패턴과 똑같다. 이러한 일관성 덕분에 RAD Server에 익숙한 개발자는 EdgeModule 서비스들을 쉽게 만들 수 있다. 여러분은 어떤 프로젝트에도 새 RAD

Server 리소스를 추가할 수 있다. RAD Studio 메뉴의 **File > New > Other... > Delphi/C++Builder > RAD Server > RAD Server Module**을 사용하면 된다.

기본적으로, 새로 생성되는 모든 리소스 모듈들에는, 자동-등록(auto-registering) 코드가 initialization 구역의 메서드 안에 구현된다. 따라서 추가 작업이 필요없다. 즉, 생성되는 모든 새 모듈들은 애플리케이션 실행될 때 자동으로 등록되도록 되어있다.

기본적인 예시를 만들기

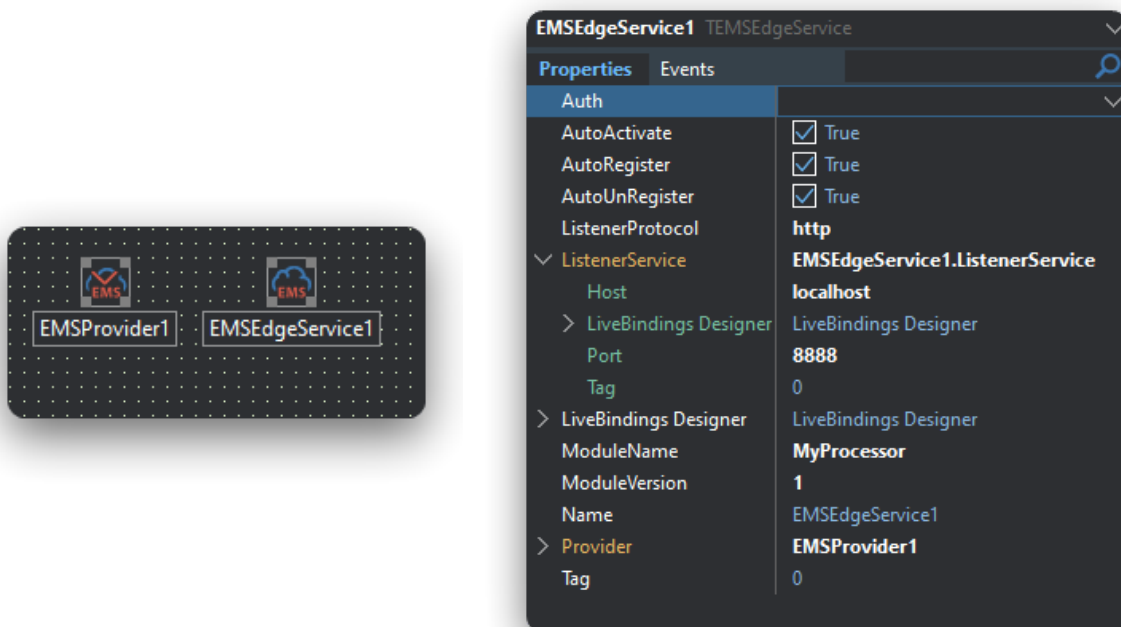
[깃허브\(GitHub\) 저장소](#)에 미리 구성된 RAD Server EdgeModule 애플리케이션의 예시가 포함되어 있다. 하지만, 기본 단계를 차례대로 진행하면서 이 예시를 구현해보자.

필수 컴포넌트들을 추가하기

RAD Server 애플리케이션이 아닌 프로젝트에서 진행한다. 그리고 우리는 반드시 **TEMSPROVIDER** 컴포넌트를 등록해야 한다. 이것이 우리의 RAD Server 인스턴스에 연결하기 때문이다. 우리는 이 컴포넌트의 여러 측면들을 직접 정의할 수 있다. 예를 들어, 호스트와 포트를 명시해 이것이 어디에서 실행되는지를 지정할 수 있다. 그 외에도 다양한 설정을 할 수 있다.

이제, **TEMSEdgeservice**를 애플리케이션에 추가한다. 그리고 그것을 EMS Provider에 연결한다. 그리고 나서, **ModuleName** 프로퍼티를 **MyProcessor**라고 명시한다. 그러면 이 이름으로 RAD Server 인스턴스 안에 엡지 모듈이 등록된다. 이 컴포넌트에는 그 외에도 여러가지 프로퍼티들이 있다(예: Auth, Version ...).

추가로, EdgeModule이 노출할 호스트와 포트를 **ListenerService** 프로퍼티에 정의해야 한다. 이 장의 기본 예시에는 각각 “localhost”와 “8888”로 지정되어 있다.



리소스를 생성하고 등록하기

우리는 이제 새 RAD Server 모듈을 프로젝트에 추가하겠다. 앞서 설명했듯이 **File > New** 메뉴를 사용하면 된다. 이 예시 안에서, 우리는 리소스를 하나 생성하는데, 그 이름을 'processor'라고 지정한다. 그리고 그 안에 엔드포인트를 하나 생성하는데, 그 이름을 'calculate'라고 지정한다. 이 엔드포인트는 GET 호출 안에 있는 파라미터 두 개를 받는다. 그리고 그 두 수의 합을 반환한다.

Delphi:

```
type
  [ResourceName('processor')]
  TProcessorResource = class
  published
    [ResourceSuffix('calculate')]
    procedure GetCalculate(const AContext: TEndpointContext;
                          const ARequest: TEndpointRequest;
                          const AResponse: TEndpointResponse);
end;

procedure TProcessorResource1.GetCalculate(const AContext: TEndpointContext;
                                          const ARequest: TEndpointRequest;
                                          const AResponse: TEndpointResponse);

var
  LParam1, LParam2: string;
  LResponse: TJSONObject;
begin
  // 입력 파라미터들을 읽는다
  if not(ARequest.Params.TryGetValue('param1', LParam1)) or
    not(ARequest.Params.TryGetValue('param2', LParam2)) then
    AResponse.RaiseBadRequest('Bad request', 'Missing data');
  LResponse := TJSONObject.Create;
  try
    // 결과를 계산한다
    var calculation := LParam1.ToDouble + LParam2.ToDouble;
    // 여러분의 처리 로직을 여기에 추가한다
    LResponse.AddPair('status', 'processed');
    LResponse.AddPair('timestamp', DateToISO8601(Now));
    LResponse.AddPair('result', calculation);
    // 응답(response)을 발송한다
    AResponse.Body.SetValue(LResponse, False);
  finally
    LResponse.Free;
  end;
end;

procedure Register;
```

```
begin
    RegisterResource(TypeInfo(TProcessorResource));
end;

initialization
    Register;
end.
```

C++Builder:

```
void TProcessorResource1::GetCalculate(TEndpointContext* AContext,
                                       TEndpointRequest* ARequest,
                                       TEndpointResponse* AResponse)
{
    String LParam1, LParam2;
    TJSONObject* LResponse;

    // 입력 파라미터들을 읽는다
    if (!(ARequest->Params->TryGetValue("param1", LParam1)) ||
        !(ARequest->Params->TryGetValue("param2", LParam2))) {
        AResponse->RaiseBadRequest("Bad request", "Missing data");
    }

    LResponse = new TJSONObject();

    try {
        // 결과를 계산한다
        double calculation = StrToFloat(LParam1) + StrToFloat(LParam2);

        // 여러분의 처리 로직을 여기에 추가한다
        LResponse->AddPair("status", "processed");
        LResponse->AddPair("timestamp", DateToISO8601(Now()));
        LResponse->AddPair("result", calculation);

        // 응답(response)을 발송한다
        AResponse->Body->SetValue(LResponse, false);
    }
    __finally {
        LResponse->Free();
    }
}

static void Register()
{

```

```
std::unique_ptr<TEMSResourceAttributes> attributes(new TEMSResourceAttributes());
attributes->ResourceName = "processor";
attributes->ResourceSuffix["GetCalculate"] = "calculate";
RegisterResource(__typeinfo(TProcessorResource1), attributes.release());
}
```

일단 이 애플리케이션이 작동하면, 클라이언트들은 RAD Server를 통해 접근할 수 있다. 이 URL 패턴을 사용한다:

<http://yourserver:8080/edgmodule/MyProcessor/processor/calculate?param1=5¶m2=10>



참고

이 GitHub 기본 예시 프로젝트에는 엔드포인트가 하나 더 있다. 그 이름은 “formValues”다. 그 엔드포인트는 데이터를 실행 중인 FMX 애플리케이션에서 받아온다. 그리고 받아온 그 현재 값을 응답(response) 안에 담는다.

실제 적용(Practical Application)들

EdgeModule들은 몇몇 실제 상황에서 능력을 발휘한다. RAD Server의 핵심 설계는 RESTful 아키텍처이지만, 여러분이 연결해야 하는 기기(device)들 또는 시스템들 중에는 상태 관리(state management) 또는 오랫동안 실행되는 작업들이 필요한 것들이 있을 수도 있다. RAD Server Edge Modules / [Thing Points](#)는 이 요구 사항을 해결할 수 있는 실질적인 도구세트를 개발자에게 제공하기 위해 고안되었다. 그리고 분산된 IoT(사물인터넷) 기기 통합부터 중앙 집중식 상태 관리까지를 커버한다.

실무 적용 예시로는 이런 것들이 있다:

1. 레거시 시스템 통합

시스템을 현대화할 때, 일부 오래된 컴포넌트를 여전히 유지하면서 다른 한편으로 점진적으로 업데이트해야 하는 경우가 있다. EdgeModule들을 사용하면 이런 것들을 할 수 있다:

- 레거시 델파이 애플리케이션들을 여전히 유지하고 서비스로 실행한다
- 기능을 현대적인 REST API를 통해 노출한다
- 점진적으로 컴포넌트를 교체한다 그래서 시스템 중단이 없도록 한다

2. 특화된 처리 서비스들

몇몇 작업들은 특정 도구 또는 라이브러리가 필요한 경우가 있다. 그런데 그런 것들이 메인 RAD Server 환경에서는 제공되지 않거나 최적화할 수 없을 수도 있다:

- 이미지 처리 모듈인데 특별한 라이브러리를 사용하는 것들
- 머신러닝 모델인데 Python(파이썬)에서 실행되는 것들
- 고성능 연산 서비스인데 C++로 되어 있는 것들

3. 크로스-플랫폼 통합

플랫폼마다 각자의 플랫폼-전용 코드가 필요한 경우가 많다. EdgeModule들을 사용하면 이런 것들을 할 수 있다:

- Windows 서비스들을 만든다. 그 서비스들이 Windows 전용 API를 사용하도록 한다
- macOS 서비스들을 구축한다. 그 서비스들이 Apple 프레임워크와 통합되도록 한다.
- Linux 서비스들을 배포한다. 그 서비스들이 네이티브 리눅스 도구를 사용하도록 한다.

4. 하드웨어 및 IoT 통합

EdgeModule들은 탁월한 솔루션을 제공해 하드웨어 기기들을 여러분의 서버 인프라에 통합할 수 있게 한다:

- 블루투스 기기 통신. 플랫폼-전용 API를 사용한다
- IoT(사물인터넷) 기기 관리. 특수 프로토콜을 사용한다
- 비콘(beacon) 추적 및 근접도(proximity) 서비스
- 산업용 장비 모니터링 및 제어
- 스마트 홈 기기 통합
- (네이티브 드라이버가 필요한) USB 기기 인터페이스 연결
- RFID 리더(reader) 및 NFC 통신
- GPS 및 위치-기반 서비스
- (독자적인 통신 프로토콜을 사용하는) 커스텀 하드웨어

EdgeModule들과 InterBase의 ChangeViews를 통합하기

EdgeModules의 활용 중 가장 강력한 예는 RAD Server를 확장할 때 고도화된 데이터베이스 기능들을 사용할 수 있다는 점이다. 대표적인 사례가 [InterBase의 ChangeViews 기술](#)과의 통합이다. 이 기술은 클라이언트와 서버 간의 효율적인 데이터 동기화를 실현한다.

InterBase ChangeViews란 무엇인가?

InterBase ChangeViews는 강력한 기능이다. 이 기능은 데이터베이스 테이블들의 변경을 추적한다. 그런데, 클라이언트와 서버 사이에 반복해서 데이터 세트 “전체”를 전송하는 방식이 아니다. ChangeViews를 사용하면 클라이언트들은 각자 마지막으로 동기화한 이후에 “변경된 레코드들”만 수신한다. 그래서 대역폭(bandwidth) 사용량이 작아지고 성능은 향상된다. 이는 특히, 모바일 애플리케이션이나 대역폭이 제한되는 연결 환경에서 효과가 크다.

