

델파이 Horse를 활용한 REST 서버 구현 세미나

0. 세미나 개요

0.1 개요

총 소요시간: 2시간 (120분)
대상: 델파이 개발 경험이 있는 개발자
목표: Horse 프레임워크를 사용하여 실무에서 활용 가능한 REST API 서버 구축

0.2 필요한 사전 준비사항

- 데이터베이스: MariaDB
- 쿼리 브라우저: DBEaver

0.3 개발 도구

- Delphi XE8 이상

1. REST API 개요

1.1 REST API란?

REST API는 웹에서 데이터를 주고받기 위한 표준 규칙입니다. 마치 레스토랑의 메뉴판처럼, 클라이언트가 서버에게 "이런 데이터를 달라" 또는 "이런 작업을 해달라"고 요청하는 방식을 정의하며, 통신 프로토콜은 HTTP/HTTPS를 데이터는 JSON을 사용합니다.

REST = **RE**presentational **S**tate **T**ransfer(표현 상태 전이)

1.2 REST API의 핵심 구조

HTTP 메소드 + URL = 작업 정의

요청	의미
GET /user	사용자 목록 조회
GET /user/123	123번 사용자 조회
POST /user	새 사용자 생성
PUT /user/123	123번 사용자 수정
DELETE /user/123	123번 사용자 삭제

1.3 REST API의 5가지 핵심 원칙

1.3.1 리소스 중심 설계

HTTP 메소드는 동사로, URL은 명사로 표현한다

요청	비고
GET /book/123	좋은 예
GET /getBook/123	나쁜 예

1.3.2 무상태(Stateless)

서버는 이전 요청을 기억하지 않음. 매 요청마다 필요한 모든 정보 포함해야 한다

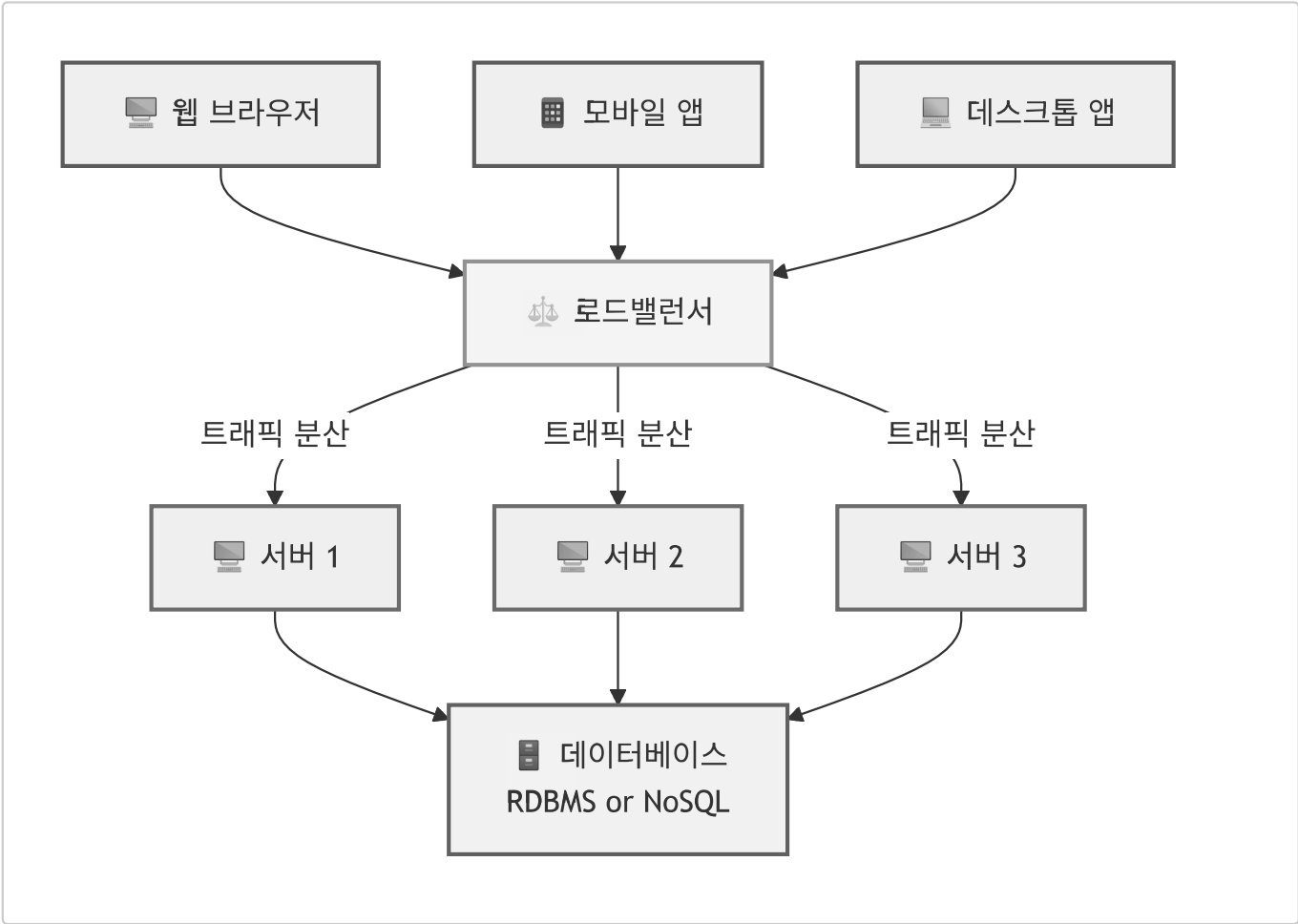
매번 인증 토큰 포함 GET /api/users Authorization: Bearer abc123token

