



Primož Gabrijelčič
<http://primoz.gabrijelcic.org>

Contents

REST na kratko.....	3
Razlogi za REST	4
Hypermedia as the Engine of Application State.....	4
JSON.....	6
Odjemalci.....	8
REST Client Library.....	8
TRESTClient.....	9
TRESTRequest.....	9
TRESTResponse.....	10
TRESTResponseDatasetAdapter	11
Avtentikacija	11
Simple	11
Basic.....	11
OAuth1	11
OAuth2	11
Strežniki	12
DataSnap	12
Avtentikacija.....	15
Odjemalec za Windows/OS X/iOS	16
Dodajanje funkcij.....	16
Odjemalci za mobilne naprave	17
Orodja.....	18
REST Debugger	18
Postman.....	20
Fiddler.....	21
Viri	22
REST	22
DataSnap	22
BaaS	23
Avtentikacija	23
HTTP	24
JSON.....	24
Druga orodja.....	24

Vsi programi, omenjeni v tem dokumentu, so na voljo na naslovu

<http://17slon.com/EA/EA-REST.zip>.

REST na kratko

V večini sodobnih programov težko ostanemo omejeni na en sam računalnik. Vedno je treba z nečim komunicirati – pa naj bo to s strežniškim programom, »oblačno« shrambo dokumentov, strežnikom državne uprave, zunanjim strojnim opremo, nadzornim programom na telefonu ...

Pred desetletji je za takšno komunikacijo (ki je je bilo takrat, priznamo, bistveno manj), vsak program uporabljal svoj protokol. Nato so se vzpostavili različni standardi, se združevali in izginjali v pozabo, enotnega pristopa k porazdeljenem povezovanju pa nismo dosegli. V poslovnih sistemih je v zadnjem desetletju primat prevzel protokol SOAP, v širnem svetu raznolikih v internet povezanih računalnikov in naprav pa zadnja leta prevladuje REST.

REST [REpresentational State Transfer] pravzaprav ni protokol. Gre bolj za *arhitekturo* porazdeljenih sistemov, ki za komunikacijo uporablja protokole in standarde svetovnega spleta, za zapisovanje strukturiranih podatkov pa v večini primerov zapisa JSON ali XML. Sama arhitektura nima formalne definicije – pravzaprav še *de facto* standarda ne. Gre le za skupek priporočil, ki se jih programi bolj ali manj (in včasih zelo »po svoje«) držijo.

Kot komunikacijski kanal se uporablja protokola *http* ali *https*. Za operacije so uporabljeni kar ukazi standarda http, torej GET, POST, PUT, DELETE in ostali. Standardni model za manipulacijo podatkov CRUD običajno mapiramo v REST tako (SQL ukazi so dodani zraven za primerjavo):

operacija	http	SQL
Create	POST	INSERT
Read	GET	SELECT
Update	PUT	UPDATE
Delete	DELETE	DELETE

Kot smo omenili, uradni standard ne obstaja, zato je tudi ta preslikava ukazov le najbolj pogosta. Za izdelavo sredstva se tako včasih uporablja tudi PUT, za spremjanje tudi PATCH, za brisanje pa POST.

Natančno definicijo sredstva (*resource*), nad katerim izvajamo ukaz, zapišemo v univerzalni identifikator (URI). Tja (ali pa v telo ukaza) zapišemo tudi parametre. Univerzalni lokator (URL) tako sestavlja:

- Naslov strežnika REST (tega sestavlja naslov računalnika in pot strežnika)
- Identifikator sredstva
- Parametri ukaza

Primer – v lokatorju <http://www.songsterr.com/a/ra/songs.json?pattern=Queen> so shranjeni:

- Naslov strežnika REST: <http://www.songsterr.com/a/ra/>
- Identifikator sredstva: songs.json
- Parametri ukaza: ?pattern=Queen

Napake se sporočajo kar z napakami http in običajno sledijo definicijam, ki jih predpisuje standard http. Na primer – strežnik naj bi vrnil »410 Gone«, kadar odjemalec poizkuša izbrisati podatek, ki ne obstaja več. Dodatni podatki o napaki pa se shranijo v telo vrjenega dokumenta. (Spet pa so to samo priporočila oziroma najpogosteje uporabljen način rabe.)

Od odjemalcev se včasih tudi pričakuje, da upoštevajo druge kode http (denimo »301 Moved Permanently«), vstavljam v zahteve prave ukaze za vmesno hranjenje podatkov (*cache*) in tako naprej. Na srečo za take »malenkosti« poskrbi kar dobra knjižnica za REST, v našem primeru Delphijev REST Client Library.

Za overovitev (*authentication*) se prav tako uporabljajo spletni standardi, na primer *http basic authentication (RFC 2617)* ali *OAuth 2*.

Razlogi za REST

Zakaj se je pravzaprav REST kljub slabi specifikaciji tako razširil? Glavni razlog je verjetno enostavnost implementacije, saj lahko uporabimo običajno spletno infrastrukturo in jo le malce nadgradimo. Poleg tega je berljiv za ljudi, kar olajša razhroščevanje in je uporaben v homogeni infrastrukturi. Prav nič pomembno ni, v katerih jezikih in na katerih operacijskih sistemih tečejo programi, ki za komunikacijo uporabljajo REST, saj prav vsi sledijo istim spletnim standardom. Odjemalce REST lahko brez težav napišemo tudi v JavaScriptu, zato jih brez težav vgradimo v programe, ki tečejo na spletnih straneh.

Zaradi uporabe spletnih standardov lahko za REST strežnike uporabimo uveljavljeno infrastrukturo strežnikov, posredniških (*proxy*) strežnikov, vodorobranov (*firewall*) in podobno in s tem enostavno dosežemo skalabilnost in hitrost v obremenjenih sistemih.

Ne nazadnje je velik razlog tudi nenatančna specifikacija, saj je REST lahko vsakdo implementiral »po svoje«. Dandanes pa sistem vzdržuje sam sebe. Zaradi tega, ker je razširjen vsepovod, tudi pri velikih (Amazon, Twitter, Google, Facebook, Dropbox ...), ga uporablja vedno več podjetij, zato postaja vedno bolj razširjen ...

Hypermedia as the Engine of Application State

V doktorski dizertaciji iz leta 2000, ki je postavila temelje arhitekturi REST, je Roy Thomas Fielding postavil zelo fleksibilen sistem. Opisal je sistem, ki ima le nekaj dobro definiranih začetnih točk (URL), aplikacija pa nato sledi hipermehijskim dokumentom (dokumentom s povezavami), ki jih vrne strežnik. V aplikacijo bi – po njegovem – vgradili le poznavanje ključnih besed, ki jih lahko vrne strežnik, ne pa tudi naslovov sredstev. Dokument, ki ga vrne strežnik, bi hkrati definiral, katere ukaze lahko v danem trenutku aplikacija sploh izvede.