ChangeViews라는 도전 과제

ChangeViews는 너무나 유용하다. 하지만, 이를 RESTful 환경 안에 구현하는 것은 어려운 도전 과제다. REST는 그 본질 상 Stateless, 즉 상태를 유지하지 않는다. 그런데, ChangeViews는 단기 세션(session)이 필요하다. 그것이

있어야 동일한 데이터베이스 트랜잭션에 대고 여러 번의 상호작용을 할 수 있고, 그래야 단계별 커밋(Multi-phase commit)이 가능하기 때문이다. (예: 요청 1은 데이터를 얻는다. 요청 2는 수신을 확인한다. 그 후에, 여러분이 ChangeView 트랜잭션의 커밋/롤백 여부 결정한다). 이 지점에서 EdgeModule들은 우아한 해법을 제공한다.



참고

이 EdgeModule + ChangeViews 패턴을 구현한 데모는 [GitHub 저장소](#) 안에 있다.

데모가 작동하는 방식

이 예시 안에 있는 EdgeModule은 상태 유지(Stateful) 계층을 RESTful 통신 위에 만든다. 그래서 ChangeViews 세션을 관리한다. 작동 방식은 다음과 같다:

- **클라이언트 식별:** 각 클라이언트는 고유한 기기 ID(Device ID)를 가진다. 서버는 동기화 상태를 추적할 때 이 기기 ID를 사용한다.
- **세션 관리:** EdgeModule은 각 기기 ID 당 하나의 세션을 유지한다. 그 세션 안에는 이 정보들을 담는다:
 - 활성 트랜잭션(스냅샷 격리 모드 안에서)
 - ChangeView 구독 정보
 - 비활성 세션 처리를 위한 타이밍 정보
- **네(4) 가지 주요 동작:**
 - **Activate:** 특정 기기 ID를 위한 ChangeView 하나를 초기화(initialize)한다
 - **Query:** 변경된 레코드들을 반환한다. 이때 그 ChangeView의 상태를 변경하지 않는다
 - **Commit:** 변경들을 승인한다. 그리고 그 클라이언트의 기준선(Baseline)을 업데이트한다.
 - **Rollback:** 현재 뷰를 취소한다. 그리고 그 클라이언트가 그 변경을 나중에 다시 볼 수 있도록 한다.
- **변경 조회(Changes Querying):** 클라이언트들은 먼저 변경 개수(count)를 조회할 수 있다. 즉 미리 알아 보고 나서 실제 데이터를 가져올 지 여부를 결정할 수 있다. 그래서 모바일 환경에 매우 적합하다.

이 데모는 최고 점수(high scores)라는 간단한 테이블 하나를 사용한다. 그 테이블에는 ID, name, score 라는 필드들이 있다. 모바일 게임이 있는데, 최고 순위에 들기 위해 전 세계 사람들이 경쟁하고 있다고 상상하면서 보자.

ChangeViews 패턴 사례 적용 시, 예를 들어, 데이터 사용량이 제한된 사람은 변경된 점수가 있는지 여부를 먼저 확인하고 나서, 최신 데이터를 다운로드 할 것인지를 결정한다. 동기화를 할 때는, 마지막 확인 시점 이후의 새 최고 점수들만 본다. 순위 정보 전체를 다운로드하는 게 아니다. 이 똑같은 방식으로, 여러 기기들을 효율적으로 동기화한다. 그래서 사람들이 항상 최신 데이터를 가지도록 보장하면서도 불필요한 네트워크 사용이 없도록 한다.

이 방식의 장점

이 통합 예시는 EdgeModule들이 제공하는 몇 가지 강력한 이점들을 보여준다:

- **능력 확장:** 이 EdgeModule은 상태 유지(Stateful) 동작을 통해 RAD Server에 추가한다. 메인 서버를 수정하지 않아도 된다.
- **효율적인 데이터 전송:** 클라이언트들은 오직 최종 동기화 이후에 변경된 데이터만 수신한다.
- **클라이언트의 통제력:** 클라이언트들은 변경 사항 수용 시점을 각자 결정할 수 있다. 저마다의 특정 요구 사항이나 제약 조건을 기준으로 결정하면 된다.
- **리소스 관리:** 이 EdgeModule은 버려지는 세션들을 깨끗이 치워준다. 타임아웃 메커니즘을 반영한다.
- **관심사 분리 원칙(Separation of Concerns) :** 이 ChangeViews 기능은 각자 자신의 모듈 안에 격리된다. 따라서, 메인 서버를 깔끔한 그대로를 유지할 수 있다.

EdgeModule들과 InterBase ChangeViews를 결합한 이 예시는 RAD Server를 확장해 고도화된 데이터 동기화 패턴을 지원하는 방법을 잘 보여준다. 이 패턴은 핵심 서버 아키텍처를 복잡하게 하기 않는다.

결론

EdgeModule들은 유연한 방법을 제공해 RAD Server의 기능을 확장할 수 있도록 한다. 그러면서도, 깔끔하고 잘 조직된 아키텍처를 그대로 유지할 수 있도록 한다. 여러분은 관리하기 좋고 확장 가능한 애플리케이션들을 구축할 수 있다. 여러분의 시스템 중에서 각 부분이 독립적으로 작동하면서도, RAD Server를 거쳐 연결되기 때문이다.

TEMSEdgeService 컴포넌트 그리고 여러분에게 익숙한 리소스 구현 패턴을 조합하면, 여러분이 새 서비스들을 만들고 그것을 여러분의 RAD Server 인프라와 통합하는 절차가 단순 명료하다. 여러분이 레거시 시스템을 통합하고 있든, 특화된 처리 기능을 추가하든, 개발 팀의 체계를 잡고 있든 상관없이, EdgeModule들은 현대적인 그리고 분산적인 애플리케이션들을 구축할 수 있는 실질적인 해법을 제공한다.

15

RAD Server와 WebStencils(웹스텐실즈)

WebStencils란 무엇인가?

WebStencils(웹스텐실즈)는 서버-쪽 스크립트-기반 통합 기술이다. 이것은 HTML 파일들을 처리하는 기술이다. 도입된 시점은 RAD Studio 12.2부터다. 개발자들은 WebStencils를 사용해 현대적이고 전문적인 모습을 갖춘 웹사이트를 구축할 수 있다. 그 웹사이트의 기반으로 어떤 자바스크립트 라이브러리든 적용할 수 있다. 또한 RAD Studio 서버-쪽 애플리케이션이 추출하고 처리하는 데이터를 활용할 수 있다.



무료 도서

WebStencils에 대해 더 자세히 알고 싶다면, 무료 백서인 'HTMX와 웹스텐실즈'를 다운로드 하기 바란다! <https://lp.embarcadero.com/HTMX-WebStencils>.

(옮긴이: 한글 번역본은 데브기어에 문의하세요)

RAD Server와 WebStencils를 통합하기

RAD Server에도 역시 WebStencils가 제공하는 혜택이 반영되어 있다. 특정 통합이 개발되어 RAD Server 반영되었고, 그 덕분에 RAD Server는 웹 개발이 가진 모든 잠재력을 활용할 수 있다. WebStencils는 프레임워크를 가리지 않는다. 즉, 여러분은 똑같은 문장, 똑같은 컴포넌트 등을 그대로 사용하면 된다. 여러분이 백엔드 서비스로 RAD Server와 WebBroker 중 무엇을 선택하든 상관 없다.



팁

현대 웹 서비스들은 일반적으로 프론트엔드 웹사이트를 제공할 뿐만 아니라 REST API를 함께 제공해 데이터에 접근할 수 있도록 하고 있다. 그래서 써드-파티 서비스들과 더 쉽게 통합할 수 있도록 한다. RAD Server를 사용하면, 이 둘 모두를 동일한 애플리케이션 안에 구현할 수 있다.

RAD Server와 WebStencils를 사용하는 방식에는 여러 가지가 있다. 가장 일반적인 두 가지 패턴을 보자:

WebStencils Processor들을 사용하기

이것은 간단한 방식이다. 각 요청(request)을 받으면 프로세서(processor)들을 사용해 HTML을 만들어 낸다. (프로세서들은 디자인 타임에 또는 프로그램 방식으로 생성된다) 그리고 만들어 낸 HTML을 RAD Server 응답(response) 안에 바로 넣어 반환한다. 프로세서는 템플릿을 읽어와서 처리해 HTML을 완성한다. 프로세서는 그 템플릿들을 파일, 상수, 변수 등으로부터 읽어 올 수 있다.

```
type
  [ResourceName('testfile')]
  TTestResource = class(TDataModule)
    [ResourceSuffix('get', './')]
    [EndpointProduce('get', 'text/html')]
    procedure Get(const AContext: TEndpointContext;
                  const ARequest: TEndpointRequest;
                  const AResponse: TEndpointResponse);

  ...

  procedure TTestResource.Get(const AContext: TEndpointContext;
                              const ARequest: TEndpointRequest;
                              const AResponse: TEndpointResponse);

var
  LTemplateFile, LHTMLContent: string;
Begin
  // 이 변수는 적용할 템플릿이 있는 실제 경로를 가리키도록 교체할 것
  LTemplateFile := 'C:\path\to\your\file.html';
  WebStencilsProcessor.InputFileName := LTemplateFile;
  LHTMLContent := WebStencilsProcessor.Content;
  AResponse.Body.SetString(LHTMLContent);
end;
```

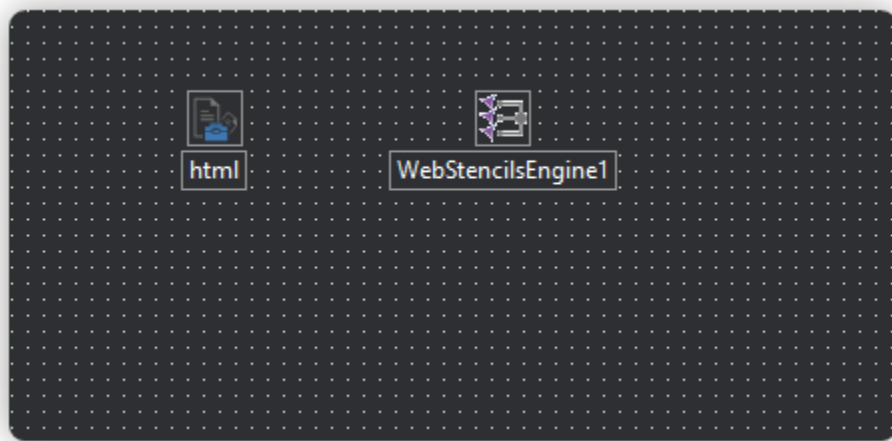
이 RAD Server 프로젝트를 실행한다. 그리고 브라우저를 열어 <http://localhost:8080/testfile> URL에 접속한다. 그러면 `LTemplateFile` 변수에 정의된 템플릿이 처리된다. 우리는 완성되어 렌더링되는 콘텐츠를 볼 수 있다.

**참고**

RAD Server의 경우, 템플릿 경로는 반드시 절대 경로로 지정해야 한다. 상대 경로를 사용하는 것은 불가능하다. 왜냐하면, RAD Server의 특성상 실전 운영 배치 시에는 Apache 또는 IIS를 사용하기 때문이다.

WebStencils Engine을 사용하기

이 옵션의 경우, 이미 존재하는 **TEMSFileResource** 컴포넌트와 **TWebStencilsEngine** 컴포넌트를 조합해 사용하기를 권장한다. 전자는 파일 시스템으로의 매핑을 수행한다. 후자는 HTTP 매핑과 템플릿 처리를 관리한다. 이 컴포넌트들을 서로 연결할 때는 WebStencils 엔진(engine)의 Dispatcher 프로퍼티를 사용한다.



TEMSFileResource를 올바르게 구성하려면, 템플릿들이 위치한 위치를 지정해야 한다. 이 컴포넌트 안에 있는 **PathTemplate** 프로퍼티를 사용하면 된다.

C:\path\to\your\templates\{filename}

{filename} 와일드카드 사용에 특히 주의하기 바란다. 이 와일드카드가 각 페이지를 참조하기 때문이다.

이 컴포넌트들에 대한 구성을 완료하고 나면, WebStencils 엔진 안에 있는 **PathsTemplates** 을 지정할 수 있다. 그 방식은 우리가 이미 봤듯이, 와일드카드 **{fileName}** 또는 템플릿 파일의 이름들을 사용하면 된다.

파일들을 올바르게 매핑하려면, 몇몇 애트리뷰트들을 “testfile” 리소스에 정의해야 한다. 앞 예시에서 했던 대로 하면 된다:

```
type
[ResourceName('testfile')]
TTestfileResource1 = class(TDataModule)
[ResourceSuffix('./')]
[ResourceSuffix('get', './{filename}')]
end;
```

```
[EndpointProduce('get', 'text/html')]
html: TEMSFileResource;
```

위 예시와 같다면, 우리는 **http://localhost:8080/testfile/{filename}** 엔드포인트에 접근할 수 있다. 여기에서 **{filename}**에는 우리가 **PathTemplate** 프로퍼티에 정의해 놓은 경로 안에 저장해 놓은 템플릿들 중 어느 파일이든 그 이름을 넣으면 된다.