1.3.3 표준 HTTP 메소드 사용

메소드	의미
GET	조회
POST	생성
PUT	수정
DELETE	삭제

1.3.4 계층화된 구조

클라이언트는 서버의 내부 구조를 몰라도 됨



1.3.5 캐시 가능

GET 요청 결과는 캐시하여 성능 향상

```
Cache-Control: max-age=3600 ETag: "abc123"
```

2. HORSE 프레임워크 소개

2.1 Horse 프레임워크 특징

2.1.1 경량화와 고성능

- 네이티브 컴파일
- 메모리 효율 (매우 적은 메모리만 사용한다)
- 빠른 실행 속도
- 경량화된 웹 프레임워크
- 간단한 라우팅 시스템

2.1.2 간단하고 직관적인 문법

- 학습 곡선이 낮음
- Node의 Express와 유사
- 가독성이 좋음.

2.1.3 다양하고 강력한 미들웨어 시스템

- 모듈화: 기능별로 미들웨어 분리
- 재사용성: 다른 프로젝트에서도 동일한 미들웨어 사용
- 확장성: 필요한 미들웨어만 추가/제거 가능

2.1.4 다양한 OS 지원

- Windows
- Linux
- macOS

2.1.5 델파이 생태계와의 통합

- 기존 델파이 라이브러리를 사용할 수 있다
- FireDAC, UniDAC 등 다양한 데이터베이스 드라이버를 사용할 수 있다

2.1.6 간편한 배포와 운영

- 단일 실행파일로 배포와 운영이 편리 (런타임 불필요)
- 도커로 배포 가능
- 클라우드에 배포 가능

2.1.7 다른 프레임워크들

델파이 기본 제공

- **DataSnap** 또는 **WebBroker** 델파이에서 기본으로 제공한다. 12.2에서는 웹스텐실을 사용해서 템플릿을 쉽게 만들 수 있다.

오픈소스

- **MARS** <https://github.com/andrea-magni/MARS>
- **WiRL** <https://github.com/delphi-blocks/WiRL> MARS와 WiRL은 같은 뿌리에서 나온 것으로 보이는데 MARS가 구현된 것이 조금 더 많지만, WiRL이 코드는 훨씬 정갈하다.
- **DelphiMVCFramework** <https://github.com/danieleteti/delphimvcframework> 가장 업데이트가 활발하다. 소스코드 정돈이 잘 안된점은 아쉽다.

2.2 소스코드 다운로드

델마당 공식 리포지토리에서 클론한다 델마당 세미나 리포지토리

클론 후 폴더의 구성은 다음과 같다.

```
01.RESTServer/
├── bin                // 실행 파일이 생성되는 곳
├── database
│   └── schema.sql    // MariaDB용 스키마
├── docs
│   └── Seminar_20250723.md // 현재 보고 있는 문서
├── lib
│   ├── Horse        // Horse 엔진 유닛들
│   ├── MiddleWare   // 미들웨어 유닛들
│   └── Swagger       // 스웨거 유닛들
└── src
    ├── 00.Raw        // Indy를 사용한 서버
    ├── 01.Basic      // 기본 서버
    ├── 02.MiddleWare // 미들웨어 적용한 서버
    ├── 03.Database   // 데이터베이스 연동 서버
    ├── 04.Client     // 클라이언트
    └── 99.NodeJS     // NodeJS 예제
```

