Getting started with OpenAPI in Lazarus and Free Pascal

Michaël Van Canneyt

January 3, 2025

Abstract

OpenAPI is a worldwide standard for describing REST interfaces. In this article we demonstrate the recently added support for OpenAPI in Free Pascal and Lazarus.

1 Introduction

In this ever more connected world, there is little doubt that a REST API using JSON as a data exchange format is the most popular and simple way to create a publicly accessible API. Today, most - if not all - online public software services are using REST and JSON: The JSON format and the HTTP protocol make it very easy to use a REST API. This wealth of available APIs begs for a universal API description format. Such a format exists, it is the OpenAPI specification.

The OpenAPI specification is a continuation of the earlier swagger API format (many still use the name swagger), and has become the de facto standard to describe a REST API. It is maintained by the OpenAPI initiative, and is backed by all the major software companies.

The OpenAPI specification is todays equivalent of an IDL (Interface Description Language or the 'type library' in Windows) for native languages, or of the WSDL for the (older) SOAP API description format.

The OpenAPI specification is currently at version 3.1, and can be consulted at:

`https://openapis.org/`

or more precisely, the spec itself is at:

`https://spec.openapis.org/`

Basically, it describes all the possible endpoints of a REST service, the available HTTP methods, and all possible data structures accepted by the service or returned by the service. These data structures are described using JSON-Schema, following the JSON schema specification:

`https://json-schema.org/`

The OpenAPI specification describes a JSON-Based REST API, although some provisions are made to signal other formats in the Open API specification. In the below, we'll assume implicitly that the requests and responses are using JSON only.

There is a wealth of tools available that allow to show nice documention for an API starting from an OpenAPI description. They will usually also allow you to try the API right in the documentation, which is of course very convenient - and no doubt one of the reasons OpenAPI became so popular.

An example of such a tool is Swagger UI:

`https://editor-next.swagger.io/`

APIdog is another:

`https://apidog.com/`

you can see it in action in figure figure 1 on page 3. The popular Postman tool also supports OpenAPI:

`https://www.postman.com/`

There are much more tools available. Most of them are paying, but many offer limited free plans. Some will generate code for you, others do not. Most can be installed locally: usually it is the online version, packaged as an electron application.

A freely available tool (written in Java) is OpenAPI generator:

`https://openapi-generator.tech/`

None of these tools has native support for the Pascal language. Even the OpenAPI generator, with it's support for 50+ programming languages, does not offer Pascal support.

Does this mean Pascal programmers cannot use OpenAPI ? No, several Delphi tools to handle OpenAPI exist. And today, a tool exists for Free Pascal.

2 JSON-Schema support in Free Pascal/Lazarus

OpenAPI uses JSON-Schema to describe the data structures used in the API. So support for OpenAPI implies support for JSON-schema. JSON-Schema can be used to describe any JSON content. In particular, it can describe well-structured JSON such as returned by REST APIs. Because it is able to describe any JSON data, it has many features, most of which are not used in a typical REST API.