팁

만약 똑같은 **TWebStencilsEngine**를 **TEMSFileResource** 하나 이상에게 연결하고 싶은 경우에는, 전역 메서드인 **AddProcessor**를 통해 추가로 할당할 수 있다.

예: **AddProcessor(FileResourceResource, WebStencilsEngine1);**

정적 JS, CSS, 이미지 등을 처리하기

RAD Server의 주요 기능 중 대표적인 것은 JSON 처리이다. 하지만, 정적 파일을 제공하는 데에도 사용할 수 있다. 즉 이미지, CSS, JS 파일 등을 제공할 수 있다: 이러한 정적 파일들을 전달하려면 **TEMSFileResource** 컴포넌트를 사용하면 된다.

WebStencils의 [RAD Server 데모](#) 안에는 독립적인 **TEMSFileResource** 컴포넌트 여러 개가 정의되어 있다. 그 컴포넌트들은 각각 해당 파일들(JS, CSS, 이미지)이 위치한 각 폴더를 매핑한다.

프론트엔드 소스

RAD Server의 요청은 리소스 기반이다. 즉, 우리가 RAD Server에 접근하려면 메인 URL의 끝에 리소스 이름을 붙여야 한다. 예: <https://localhost:8080/web>. 웹사이트를 제공하면서 동시에 REST API도 제공하는 경우에는, 이 방식 덕분에 우리가 두 엔드포인트를 명확히 분리할 수 있다. 그래서 동일한 RAD Server 인스턴스에서 그 두 가지 모두를 호스팅할 수 있다.

더 자세히 알아보기

이것은 RAD Server와 WebStencils 사이의 통합을 살짝 미리 보여준 정도에 불과하다. 이 라이브러리 그리고 이것이 제공하는 모든 능력들을 더 자세히 알아보려면 [무료 전자책\(e-book\)을 다운로드](#)해 읽어 보기 바란다. (옮긴이: 한글 번역본은 데브기어에 문의하세요)

16

RAD Server 멀티-테넌시 지원

멀티-테넌시(Multi-Tenancy) 소개

멀티-테넌시(Multi-Tenancy)는 아키텍처 패턴이다. 이 패턴에서는 단일 소프트웨어 인스턴스가 여러 개의 독립된 클라이언트 기관(테넌트, tenant)들을 위해 동작한다. 엔터프라이즈 애플리케이션들 안에서 멀티-테넌시를 적용하면 리소스 활용이 효율적이다. 그러면서도 각 테넌트 사이에 데이터 격리를 엄격하게 유지할 수 있다. 이 방식은 특히 SaaS([Software as a Service](#)) 애플리케이션에서 가치가 높다. 왜냐하면, 많은 클라이언트들이 동일한 서버 애플리케이션에 접근해야 하지만, 각자의 전용(private) 데이터와 커스터마이징도 필요하기 때문이다.

RAD Server는 멀티-테넌시를 견고하게 지원한다. 즉, RAD Server 인스턴스 하나는 데이터베이스 연결을 하나만 가지고 있으면서, 여러 개의 격리된 테넌트들에서 서비스를 제공할 수 있다. 각 테넌트는 자신만의 독립적이고 안전한 실행 환경 안에서 동작한다. 그리고 각자의 리소스들은 다른 테넌트와 완전히 분리된 상태로 관리된다. 즉, 사용자, 그룹, 설치 정보, 엣지 모듈, 기타 데이터 등을 테넌트마다 독립적으로 다룬다.

테넌트 격리는 요청(request) 수준에서 강제된다. 즉 각 요청마다 테넌트 자격 증명을 확인하고, 각 테넌트에 해당하는 데이터만 가져오도록 모든 데이터베이스 작업에 필터를 적용한다. 이는 RAD Server의 인증 및 권한 부여 파이프라인 안에서 투명하게 처리된다. 그리고 테넌트 컨텍스트는 요청 처리 전 과정에서 사용할 수 있게 제공된다.

RAD Server 멀티-테넌시의 주요 장점

- **리소스 효율성:** 단 하나의 RAD Server 인스턴스로 여러 개의 클라이언트들에서 서비스를 제공한다. 그래서 하드웨어와 유지보수 비용을 절감한다.
- **데이터 격리:** 테넌트 데이터의 완전하게 분리한다. 그리고 그 보안 경계가 견고하다.

- **관리 단순화:** 애플리케이션을 중앙에서 관리한다. 그러면서도 테넌트별 구성을 유지한다.
- **확장성:** 새로운 테넌트를 추가할 수 있다. 그래도 추가 인프라를 배포하지 않아도 된다.
- **사용자 정의(custom) 리소스:** 각 테넌트는 각자 자신만의 사용자 정의 리소스들을 가질 수 있다.

RAD Server 안에 있는 테넌트 아키텍처를 이해하기

RAD Server 안에서, 테넌트 하나는 고유한 기관 단위(organizational entity)에 해당한다. 각 기관은 각자 자신만의 격리된 환경이 필요하다. 따라서 각 테넌트는 이런 것들을 가지고 작동한다:

- 고유한 **TenantID** (전형적으로 GUID) 및 **TenantSecret** (비밀번호)
- 격리된 인증 컨텍스트 (테넌트별 사용자와 그룹들)
- 리소스 접근 제어가 분리되어 있음
- 테넌트별 데이터 필터링
- 독립적인 구성

RAD Server 안에 반영된 멀티-테넌시 모델은 "모든 것을 공유하되, 논리적으로는 완전히 분리" 방식을 따른다: 하나의 서버, 하나의 데이터베이스, 하지만, 엄격하게 통제되는 접근 경계를 테넌트들 사이에 둔다.

RAD Server 안에서 멀티-테넌시를 활성화하기

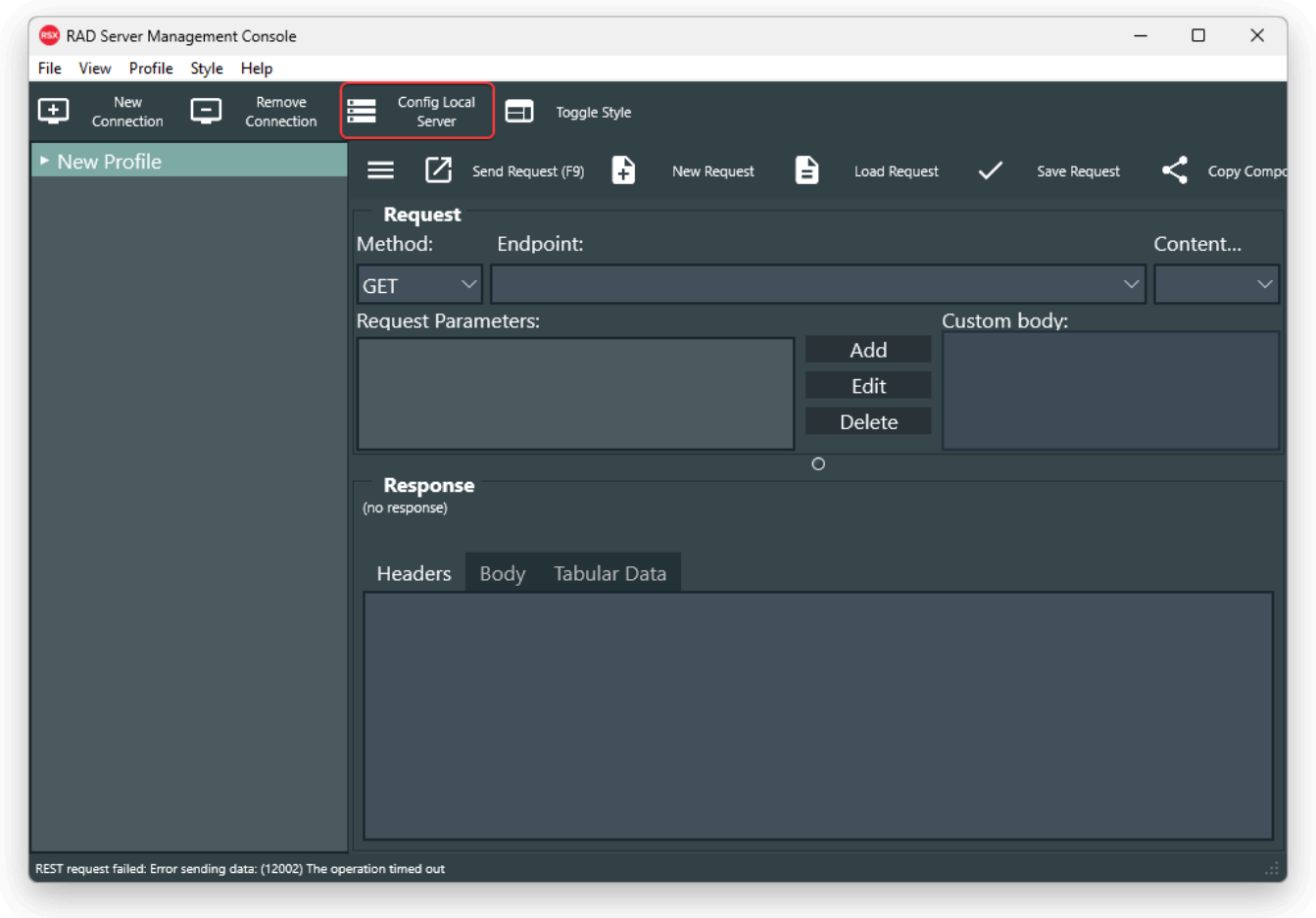
멀티-테넌시 지원을 여러분의 RAD Server 배포에서 활성화하려면:

1. `emsserver.ini` 설정 파일을 연다 (전형적으로 다음 경로에 위치함:
`C:\Users\Public\Documents\Embarcadero\EMS\emsserver.ini`)
2. `[Server.Tenants]` 구역 안에서, `MultiTenantMode=1` 로 지정한다.
3. RAD Server 인스턴스를 재시작한다. 그래서 변경들이 적용되도록 한다.

```
# 예시: emsserver.ini 구성(configuration)
[Server.Tenants]
MultiTenantMode=1
```

RAD Server Console을 사용하기

멀티-테넌시 활성화를 RAD Server Console을 사용해 할 수도 있다. 이 도구를 찾으려면, IDE 안에서 Tools > RAD Server Console 메뉴로 가면 된다. 콘솔이 열리면 “Config Local Server” 버튼을 클릭하고, “Locate”를 클릭해 .ini 파일 위치로 간다. 그러면 해당 구성(configuration)을 그래픽 UI 환경에서 볼 수 있고, 여러분이 구성을 변경할 수 있다. 즉, 멀티-테넌시뿐만 아니라 RAD Server의 나머지 파라미터들도 그래픽 UI에서 설정할 수 있다.



이 장의 뒷부분에서 더 자세히 살펴볼텐데, 일단 멀티-테넌시가 활성화되면, RAD Server로 전송되는 모든 요청들은 그 요청을 보내는 테넌트 ID 정보를 반드시 담고 있어야 한다. 그 정보는 자동으로 요청 컨텍스트 안에 주입된다. 그리고 그 정보는 `AContext.Tenant` 프로퍼티를 통해 접근할 수 있다.

테넌트 관리

RAD Server는 테넌트 관리를 위한 표준 엔드포인트를 제공하지 않는다. 그 대신, 전용 도구를 제공한다. 그래서 관리자들이 테넌트를 관리할 수 있도록 한다.

EMS Multi-Tenant Console

RAD Server에는 명령줄 유틸리티가 포함되어 있다. 이것은 테넌트 관리를 위해 특별히 설계된 것이다. 이 EMS Multi-Tenant Console은 이런 명령(command)들을 제공한다:

- `add` - 새 테넌트를 생성한다. 우리가 name(이름), Secret,(activation status(활성화 상태))를 지정한다.
- `view` - 특정 테넌트의 상세 정보를 표시한다
- `edit` - 테넌트의 프로퍼티들을 변경한다. name, Secret, activation status 등.

- **delete** - 테넌트 그리고 그와 관련된 모든 데이터를 삭제한다
- **list** - 시스템 안에 있는 모든 테넌트들을 나열한다

각 테넌트는 이런 것들을 가지고 있다:

- 고유 테넌트 ID (GUID 형식)
- 테넌트 이름 (tenant name, 사용자가 읽을 수 있는 식별자)
- 활성화 상태 (activation status)
- Secret 자격 증명 (secret credential, password)
- [옵션] 사용자 정의(custom) 메타데이터

이 콘솔을 사용해 테넌트를 관리하는 예시:

```
> add
tenant name: northregion
tenant secret: xxxxxxxx
Save changes? [y/n]? y

> list
Tenant ID: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX, Tenant Name: northregion, Is Active:
True, Has secret: True
```



참고

EMS Multi-Tenant Console은 다음 경로에 있다:

C:\Program Files (x86)\Embarcadero\Studio\XX.0\bin\EMSMultiTenantConsole.exe

이 도구에 대한 더 자세한 정보는 이 [DocWiki 링크](#)에서 확인할 수 있다.