Oglejmo si primer z Wikipedie. Denimo, da imamo bančni sistem in mu pošljemo zahtevko za vpogled v račun 12345. (Pri tem odmislimo varnostne zahteve, overovitev itd, da primer ne postane preveč zapleten.)

```
GET /account/12345
```

Strežnik vrne dokument s povezavami do veljavnih storitev.

```
<?xml version="1.0"?>
<account>
  <account_number>12345</account_number>
  <balance currency="usd">100.00</balance>
  <link rel="deposit" href="http://somebank.org/account/12345/deposit" />
  <link rel="withdraw" href="http://somebank.org/account/12345/withdraw" />
  <link rel="transfer" href="http://somebank.org/account/12345/transfer" />
  <link rel="close" href="http://somebank.org/account/12345/close" />
</account>
```

Če želi program na primer prenesti denar na drug račun, mora poslati ukaz na naslov <http://somebank.org/account/12345/transfer> (tu vidimo, kako včasih postane parameter – 12345 – kar del specifikacije sredstva). Te povezave ne smemo imeti shranjene kar v programu, saj se lahko s časom spremeni, temveč jo moramo vedno prebrati iz dokumenta. Vedeti moramo le, da je operacija prenosa sredstev vedno označena s ključno besedo »transfer«. To je osnova pristopa »Hypermedia as the Engine of Application State«.

Denimo, da smo prenesli 125 dolarjev in je račun sedaj 25 dolarjev »v minusu«. Ponovna izvedba ukaza

```
GET /account/12345
```

bi sedaj vrnila drugačen dokument:

```
<?xml version="1.0"?>
<account>
  <account_number>12345</account_number>
  <balance currency="usd">-25.00</balance>
  <link rel="deposit" href="http://somebank.org/account/12345/deposit" />
</account>
```

Ker je stanje računa negativno, strežnik sedaj dopušča le polog na račun, onemogočenih storitev pa nam niti ne objavi.

Tako zamišljen pristop je izredno fleksibilen, vendar se v praksi skoraj ne uporablja. Večina spletnih storitev, ki nudijo nadzor z arhitekturo REST, uporablja kar dobro definiran nabor znanih lokatorjev URL, ki jih vgradimo v program.

JSON

Za prenos strukturiranih podatkov po arhitekturi REST se pogosto uporablja format JSON [JavaScript Object Notation], zato se spodobi, da ga na kratko predstavimo tudi tu.

Gre za zelo enostaven format, v katerem osnovni element predstavlja par *ime:vrednost*. Več parov lahko sestavimo v seznam in jih ločimo z vejicami. *Ime* para je vedno enostaven niz, zaprt v dvojne narekovaje, *vrednost* pa lahko predstavlja nekaj osnovnih tipov:

- Število je lahko celo ali racionalno, zapišemo pa ga brez narekovajev.
- Niz zapremo v dvojne narekovaje.
- Logično vrednost predstavlja niza znakov *true* in *false*.
- Objekt zapremo v zavite oklepaje {}. Objekt lahko vsebuje nove pare podatkov.
- Polje zapremo v oglate oklepaje []. Polje lahko vsebuje nič ali več vrednosti, ki niso poimenovane.
- Null predstavlja prazno, neobstoječo vrednost.

Primer:

```
{  
  "glossary": {  
    "title": "example glossary",  
    "GlossDiv": {  
      "title": "S",  
      "GlossList": {  
        "GlossEntry": {  
          "ID": "SGML",  
          "SortAs": "SGML",  
          "GlossTerm": "Standard Generalized Markup Language",  
          "Acronym": "SGML",  
          "Abbrev": "ISO 8879:1986",  
          "GlossDef": {  
            "para": "A meta-markup language, used to create markup languages such  
as DocBook.",  
            "GlossSeeAlso": ["GML", "XML"]  
          },  
          "GlossSee": "markup"  
        }  
      }  
    }  
  }  
}
```

Od različice XE6 dalje vsebuje Delphi enoto *System.JSON*, ki definira osnovne tipe formata JSON.

Podatki, ki jih prenašamo v arhitekturi REST so skoraj vedno zaviti v JSON objekt, ki ga v Delphiju predstavlja tip *TJSONObject*. Ta implementira metodi *Parse* in *ParseJsonValue*, ki pretvorita format JSON v interno hierarhijo objektov. Za pretvorbo v obratno smer poskrbi metoda *ToString*.

Delo s hierarhijo objektov tipa TJSONxxxx zna biti nerodno, zato vsebuje Delphi (od različice XE6 dalje) tudi funkcije za pretvorbo Delphi objektov v format JSON in nazaj. Implementira jih razred *TJson*, ki ga najdemo v enoti *REST.Json*.

Primer rabe:

```
uses REST.JSON;

type
  TFOO = class
  private
    FFOO: string;
    FFee: integer;
  end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Foo: TFOO;
begin
  Foo := TFOO.Create;
  try
    Foo.FOO := 'Hello World';
    Foo.Fee := 42;
    Memo1.Lines.Text := TJson.ObjectToJsonString(Foo);
  finally
    Foo.Free;
  end;
  Foo := TJson.JsonToObject<TFOO>(Memo1.Lines.Text);
  try
    Foo.Fee := 100;
    Memo1.Lines.Add(TJson.ObjectToJsonString(Foo));
  finally
    Foo.Free;
  end;
end;
```

Pretvorniki razreda *TJson* upoštevajo naslednja pravila:

- Pretvarjajo le polja (*fields*), lastnosti (*properties*) ignorirajo.
- Ime JSON para je enako imenu polja brez uvodnega »F«. Prva črka imena para se spremeni v malo črko (»FFoo« postane »foo«).
- Če pretvornik ne najde polja, ki bi ustrezalo imenu v dokumentu JSON, ta element dokumenta ignorira.

V definiciji razreda lahko uporabite atribut *[JSONMarshalled(false)]* če polja ne želite pretvoriti v JSON. Z atributom *[JSONName('ime')]* lahko polju določite ime v formatu JSON. Atributi so definirani v enoti *REST.Json.Types*.

V starejših Delphijih lahko za obdelavo zapisa JSON uporabite knjižnico *SuperObject* ali funkcijo *JsonToDelphiClass*. Povezavi do njiju najdete proti koncu brošure.

Odjemalci

Za pisanje odjemalcev REST ne potrebujemo nič posebnega – le komponento, ki zna poslati http zahtevo in prejeti odgovor. Takšno najdemo v Indyju, pa v ICS, uporabimo lahko kar Windowsov WinInet, v novejših časih nam je na voljo od platforme neodvisni System.Net.HttpClient in še mnogo podobnih rešitev bi lahko našli.

Od različice XE5 dalje nam je na voljo še lepša rešitev – set komponent *REST Client Library*. Tega najdemo v enoti REST.Client in njenih sorodnikih (REST.HttpClient, REST.Authenticator.*., REST.Json ...). Knjižnica se je skozi leta posodabljala in v zadnji različici (10 Seattle) za prenos podatkov uporablja System.Net.HttpClient. Zaradi tega nam za rabo protokola https programu ni treba prilagati knjižnic SSL, saj za vse prenose podatkov (http in https) poskrbi operacijski sistem (Windows, OS X, iOS ali Android). Knjižnica deluje na platformah VCL in FireMonkey.

Če se bo naš odjemalec pogovarjal s strežnikom DataSnap, ki uporablja arhitekturo REST, potem lahko povezavo vzpostavimo s komponento *TDSRestConnection*. Več o tem bomo povedali v nadaljevanju, ko bomo opisali programiranje strežnikov REST s tehnologijo DataSnap.