2.3 Indy를 사용하여 구현하기

프레임워크의 도움없이 서버를 구현해 보면서, 프레임워크의 필요성에 대해 알아본다

```
unit uMain;

interface

uses
  Winapi.Windows,
  Winapi.Messages,
  System.SysUtils,
  System.Variants,
  System.Classes,
```

```
Vcl.Graphics,  
Vcl.Controls,  
Vcl.Forms,  
Vcl.Dialogs,  
Vcl.StdCtrls,
```

```
IdContext,  
IdHttp,  
IdCustomHttpServer,  
IdHttpServer;
```

type

```
TfrmMain = class(TForm)  
    ePort: TEdit;  
    btnStart: TButton;  
    btnStop: TButton;  
    btnExecute: TButton;  
    procedure btnExecuteClick(Sender: TObject);  
    procedure btnStartClick(Sender: TObject);  
    procedure btnStopClick(Sender: TObject);  
private  
    FServer: TIdHttpServer;  
  
    procedure Start(const APort: Integer = 8080);  
    procedure Stop;  
    procedure TriggerCommandGet(AContext: TIdContext; ARequestInfo:  
        TIdHttpRequestInfo; AResponseInfo: TIdHttpResponseInfo);  
public  
    constructor Create(AOwner: TComponent); override;  
    destructor Destroy; override;  
end;
```

var

```
    frmMain: TfrmMain;
```

implementation

```
{ $R *.dfm }
```

uses

```
    Winapi.ShellAPI;
```

```
constructor TfrmMain.Create(AOwner: TComponent);
```

```
begin
```

```
    inherited;
```

```
    FServer := TIdHttpServer.Create(Self);
```

```
end;
```

```
destructor TfrmMain.Destroy;
```

```
begin
```

```
    FServer.Active := False;
```

```
    FServer.Free;
```

```
    inherited;
```

```

end;

procedure TfrmMain.btnExecuteClick(Sender: TObject);
begin
    var LURL := Format('http://localhost:%d/hello', [FServer.Bindings[0].Port]);
    ShellExecute(0, 'open', PChar(LURL), nil, nil, SW_SHOW);
end;

procedure TfrmMain.btnStartClick(Sender: TObject);
begin
    Start;
end;

procedure TfrmMain.btnStopClick(Sender: TObject);
begin
    Stop;
end;

procedure TfrmMain.Start(const APort: Integer);
begin
    FServer.Bindings.Clear;

    with FServer.Bindings.Add do
    begin
        IP := '0.0.0.0';
        Port := APort;
    end;

    FServer.OnCommandGet := TriggerCommandGet;
    FServer.Active := True;
end;

procedure TfrmMain.Stop;
begin
    FServer.Active := False;
end;

procedure TfrmMain.TriggerCommandGet(AContext: TIdContext;
    ARequestInfo: TIdHTTPRequestInfo; AResponseInfo: TIdHTTPResponseInfo);
begin
    if SameText(ARequestInfo.URI, '/hello') then
    begin
        AResponseInfo.ContentType := 'application/json';
        AResponseInfo.ContentText := '{"response": "Hello, world"}';
    end;
end;

```

2.4 기본 서버 구현

가장 기본이 되는 서버를 구현해 보면서 라우터의 사용법에 대해 익힌다.

```

program BasicServer;

uses
  System.SysUtils,
  Horse;

begin
  // 라우터에 등록한다
  THorse.Get('/hello',
    procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
    begin
      Res.Send('Hello World!');
    end);

  // 9000번 포트를 리스닝한다
  THorse.Listen(9000);
end.

```

3. 미들웨어

3.1 미들웨어란?

- 요청-응답 사이클에서의 중간 처리
- 공식 미들웨어
 - **horse/json** SON 처리를 위한 미들웨어
 - **horse/basic-auth** 기본 인증지원
 - **horse/cors** Cross-Origin Resource Sharing 지원
 - **horse/stream** 바이너리 스트림 전송 지원
 - **horse/jwt** JWT 인증 지원
 - **horse/exception** 익셉션 처리 지원
 - **horse/logger** 로깅 지원
 - **horse/compression** 응답 압축 지원
- 비공식 미들웨어
 - 사용자들이 개발한 다양한 미들웨어가 존재하며
 - 필요에 따라 미들웨어를 직접 만들어서 사용할 수 있다

3.2 미들웨어 사용해보기

3.2.1 JSON 미들웨어

JSON 미들웨어는 웹 서버나 API 서버에서 JSON 데이터를 자동으로 파싱하고 처리하는 중간 계층(middleware) 소프트웨어입니다.