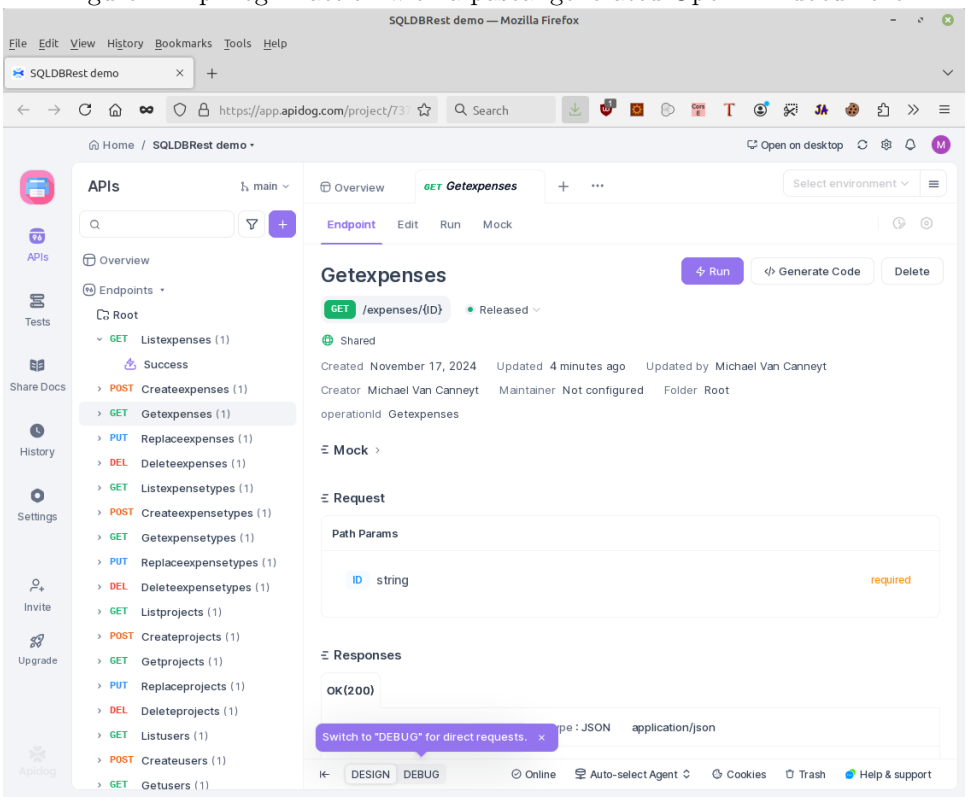
Recently, a set of units was developed for Free Pascal to be able to read, describe and write a JSON schema document: This is a set of classes that contain all the info in a JSON-Schema document. This allows you to access the information in a JSON-Schema in an easy-to-use set of classes. Any JSON-schema document can be read, and written.

Using these classes, code can be generated for the pascal data structures (records, classes, interfaces) that are described by the JSON-Schema. The following units exist:

fpjson.schema.types Various simple types (mostly enumerateds) that are used in the schema.

fpjson.schema.schema The actual JSON schema classes: these classes describe a JSON-Schema document. The basic class is - no surprise there - called `TJSONSchema`. It has properties for all constructs found in a JSON-Schema document.

Figure 1: ApiDog in action with a pascal-generated OpenAPI document



fpjson.schema.loader This contains a class that allows to load a JSON schema from a previously loaded or constructed `TJSONData` object.

fpjson.schema.reader This contains a class that allows to read a JSON schema from file/stream. No intermediate JSON structure is created, so it is faster and uses less memory than the loader.

fpjson.schema.writer This contains a class that allows to write a JSON schema to a `TJSONData` object or to stream/file.

fpjson.schema.validator This class allows you to determine whether a given JSON object corresponds to a JSON schema.

fpjson.schema.pascaltypes This unit contains a set of classes that describe a Pascal type corresponding to a JSON Schema. Basically, this converts a JSON Schema into one or more pascal structures.

fpjson.schema.codegen This unit contains classes that can be used to actually write out the pascal types defined using the types in `fpjson.schema.pascaltypes`.

The last unit contains one way to convert the types to pascal: JSON structures are mapped to pascal types: classes or records. These pascal types are described by the 3 classes in the `fpjson.schema.pascaltypes` unit:

TSchemaData This class is simply a container for all the type data; it has some methods to look up data structures by their name, and to add new structures to the collection.

TPascalTypeData Describes a pascal type. This can be a simple type (string, float, boolean, enum) or a complex type (record, class).

TPascalPropertyData As the name implies, it describes the elements (properties or fields) in a complex type.

In the code generator unit `fpjson.schema.codegen` you will find 2 classes:

TTypeCodeGenerator will generate a type description. For sequences this will be an array, for complex structures this can be a class or a record.

TSerializerCodeGenerator will generate a serializer helper for a given type. The helpers will read the fields of a pascal class or record from JSON, or write out the contents of the structure as JSON.