Med REST odjemalce spadajo tudi programi, ki uporabljajo knjižnice za dostop do BaaS (Backend as a Service) platform Parse in Kinvey ter do Embarcaderove platforme EMS (Enterprise Mobility Services). Ker je tam REST uporabljen le kot komunikacijski medij in ni izpostavljen v programske vmesnikih, se z njimi v tej skripti ne bomo posebej ukvarjali. Nekaj podatkov najdete v povezavah na koncu brošure.

REST Client Library

Za prenos zahteve REST do strežnika potrebujemo nekaj gradnikov:

- *TRESTClient* vzpostavi povezavo (http ali https), pošlje zahtevo, počaka na odgovor in nas o njem obvesti.
- *TRESTRequest* opisuje zahtevo. Tu vpišemo ime sredstva, nastavimo parametre in podobno.
- *TRESTResponse* vsebuje odgovor, ki ga je poslal strežnik. Vsebuje tudi numerično kodo statusa (200 za OK oziroma kaj drugega za signalizacijo napake).
- Za predelavo odgovora v formatu JSON v tabelično obliko lahko uporabimo komponento *TRESTResponseDatasetAdapter*.
- Za overovitev (avtentifikacijo) uporabnika ali aplikacije uporabimo eno od komponent *TXXXAuthenticator*, ki jih bomo opisali v nadaljevanju.

Začetno stanje komponent najlažje vzpostavimo z orodjem *REST Debugger*, ki ga bomo bolj podrobno obdelali v nadaljevanju.

TRESTClient

Komponenti *TRESTClient* lahko nastavimo naslednje pomembne lastnosti:

- *Accept*
Vrednost http headerja »Accept«. Kadar je potrebno (če API to zahteva), vpišemo tip podatkov, ki ga pričakujemo (na primer *application/json* ali *application/xml*).
- *AcceptCharset*
Vrednost http headerja »Accept-Charset«. Znakovni nabor, ki naj ga uporabi strežnik. Običajno bo to *UTF-8*.
- *Authenticator*
Povezava na komponento za avtentifikacijo.
- *BaseURL*
Pot do REST strežnika.
- *Params*

Vsebina http parametrov, ki jih bomo poslali z zahtevo. Običajno parametre definiramo ob zahtevi (*TRESTRequest*), na tem mestu pa samo vpišemo parametre, ki vplivajo na http povezavo, ne pa na API klic REST strežnika. Parametre bomo opisali ob komponenti *TRESTRequest*.

- *SynchronizedEvents*
Če je vrednost *True*, se dogodki komponente prožijo v glavni niti, sicer v delovni niti.

Accept	application/json, text/plain; q=0.9, text/html;q=0.8
AcceptCharset	UTF-8, *;q=0.8
AcceptEncoding	<input checked="" type="checkbox"/> True
AllowCookies	<input checked="" type="checkbox"/> True
Authenticator	<input checked="" type="checkbox"/> True
AutoCreateParams	<input checked="" type="checkbox"/>
BaseUrl	http://www.songsterr.com/a/r/a
BindSource	RESTClient1.BindSource
ContentType	
FallbackCharsetEncoding	UTF-8
HandleRedirects	<input checked="" type="checkbox"/> True
IPIImplementationID	
LiveBindings Designer	LiveBindings Designer
Name	RESTClient1
Params	(TRESTRequestParameterList)
ProxyPassword	
ProxyPort	0
ProxyServer	
ProxyUsername	
RaiseExceptionOn500	<input type="checkbox"/> False
SynchronizedEvents	<input checked="" type="checkbox"/> True
Tag	0
UserAgent	Embarcadero RESTClient/1.0

TRESTRequest

Komponenti *TRESTRequest* lahko nastavimo naslednje pomembne lastnosti:

- *Accept, AcceptCharset, Params, SynchronizedEvents*
Enako kot v komponenti *TRESTClient*.
- *Client*
Povezava na komponento *TRESTClient*.
- *Method*
Http metoda, ki jo želimo izvesti (GET, PUT, ...).
- *Resource*
Opis sredstva na strežnikovi strani + morebitni sklici na parametre.
- *ResourceSuffix*
Morebiten tekst, ki bo dodan na konec zahteve, preden se pošlje strežniku.
- *Response*
Povezava na komponento *TRESTResponse*.

Accept	application/json, text/plain; q=0.9, text/html;q=0.8
AcceptCharset	UTF-8, *;q=0.8
AcceptEncoding	<input checked="" type="checkbox"/> True
AutoCreateParams	<input checked="" type="checkbox"/>
BindSource	RESTRequest1.BindSource
Client	RESTClient1
HandleRedirects	<input checked="" type="checkbox"/> True
LiveBindings Designer	LiveBindings Designer
Method	rmGET
Name	RESTRequest1
Params	(TRESTRequestParameterList)
Resource	songs.json?pattern={PATTERN}
ResourceSuffix	
Response	RESTResponse1
SynchronizedEvents	<input type="checkbox"/> False
Tag	0
Timeout	30000

TRESTRequest implementira tudi metodi, ki zahtevo pošljeta strežniku – *Execute* in *ExecuteAsync*. Prva izvede zahtevo v glavni, druga pa v delovni niti. Na mobilnih platformah vedno uporabljajte *ExecuteAsync*.

Parametri

Parametri zahteve REST vsebujejo ime, vrednost, ter način, kako naj se vstavijo v zahtevo. Slednji lahko zasede naslednje vrednosti.

- *pkCOOKIE*
Parameter se bo zapisal v piškotek (*cookie*).
- *pkGETorPOST*
Parameter bo poslan kot del URL (če je metoda GET) ali kot del telesa (če je metoda POST ali PUT).
- *pkURLSEGMENT*
Parameter bo vstavljen neposredno v URL. Primer take rabe smo videli na začetku, v poglavju »Hypermedia as the Engine of Application State«, kjer smo imeli URI »/account/{id}«. V tem primeru je {id} parameter tipa *pkURLSEGMENT*.
- *pkHTTPHEADER*
Parameter bo poslan kot del http headerja.
- *pkREQUESTBODY*
Parameter bo uporabljen kot (celotno) telo zahteve.

TRESTResponse

Ko se zahteva izvede, lahko iz komponente *TRESTResponse* preberemo naslednje pomembne vrednosti:

BindSource	RESTResponse1.BindSource
Content	(empty)
ContentEncoding	
ContentLength	0
ContentType	application/json
LiveBindings Designer	LiveBindings Designer
Name	RESTResponse1
RootElement	
Tag	0

- *ContentEncoding*
Uporabljeni znakovni nabor.
- *Content*
Odgovor.
- *ContentLength*
Dolžina odgovora.
- *ContentType*
Vrnjeni tip podakov (JSON, XML, ...).
- *ErrorMessage*
Morebitno http sporočilo o napaki.
- *Headers*
Http headerji.
- *JsonValue*
Odgovor, pretvorjen v format JSON (TJsonValue). Če je nastavljena lastnost *RootElement*, se bo procesiranje začelo šele pri elementu s tem imenom.
- *StatusCode*
Http status, kot ga je vrnil strežnik (200 pomeni OK).
- *StatusText*
Tekstovni del http statusa.

TRESTResponseDataSetAdapter

Komponenti *TRESTResponseDataSetAdapter* lahko nastavimo naslednje pomembne lastnosti:

- *Dataset*
Povezava do komponente, ki bo sprejela obdelane podatke.
- *FieldDefs*
Definicije podatkovnih polj (v kolikor nismo zadovoljni s privzetimi nastavitevami).
- *Response*
Povezava do komponente *TRESTResponse*.
- *RootElement*
Ime JSON para, kjer se začnejo podatki.