JSON 미들웨어의 역할

- 요청 처리 HTTP 요청 본문의 JSON을 자동 파싱 Content-Type 검증 (application/json) 파싱 에러 처리
- 응답 처리 객체를 JSON으로 자동 직렬화 적절한 Content-Type 헤더 설정 JSON 포매팅 (들여쓰기, 압축 등)

```

program Middleware01;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  System.JSON,
  Horse,
  Horse.Jhonson;

begin
  // 미들웨어 등록
  THorse.Use(Jhonson());

  // 라우터에 등록한다
  THorse.Get('/hello',
    procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
    var
      LResult: TJSONObject;
    begin
      LResult := TJSONObject.Create;
      LResult.AddPair('response', 'Hello, World');
      Res.Send<TJSONObject>(LResult);
    end);

  // 9000번 포트를 리스닝한다
  THorse.Listen(9000);
end.

```

3.2.2 basic-auth

Basic Authentication은 HTTP 프로토콜에서 가장 기본적인 인증 방식입니다. Basic Authentication 동작 원리

1. 인코딩 방식

- 사용자명과 패스워드를 콜론(:)으로 연결
- Base64로 인코딩
- HTTP 헤더에 포함하여 전송

```
username:password → Base64 인코딩 → dXNlcm5hbWU6cGFzc3dvcmQ=
```

2. HTTP 헤더 형식

```
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

3. 보안 고려사항

장점:

- 구현이 간단함
- 모든 HTTP 클라이언트에서 지원
- 서버 세션 상태 불필요

단점:

- Base64는 암호화가 아닌 인코딩 (쉽게 디코딩 가능)
- 매 요청마다 인증 정보 전송
- 브라우저에 패스워드 저장 위험

```
program Middleware02;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  System.JSON,

  Horse,
  Horse.Jhonson,
  Horse.BasicAuthentication;

begin
  // 미들웨어 등록
  THorse.Use(Jhonson());

  // BasicAuth를 처리하기 위한 미들웨어
  THorse.Use(HorseBasicAuthentication(
    function(const AUsername, APassword: string): Boolean
    begin
      Result := AUsername.Equals('user') and APassword.Equals('password');
    end)));

  // 라우터에 등록한다
  THorse.Get('/hello',
    procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
    var
      LResult: TJSONObject;
    begin
      LResult := TJSONObject.Create;
      LResult.AddPair('response', 'Hello, World');
      Res.Send<TJSONObject>(LResult);
    end);

  // 9000번 포트를 리스닝한다
  THorse.Listen(9000);
end.
```

브라우저에서 테스트 하기

```
http://user:password@localhost:9000/hello
```

3.2.3 cors

CORS(Cross-Origin Resource Sharing)는 웹 브라우저의 Same-Origin Policy(동일 출처 정책) 제한을 우회하기 위해 사용됩니다.

Same-Origin Policy란? 웹 브라우저는 보안상 이유로 다음 조건이 모두 같은 경우에만 리소스 접근을 허용합니다:

- 프로토콜 (http/https)
- 도메인 (example.com)
- 포트 (80, 443, 8080 등)

CORS 동작 과정

- Preflight Request: 브라우저가 OPTIONS 요청으로 서버에 허가 요청
- 서버 응답: 허용된 origin, 메서드, 헤더 정보 반환
- 실제 요청: 허가받은 경우에만 실제 API 호출 실행

주의사항

- 보안: * (모든 도메인 허용)은 운영 환경에서 위험
- 성능: Preflight 요청으로 인한 추가 네트워크 비용
- 설정: 필요한 origin만 정확히 허용해야 함

CORS는 웹 브라우저에서만 적용되는 정책이므로, 데스크톱 애플리케이션이나 서버 간 통신에는 영향을 주지 않습니다.

```
program Middleware03;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  System.JSON,

  Horse,
  Horse.Jhonson,
  Horse.BasicAuthentication,
  Horse.CORS;

begin
  // 미들웨어 등록
  HorseCORS
    .AllowedOrigin('*')
    .AllowedCredentials(true)
    .AllowedHeaders('*')
    .AllowedMethods('*')
    .ExposedHeaders('*');
```

```

// It's necessary to add the middleware in the Horse:
THorse.Use(CORS);

THorse.Use(Jhonson());

// BasicAuth를 처리하기 위한 미들웨어
THorse.Use(HorseBasicAuthentication(
  function(const AUsername, APassword: string): Boolean
  begin
    Result := AUsername.Equals('user') and APassword.Equals('password');
  end));