멀티-테넌시 구현 모델들

RAD Server에 내장되어 있는 멀티-테넌시는 공유 데이터베이스 모델을 사용하고, 테넌트별 필터링을 적용한다. 하지만, 대안 방식(alternative approach)들도 가능하다.

공유 데이터베이스에서 테넌트 필터링을 적용

이 유형의 구현은 모든 테넌트들이 이런 것들을 공유한다:

- 단 하나의 데이터베이스 인스턴스
- 동일한 하나의 데이터베이스 스키마
- 공통 테이블들 (여기에서 테넌트 필터링을 적용)

데이터 격리를 실현하기 위해 각 테이블마다 **TenantID** 필드가 있다. 요청(request)이 발생하면, 해당 테넌트 컨텍스트가 확립된다. 인증 헤더(authentication header)들로부터 받아서 만들고, 모든 데이터베이스 쿼리를 필터링하는데 사용된다.

테이블 스키마 예시:

```
CREATE TABLE STORE_ITEMS
(
  "ITEMID"          INTEGER NOT NULL,
  "TENANTID"        VARCHAR(50) NOT NULL,  -- Tenant 식별자 필드
  "BARCODE"         VARCHAR(20) NOT NULL,
  "ITEMNAME"        VARCHAR(50) NOT NULL,
  "DESCRIPTION"     VARCHAR(300) NOT NULL,
  "QTY"             INTEGER NOT NULL,
  PRIMARY KEY ("ITEMID")
);
```

장점:

- 더 간단하게 배포하고 유지보수한다
- 비용 효과적으로 리소스를 활용한다
- RAD Server에서 내장되어 있는 지원 기능이다

고려 사항:

- 세심한 쿼리 필터링이 필요하다
- 리소스 한도를 테넌트들이 공유한다



참고

RAD Server의 emsserver.ib 데이터베이스는 이 패러다임을 따른다. 멀티-테넌시를 활성화할 경우, 모든 공유 테이블들은 tenant ID 필드를 가진다. 그래서 각 요청을 자동으로 검증한다.

테넌트별-데이터베이스

더 강력한 격리가 필요하거나 기존의 다중-데이터베이스 환경을 통합하는 경우, 여러분이 사용자 정의 테넌트별-데이터베이스 아키텍처를 구현할 수 있다. tenant ID를 이용해 특정 데이터베이스 이름을 매핑할 수 있고, 이 데이터베이스 매핑 정보를 설정 파일 안 또는 DB의 룩업(lookup) 테이블 안에 저장할 수도 있다.

장점:

- (테넌트들 사이에) 완전하게 데이터를 격리한다
- (테넌트별로) 독립적으로 확장하고 커스터마이징할 수 있다
- 더 강화된 보안 경계

고려 사항:

- 운영 복잡도가 증가한다
- 리소스 오버헤드가 증가한다
- 커스텀(사용자 정의) 연결 관리가 필요하다

이 방식은 테넌트별로 완전한 데이터 격리와 커스터마이징을 제공한다. 하지만, 운영 상의 복잡성이 커질 수 있다.

멀티-테넌트 RAD Server 애플리케이션을 개발하기

멀티-테넌트 애플리케이션을 RAD Server를 사용해 구축할 때, 여러분은 몇 가지 핵심 사항을 고려해야 한다. 이 고려사항들은 여러분이 어떤 데이터베이스 아키텍처를 선택하든 상관없다.

모든 리소스들은 테넌트 격리를 염두하고 설계되어야 한다. 비록 RAD Server는 여러 테넌트가 공통으로 접근할 수 있는 리소스/메서드를 정의하는 것도 허용하지만 말이다.

리소스 구현 안에서 테넌트 정보를 접근하기

여러분의 리소스 코드 안에서, 테넌트 정보에 접근할 수 있다. 엔드포인트 컨텍스트 (endpoint context)를 사용하면 된다. `TEndpointContext.Tenant` 프로퍼티는 현재 테넌트의 ID와 name에 접근하도록 한다.

```
procedure TCustomLogonResource.Get(const AContext: TEndpointContext;
                                   const ARequest: TEndpointRequest;
                                   const AResponse: TEndpointResponse);

begin
    var lTenantId := AContext.Tenant.Id;
    var lTenantName := AContext.Tenant.TenantName;
    // 이어서, 요청(request) 처리를 계속 진행한다...
end;
```

테넌트 컨텍스트는 자동으로 확립된다. 즉, 요청 인증(authentication) 시 `X-Embarcadero-Tenant-Id`와 `X-Embarcadero-Tenant-Secret` 헤더를 설정해 테넌트 컨텍스트를 확립한다. 그 컨텍스트는 여러분의 리소스 메서드들이 사용할 수 있다. 여러분이 추가로 코드를 작성하지 않아도 된다.

테넌트별-필터가 적용되는 리소스를 구현하기

구현 방식은 여러분이 선택한 데이터베이스 아키텍처에 따라 달라진다:

공유 데이터베이스 방식

공유 데이터베이스 모델인 경우, 데이터 필터링은 여러분의 쿼리 안에 있는 tenant ID를 기준으로 처리된다:

```
procedure TStoreResource.GetItems(const AContext: TEndpointContext;
                                   const ARequest: TEndpointRequest;
```

```

                                const AResponse: TEndpointResponse);
begin
    // 테넌트 필터링은 데이터를 추출할 때 적용된다
    var LTenantId := AContext.Tenant.Id;
    FDQueryGetItems.Params.ParamByName('TenantId').AsString := LTenantId;
    FDQueryGetItems.Open;
    // 이어서, 요청(request) 처리를 계속 진행한다...
end;

```

테넌트별-데이터베이스 방식

테넌트별로 데이터베이스를 분리한 경우, 동적으로 알맞은 데이터베이스 연결을 선택해야 한다. 아래 예시에서는 `TDictionary<string, TFDBConnection>`을 사용한다. 그 이름은 `FTenantConnections` 이다. 그 안에 연결들을 저장해 두고 재사용한다. 그러면 이어지는 요청들을 위한 연결을 더 쉽게 찾을 수 있다. 매번 새 연결을 생성하지 않아도 된다.

```

// 동적 연결 관리로 멀티 테넌트 데이터베이스들을 다루는 예시
function GetConnectionForTenant(const ATenantId: string): TFDBConnection;
var
    LConnection: TFDBConnection;
begin
    // 이 테넌트를 위한 연결(connection)을 열거나 생성한다
    if not FTenantConnections.TryGetValue(ATenantId, LConnection) then
    begin
        LConnection := TFDBConnection.Create(nil);
        // 이 특정 테넌트를 위한 연결을 구성한다
        LConnection.Params.Database := Format('Tenant_%s.ib', [ATenantId]);
        // 기타 연결 파라미터들...
        FTenantConnections.Add(ATenantId, LConnection);
    end;
    Result := LConnection;
end;

// 리소스 안에서, 테넌트별 고유 연결을 사용하기
procedure TMyResource.Get(const AContext: TEndpointContext;
                          const ARequest: TEndpointRequest;
                          const AResponse: TEndpointResponse);
begin
    Query.Connection := GetConnectionForTenant(AContext.Tenant.Id);
    // 이어서, 요청(request) 처리를 계속 진행한다...
end;

```

익명 테넌트 리소스들

몇몇 리소스들 또는 특정 엔드포인트들은 여러 테넌트들이 접근하거나 또는 테넌트 인증 없이 접근해야 할 수 있다. 이 경우, `[AllowAnonymousTenant]` 애트리뷰트를 사용한다.

리소스-수준에서 익명으로 접근

이 애트리뷰트가 리소스 수준에서 정의되면, 해당 리소스 전체는 모든 테넌트들이 사용할 수 있도록 제공된다.

Delphi

```
[ResourceName('settings')]
[AllowAnonymousTenant]
TSettingsResource = class(TDataModule)
    // 이 리소스를 접근할 때는 테넌트 인증이 없어도 된다
end;
```

C++

```
static void Register()
{
    std::unique_ptr<TEMSResourceAttributes> attributes(new TEMSResourceAttributes());
    attributes->ResourceName = "settings";
    attributes->ResourceTenantAuthorization["settings"] = TTenantAuthorization::SkipAll;
    RegisterResource(__typeid(TEMSResource1), attributes.release());
}
```

메서드-수준에서 익명으로 접근

동일한 애트리뷰트는 특정 메서드에서 사용할 수 있다. 그러면 리소스 안에서 그 엔드포인트에만 익명 접근이 된다.

Delphi

```
[ResourceName('settings')]
TTestResource1 = class(TDataModule)
published
    [ResourceSuffix('./user')]
    procedure GetUser(const AContext: TEndpointContext;
                     const ARequest: TEndpointRequest;
                     const AResponse: TEndpointResponse);
    [ResourceSuffix('./general')]
```

```
[AllowAnonymousTenant] // 이 엔드포인트 설정(즉 general)은 테넌트를 가리지 않는다
procedure GetGeneral(const AContext: TEndpointContext;
                     const ARequest: TEndpointRequest;
                     const AResponse: TEndpointResponse);

end;
```

C++

```
static void Register()
{
    std::unique_ptr<TEMSResourceAttributes> attributes(new TEMSResourceAttributes());
    attributes->ResourceName = "settings";
    attributes->ResourceSuffix["GetUser"] = "./user";
    attributes->ResourceSuffix["GetGeneral"] = "./general";
    attributes->EndPointTenantAuthorization["./general"] =
        TTenantAuthorization::SkipAll;
    RegisterResource(__typeinfo(TEMSResource1), attributes.release());
}
```

클라이언트-측 구현

RAD 스튜디오 애플리케이션에서 TEMSProvider 컴포넌트를 사용해 접근하기

RAD Server 인스턴스를 RAD 스튜디오 애플리케이션에서 연결하는, 가장 편한 방법은 **TEMSProvider** 컴포넌트를 사용하는 것이다. 클라이언트 측에서, 여러분은 테넌트 자격 증명을 사용자 인증과 함께 제공해야 한다.

```
// 테넌트 자격 증명을 수립한다
EMSPProvider.TenantId := 'your-tenant-id-guid';
EMSPProvider.TenantSecret := 'tenant-password';

// 그리고 나서, 사용자 인증을 해당 테넌트 컨텍스트 안에서 수행한다
BackendAuth.UserName := 'username';
BackendAuth.Password := 'password';
BackendAuth.Login;
```

API 엔드포인트를 통해 액세스하기

이 경우, 특정 테넌트에 해당하는 tenant ID와 tenant Secret을 반드시 각 요청의 헤더 안에 정의해야 한다. 필요한 헤더 파라미터들은 다음과 같다:

X-Embarcadero-Tenant-Id

X-Embarcadero-Tenant-Secret

**참고**

멀티-테넌시 환경 안에, 이 두 파라미터가 반드시 각 요청 헤더마다 들어 있어야 한다. 그래야 그 테넌트를 올바르게 식별하고 검증한다. 단, [AllowAnonymousTenant] 애트리뷰트가 엔드포인트 안에 정의되어 있다면 그렇지 않다.

TEMSPROVIDER 컴포넌트 (멀티-테넌트 RAD Studio 클라이언트 앱 안)

TEMSPROVIDER 컴포넌트는 멀티-테넌트 RAD Server와 상호작용하는 클라이언트 애플리케이션들의 중심이다. 이 컴포넌트가 관리하는 것들:

1. **Server Connection(서버 연결):** RAD Server의 호스트 이름과 포트
2. **Tenant Identity(테넌트 신원):** TenantID와 TenantSecret. 어느 테넌트 컨텍스트를 사용할 지를 식별
3. **Authentication(인증):** 사용자 자격 증명 (그 테넌트 컨텍스트 안에서)

전형적인 구현:

```
procedure TClientDM.SetHost(const AHost: string; APort: Integer);
begin
    EMSPROVIDER.URLHost := AHost;
    EMSPROVIDER.URLPort := APort;
end;

procedure TClientDM.SetTenant(const ATenantId, ATenantSecret: string);
begin
    EMSPROVIDER.TenantId := ATenantId;
    EMSPROVIDER.TenantSecret := ATenantSecret;
end;

procedure TClientDM.SetUserCredentials(const AUserName, APassword: string);
begin
    BackendAuth.UserName := AUserName;
    BackendAuth.Password := APassword;
    BackendAuth.Authentication := TBackendAuthentication.Default;
end;
```

일단 구성이 되면, 이 TEMSPROVIDER를 통하는 모든 요청들은 해당 테넌트 상세 정보를 담게 된다. 그래서 사용자들이 오직 자신들의 테넌트를 위한 데이터에만 접근할 수 있도록 보장한다.