Active	<input type="checkbox"/> False
AutoUpdate	<input checked="" type="checkbox"/> True
Dataset	FDMemTable1 (TFieldDefs)
FieldDefs	
LiveBindings Designer	LiveBindings Designer
Name	RESTResponseDataSetAdapter1
NestedElements	<input type="checkbox"/> False
NestedElementsDepth	0
Response	RESTResponse1
ResponseJSON	
RootElement	
Tag	0

Avtentikacija

Za overovitev uporabnika ali aplikacije na formo postavimo eno od naslednjih komponent in jo povežemo s komponento *TRESTClient*.

Simple

TSimpleAuthenticator pošlje uporabniško ime in geslo kot del URL (GET) ali kot del telesa zahteve (POST). Čeprav je to videti zelo nevarno, je tak način čisto primeren, če uporabljam povezavo *https* in preverjam certifikate strežnika, tako da smo prepričani, da se v povezavo ni vključil prisluškovalec (*man in the middle attack*).

Basic

THTTPBasicAuthenticator vstavi uporabniško ime in geslo v http header po specifikaciji RFC2617. Tudi tak način je primeren, če je povezava varna (*https*).

OAuth1

TOAuth1Authenticator implementira avtentikacijo po standardu OAuth 1.0a. Primer rabe najdete v Delphiju priloženem primeru v mapi *Samples\Object Pascal\RESTDemo*. Avtentikacijo OAuth 1.0a je v večini strežnikov že zamenjal novejši in varnejši OAuth 2, zato se boste s tem načinom prijave le redko srečali.

OAuth2

TOAuth2Authenticator implementira avtentikacijo po standardu OAuth 2. Tudi primer tega avtentikatorja najdete v primeru *Samples\Object Pascal\RESTDemo*. Še bolje pa bo, če si ogledate videa »An Overview of OAuth2 using the REST Client Components« in »The New REST Client Library, Dive into the Details«, ki sta na voljo na YouTubu.

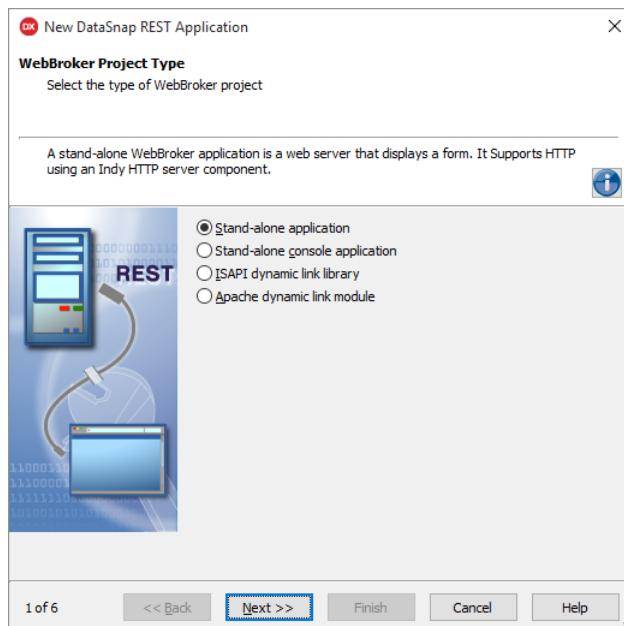
Strežniki

Tudi strežniški del lahko izdelamo z običajnimi TCP/IP komponentami ali (lažje) na podlagi komponente, ki vsebuje strežnik http (denimo *TidHTTPServer* iz knjižnice Indy ali *THttpServer* iz knjižnice ICS). Uporabite lahko tudi kak drugi strežnik, denimo *mORMot*. V največ primerih pa boste REST strežnik v Delphiju najlaže naredili kar s tehnologijo *DataSnap*.

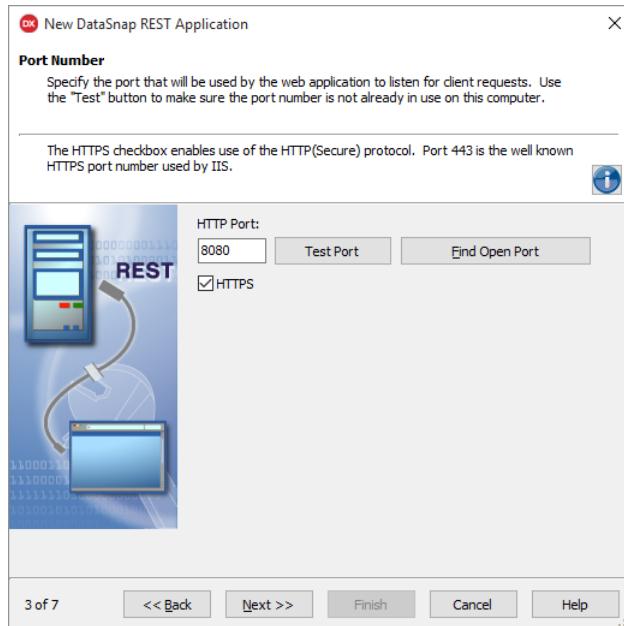
DataSnap

Strežnik *DataSnap* je možnost povezovanja z arhitekturo REST dobil v različici XE, tako da starejše različice Delphija za ta pristop niso primerne.

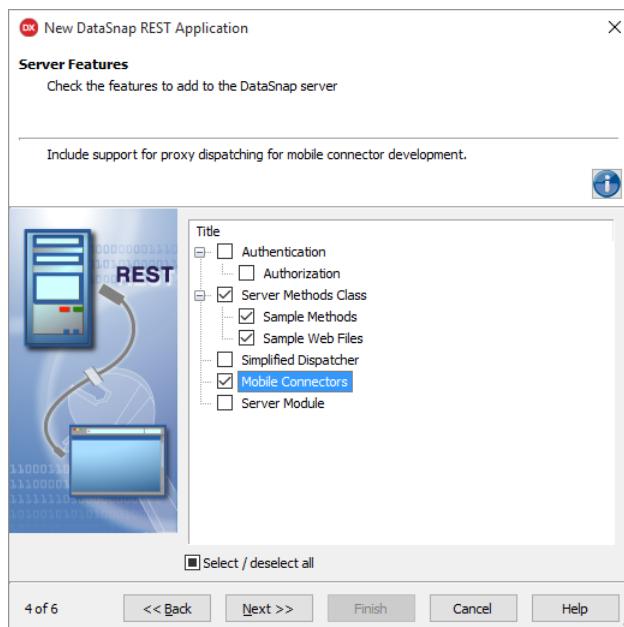
Za izdelavo strežnika uporabimo čarovnik *DataSnap REST Application*. Na voljo imamo naslednje tipe strežnikov: aplikacijo VCL ali FireMonkey, konzolno aplikacijo, dinamično knjižnico ISAPI, ki jo lahko integriramo v strežnik IIS, ali modul za strežnik Apache.



Podati moramo vrata, na katerih bo poslušal spletni strežnik, ki bo uporabljal protokola HTTP in/ali HTTPS. Če uporabljamo slednjega, moramo nastaviti tudi spletnne certifikate, zato bomo v našem enostavnem primeru ta korak izpustili.