// 라우터에 등록한다
THorse.Get('/hello',
  procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
  var
    LResult: TJSONObject;
  begin
    LResult := TJSONObject.Create;
    LResult.AddPair('response', 'Hello, World');
    Res.Send<TJSONObject>(LResult);
  end);

// 9000번 포트를 리스닝한다
THorse.Listen(9000);
end.

```

3.2.4 stream

Stream을 다루기 위한 공식 미들웨어입니다. 이 미들웨어를 사용하면 웹 API에서 파일이나 바이너리 데이터를 스트림으로 전송할 수 있습니다.

주요 특징

- 파일 전송: PDF, 이미지, 동영상 등 다양한 파일 형식을 스트림으로 전송
- 메모리 효율성: 대용량 파일도 메모리에 모두 로드하지 않고 스트림으로 처리
- Content-Type 설정: 파일 형식에 맞는 MIME 타입 자동 설정
- MIT 라이선스: 무료 오픈소스

```

program Middleware04;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  System.Classes,
  System.IOUtils,

  Horse,
  Horse.OctetStream;

begin

```

```

THorse.Use(OctetStream);

THorse.Get('/stream',
  procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
  var
    LStream: TFileStream;
  begin
    LStream := TFileStream.Create(ExtractFilePath(ParamStr(0)) + 'horse.pdf',
fmOpenRead);
    Res.Send<TStream>(LStream).ContentType('application/pdf');
  end);

THorse.Get('/download/:filename',
  procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
  var
    LPath: string;
    LStream: TFileStream;
  begin
    LPath := TPath.Combine(ExtractFilePath(ParamStr(0)),
Req.Params['filename']);
    if TFile.Exists(LPath) then
      begin
        LStream := TFileStream.Create(LPath, fmOpenRead);
        Res.Send<TStream>(LStream).ContentType('application/pdf');
      end
    else
      begin
        Res.Status(THTTPStatus.NotFound);
      end;
    end);

  // 9000번 포트를 리스닝한다
  THorse.Listen(9000);
end.

```

3.2.5 jwt

JWT란? JWT는 당사자 간에 정보를 JSON 객체로 안전하게 전송하기 위한 개방형 표준(RFC 7519)입니다. 주로 **인증 (Authentication)**과 정보 교환에 사용됩니다.

JWT 구조 JWT는 점(.)으로 구분된 세 부분으로 구성됩니다:

```
Header.Payload.Signature
```

1.Header (헤더)

- 토큰 타입(JWT)과 해싱 알고리즘 정보
- Base64Url로 인코딩

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

2. Payload (페이로드)

- 실제 전송할 데이터(클레임)
- 사용자 정보, 권한, 만료시간 등

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022,
  "exp": 1516242622
}
```

3. Signature (서명)

- 토큰의 무결성 검증
- Header + Payload + Secret Key로 생성

동작 방식

- 로그인: 사용자가 인증 정보 제공
- 토큰 발급: 서버가 JWT 생성 후 클라이언트에 전송
- 토큰 사용: 클라이언트가 요청 시 JWT를 Header에 포함
- 검증: 서버가 JWT 서명을 검증하여 사용자 인증

장점

- 상태 없음(Stateless): 서버에 세션 정보 저장 불필요
- 확장성: 마이크로서비스 환경에 적합
- 크로스 도메인: CORS 문제 해결
- 모바일 친화적: 쿠키 대신 사용 가능

단점

- 토큰 크기: 쿠키보다 크기가 큼
- 보안: 토큰 탈취 시 만료 전까지 사용 가능
- 서버 측 무효화 어려움: 블랙리스트 관리 필요

주요 사용 사례

- 인증/인가: 로그인 후 API 접근 제어
- 정보 교환: 서비스 간 안전한 데이터 전송
- SSO(Single Sign-On): 여러 서비스 간 인증 공유

JWT는 현대 웹 애플리케이션에서 표준적으로 사용되는 인증 방식으로, REST API나 마이크로서비스 아키텍처에서 특히 유용합니다.

```
uses
  Horse,
  Horse.JWT;

begin
  THorse.Use(HorseJWT('MY-PASSWORD'));

  THorse.Post('ping',
    procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
    begin
      Res.Send('pong');
    end);

  THorse.Listen(9000);
end.
```