(테넌트 안의) 사용자 및 그룹 관리를 구현하기

멀티-테넌트 애플리케이션들은 전형적으로 각 테넌트 안에서 서로 다른 사용자 역할들이 필요하다. RAD Server 안에는 이를 지원하도록 내장된 사용자 및 그룹 시스템 관리 기능이 있다. 그 기능은 이미 이전 장들에서 살펴봤다.

기능 면에서, 동일한 기능들과 도구들이 제공되므로 사용자와 그룹을 관리할 수 있다. 하지만, 이것들이 그 테넌트의 컨텍스트에만 독점적으로 연결되도록 할 수 있다.

예를 들어, 새 사용자를 하나 만들고 싶다고 하자. 프로그래밍 방식으로는 `TEMSInternalAPI` 클래스를 사용하고, 그것의 인스턴스를 생성할 때 컨텍스트 안에 명시하면 된다. 또는 제공되는 엔드포인트들을 사용할 수도 있다: 만약 `POST /users`를 사용한다면, 그 사용자는 헤더 파라미터 안에 정의된 그 테넌트 안에 생성된다.

샘플 애플리케이션: RAD Server 멀티-테넌트 데모

완전한 멀티-테넌시 예시가 작동하는 것을 직접 보려면, RAD Server Multi-Tenant Demo 애플리케이션을 참조하면 된다. 이 데모는 RAD 스튜디오 안에 포함되어 있다. 그 경로는:

`C:\Users\Public\Documents\Embarcadero\Studio\XX.0\Samples\Object Pascal\Database\EMS\Multi-Tenancy Demo`

이 샘플은 장난감 체인점의 구현 사례를 보여준다. 여기에서:

- 각 매장 위치는 분리된 개별 테넌트로 작동한다
- 매장 직원들은 서로 다른 역할을 가진다 (관리자, 계산원)
- 재고는 분리되어 각 매장별로 관리된다
- (매장들 사이에서) 데이터가 완전하게 격리되어 있다

이 샘플에 대한 자세한 정보는 [RAD Server Multi-Tenant Application 문서](#)에서 확인할 수 있다.

결론

RAD Server의 멀티-테넌시 지원은 확장 가능하고 보안성이 뛰어난 애플리케이션들을 구축하는 강력한 기반을 제공한다. 그래서 단 한 곳에 배포한 애플리케이션이 여러 개의 클라이언트 기관(organization)들에게 서비스를 제공할 수 있다. 이 장에서 설명한 패턴들과 관례들을 따라하면, 여러분은 효율적인 멀티-테넌트 아키텍처를 만들 수 있다. 그래서 엄격한 데이터 격리를 유지하면서도 중앙 집중식 배포와 관리가 주는 이점들을 누릴 수 있다.

명심할 점이 있다. 성공적인 멀티-테넌트 애플리케이션들은 균형있게 효율성(공유 리소스들)과 보안 요구 사항(테넌트들 사이의 강력한 경계 유지)을 다룬다. RAD Server는 이러한 균형을 효과적으로 달성하는 데 필요한 도구들을 제공한다.

17

엔드포인트를 외부 클래스에 매핑하기

RAD Server에는 사용자 정의(custom) 리소스 API를 위한 기능이 있다. 그래서 요청(request) 처리를 리소스 모듈 안에 있는 Delphi 필드, 즉 사용자 정의 엔드포인트 제공자 클래스/컴포넌트에게 위임할 수 있다.

요청 처리는 오브젝트 타입인 리소스 모듈 필드에게 위임될 수 있다. 오브젝트 타입으로는 (TComponent 자손 등) 어떤 클래스든지 상관없다. 그저 IEMSEndpointPublisher 인터페이스를 구현하는 클래스면 된다. 그런 클래스는 엔드포인트 퍼블리셔 클래스가 된다. 기본 설정에 따르면, 모든 클래스 엔드포인트들은 리소스 접미사(resource suffix)를 가지게 된다. 그리고 그것은 그 엔드포인트들을 담고 있는 리소스 모듈 상의 필드 이름과 동일하다. 리소스 접미사를 여러분이 덮어쓸 수도 있다. 애트리뷰트를 그 리소스 모듈 필드에서 명시하면 된다:

- ResourceName – 필드 이름을 덮어쓴다.
- ResourceSuffix – 리소스 접미사를 덮어쓴다.

클래스 엔드포인트는 **ResourceSuffix** 애트리뷰트를 사용해 그것의 리소스들이 어떻게 관리되는 지를 커스터마이징할 수 있다. 만약 접미사가 "/"로 시작하면, 기본 리소스 즉 리소스 모듈 필드에서 정의된 접미사 뒤에 붙는다. 리소스 모듈 엔드포인트들에 붙일 수 있는 유효한 애트리뷰트라면 무엇이든 클래스 엔드포인트들에서 사용할 수 있다. 동일한 애트리뷰트가 여러 곳에 정의되어 있는 경우, 우선순위가 더 높은 것이 적용된다. 우선순위는 오름차순이다. 즉 가장 낮은 것이 가장 우선 순위가 높다. 따라서, 더 나중에 정의된 것이 더 먼저 정의된 것을 덮어쓴다:

- 클래스 엔드포인트
- 리소스 모듈 필드 (어떤 애트리뷰트는 첫 번째 인자로 클래스 엔드포인트 이름을 받을 수도 있다)

- TEMSResourceAttributes (리소스 모듈을 등록하는 곳에 명시된다). TEMSResourceAttributes 호출을 위해 사용되는 이름은 <리소스 모듈 필드 이름>.<클래스 엔드포인트 이름>이다.

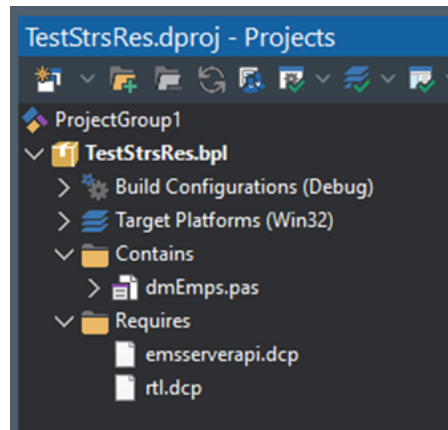
**경고**

이 매핑 기능이 크게 의존하는 기술은 RTTI(Run-Time Type Information)다. 따라서 이 방식은 현재 오직 델파이(Delphi)에서만 사용할 수 있다.

RAD Server 패키지 애플리케이션을 만들기

RAD Server Package 마법사를 사용하여 패키지와 리소스 모듈을 새로 만든다. 리소스 이름은 “Test”라고 지정하고 해당 모든 엔드포인트들을 추가하도록 옵션을 선택한다.

Delphi:



RAD Server 애플리케이션 프로젝트

TComponent의 자손 클래스를 하나 만든다. 이름을 TEmpsProvider라고 지정하고, 이 클래스가 인터페이스 즉 IEMSEndpointPublisher를 구현하게 한다. 이 클래스의 선언을 작성한다. 이때 ResourceSuffix들을 명시한다. 그리고 이 TEmpsProvider 엔드포인트를 위한 구현을 만든다. class var를 하나 추가해 JSONArray를 담을 수 있도록 한다. Test 리소스의 public 구역에 TEmpsProvider용 필드를 선언한다. 아래 코드를 참고해 interface 구역과 implementation 구역을 완성한다. TEmpsProvider와 Test 리소스를 위한 interface 구역의 선언 코드:

```
type
  TEmpsProvider = class(TComponent, IEMSEndpointPublisher)
  private
    class var FArray: TJSONArray;
  private
    function GetItemByID(ARequest: TEndpointRequest): Integer;
    class constructor Create;
    class destructor Destroy;
  public
```

```

procedure Get(const AContext: TEndpointContext;
              const ARequest: TEndpointRequest;
              const AResponse: TEndpointResponse);
[ResourceSuffix('./{id}')]
procedure GetItem(const AContext: TEndpointContext;
                  const ARequest: TEndpointRequest;
                  const AResponse: TEndpointResponse);
[ResourceSuffix('./{id}')]
procedure Put(const AContext: TEndpointContext;
              const ARequest: TEndpointRequest;
              const AResponse: TEndpointResponse);
procedure Post(const AContext: TEndpointContext;
               const ARequest: TEndpointRequest;
               const AResponse: TEndpointResponse);
[ResourceSuffix('./{id}')]
procedure Delete(const AContext: TEndpointContext;
                 const ARequest: TEndpointRequest;
                 const AResponse: TEndpointResponse);

end;

[ResourceName('test')]
TTestResource1 = class(TDataModule)
public
    Emps: TEmpsProvider;
    constructor Create(AOwner: TComponent);
published
    procedure Get(const AContext: TEndpointContext;
                  const ARequest: TEndpointRequest;
                  const AResponse: TEndpointResponse);
end;

```

TEmpsProvider와 Test 리소스를 위한 implementation 구역의 구현 코드:

```

uses
    System.JSON.Builders, System.JSON.Writers, System.JSON.Readers;

{ TEmpsProvider }
class constructor TEmpsProvider.Create;
var
    oWriter: TJsonObjectWriter;
    oBuilder: TJSONArrayBuilder;
begin
    oWriter := TJsonObjectWriter.Create(False);
    oBuilder := TJSONArrayBuilder.Create(oWriter);
try

```

```

oBuilder
  .BeginArray
    .BeginObject
      .Add('id', '1')
      .Add('first_name', 'John')
      .Add('second_name', 'Smith')
    .EndObject
    .BeginObject
      .Add('id', '2')
      .Add('first_name', 'Poll')
      .Add('second_name', 'Scott')
    .EndObject
    .BeginObject
      .Add('id', '3')
      .Add('first_name', 'Harris')
      .Add('second_name', 'Hill')
    .EndObject
  .EndArray;
FArray := oWriter.JSON as TJSONArray;
finally
  oBuilder.Free;
  oWriter.Free;
end;
end;

class destructor TEmpsProvider.Destroy;
begin
  FArray.Free;
end;

function TEmpsProvider.GetItemByID(ARequest: TEndpointRequest): Integer;
var
  i: Integer;
begin
  Result := -1;
  for i := 0 to FArray.Count - 1 do
    if (FArray.Items[i] as TJSONObject).GetValue('id').Value =
      ARequest.Params.Values['id'] then
      Exit(i);
  end;
end;

procedure TEmpsProvider.Get(const AContext: TEndpointContext;
                           const ARequest: TEndpointRequest;
                           const AResponse: TEndpointResponse);
begin
  AResponse.Body.SetValue(FArray, False);
end;

```

```

procedure TEmpsProvider.GetItem(const AContext: TEndpointContext;
                                const ARequest: TEndpointRequest;
                                const AResponse: TEndpointResponse);

var
  i: Integer;
begin
  i := GetItemByID(ARequest);
  if i >= 0 then
    AResponse.Body.SetValue(FArray.Items[i], False)
  else
    AResponse.RaiseNotFound('Item is not found');
end;

procedure TEmpsProvider.Put(const AContext: TEndpointContext;
                             const ARequest: TEndpointRequest;
                             const AResponse: TEndpointResponse);

var
  i: Integer;
begin
  i := GetItemByID(ARequest);
  if i >= 0 then
    FArray.Remove(i);
  FArray.Add(ARequest.Body.GetObject.Clone as TJSONObject);
end;

procedure TEmpsProvider.Post(const AContext: TEndpointContext;
                              const ARequest: TEndpointRequest;
                              const AResponse: TEndpointResponse);

var
  i: Integer;
begin
  i := GetItemByID(ARequest);
  if i >= 0 then
    AResponse.RaiseDuplicate('Item already exists')
  else
    FArray.Add(ARequest.Body.GetObject.Clone as TJSONObject);
end;

procedure TEmpsProvider.Delete(const AContext: TEndpointContext;
                                const ARequest: TEndpointRequest;
                                const AResponse: TEndpointResponse);

var
  i: Integer;
begin
  i := GetItemByID(ARequest);
  if i >= 0 then