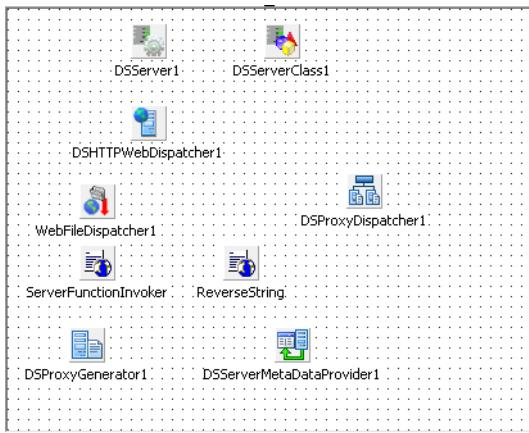


Določiti moramo še, katere dele programa naj zgradi strežnik. Poleg privzetih bomo vključili še možnost *Mobile Connectors*, ki pripravi knjižnice za izdelavo odjemalcev na mobilnih napravah.



Na koncu moramo določiti vrsto strežniškega modula. Uporabili bomo možnost *TdataModule*, ki zgradi *data module* in nanj postavi komponente.

Čarovnik bo naredil projekt, ki vsebuje spletni modul (Web Module), na njem pa je cela množica komponent.



TDSServer je glavna strežniška komponenta. Njena najpomembnejša lastnost, *AutoStart*, je že privzeto vključena – to pomeni, da se bo strežnik samodejno zagnal.

TDSServerClass določa *razred*, skupek funkcionalnosti, ki jo nudimo odjemalcem. Komponenta povezuje strežnik (*DSServer1*) z razredom, ki implementira funkcionalnost (*TServerMethods1* v enoti *ServerMethodsUnit1*). Povezava je narejena tako, da lastnost *Server* kaže na instanco serverja (*DSServer1*), v kodi pa sprogramiramo dogodek *OnGetClass*, ki vrne ime razreda (*TServerMethods1*). To ime bomo kasneje potrebovali v odjemalski kodi.

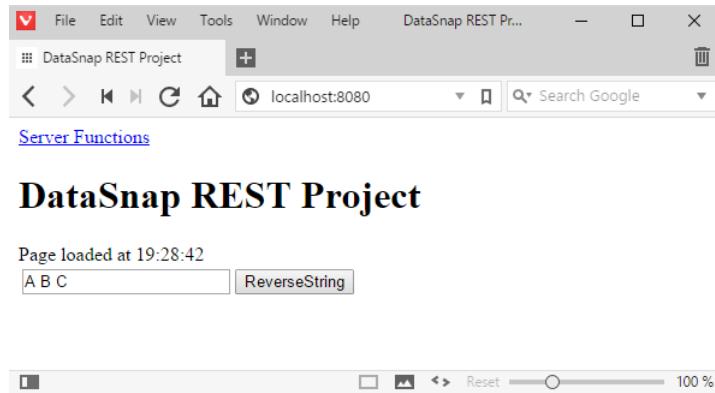
Preostanek komponent skrbi za pravilno vračanje spletnih strani in klicanje funkcij iz strežniškega razreda (*TServerClass1*). Pomembna sta *ServerFunctionInvoker* in *ReverseString* – komponenti, ki kličeta strežniške funkcije in rezultat vstavita v predlogo HTML. Dve sta tu zato, da vidite, da ni nujno, da bi se vse funkcije izvajale po istem kopitu in vračale strani na enak način. Vsebino, ki ju vračata, si lahko ogledate, tako da se iz spletnega brskalnika priključite na <http://localhost:8080> (na vratih 8080 posluša naš strežnik). (Mimogrede, če boste do strežnika dostopali le z odjemalskimi aplikacijami, spletnne predloge in na tak način sestavljeni rezultati niso pomembni. Ves prenos podatkov bo potekal po standardu JSON.)

Pomembna je tudi komponenta *TDSSProxyGenerator*, ki omogoča izdelavo posredniških (proxy) knjižnic za mobilne naprave. V lastnosti *Writer* si lahko ogledate, katere knjižnice so podprtne. Knjižnico za specifično mobilno napravo najlažje izdelamo tako, da poženemo strežnik, ter uporabimo priloženi program *Win32ProxyDownloader*. Postopek je opisan v dokumentaciji; povezavo (*Mobile Connectors*) najdete na koncu skripte.

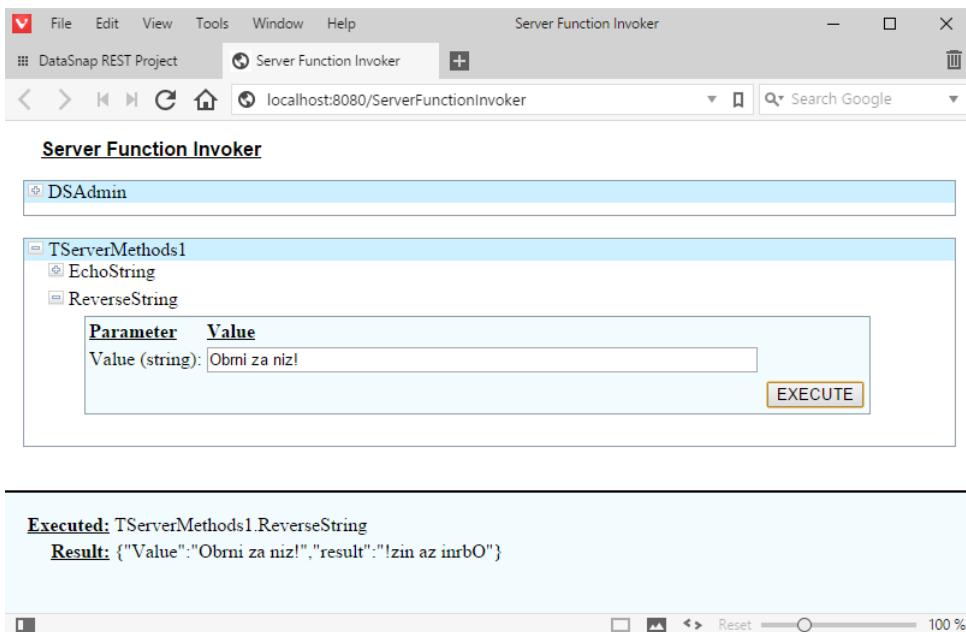
Čarovnik izdelo tudi obsežno strukturo imenikov, od katere je najpomembnejša veja *proxy*, v kateri so shranjene predloge za izdelavo posredniških knjižnic. V veji *templates* so shranjene predloge spletnih strani, v mapi *css* slogovne (oblikovalne) knjižnice, v mapi *images* slike, v mapi *js* pa podporne datoteke JavaScript.

Java Script REST
C# Silverlight REST
C++Builder DBX
C++Builder REST
FreePascal iOS 4.2 REST
FreePascal iOS 5.0 REST
Java (Android) REST
Java (Black Berry) REST
Java Script REST
Object Pascal DBX
Object Pascal REST
Objective-C iOS 4.2 REST
Objective-C iOS 7.1 REST
Objective-C iOS 8.1 REST

Strežnik lahko preizkusimo kar brez odjemalca. Poženemo brskalnik in se odpravimo na localhost:8080. Dobimo privzeto stran ki prikazuje delovanje funkcije *ReverseString*.