3.2.6 exception

Handle-exception은 Horse 프레임워크에서 예외를 처리하기 위한 공식 미들웨어입니다. 이 미들웨어는 API에서 발생하는 예외를 자동으로 캐치하고, 클라이언트에게 적절한 JSON 형태의 오류 응답을 반환합니다.

```
uses
  Horse,
  Horse.Jhonson, // It's necessary to use the unit
  Horse.HandleException, // It's necessary to use the unit
  System.JSON;

begin
  // It's necessary to add the middlewares in the Horse:
  THorse
    .Use(Jhonson) // It has to be before the exceptions middleware
    .Use(HandleException);

  THorse.Post('/ping',
    procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
    begin
      // Manage your exceptions:
      raise EHorseException.New.Error('My Error!');
    end);

  THorse.Listen(9000);
end.
```

3.2.7 logger

개발된 API의 로그를 등록하기 위한 공식 미들웨어입니다. 이 미들웨어는 HTTP 요청과 응답에 대한 접근 로그를 자동으로 기록하여 API의 모니터링과 디버깅을 지원합니다.

```
uses
    Horse,
    Horse.Jhonson, // It's necessary to use the unit
    Horse.HandleException, // It's necessary to use the unit

begin
    THorse.Use(
        procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
        begin
            Writeln(Format('[%s] %s %s', [
                FormatDateTime('yyyy-mm-dd hh:nn:ss', Now),
                Req.Method,
                Req.PathInfo
            ]));
            Next();
        end);

    THorse.Listen(9000);
end.
```

3.2.8 compression

API의 콘텐츠를 압축하기 위한 공식 미들웨어입니다. 이 미들웨어는 HTTP 응답을 자동으로 압축하여 네트워크 트래픽을 줄이고 전송 속도를 향상시킵니다.

지원하는 압축 방식 현재 미들웨어는 DEFLATE와 GZIP을 사용하여 데이터를 압축할 수 있도록 준비되어 있습니다.

압축 타입	Delphi	Lazarus
DEFLATE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
GZIP	<input checked="" type="checkbox"/>	<input type="checkbox"/>

```
uses
    System.SysUtils,
    System.JSON,

    Horse,
    Horse.Jhonson,
    Horse.Compression;

begin
    THorse
        .Use(Compression())
        .Use(Jhonson);

    // 일정 크기 이상만 압축하도록 설정할 수 있다
    // THorse.Use(Compression(1024));
```

```

THorse.Get('/ping',
  procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
  var
    I: Integer;
    LPong: TJSONArray;
  begin
    LPong := TJSONArray.Create;
    for I := 0 to 1000 do
      LPong.Add(TJSONObject.Create(TJSONPair.Create('ping', 'pong')));
    Res.Send(LPong);
  end);

THorse.Listen(9000);
end.

```

4. 데이터베이스 연동 서버

4.1 데이터베이스 및 샘플 데이터 생성

```

DROP TABLE IF EXISTS users;
CREATE TABLE users (
  seq INTEGER AUTO_INCREMENT PRIMARY KEY,
  usr_id VARCHAR(12) NOT NULL,
  usr_nm VARCHAR(50) NOT NULL,
  usr_pw VARCHAR(64) NOT NULL,
  usr_lvl INTEGER
);

```

4.2 FireDAC 연결 풀 만들기

- 데이터베이스 연결 풀을 사용해야 하는 이유
 - 성능 저하
 - 리소스 낭비
 - 연결 한계 초과시 데이터베이스 성능 저하 또는 다운 MySQL 기본 max_connections = 151
PostgreSQL 기본 max_connections = 100
- FireDAC은 풀링을 기본으로 지원한다

```

unit uDBPool;

interface

{$REGION 'USES'}
uses
  System.SysUtils,
  System.Classes,
  System.Generics.Collections,
  System.NetEncoding,

```



```

Data.DB,

FireDAC.Comp.UI,
FireDAC.UI.Intf,
FireDAC.VCLUI.Wait,
FireDAC.Stan.Def,
FireDAC.Stan.Intf,
FireDAC.Stan.Pool,
FireDAC.Stan.Async,
FireDAC.Stan.Param,
FireDAC.DApt,
FireDAC.Phys.MySQL,
FireDAC.Comp.Client,
FireDAC.Comp.Script;
{$ENDREGION}

type
  TDatabase = class sealed
  strict private
    const
      CONNECTION_NAME = '_pooled_connection_';

    class constructor Create;
    class destructor Destroy;
  public
    class function GetConnection: TFDConnection; static;
  end;

implementation

{$REGION 'TDatabase'}

class constructor TDatabase.Create;
begin
  var LParams := TStringList.Create;
  try
    LParams.Add('Server=127.0.0.1');
    LParams.Add('Port=3306');
    LParams.Add('Database=pos_db');
    LParams.Add('User_Name=root');
    LParams.Add('Password=1');
    LParams.Add('Pooled=True');
    LParams.Add('CharacterSet=utf8');

    FDManager.AddConnectionDef(CONNECTION_NAME, 'MySQL', LParams);
  finally
    LParams.Free;
  end;
end;

class destructor TDatabase.Destroy;
begin
  FDManager.CloseConnectionDef(CONNECTION_NAME);
end;