```

```

        FArray.Remove(i)
    else
        AResponse.RaiseNotFound('Item is not found');
    end;

constructor TTestResource1.Create(AOwner: TComponent);
begin
    inherited;
    Emps := TEmpsProvider.Create(Self);
end;

{ TTestResource }

procedure TTestResource1.Get(const AContext: TEndpointContext;
                             const ARequest: TEndpointRequest;
                             const AResponse: TEndpointResponse);
begin
    // Sample code
    AResponse.Body.SetValue(TJSONString.Create('test'), True);
end;

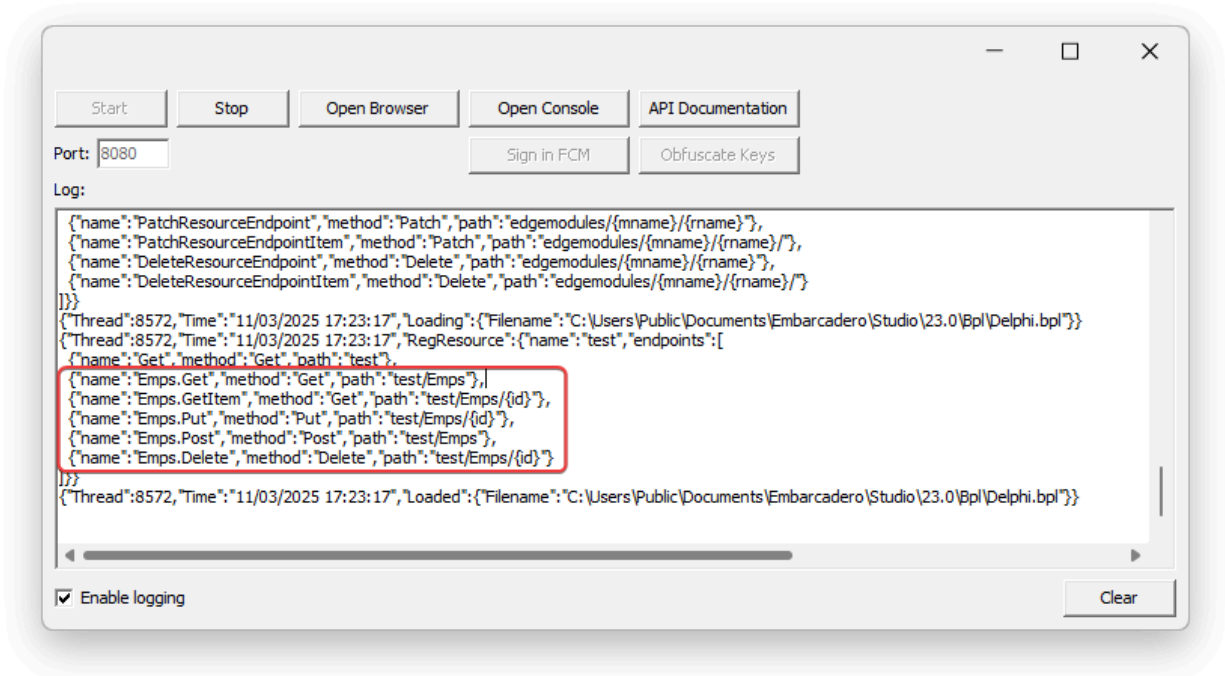
procedure Register;
begin
    RegisterResource(TypeInfo(TTestResource1));
end;

initialization
    Register;
end.

```

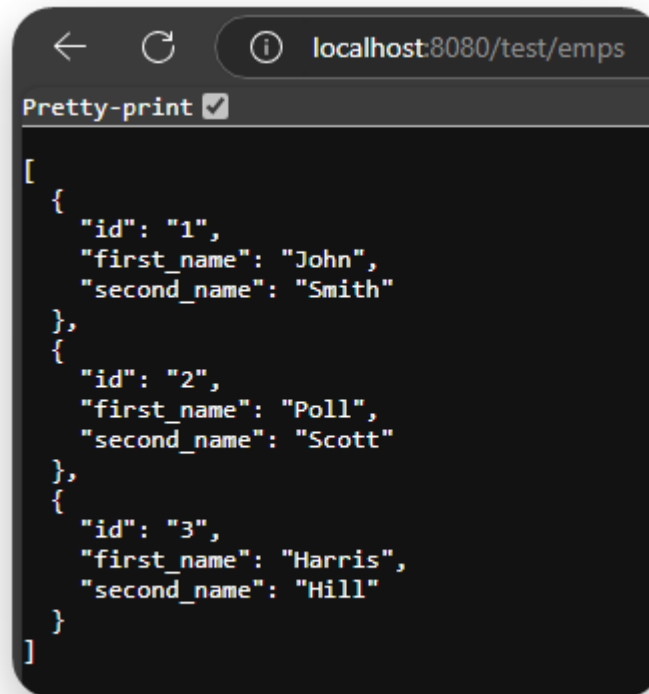
만든 RAD Server 애플리케이션을 테스트하기

이 RAD Server 애플리케이션을 컴파일하고 실행한다. 로그가 다음과 같이 표시된다:

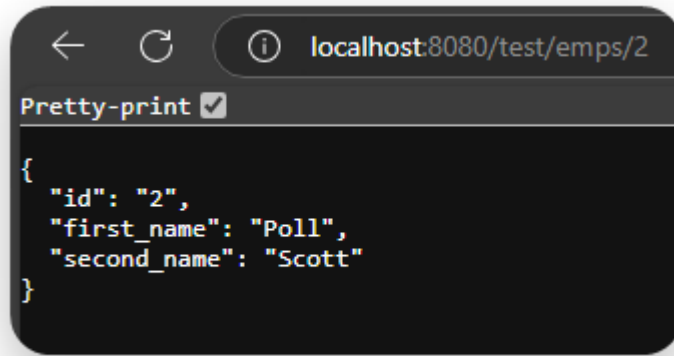


RAD Server 애플리케이션 로그를 보면, Test 리소스와 "Emps" 접두사가 붙어 있다

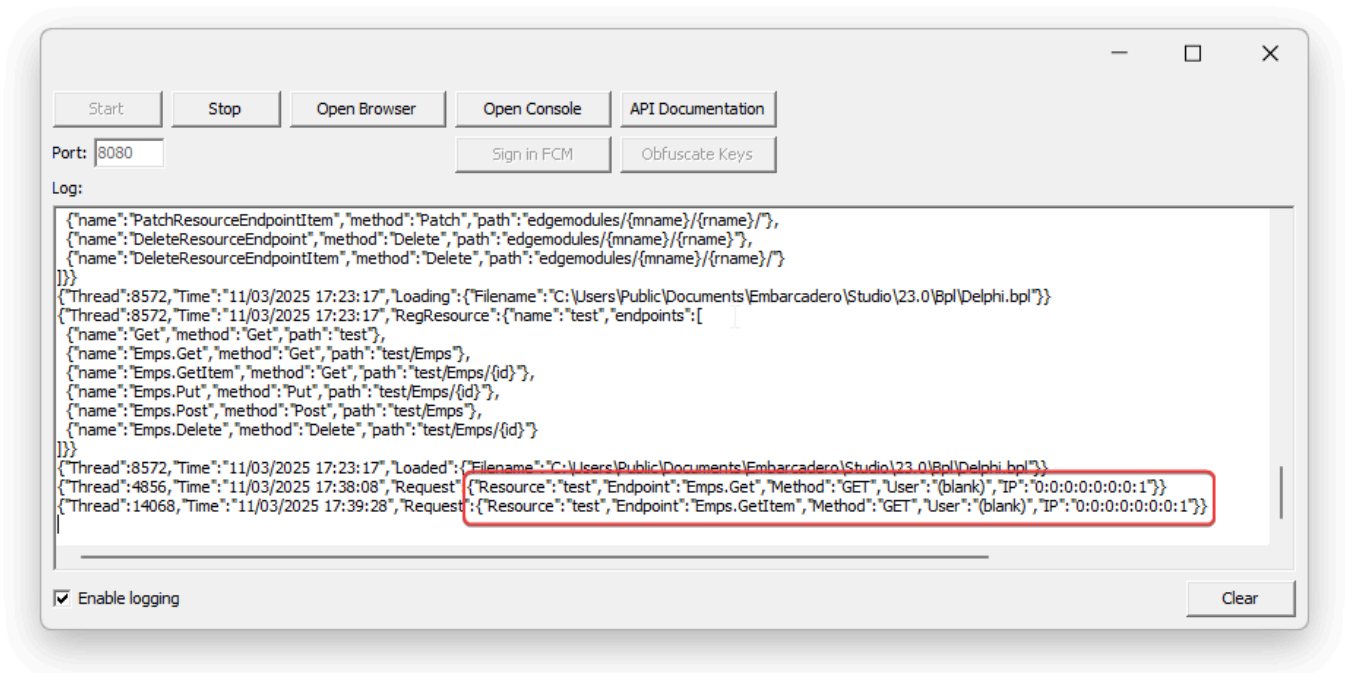
위 로그에서 잘 봐줄 것이 있다. 리소스 이름이 Test이고, 각 엔드포인트마다 접두사로 “Emps”가 붙어 있다. 지금부터 처리할 요청들은 Emps 오브젝트에 들어있는 TEmpsProvider의 메서드들이 처리하게 된다:



<http://localhost:8080/test/emp> – TEmpsProvider.Get 메서드를 호출해, 배열의 모든 요소들을 반환한다



http://localhost:8080/test/emps/2 – TEmpsProvider.GetItem 메서드를 호출한다. 그래서 배열의 두 번째 요소를 반환한다



RAD Server 애플리케이션 로그는 Emps.Get 및 Emps.GetItem이 실행되었음을 보여준다.

참고 자료

- 이 장을 위한 코드 예시는 [GitHub 저장소](#)의 “17 - Mapping Endpoints to External Classes”에 있다.
- RAD Server 컴포넌트들을 사용하기:
http://docwiki.embarcadero.com/RADStudio/en/Using_RAD_Server_Components

18

리소스 인터셉터를 통해 RAD Server를 확장하기

RAD Server는 언제나 API 개발 과정을 단순 명료하게 하는 것이 목적이다. 그러면서도 견고한 기능들을 바로 쓸 수 있게 담아서 제공한다. 새로 추가된 `IEMSResourceInterceptor` 인터페이스는 그 철학을 바탕으로 생겨났다. 이것은 사용자가 지정하는(custom) 미들웨어-식 처리를 여러분의 데이터모듈에게 추가할 수 있도록 한다. 여러분은 이런 처리가 실제로 필요한 경우에 사용하면 된다. 그래서 그 단순한 경우들을 복잡하게 만들지 않는다.

리소스 인터셉터들을 이해하기

이 인터셉터 기능은 간단한 인터페이스 하나를 통해 작동한다. 이 인터페이스를 여러분의 데이터모듈 또는 클래스에 구현할지 말지는 여러분이 선택하면 된다. RAD Server는 요청을 받아 처리할 때, 여러분의 데이터모듈이 인터페이스인 `IEMSResourceInterceptor`를 구현하고 있는지 확인한다. 만약 그렇다면, 이 프레임워크는 여러분의 요청 인터셉터 메서드들을 호출한다. 그 시점은 그 요청 생명 주기 안 특정 시점들 중 하나다.

```
// IEMSResourceInterceptor 인터페이스를 구현하는 예시
TMyDataModule = class(TDataModule, IEMSResourceInterceptor)
procedure BeforeRequest(const AContext: TEndpointContext;
                        const ARequest: TEndpointRequest;
                        const AResponse: TEndpointResponse;
                        var AHandled: Boolean);
procedure AfterRequest(const AContext: TEndpointContext;
                       const ARequest: TEndpointRequest;
```

```
const AResponse: TEndpointResponse);
```

BeforeRequest 메서드는 여러분의 엔드포인트 메서드가 동작하기 전에 실행된다. 따라서 여러분이 데이터를 준비하고, 요청을 검증하고, 심지어 처리를 더 일찍 끝내기에 좋은 기회다. **AHandled := True** 라고 지정하면 된다. **AfterRequest** 메서드는 여러분의 엔드포인트가 처리를 완료한 후에 실행된다. 따라서 여러분이 깨끗이 정리하기, 로그를 기록하기, 응답을 변경하기 등을 하기에 가장 좋은 곳이다.

RAD Server는 각 요청마다 완전히 새로운 데이터모듈(또는 클래스) 인스턴스를 생성한다. 그러므로, 여러분은 각 요청별 데이터를 인스턴스의 필드들 안에 안전하게 저장할 수 있다. 스레드 안전성을 걱정하지 않아도 된다.

표준 RAD Server 기능들 만으로 부족할 때

RAD Server에는 뛰어난 내장 기능들이 있다: 사용자 인증, 분석, 로그 기록, 멀티-테넌트 지원 등등. 이 기능들은 일반적인 API 요구사항 대부분을 쉽게 처리한다. 하지만, 때로는 제공되는 기능 이상이 필요한 경우가 있다. 그리고 바로 이 지점에서 인터셉터들이 진가를 발휘한다.

인터셉터들은 RAD Server의 기존 기능들을 바꾸는 것이 아니라 보완한다는 사실을 이해하는 것이 핵심이다. 일종의 탈출구라고 보면 된다. 즉 표준 방식이 특정 요구사항들에 적합하지 않은 상황에 사용하면 된다.