S klikom povezave *Server Functions* pridemo do strani, ki dokumentira vse strežniške funkcije, tako administrativnih (*DSAdmin* – te lahko tudi izklopimo) kakor tudi uporabniške, torej tiste, ki smo jih dodali sami. Te so zbrane v veji z imenom strežniškega razreda (v našem primeru *TServerMethods1*). Če je definiranih več strežniških razredov, bo vidnih več vej. Vsako funkcijo lahko takoj tudi preizkusimo. Ker smo pri izdelavi strežnika izbrali možnost *Sample Methods*, sta nam na voljo testni funkciji *EchoString* in *ReverseString*.



Na primeru na sliki vidimo, da se podatki prenašajo po standardu JSON (vrednost *Result* na dnu zaslona).

Avtentikacija

Overavljanje dodamo s komponento *TDSAuthenticationManager*, ki ji nastavimo dogodek *OnAuthenticate*, v katerem preverjamo uporabniško ime in geslo. Čarownik bo že sam dodal to komponento, če izberemo konfiguracijsko možnost *Authentication*.

DataSet v JSON

Za pretvorbo poljubnega dataseta v JSON je najbolj uporabiti komponento *TDataSetRESTRequestAdapter*, ki jo najdete na githubu.

Odjemalec za Windows/OS X/iOS

Odjemalec za operacijske sisteme, ki jih podpira RAD Studio, najlažje naredimo kar s čarovnikom. Izdelamo novo aplikacijo VCL ali FireMonkey, nato pa izberemo *File, New, Other*. V veji *DataSnap Server* najdemo ikone, s katerimi naredimo odjemalske module. Ker uporabljamo strežnik REST, moramo narediti odjemalski modul *DataSnap REST Client Module*.

Določiti moramo lokacijo strežnika (krajevni ali oddaljeni; tudi če izberemo slednjega, lahko kasneje za ime nastavimo *localhost* in delamo s krajevnim strežnikom), vrsto strežnika (DataSnap, WebBroker ali IIS) ter naslov, vrata ter uporabniško ime in geslo. Če ne uporabljamo avtorizacije, lahko slednji dve polji pustimo prazni. Ko kliknete zaključni *Finish*, mora biti strežnik pognan.

Čarovnik bo naredil majhen podatkovni modul, na katerem je le komponenta *TDSRestConnection*, ki skrbi za povezavo s strežnikom, poleg tega pa zelo pomemben razred *ClientClasses* (v enoti *ClientClassesUnit1*, ki vsebuje posredniške (*proxy*) funkcije z enakimi imeni in tipi parametrov, kot jih definira strežnik. Ko bi radi v odjemalcu uporabili strežniško funkcijo, le pokličemo funkcijo iz tega razreda in podatki se transparentno prenesejo do strežnika, kjer se funkcija izvede, vrne rezultat, ki se prenese nazaj do našega program in vrne kot rezultat posredniške funkcije.

V testnem programu imamo nekaj gumbov, ki kličejo strežniške funkcije. Na njihovem primeru natančno vidimo, kako je treba opraviti tak klic.

```
procedure TfrmDataSnapRESTClient.Button1Click(Sender: TObject);
begin
  Edit2.Text := ClientModule1.ServerMethods1Client.EchoString(Edit1.Text);
end;

procedure TfrmDataSnapRESTClient.Button2Click(Sender: TObject);
begin
  Edit2.Text := ClientModule1.ServerMethods1Client.ReverseString(
    Edit1.Text);
end;
```

Razreda *ClientClasses* nikoli ne urejamo ročno, saj ga je treba ponovno zgenerirati, kadar spremenjamo strežniške funkcije.

Dodajanje funkcij

Na enostavnem primeru si oglejmo, kako v strežnik dodamo novo funkcijo, ki jo bo uporabljal odjemalec. Ker pri uporabi mobilnih odjemalcev, ki niso napisani v Delphiju, ne moremo uporabljati komponent *TClientDataSet*, moramo celotno delovanje, tudi če upravljam podatkovno bazo, zasnovati okoli lastnih funkcij, ki se na strežniku preslikajo v podatkovne funkcije. (Če tudi odjemalca programirate v Delphiju, lahko za dostop do DataSnap REST strežnika uporabite kar običajno komponento *TDSProviderConnection*.)

Naš testni program poleg funkcij *EchoString* in *ReverseString*, ki jih je naredil čarovnik, implementira še funkcijo *PlusOne*, ki vrednosti polja prišteje 1, to shrani nazaj v polje in vrne rezultat.

```
type
  TServerMethods1 = class(TDSServerModule)
  private
    FValue: integer;
  public
    function EchoString(Value: string): string;
    function ReverseString(Value: string): string;
    function PlusOne: integer;
  end;

function TServerMethods1.PlusOne: integer;
begin
  Inc(FValue);
  Result := FValue;
end;
```

Ko spremenimo število ali parametre strežniških funkcij, moramo na novo izdelati tudi posredniške knjižnice odjemalcev – tako navadnih kakor tudi mobilnih. Za slednje smo že povedali, da to naredimo s programom *Win32ProxyDownloader*, v navadnih odjemalcih pa kliknemo z desno tipko na *TDSRestConnection* in izberemo *Generate DataSnap client classes*. Pri tem mora biti strežnik pognan.

V urejevalniku se pojavi nov zavihek z novo vsebino posredniškega razreda. Najbolje je, da vsebino prekopiramo v *ClientClasses* in novi zavihek zavrzemo. S tem je že vse pripravljeno za uporabno novih funkcij.

Odjemalci za mobilne naprave

Za nekatere vrste mobilnih naprav lahko odjemalce izdelamo v domorodnih (native) orodjih in za dostop do strežnika uporabimo posredniške knjižnice, ki jih je pripravil *Win32ProxyDownloader*. To lahko storimo na Androidih od 2 do 4, BlackBerryju (knjižnice v Javi), Windows Phone 7 (knjižnice v C#), iOS od različice 4 dalje (knjižnice v ObjectiveC), v drugih okoljih pa si lahko pomagamo s posredniškimi knjižnicami v jeziku JavaScript ali pa delamo neposredno s strežnikom z uporabo protokola REST.