```

```
class function TDatabase.GetConnection: TFDCConnection;
begin
    Result := TFDCConnection.Create(nil);
    Result.ConnectionDefName := CONNECTION_NAME;
    Result.Connected := True;
end;

{$ENDREGION}

end.
```

4.3 API 설계

HTTP 메소드	URL	동작
GET	/api/user	전체 사용자 목록 조회
GET	/api/user/:id	특정 사용자 조회
POST	/api/user	사용자 생성
PUT	/api/user/:id	사용자 수정
DELETE	/api/user/:id	특정 사용자 삭제

4.4 구현

```

program DBServer;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  System.Classes,
  System.JSON,

  Data.DB,
  uDBPool,

  Horse;

begin
  // 전체 조회
  THorse.Get('/user',
    procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
    begin
      var LConnection := TDatabase.GetConnection;
      var LQuery := LConnection.GetQuery;
      try
        LQuery.SQL.Text := ''
        SELECT
          USR_ID
          , USR_NM

```

```

        , USR_LVL
    FROM
        USRS
    '';
    LQuery.Open;

    Res.ContentType('application/json');
    Res.Send(LQuery.ToJSON);
finally
    LQuery.Free;
    LConnection.Free;
end;
end);

```

// 단건 조회

```

THorse.Get('/user/:id',
procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
begin
    var LConnection := TDatabase.GetConnection;
    var LQuery := LConnection.GetQuery;
    try
        LQuery.SQL.Text := ''
        SELECT
            USR_ID
            , USR_NM
            , USR_LVL
        FROM
            USRS
        WHERE
            USR_ID = :USR_ID
        '';
        LQuery.Params.ParamByName('USR_ID').AsString := Req.Params['id'];
        LQuery.Open;

        Res.ContentType('application/json');
        Res.Send(LQuery.ToJSON);
    finally
        LQuery.Free;
        LConnection.Free;
    end;
end);

```

// 삭제

```

THorse.Delete('/user/:id',
procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
begin
    var LConnection := TDatabase.GetConnection;
    var LQuery := LConnection.GetQuery;
    try
        LQuery.SQL.Text := ''
        DELETE
        FROM
            USRS
        WHERE
            USR_ID = :USR_ID
    
```

```

        '';
        LQuery.Params.ParamByName('USR_ID').AsString := Req.Params['id'];
        LQuery.ExecSQL;

        Res.ContentType('application/json');
        Res.Send('{"result": "OK"}');
    finally
        LQuery.Free;
        LConnection.Free;
    end;
end);

// 등록
THorse.Post('/user',
procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
begin
    var LConnection := TDatabase.GetConnection;
    var LQuery := LConnection.GetQuery;
    try
        LQuery.SQL.Text := ''
        INSERT INTO USRS
        (
            USR_ID
            , USR_PW
            , USR_NM
            , USR_LVL
        )
        VALUES
        (
            :USR_ID
            , :USR_PW
            , :USR_NM
            , :USR_LVL
        )
        '';

        var LJJSON := TJJSONObject.ParseJSONValue(Req.Body) as TJJSONObject;
        LQuery.Params.ParamByName('USR_ID').AsString := LJJSON.GetValue<string>
('usr_id');
        LQuery.Params.ParamByName('USR_PW').AsString := '1234';
        LQuery.Params.ParamByName('USR_NM').AsString := LJJSON.GetValue<string>
('usr_nm');
        LQuery.Params.ParamByName('USR_LVL').AsInteger := LJJSON.GetValue<Integer>
('usr_lvl');
        LQuery.ExecSQL;
        LJJSON.Free;
        Res.ContentType('application/json');
        Res.Send(LQuery.ToJSON);
    finally
        LQuery.Free;
        LConnection.Free;
    end;
end);

// 수정

```

```

THorse.Put('/user/:id',
  procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
  begin
    var LConnection := TDatabase.GetConnection;
    var LQuery := LConnection.GetQuery;
    try
      LQuery.SQL.Text := ''
      UPDATE USRS
      SET
        USR_NM = :USR_NM
        , USR_LVL = :USR_LVL
      WHERE
        USR_ID = :USR_ID
      '';

      var LJJSON := TJSONObject.ParseJSONValue(Req.Body) as TJSONObject;
      LQuery.Params.ParamByName('USR_ID').AsString := Req.Params['id'];
      LQuery.Params.ParamByName('USR_NM').AsString := LJJSON.GetValue<string>
('usr_nm');
      LQuery.Params.ParamByName('USR_LVL').AsInteger := LJJSON.GetValue<Integer>
('usr_lvl');
      LQuery.ExecSQL;
      LJJSON.Free;

      Res.ContentType('application/json');
      Res.Send(LQuery.ToJSON);
    finally
      LQuery.Free;
      LConnection.Free;
    end;
  end);

// 9000번 포트를 리스닝한다
THorse.Listen(9000);
end.