JSON-Schema can describe any JSON content, some content cannot be easily expressed as Pascal structures. To make the conversion easier, some limitations are in place:

- Advanced JSON-Schema features that introduce logic (for example the `if`, `then` and `else` tags) are not supported.
- Property names are well-behaved: they must be usable as Pascal identifiers: if a poperty name is encountered which is a keyword (`type` is partcularly popular), it will be escaped.

Meaning that the pascal code generator assumes 'simple' data objects, which translate easily to records, arrays and simple types. For most REST apis, this should not be an issue.

The code in these generator units also show certain decisions on how to map JSON to data:

- The data objects are not using properties but plain fields.
- The serializer does not use RTTI (no need, since it knows what is needed).
- Sequences are coded using dynamic arrays (as opposed to a TList).
- In case of classes, memory management is left to the user of the objects.

Some configuration flags (properties on the code generator classes) exist to allow to modify the behaviour of the code generator, the current implementation is designed to provide basic support for OpenAPI.

In case you don't like these decisions, you can of course simply use this unit as a inspiration to create your own writer: the main work of converting the JSON-Schema to pascal structures is implemented in the `fpjson.schema.pascaltypes` unit. Since this is just a first implementation, it is likely that other flags and/or class writers will be added.

To make a long story short, the combination of the above units will convert the following JSON-Schema description

```
{
  "type": "object",
  "properties": {
    "b": {
```

```
      "type": "string"
    },
    "c": {
      "type": "integer"
    }
  }
}
```

Depending on the value of the `WriteDtoClass` property on the code generator class, this will generate either a record:

```
TA = record
  b : string;
  c : integer;
end;
```

or an object:

```
TA = Class(TObject)
  b : string;
  c : integer;
end;
```

Of course, these are but simple examples, more elaborate schemas involving nested structures can be handled as well.

The JSON-Schema code generator will also create code to deserialize the object from a JSON structure to an actual instance. Every generated type will get a helper which introduces 4 methods:

```
class function deserialize(aJSONData : TJSONData) : Ta;
class function deserialize(aJSONData : String) : Ta;
function serializeObject : TJSONData;
function serialize : String;
```

The `deserialize` calls will create and fill an instance of `Ta`. Inversely, `Serialize` will create JSON from a `Ta` instance.

So, given the following JSON

```
{
  "b" : "hello, world!",
  "c" : 42
}
```

The following code

```
var
  a : TA;
begin
  a:=TA.Deserialize('{"b":"hello, world!","c": 42}');
end;
```

will load the data in the json and set the appropriate fields. Likewise, the following code

```
var
  a : TA;
begin
  a.b:='something';
  a.c:=12;
  Writeln(a.Serialize);
end;
```

Will write

```
{ "b" : "something", "c" : 12 }
```

The code which serializes the objects does not use any RTTI - since the structure of the objects is known, there is no reason to use RTTI. There is a configuration setting that will switch between Delphi or FPC JSON support, so these code generators can create code which is usable in Delphi.

3 OpenAPI support in Free Pascal/Lazarus

Similar in design to the JSON-Schema, a set of units was created to support reading and writing an OpenAPI specification document, and to generate classes that describe the REST services contained in the OpenAPI description. Inversely the classes can be constructed from your data and can then write out an OpenAPI description. The available units are currently:

fpopenapi.types contains many enumerated types. Members in all OpenAPI classes are implemented as properties, and the properties are created on the fly in their getter. To know whether a certain property was read from the description or not, all classes have a `HasKeyWord` function which accepts an enumerated (there is a value for each possible member) and returns `True` if the property was read from file.

fpopenapi.objects This unit contains the classes that describe the OpenAPI document: for every OpenAPI construct, a class exists. The openAPI document is described by the `TOpenAPI` class, the 'path' items are described using `TPathItem`, an operation using `TAPIOperation` and so on. The code completion of Lazarus will help you navigate through all classes.

fpopenapi.reader This unit contains a reader for an OpenAPI description (in JSON format, YAML is not supported at the moment). It will read the file and populate all classes from the `fpopenapi.objects` unit as needed.