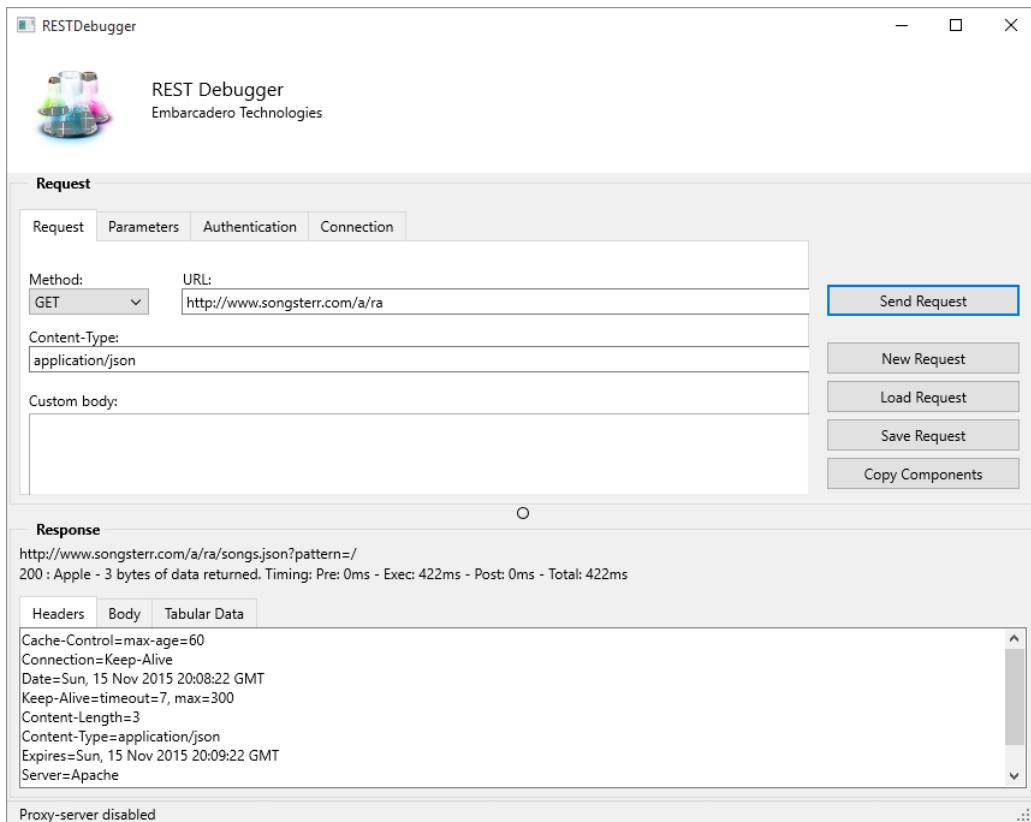
Orodja

Za zaključek si oglejmo še nekaj odličnih orodij, ki nam bodo pomagala pri razvoju in razhroščevanju REST aplikacij.

REST Debugger

V Delphi je že od različice XE5 dalje vgrajeno orodje *REST Debugger*. Gre za zelo uporabno, a rahlo okorno aplikacijo, ki bi si zaslužila nekaj nujnih popravkov. Na srečo je priložena tudi celotna izvorna koda, tako da lahko kaj popravite tudi sami. (Najdete jo v mapi z izvorno kodo – C:\Program Files (x86)\Embarcadero\Studio\17.0\source\data\rest\restdebugger za Delphi/RAD Studio 10 Seattle.)

REST Debugger je namenjen hitremu izvajanju zahtev REST. Nastavite metodo, URL, parametre, kliknete *Send Request* in preverite, ali se je zahteva pravilno izvedla in ali je strežnik vrnil smiselne rezultate.



Nastavite lahko tudi avtentikacijske podatke, posredniški strežnik in prikažete rezultat v tabelarični obliki.

The screenshot shows the REST Debugger interface. In the Request tab, the URL is set to `songs.json?pattern={PATTERN}` and the parameter `[URL-SEGMENT] PATTERN=Queen` is selected. The Response tab displays the JSON data returned from the API call `http://www.songsterr.com/a/ra/songs.json?pattern=Queen`. The response is a table of songs:

id	type	title	artist	chordsPresent	tabTypes
371	Song	Another One Bites the Dust	{"id":55,"type":"Artist","na... True	True	["PLAYER",...
270	Song	Bohemian Rhapsody	{"id":55,"type":"Artist","na... True	True	["PLAYER",...
527	Song	Rocket Queen	{"id":22,"type":"Artist","na... False	False	["PLAYER",...
23541	Song	Crazy Little Thing Called Love	{"id":55,"type":"Artist","na... True	True	["PLAYER",...
460	Song	Mississippi Queen	{"id":202,"type":"Artist","n... True	True	["PLAYER",...
119	Song	Love Of My Life	{"id":55,"type":"Artist","na... True	True	["PLAYER",...

Posamezne zahtevek lahko shranite v datoteke in jih kasneje po želji spet prikličete.

Najlepše od vsega pa je, da lahko z gumbom *Copy Components* na odložišče skopirate nabor skonfiguriranih komponent, ki bo izvedel želeni ukaz. Potem jih le še prilepite na formo.

Postman

Postman je razširitev za Chrome, ki omogoča enako funkcionalnost v drugačni preobleki. Zna pokazati nekaj več podatkov, delo lahko avtomatiziramo, ne zna pa prikazati rezultatov v tabelični obliki.

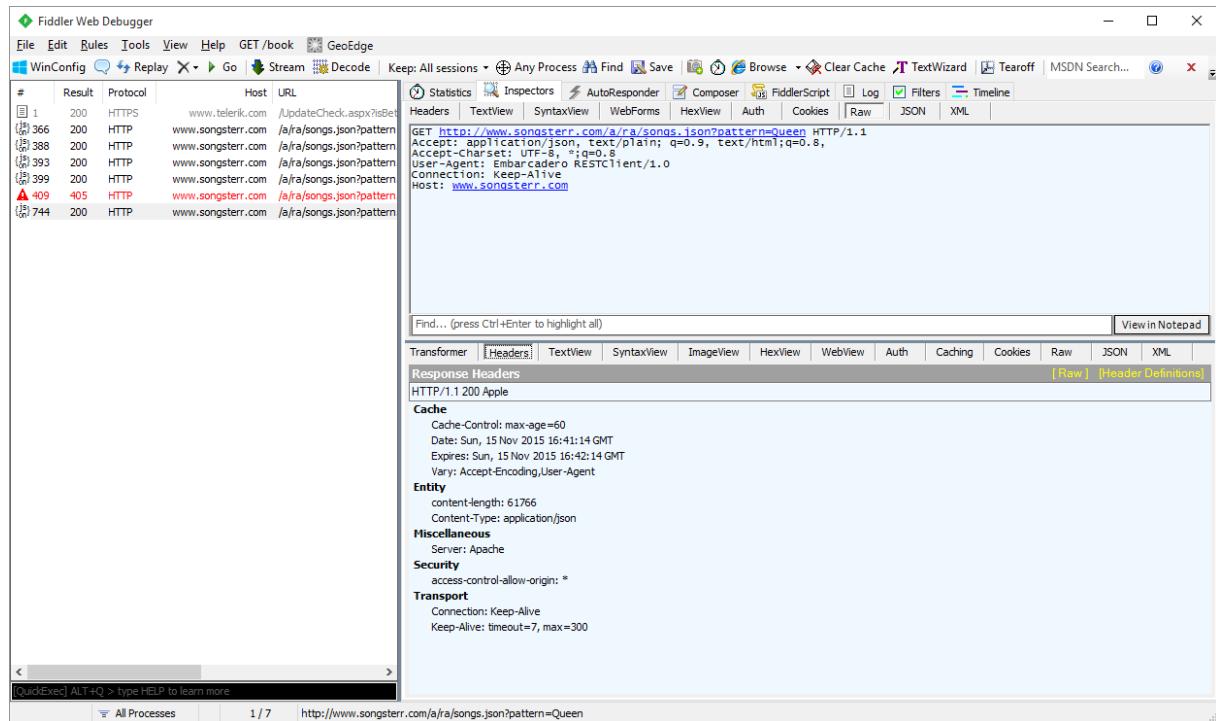
The screenshot shows the Postman application interface. At the top, there's a search bar, tabs for 'Builder' and 'Runner', and various icons for sync, import, and sign-in. The URL in the address bar is 'http://www.songsterr.com/a/ra/songs.json?pattern=Midr'. Below the address bar, there are tabs for 'GET', 'Params', 'Send', and 'Collection'. The 'Send' button is highlighted in blue. Underneath these tabs, there are sections for 'Authorization', 'Headers (0)', 'Body', 'Pre-request script', and 'Tests'. The 'Body' section is selected and has a dropdown menu showing 'No Auth'. The 'Body' tab is also selected. Below the tabs, there are buttons for 'Pretty', 'Raw', 'Preview', and 'JSON'. The 'JSON' button is selected and highlighted in green. The main content area displays a JSON response with line numbers from 1 to 20. The response object contains details about a song:

```
1  {
2   "id": 29537,
3   "type": "Song",
4   "title": "Beds Are Burning",
5   "artist": {
6     "id": 3747,
7     "type": "Artist",
8     "nameWithoutThePrefix": "Midnight Oil",
9     "useThePrefix": false,
10    "name": "Midnight Oil"
11  },
12  "chordsPresent": true,
13  "tabTypes": [
14    "PLAYER",
15    "TEXT_GUITAR_TAB",
16    "CHORDS",
17    "TEXT_BASS_TAB"
18  ],
19 }
20 }
```