이 장의 나머지 부분에서, 우리는 다양한 상황들을 살펴본다. 그래서, **IEMSResourceInterceptor**가 여러분의 개발을 훨씬 더 쉽게 만들어 준다는 것을 알 수 있을 것이다. 하지만, 이런 유연성을 고려하면, 그 가능성은 그저 이 예시들에만 국한되지 않고 훨씬 더 크다는 점을 이해하고 보기 바란다.

내장된 방식을 넘어서는 멀티-테넌시

RAD Server에 내장된 멀티-테넌시 기능은, 여러분이 다수의 테넌트들을 가지고 있고 단 하나의 데이터베이스를 공유하면서 데이터 격리를 알맞게 적용하는 환경에서 매우 잘 작동한다. 하지만, 만약 여러분의 아키텍처 상 테넌트별로 개별 데이터베이스를 사용해야 한다면 어떻게 할까?

테넌트별-데이터베이스 (Database-Per-Tenant) 아키텍처를 구현하기

애플리케이션이 있는데, 규제 요구 사항들 또는 고객의 요청으로 인해 각 테넌트마다 완전히 격리된 데이터베이스가 필요한 경우를 생각해 보자. 인터셉터를 사용해, 여러분은 깔끔하게 구현할 수 있다:

```
procedure TMyDataModule.BeforeRequest(const AContext: TEndpointContext;
                                     const ARequest: TEndpointRequest;
                                     const AResponse: TEndpointResponse;
                                     var AHandled: Boolean);

var
  TenantID: string;
  ConnectionString: string;
begin
```

```
// 요청으로부터 테넌트를 추출한다
TenantID := ExtractTenantFromRequest(ARequest);

if not IsValidTenant(TenantID) then
begin
    AResponse.StatusCode := 404;
    AHandled := True;
    Exit;
end;

// 이 특정 테넌트를 위한 데이터베이스 연결을 구성한다
ConnectionString := GetTenantConnectionString(TenantID);
ConfigureDatabaseConnection(ConnectionString);

// 테넌트 정보를 저장한다. 엔드포인트 메서드들 안에서 사용하기 위해서다
FCurrentTenantID := TenantID;
end;
```

이 방식은 테넌트 탐지와 데이터베이스 구성을 중앙 집중화한다. 그래서 여러분의 리소스 안에 있는 모든 엔드포인트들이 각자 자동으로 올바른 테넌트의 데이터 위에서 작동하도록 보장한다. 엔드포인트 메서드 안에 코드를 추가로 작성하지 않아도 된다.

고급 테넌트 격리 시나리오

단순히 데이터베이스를 전환하는 것을 넘어, 인터셉터들은 정교한 테넌트 격리 패턴을 구현할 수 있다. 여러분은 테넌트마다 다른 암호화 키를 사용하기, 테넌트 전용 기능 플래그(Feature flag)들, 구독 등급에 따른 접속 빈도 차등화 등이 필요할 수 있다. 인터셉터 패턴은 이 모든 상황들을 다룰 수 있다. 왜냐하면 설정 로직이 먼저 실행된 이후에 모든 비즈니스 로직이 실행되기 때문이다.

사용자 정의 인증과 권한 부여

RAD Server에 내장된 인증 시스템은 많은 사용 사례들을 커버한다. 하지만, 현대 애플리케이션들은 종종 외부 ID 공급자들과 통합, 사용자가 지정한 검증, 복잡한 역할 기반 접근 제어(RBAC, Role-Based Access Control)가 필요한 경우가 많다. 이런 제어는 표준 사용자/그룹 모델을 넘어서는 것들이다.

엔터프라이즈 ID 통합

엔터프라이즈 애플리케이션들은 이미 가지고 있는 사내 인증(identity) 공급자들과 인증 시스템들과 통합되어야 하는 경우가 많다. 인터셉터 하나가 이러한 통합 작업을 쉽게 처리할 수 있다:

```
procedure TSecureDataModule.BeforeRequest(const AContext: TEndpointContext;
    const ARequest: TEndpointRequest;
    const AResponse: TEndpointResponse;
```

```

var AHandled: Boolean);

var
  Token: string;
  UserInfo: TUserInfo;
begin
  Token := ExtractToken(ARequest);

  if not ValidateWithExternalProvider(Token, UserInfo) then
  begin
    AResponse.StatusCode := 401;
    AHandled := True;
    Exit;
  end;

  // 사용자 컨텍스트를 저장한다. 엔드포인트 메서드들을 위해서다
  FCurrentUser := UserInfo;
  FUserPermissions := GetUserPermissions(UserInfo.UserID);
end;

```

이 패턴을 사용하면 여러분의 엔드포인트 메서드 안에 RAD Server의 단순함을 그대로 유지할 수 있다. 그러면서도, 정교한 인증 요구 사항들을 지원한다. 그래서 표준 표준 프레임워크 패턴과 잘 맞지 않는 요구 사항을 해소한다.

더 수준이 높은 로그 기록과 모니터링

RAD Server에는 견고한 로깅 기능이 포함되어 있다. 하지만, 몇몇 애플리케이션은 특화된 모니터링, 컴플라이언스 감사, (Application Insights, New Relic 등) 외부 모니터링 시스템, 커스텀 메트릭 플랫폼들과 통합이 필요하다.

광범위하게 요청을 추적하기

인터셉터 하나가 표준 로깅을 넘어서는 상세한 요청 추적 기능을 구현할 수 있다:

```

procedure TMonitoredDataModule.BeforeRequest(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest;
  const AResponse: TEndpointResponse;
  var AHandled: Boolean);

begin
  FRequestStartTime := Now;
  FRequestID := GenerateRequestID;

  // 일관된 구조로 정리된 요청 데이터를 로그에 기록한다
  LogRequestStart(FRequestID, ARequest.Method, ARequest.PathInfo,
    ExtractUserID(ARequest), ARequest.ContentLength);

  // 성능 모니터링을 시작한다
  StartPerformanceCounters;

```

```

end;

procedure TMonitoredDataModule.AfterRequest(const AContext: TEndpointContext;
                                           const ARequest: TEndpointRequest;
                                           const AResponse: TEndpointResponse);

var
    Duration: Integer;
    Metrics: TPerformanceMetrics;
begin
    Duration := MillisecondsBetween(Now, FRequestStartTime);
    Metrics := CollectPerformanceMetrics;

    // 상세한 응답 데이터를 로그에 기록한다
    LogRequestComplete(FRequestID, AResponse.StatusCode, Duration,
        AResponse.ContentLength, Metrics);

    // 파악된 성능 지표들을 외부 모니터링 시스템에게 보낸다
    SendMetricsToMonitoringSystem(FRequestID, Duration, Metrics);
end;

```

이 방식은 API가 실제로 어떻게 동작하는지를 여러분이 완전히 투명하게 들여다볼 수 있게 한다. 그러면서도 여러분의 엔드포인트 메서드들은 비즈니스 로직에만 집중하게 한다.

데이터 검증과 변환

복잡한 애플리케이션들은 정교한 입력값 검증, 데이터 변환, 응답 서식 반영 등이 필요한 경우가 많다. 이런 것들은 여러분이 개별 엔드포인트 메서드 안에 구현하려고 하는 로직을 크게 벗어난다.

중앙 집중식 입력 처리

인터셉터 하나가 모든 엔드포인트에 걸쳐 복잡한 검증 상황들을 다룰 수 있다:

```

procedure TValidatedDataModule.BeforeRequest(const AContext: TEndpointContext;
                                           const ARequest: TEndpointRequest;
                                           const AResponse: TEndpointResponse;
                                           var AHandled: Boolean);

var
    ValidationResult: TValidationResult;
begin
    // 테넌트별 고유한 검증 규칙을 적용한다
    ValidationResult := ValidateRequestForTenant(ARequest, AContext.Tenant.id);

    if not ValidationResult.IsValid then
    begin
        AResponse.StatusCode := 400;
    end;
end;

```

```

    AResponse.JSONValue := CreateValidationErrorResponse(ValidationResult);
    AHandled := True;
    Exit;
end;

// 입력 데이터를 변형하고 안전하도록 정제한다
TransformRequestData(ARequest);

// 검증된 데이터를 저장한다. 엔드포인트에서 사용하기 위해서다
FValidatedInput := ValidationResult.ProcessedData;
end;

```

이 패턴은 일관된 데이터 품질을 전체 API에 걸쳐 보장한다. 그러면서도 검증 로직을 중앙에서 관리하고 유지보수성을 향상한다.

성능 최적화와 캐싱(caching)

RAD Server는 우수한 성능을 그 자체에서 제공한다. 하지만, 특정 애플리케이션들은 커스텀 캐싱 전략들, 특정 응답 압축, 특정 요청 패턴들에 대한 조기 종료 로직 등을 통해 이득을 얻기도 한다.

지능형 응답 캐싱

인터셉터 하나가 정교한 캐싱을 구현할 수 있다. 그래서 여러 요소들을 고려하거나, 널리 사용되는 캐싱 시스템들 (예: [Redis](#), [Memcached](#))과 통합하거나, 심지어, 인-메모리 솔루션들을 구현할 수도 있다:

```

procedure TCachedDataModule.BeforeRequest(const AContext: TEndpointContext;
                                           const ARequest: TEndpointRequest;
                                           const AResponse: TEndpointResponse;
                                           var AHandled: Boolean);

var
    CacheKey: string;
    CachedResponse: TJSONValue;
begin
    // 오직 GET 요청만 캐시에 저장한다
    if ARequest.Method <> 'GET' then
        Exit;

    CacheKey := GenerateCacheKey(ARequest, AContext.Tenant.id);

    // Redis를 먼저 시도한다. 그러지 못하는 경우 로컬 캐시에서 처리한다
    CachedResponse := GetFromRedis(CacheKey);
    if not Assigned(CachedResponse) then
        CachedResponse := GetFromLocalCache(CacheKey);

```



```

if Assigned(CachedResponse) then
begin
    AResponse.JSONValue := CachedResponse;
    AResponse.StatusCode := 200;
    AHandled := True; // 엔드포인트 처리하기를 완전히 생략하고 건너뛰다
end;
end;

procedure TCachedDataModule.AfterRequest(const AContext: TEndpointContext;
                                          const ARequest: TEndpointRequest;
                                          const AResponse: TEndpointResponse);

var
    CacheKey: string;
    CacheDuration: Integer;
begin
    // 성공한 GET 응답들을 캐시에 저장한다
    if (ARequest.Method = 'GET') and (AResponse.StatusCode = 200) then
    begin
        CacheKey := GenerateCacheKey(ARequest, AContext.Tenant.id);
        CacheDuration := GetCacheDuration;

        // Redis와 로컬 캐시 둘 다에 저장한다
        StoreInRedis(CacheKey, AResponse.JSONValue, CacheDuration);
        StoreInLocalCache(CacheKey, AResponse.JSONValue, CacheDuration div 4);
    end;
end;

```

이 방식은 읽기 작업이 매우 많은 API의 성능을 획기적으로 향상시킨다. 그러면서도 여러분의 엔드포인트 구현들은 변함없이 단순하다. 이 이중 캐시 전략은 속도(로컬 캐시)와 확장성(예: Redis를 사용한 분산 캐시) 둘 모두를 제공한다.

요청 횟수 제한하기(Rate Limiting) 및 API 보호

기본적인 보안을 넘어, 실제 운영 API들은 정교한 요청 횟수 제한, 요청 스로틀링(Throttling), 특정 공격 패턴들로부터의 보호 등이 필요한 경우가 많다.

수준높게 요청 횟수를 제한하기

인터셉터 하나가 정교한 요청 횟수 제한 로직을 구현할 수 있다. 그 로직 안에서 사용자 등급 또는 특정 요청 패턴들을 고려할 수 있다:

```

procedure TRateLimitedDataModule.BeforeRequest(const AContext: TEndpointContext;
                                                const ARequest: TEndpointRequest;
                                                const AResponse: TEndpointResponse;

```

```

var AHandled: Boolean);

var
  UserID: string;
  RateLimit: TRateLimit;
  Usage: TUsageInfo;
begin
  UserID := ExtractUserID(ARequest);
  RateLimit := GetRateLimitForUser(UserID);
  Usage := GetCurrentUsage(UserID);

  if Usage.ExceedsLimit(RateLimit) then
  begin
    AResponse.StatusCode := 429; // 허용 횟수를 넘어서는 과도한 요청들
    AResponse.SetHeaderValue('Retry-After',
      IntToStr(Usage.GetRetryAfterSeconds));
    AHandled := True;
    Exit;
  end;

  // 카운터(사용량을 세어 기록한) 값을 업데이트한다
  IncrementUsage(UserID);
end;

```

상속을 통해 재사용하는 인터셉터 로직

여러 리소스들을 가진 복잡한 API들 안에서, 여러분은 어떤 인터셉터 기능을 공유해 서로 다른 리소스들 사이에서 사용할 수 있도록 해야 하는 경우가 많이 있다. 가장 흔한 상황으로는 표준화된 로그 기록, 범용 입력 검증, 일관된 인증 패턴들 등이 있다. 이런 경우에는 중복되는 코드를 여러 여러 리소스들에 펼쳐놓기 보다는, 기반 클래스를 하나 만들어서 그 안에 공통 인터셉터 로직을 구현하는 것이 더 좋다.

