BesaFHIR Guide

http://www.besasoftware.com

Version 1.X
2021-04-05

Contents

Introduction

A light-weight and fast FHIR client. It use Indy's HTTP component *(TIdHTTP)* and Windows native http client (*TNetHTTPClient*) for connecting. For FHIR object model, with support for efficiently parsing and writing in *JSON* and *XML* format.

BesaFHIR is Delphi components for handling FHIR (Fast Healthcare Interoperability Resources) messages. Besa FHIR can generate and parse JSON and XML messages. You can develop easy your applications for communicate your healthcare systems.

- Supports STU3,R4 version.
- You can use with Delphi 2009, 2010, XE, XE2, XE3, XE4, XE5, XE6, XE7, XE8, 10, 10.1,10.2, 10.3, 10.4.
- %100 Native pascal code. No DLL No OCX.
- It's support message groups and nested groups.
- Support for all FHIR data types.
- Licensed royalty-free per developer, per team, or per site.

Installation

Delphi versions

1. Run your setup file and install it.
2. Please add installation path to Delphi Editor’s Lib and Search Path.

Quick Start

This is the documentation of the the settings of the **TbsFhirClient** and the **CRUD** interactions you can perform with it. We will then give some examples of search interactions, and explain how to perform operations and transactions.

Before you create a fhir client, you need to load the message library. This library contains FHIR definitions. A library was created for each FHIR version.

Library names formatted as BSFRXX.BSL, XX is FHIR release name.
For FHIR STU 3 version library name BSFRSTU3.BSL.

```
//Load library...
procedure TForm1.FormCreate(Sender: TObject);
begin
  BSFHIRLibrary.LoadFromFile('BSFRSTU3.BSL');
end;
```

Create a FHIR Client

Before we can do any of the interactions, we have to create a new TbsFhirClient instance. This is done by passing the url of the FHIR server's endpoint as a parameter to the constructor:

```
var
  client: TbsFHIRClient;
begin
  client := TbsFHIRClient.Create('http://server.url/');
  //for use Indy
  client.HttpType:=hcIndy;
  client.HttpIndy:=IdHttp1;

  //for use TNetHTTPClient
  //client.HttpType:=hcNetClient;
  //client.HttpNetClient:=NetHTTPClient1;

end;
```

FhirClient communication options

To specify the preferred format JSON or XML of the content to be used when communicating with the FHIR server, you can use the EncodingFormat attribute:

```
client.EncodingFormat := ffmtJSON; // JSON is default encoding format.
```

The FHIR client will send all requests in the specified format. The default setting for this field is JSON.

When communicating the preferred format to the server, this can either be done by appending _format=[format] to the URL. The client uses the _format by default, but if you want, you can disable _format parameter:

```
client.UseFormatParam := false;
```

CRUD interactions

A TbsFhirClient named client has been setup in the previous topic, now let's do something with it.

Create a new resource

Assume we want to create a new resource instance and want to ask the server to store it for us. This is done using Create.

```
var
  client: TbsFHIRClient;
  patient : TPatient;
  id : string;

begin
  client := TbsFHIRClient.Create('http://server.url/');
  patient:=TPatient.Create;
  // ...
  // set up data
  // ...
  id := client.CreateResource(patient);
end;
```

Sample Code:

```
var
  client: TbsFHIRClient;
  patient : TPatient;
  id : string;

begin
  client := TbsFHIRClient.Create('http://server.url/');
  client.HttpIndy:=IdHTTP1;

  patient:=TPatient.Create;
  patient.active:=True;

  patient.text:=TNarrative.Create;
  patient.text.status:='generated';

  patient.identifier.Add;
  patient.identifier[0].system:='urn:system';
  patient.identifier[0].value:='12345';

  patient.name_.Add; //Adds new THumanName
  patient.name_[0].family:='James';
  patient.name_[0].given[0]:='John';

  Id:= client.CreateResource(patient);

  patient.Free;
  client.Free;
end;
```

Reading an existing resource

To read the data for a given resource instance from a server, you'll need its technical id. You may have previously stored this after a Create, or you have found its address in a ResourceReference (e.g. Observation.Subject.Reference).