At the bottom right of the JSON preview area, there's a button labeled 'Scroll to response'.

Fiddler

Pri iskanju zoprnih problemov ter ugotavljanju, kako druge aplikacije uporabljajo storitve REST, vam bo prišel prav sistemski posredniški strežnik Fiddler. V njem lahko analizirate vse povezave http, prikazati pa zna prav vse detajle.



Viri

REST

Representational state transfer [Wikipedia]

https://en.wikipedia.org/wiki/Representational_state_transfer

Hypermedia as the Engine of Application State

<https://en.wikipedia.org/wiki/HATEOAS>

REST Debugger

http://docwiki.embarcadero.com/RADStudio/Seattle/en/REST_Debugger

REST Client Library

http://docwiki.embarcadero.com/RADStudio/Seattle/en/REST_Client_Library

Developing REST Servers in Delphi XE using DataSnap

<http://www.embarcadero.com/rad-in-action/datasnap-rest>

Mobile jQuery Client for Delphi REST Server

http://blog.marcocantu.com/blog/mobile_jquery_delphi_rest.html

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

http://docwiki.embarcadero.com/RADStudio/Seattle/en/Tutorial:_Using_the_REST_Client_Library_to_Access_REST-based_Web_Services

REST Demo Sample

http://docwiki.embarcadero.com/CodeExamples/Seattle/en/REST.RESTDemo_Sample

The New REST Client Library, Dive into the Details

<https://www.youtube.com/watch?v=MpZHOSmIgSU>

The New REST Client Library: A Tool of Many Trades

<https://www.youtube.com/watch?v=nPXYLK4JZvM>

DataSnap

DataSnap.FireDACJSONReflect REST Server Client Sample

http://docwiki.embarcadero.com/CodeExamples/Seattle/en/DataSnap.FireDACJSONReflect_REST_Server_Client_Sample

Tutorial: Using a REST DataSnap Server with an Application and FireDAC

http://docwiki.embarcadero.com/RADStudio/Seattle/en/Tutorial:_Using_a_REST_DataSnap_Server_with_an_Application_and_FireDAC

Tutorial: Using a REST DataSnap Server with an Application

http://docwiki.embarcadero.com/RADStudio/Seattle/en/Tutorial:_Using_a_REST_DataSnap_Server_with_an_Application

DataSnap.FireDACJSONReflect REST Server Client Sample

http://docwiki.embarcadero.com/CodeExamples/Seattle/en/DataSnap.FireDACJSONReflect_REST_Server_Client_Sample

REST Servers in Delphi XE Part I - Building a REST Server

<https://www.youtube.com/watch?v=Qt6hP5EzBME>

REST Servers in Delphi XE Part II - Extending the REST Server

<https://www.youtube.com/watch?v=DdW8zssbQ0g>

DataSnap REST Application Wizard

http://docwiki.embarcadero.com/RADStudio/Seattle/en/DataSnap_REST_Application_Wizard

Using jQuery with DataSnap REST Applications

<https://www.youtube.com/watch?v=KYGV8ovjipk>

REST and Mobile DataSnap Client Development

<http://cc.embarcadero.com/Item/28543>

Mobile Connectors

http://docwiki.embarcadero.com/RADStudio/Seattle/en/Getting_Started_with_DataSnap_Mobile_Connectors

BaaS

REST BaaS Framework

<http://docwiki.embarcadero.com/Libraries/Seattle/en/REST.Backend>

REST Kinvey Provider

<http://docwiki.embarcadero.com/Libraries/Seattle/en/REST.Backend.KinveyProvider>

REST Parse Provider

<http://docwiki.embarcadero.com/Libraries/Seattle/en/REST.Backend.ParseProvider>

BaaS ToDo Sample

http://docwiki.embarcadero.com/CodeExamples/Seattle/en/REST.BaaS_ToDo_Sample

Enterprise Mobility Services Overview

http://docwiki.embarcadero.com/RADStudio/Seattle/en/Enterprise_Mobility_Services

<https://www.youtube.com/watch?v=LmfkUdcbqhs>

Autentifikacija

RFC 2617: HTTP Authentication: Basic and Digest Access Authentication

<https://tools.ietf.org/html/rfc2617>

OAuth 1.0a

<http://oauth.net/core/1.0a/>

OAuth 2

<http://oauth.net/2/>

An Overview of OAuth2 using the REST Client Components

<https://www.youtube.com/watch?v=EuEovgmVCUs>

Developer Skill Sprint - Closing the Loop on OAuth2.0

<http://community.embarcadero.com/fr/blogs/entry/developer-skill-sprint-closing-the-loop-on-oauth2-0>

Developing the Internet of Things: MISFIT Shine

<http://blog.appmethod.com/developing-the-internet-of-things-misfit-shine>

HTTP

Hypertext Transfer Protocol

https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

List of HTTP status codes

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

JSON

All about JSON

<http://json.org>

JSON in Delphi

<http://docwiki.embarcadero.com/RADStudio/Seattle/en/JSON>

TObject to JSON and JSON to TObject

<https://www.youtube.com/watch?v=TSqWoFvjj5g>

SuperObject

<https://github.com/onryldz/x-superobject>

<https://github.com/hgourvest/superobject>

JsonToDelphiClass

<http://www.pgeorgiev.com/?p=1832>

TDataSetRESTRequestAdapter

<https://github.com/andrea-magni/TDataSetRESTRequestAdapter>

Druga orodja

DataSnap-like Client-Server JSON RESTful Services in Delphi 6-XE5 with mORMot

<http://blog.synopse.info/post/2010/07/18/DataSnap-like-Client-Server-JSON-RESTful-Services-in-Delphi-7-2010>

kbmMW

<http://www.components4programmers.com/products/kbmww/index.htm>

Simple REST Client for Delphi with IdHttp

<http://jamiei.com/blog/2013/01/simple-rest-client-for-delphi/>

<https://github.com/jamiei/SimpleRestClient>

Fiddler

<http://www.telerik.com/download/fiddler>

Postman

<https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjb dggehcddcbn cddd m op>