```

5. API 문서화와 테스트

5.1 Swagger를 사용해서 문서화 하기

SWAGGER의 특징

- API 문서화 자동화 코드에서 주석이나 어노테이션을 통해 API 문서를 자동으로 생성합니다. 수동으로 문서를 작성하고 업데이트할 필요가 없어 개발 효율성이 높아집니다.
- OpenAPI 사양 준수 OpenAPI Specification(OAS)를 기반으로 하여 표준화된 방식으로 REST API를 정의하고 문서화합니다.
- 인터랙티브 문서 Swagger UI를 통해 웹 브라우저에서 직접 API를 테스트할 수 있는 대화형 문서를 제공합니다. 실제 API 호출을 해보면서 응답을 확인할 수 있습니다.
- 코드 생성 API 사양서를 기반으로 클라이언트 SDK, 서버 스텝 코드 등을 다양한 프로그래밍 언어로 자동 생성할 수 있습니다.
- API 설계 우선 개발 코드 작성 전에 API를 먼저 설계하고 문서화하여 팀 간 협업과 API 일관성을 개선할 수 있습니다.

- 다양한 도구 지원 Swagger Editor, Swagger UI, Swagger Codegen 등 다양한 도구들이 API 개발 생명주기 전반을 지원합니다.

이러한 특징들로 인해 Swagger는 현재 가장 널리 사용되는 API 문서화 및 개발 도구 중 하나가 되었습니다.

```
program Middleware05;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  System.Classes,

  Horse,
  Horse.GBSwagger;

type
  TUser = class
  private
    FID: Double;
    FName: String;
    FLastName: string;
  public
    property ID: Double read FID write FID;
    property Name: String read FName write FName;
    property LastName: string read FLastName write FLastName;
  end;

  TAPIError = class
  private
    FError: string;
  public
    property Error: string read FError write FError;
  end;

var
  App: THorse;

begin
  App := THorse.Create;

  // Access http://localhost:9000/swagger/doc/html
  App.Use(HorseSwagger);

  APP.Get('/user',
    procedure(Req: THorseRequest; Res: THorseResponse; Next: TProc)
    begin
      end);

  APP.Post('user',
    procedure (Req: THorseRequest; Resp: THorseResponse; Next: TProc)
    begin
      end);
```

```
APP.Delete('user',  
  procedure (Req: THorseRequest; Resp: THorseResponse; Next: TProc)  
  begin  
  end);
```

```
Swagger
```

```
  .Info  
    .Title('Database CRUD Sample')  
    .Description('API Horse')  
    .Contact  
      .Name('시골프로그래머')  
      .Email('civilian7@email.com')  
      .URL('https://cafe.naver.com/delmadang')  
    .&End  
  .&End  
  .BasePath('v1')  
  .Path('user')  
    .Tag('User')  
    .GET('List All', 'List All Users')  
      .AddResponse(200, 'successful operation')  
        .Schema(TUser)  
        .IsArray(True)  
      .&End  
      .AddResponse(400).&End  
      .AddResponse(500).&End  
    .&End  
    .POST('Add user', 'Add a new user')  
      .AddParamBody('User data', 'User data')  
        .Required(True)  
        .Schema('string')  
      .&End  
      .AddResponse(201, 'Created')  
        .Schema(TUser)  
      .&End  
      .AddResponse(400, 'BadRequest')  
        .Schema(TAPIError)  
      .&End  
      .AddResponse(500).&End  
    .&End  
    .DELETE('Delete user', 'Delete user')  
      .AddParamQuery('userid', 'User ID')  
        .Required(True)  
        .Schema('string')  
      .&End  
      .AddResponse(201, 'Deleted')  
        .Schema(TUser)  
      .&End  
      .AddResponse(400, 'BadRequest')  
        .Schema(TAPIError)  
      .&End  
      .AddResponse(500).&End  
    .&End  
  .&End  
.&End;
```

```
// 9000번 포트를 리스닝한다
App.Listen(9000);
end.
```