The Read interaction on the TbsFhirClient has two overloads to cover both cases. Furthermore, it accepts both relative paths and absolute paths (as long as they are within the endpoint passed to the constructor of the TbsFhirClient).

```
// Read the current version of a Patient resource with technical id '1'

var
  client: TbsFHIRClient;
  patient : TPatient;

begin
  client := TbsFHIRClient.Create('http://server.url/');
  client.HttpIndy:=IdHTTP1;

  patient:=TPatient( client.ReadResource('Patient','1'));

  patient.Free;
  client.Free;
end;
```

Note that Read can be used to get the most recent version of a resource as well as a specific version, and thus covers the two '~logical' REST interactions ReadResource and ReadResourceV.

Updating a resource

Once you have retrieved a resource, you may edit its contents and send it back to the server. This is done using the Update interaction. It takes the resource instance previously retrieved as a parameter:

```
// Read the current version of a Patient resource with technical id '1'

var
  client: TbsFHIRClient;
  patient : TPatient;

begin
  client := TbsFHIRClient.Create('http://server.url/');
  client.HttpIndy:=IdHTTP1;

  // Read patient info.
  patient:=TPatient( client.ReadResource('Patient','1'));

  // Add a name to the patient, and update
  patient.name_[0].family:='James1';
  patient.name_[0].given[0]:='Given';

  client.UpdateResource(patient);

  patient.Free;
  client.Free;
end;
```

There's always a chance that between retrieving the resource and sending an update, someone else has updated the resource as well. Servers supporting version-aware updates may

refuse your update in this case and return a HTTP status code 409 (Conflict), which causes the Update interaction to throw a Exception with the same status code.

Deleting a Resource

The Delete interaction on the TbsFHIRClient deletes a resource from the server. It is up to the server to decide whether the resource is actually removed from storage, or whether previous versions are still available for retrieval. The Delete interaction has multiple overloads to allow you to delete based on a url or a resource instance:

```
var
  client: TbsFHIRClient;
  //patient : TPatient;
begin
  client := TbsFHIRClient.Create('http://server.url/');
  client.HttpIndy:=IdHTTP1;

  //patient:=TPatient( client.ReadResource('Patient','1'));

  //client.DeleteResource(patient);
  // or
  client.DeleteResource('Patient','1');

  client.UpdateResource(patient);

  //patient.Free;
  client.Free;
end;
```

The Delete interaction will fail and throw a Exception if the resource was already deleted or if the resource did not exist before deletion, and the server returned an error indicating that.

Note that sending an update to a resource after it has been deleted is not considered an error and may effectively "undelete" it.

Looking at LastResult

After the TbsFhirClient has received a response from the server, you will usually work with the resource instance that was returned. If you still need to check the response from the server, for example to lookup the technical id or version id the server has assigned to your resource instance, you can do this by looking at the LastResponseCode property of the FhirClient.

```
client.CreateResource(patient);

if (client.LastResponseCode = 201) then
begin
  ShowMessage('The response text for the resource is: ' +
client.LastResponseText);
end;
```

Searching for resources

FHIR has extensive support for searching resources through the use of the REST interface. Describing all the possibilities is outside the scope of this document, but much more details can be found online in the specification.

The FHIR client has a few operations to do basic search.

Searching within a specific type of resource

The most basic search is the client's Search(AResourceName: String; Params: String) function. It searches all resources of a specific type based on zero or more criteria. Criteria must conform to the parameters as they would be specified on the search URL in the REST interface, so for example searching for all patients named 'James' would look like this

```
var
  results:TBundle;
begin
  results:= client.Search('Patient', 'family=James');
end;
```

The search will return a Bundle containing entries for each resource found. It is even possible to leave out all criteria, effectively resulting in a search that returns all resources of the given type.

Sample code:

```
var
  client:TbsFHIRClient;
  bundle:TBundle;
  i:integer;
begin
  client:=TbsFHIRClient.Create('http://server.url/');
  client.HttpIndy:=IdHTTP1;

  bundle :=TBundle(client.Search('Patient', 'family=James'));

  ShowMessage('Total:'+IntToStr( bundle.total));

  if bundle.total > 0 then
  begin
    mLog.Lines.Append('--- id(s)---');
    for i:=0 to bundle.entry.Count-1 do
    begin
      mLog.Lines.Add(
        TPatient(bundle.entry[i].resource).id
      );
    end;
  end;

  bundle.Free;
  client.Free;
end;
```