이 방식은 RAD 스튜디오 안에서 완벽하게 활용할 수 있다. 왜냐하면 RAD Server는 `TDataModule`이나 `TComponent`를 리소스들을 위한 기본 클래스로 사용하기 때문이다. 그리고 상속에서 인터페이스를 함께 사용하는 것은 자연스럽다. 우리가 이 패턴을 어떻게 구현하는 지 구체적인 방법에 대해서는 더 상세하게 다룰 것이다.



팁

인터셉터를 상속할 수 있도록 하는 가장 작은 클래스는 `TInterfacedPersistent`다. 요구사항들과 사용되는 아키텍처에 따라, `TComponent`와 `TDataModule`를 사용해도 된다.

기반 인터셉터(Base Interceptor)를 만들기

```

// 기반 인터셉터 클래스는 공통 기능들을 가진다

```

```

TBaseInterceptor = class(TInterfacedCoPersistent, IEMSResourceInterceptor)
protected
  FRequestID: string;
  FStartTime: TDateTime;
  FCurrentUserID: string;
  FValidatedData: TJSONObject;

  // 가상(virtual) 메서드들: 이것들은 자손들에 의해 덮어써질 수 있다
  procedure DoCustomValidation(const ARequest: TEndpointRequest;
                               var AValid: Boolean;
                               var AErrorMessage: string); virtual;
  procedure DoCustomLogging(const ARequest: TEndpointRequest;
                             const AResponse: TEndpointResponse); virtual;
  procedure DoAuthentication(const ARequest: TEndpointRequest;
                              var AAuthenticated: Boolean;
                              var AUserID: string); virtual;

public
  // IEMSResourceInterceptor 구현
  procedure BeforeRequest(const AContext: TEndpointContext;
                          const ARequest: TEndpointRequest;
                          const AResponse: TEndpointResponse;
                          var AHandled: Boolean);
  procedure AfterRequest(const AContext: TEndpointContext;
                         const ARequest: TEndpointRequest;
                         const AResponse: TEndpointResponse);

end;

procedure TBaseInterceptor.BeforeRequest(const AContext: TEndpointContext;
                                         const ARequest: TEndpointRequest;
                                         const AResponse: TEndpointResponse;

                                         var AHandled: Boolean);
var
  IsAuthenticated: Boolean;
  IsValid: Boolean;
  ErrorMessage: string;
begin
  FRequestID := GenerateRequestID;
  FStartTime := Now;

  // 표준 인증(authentication) - 자손들에 의해 커스터마이징 될 수 있다
  DoAuthentication(ARequest, IsAuthenticated, FCurrentUserID);
  if not IsAuthenticated then
  begin
    AResponse.StatusCode := 401;
    AHandled := True;
    Exit;
  end;
end;

```

```

end;

// 표준 검증(validation) - 자손들에 의해 커스터마이징 될 수 있다
DoCustomValidation(ARequest, IsValid, ErrorMessage);
if not IsValid then
begin
    AResponse.StatusCode := 400;
    AResponse.JSONValue := TJSONObject.Create(TJSONPair.Create('error',
ErrorMessage));
    AHandled := True;
    Exit;
end;

// 공통 로그 기록
LogRequestStart(FRequestID, ARequest.Method, ARequest.PathInfo, FCurrentUserID);
end;

procedure TBaseInterceptor.AfterRequest(const AContext: TEndpointContext;
const ARequest: TEndpointRequest;
const AResponse: TEndpointResponse);

var
    Duration: Integer;
begin
    Duration := MillisecondsBetween(Now, FStartTime);

    // 표준 로그 기록
    LogRequestComplete(FRequestID, AResponse.StatusCode, Duration);

    // 자손들이 사용자 정의 로그 기록을 추가할 수 있도록 허용한다
    DoCustomLogging(ARequest, AResponse);
end;

// 기본(default) 구현 - 자손들이 덮어쓸 수 있다.
procedure TBaseInterceptor.DoAuthentication(const ARequest: TEndpointRequest;
var AAuthenticated: Boolean;
var AUserID: string);

begin
    // 기본(default) JWT 토큰 검증(validation)
    AUserID := ExtractUserFromJWT(ARequest);
    AAuthenticated := AUserID <> '';
end;

procedure TBaseInterceptor.DoCustomValidation(const ARequest: TEndpointRequest;
var AValid: Boolean;
var AErrorMessage: string);

begin
    // 기본(default) 검증(validation) - 콘텐츠 타입을 점검을 POST/PUT인 경우에만 한다

```

```

AValid := True;
if (ARequest.Method = 'POST') or (ARequest.Method = 'PUT') then
begin
  if not ARequest.ContentType.StartsWith('application/json') then
  begin
    AValid := False;
    AErrorMessage := 'Content-Type must be application/json';
  end;
end;
end;

procedure TBaseInterceptor.DoCustomLogging(const ARequest: TEndpointRequest;
                                           const AResponse: TEndpointResponse);

begin
  // 기본(default) 구현은 아무것도 하지 않는다 - 자손들이 덮어쓸 수 있다.
end;

```

기반 인터셉터를 확장하기

이제 여러분의 특정 리소스들은 이 기반 클래스로부터 상속을 받고, 오직 필요한 것만 커스터마이징할 수 있다:

```

// Users API 리소스 - 사용자 정의 검증(custom validation)을 적용한다
TUsersResource = class(TBaseInterceptor)
protected
  procedure DoCustomValidation(const ARequest: TEndpointRequest;
                              var AValid: Boolean;
                              var AErrorMessage: string); override;
  procedure DoCustomLogging(const ARequest: TEndpointRequest;
                            const AResponse: TEndpointResponse); override;
public
  // 여러분의 일반(regular) 엔드포인트 메서드들
  [EndPoint(rmGET, '/users')]
  procedure GetUsers(const AContext: TEndpointContext;
                    const ARequest: TEndpointRequest;
                    const AResponse: TEndpointResponse);

  // ... 기타 엔드포인트들
end;

procedure TUsersResource.DoCustomValidation(const ARequest: TEndpointRequest;
                                           var AValid: Boolean;
                                           var AErrorMessage: string);

begin
  // 부모의 검증(parent validation)을 먼저 호출한다
  inherited DoCustomValidation(ARequest, AValid, AErrorMessage);

```

```

if not AValid then
    Exit;

// 사용자별 고유한 검증(user-specific validation)을 추가한다
if (ARequest.Method = 'POST') and Assigned(ARequest.JSONValue) then
begin
    if not ARequest.JSONValue.TryGetValue<string>('email', FValidatedEmail) then
    begin
        AValid := False;
        AErrorMessage := 'Email field is required';
        Exit;
    end;

    if not IsValidEmail(FValidatedEmail) then
    begin
        AValid := False;
        AErrorMessage := 'Invalid email format';
    end;
end;
end;

procedure TUsersResource.DoCustomLogging(const ARequest: TEndpointRequest;
                                          const AResponse: TEndpointResponse);
begin
    // 특정 로그 기록을 추가한다. 사용자 동작을 기록하기 위해서다
    if AResponse.StatusCode = 201 then
        LogUserCreated(FCurrentUserID, FValidatedEmail);
end;
// 나머지 메서드들 (GetUsers 등등)

```

이제 `TBaseInterceptor`를 다른 어떤 클래스에서도 상속받을 수 있다. 그리고 심지어 특정 메서드를 덮어써서 더 세밀하게 제어할 수 있다. 그 제어는 리소스 레벨에서 필요한대로 구현하면 된다.

```

// 테넌트를 인식하는 API들을 위해 특화된 기반
TTenantAwareInterceptor = class(TBaseInterceptor)
protected
    FCurrentTenantID: string;
    procedure DoAuthentication(const ARequest: TEndpointRequest;
                              var AAuthenticated: Boolean;
                              var AUserID: string); override;
end;

// 관리자 API들을 위해 특화된 기반
TAdminInterceptor = class(TBaseInterceptor)

```

```
protected
  procedure DoAuthentication(const ARequest: TEndpointRequest;
                             var AAuthenticated: Boolean;
                             var AUserID: string); override;
end;
```

이 방식은 특히 매우 많은 API들이 있는 곳에서 효과적이다. 예를 들어 여러분이 수십 개의 리소스를 가지고 있는데 그것들 모두에 일관된 동작을 넣고 싶은 경우를 들수 있다. 이 가상 메서드(Virtual method) 패턴은 완벽하게 균형있게 “공유 기능들”과 “필요한 곳에서의 커스터마이징”을 조화시킨다.

외부 시스템들과의 통합

현대 애플리케이션은 단독으로 격리되어 존재하는 경우가 거의 없다. 여러분은 외부 API들, 메시지 큐들, 이벤트 시스템들과 통합해야 하는 경우가 있다. 그래서 RAD Server에 내장된 기능을 넘어 다른 방식들이 필요할 수 있다.

이벤트-주도(Event-Driven) 아키텍처 통합

인터셉터 하나가 가교 역할을 해 외부 이벤트 시스템을 연결할 수 있다:

```
procedure TEventDrivenInterceptor.AfterRequest(const AContext: TEndpointContext;
                                               const ARequest: TEndpointRequest;
                                               const AResponse: TEndpointResponse);

var
  Event: TAPIEvent;
begin
  // 이벤트 밖으로 내보낸다. 단, 오직 성공한 동작들에 대해서만 그렇게 한다
  if AResponse.StatusCode < 400 then
  begin
    Event := CreateAPIEvent(ARequest, AResponse, FCurrentUser);
    PublishToEventBus(Event);
    // 웹훅(webhook)들을 발동한다. 단, 특정 동작에 대해서만 그렇게 한다
    if ShouldTriggerWebhook(ARequest.Method, ARequest.PathInfo) then
      TriggerWebhookAsync(Event);
  end;
end;
```

최고의 실무 적용 관행 및 고려 사항

인터셉터를 구현할 때 명심할 것이 있다. 인터셉터(interceptor)들은 모든 요청마다 처리(processing)를 추가한다. 즉, 매우 큰 영향을 주는 경로가 된다. 따라서, 설계할 때는 인터셉터가 빠르도록 그리고 비용이 큰 동작을 피하도록 한다. `BeforeRequest` 메서드를 사용해 설정이나 검증을 하라. `AfterRequest` 메서드들을 사용해 깨끗이 정리하거나 공지(notifications)들을 수행 하라.

인터셉터는 그 리소스 안의 모든 엔드포인트들에 대해 실행된다는 점을 기억하라. 엔드포인트마다 다른 동작이 필요하다면, 여러분이 요청의 경로(path)나 메서드(method)를 확인해 알맞은 행위(action)를 결정할 수 있다. 또는 별도의 리소스들이 각자 다른 인터셉터 구현을 사용하도록 하는 것을 고려할 수 있다.

인터셉터 패턴은 크로스-컷팅(cross-cutting) 즉, 한 기능을 여러 엔드포인트들에 적용해야 할 때 가장 효과적이다. 특정 엔드포인트 하나를 위한 로직이라면 그 엔드포인트 메서드 안에 직접 구현하는 것이 일반적으로 더 명확하다.

결론

리소스 인터셉터(interceptor)들은 RAD Server 설계 철학의 진화가 자연스럽게 진행되고 있음을 잘 보여준다: 단순한 것은 단순하게(simple) 유지하라. 하지만, 복잡한 경우에 필요한 유연성(flexibility)을 제공하라. 인터셉터들은 RAD Server의 뛰어난 내장 기능들을 대체하는 것이 아니다. 오히려 그것들을 확장한다. 그래서 여러분의 특정 요구 사항이 표준 패턴을 벗어날 때 사용할 수 있도록 한다.

여러분이 복잡한 멀티-테넌시를 구현하든, 엔터프라이즈 시스템들과 통합하든, 또는 정교한 모니터링 기능을 추가하든 상관없이, 인터셉터들은 깔끔한 아키텍처 패턴들을 제공한다. 그래서 여러분이 계속 명확성과 유지보수성을 유지할 수 있도록 한다. 그 덕분에 RAD Server 개발은 더욱 효과적이다. 핵심은 인터셉터들을 신중하게 사용하는 것이다. 인터셉터는 보완이 필요할 때 사용하면 좋다. 다시 말해, 이 프레임워크에 이미 있는 장점들과 경쟁하는 목적으로 사용하는 것은 좋지 않다.



참고

[IEMSResourceInterceptor](#) 인터페이스는 RAD 스튜디오 13 버전부터 사용할 수 있다.