fpopenapi.writer This unit contains a class that writes out a `TOpenAPI` instance.

fpopenapi.pascaltypes Similar to the `fpjson.schema.pascaltypes` unit, this unit contains classes that describe the services (class `TAPIservice)` and methods (class `TAPIServiceMethod`) from the OpenAPI description. All service definitions are contained in a `TAPIData` class.

fpopenapi.generators In this unit you'll find all classes that write out the services (class `TAPIservice)` and methods (class `TAPIServiceMethod`) as pascal objects.

fpopenapi.codegen this unit contains a class that will create and use all the classes to write all needed units to create a service client and/or server.

The `fpopenapi.pascaltypes` and `fpopenapi.codegen` units allow you to create one or more service classes from an openAPI description. The `TAPIData` class contains the following methods:

```
Procedure CreateDefaultTypeMaps; virtual;
Procedure CreateDefaultAPITypeMaps(aIncludeServer : Boolean);
Procedure CreateServiceDefs; virtual;
```

each of these methods must be called:

CreateDefaultTypeMaps will create type definitions for simple types such as integer, boolean, string etc.

CreateDefaultAPITypeMaps will examine the 'Components' property of OpenAPI and will generate classes (or records) based on what is found there. This uses the JSON-Schema to create the actual classes.

CreateServiceDefs Creates service definitions for all operations that have 'application/json' content.

An OpenAPI document can describe many endpoints. Often, these endpoints form logical groups, for example:

- a series of endpoints to manage contacts.
- a series of endpoints to manage a calendar.
- a series of endpoints to manage a user profile.

Typically these would be grouped in 3 distinct "services": a contacts service, calendar service and user service, each with several methods, corresponding to the operations for that service, for example: the contact service would contain methods called CreateContact, UpdateContact, DeleteContact, ListContacts, and similar methods for the calendar and profile endpoints.

How does the `CreateServiceDefs` decide what service names and method names to use ? For the services, there are 2 mechanisms:

- If tags are present in the path item, the first tag is used as the service name.
- If no tag is present, the first component in the path is used.

That means that an OpenAPI document with path items

```
/contact/{ID}
/contact
/calendar/{ID}
/calendar
/profile/{ID}
/profile/
```

would result in 3 services : `ContactService` `CalendarService` and `ProfileService`.

The method name is determined from the `operationId` key in the OpenAPI description, if present. If not present, it will use the method and path. Given the following 'paths' in an OpenAPI:

```
"paths" : {
  "/contact/{ID}" : {
    "get" : {
      operationId: "DeleteContact"
    },
    "put" : {
      operationId: "UpdateContact"
    }
  }
  "/contact/" : {
    "get" : {
      "get" : {
        operationId: "ListContacts"
      },
      "post" : {
        operationId: "CreateContact"
      }
    }
  }
}
```

This will result in a `ContactService` service definition with methods `DeleteContact`, `UpdateContact`,`ListContacts`, `CreateContact`.

If you wish to change the method names to something more to your liking, it is possible to specify a map of the following form:

```
OperationId=ServiceName.methodname
Verb:Path=ServiceName.MethodName
```

where 'Verb' is the HTTP verb (one of get, post, put etc.) and `Path` is the endpoint path in the OpenAPI description.

If the `CreateServiceDefs` method encounters an entry for `OperationId` or `Verb:Path` combination in the map, then the servicename and method name from the map will be used.

4 Creating an OpenAPI client

So, if we have an OpenAPI description file, how can these classes be used ?

Imagine the following `OpenAPI` description:

```
{
  "openapi": "3.1.0",
  "info": {
    "title": "Tic Tac Toe",
    "version": "1.0.0"
  },
  "paths": {
    "/simple": {
      "get": {
        "operationId": "list",
        "responses": {
          "200": {
```

```
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/a"
                }
              }
            }
          }
        }
      }
    }
  },
  "components": {
    "schemas": {
      "a": {
        "type": "object",
        "properties": {
          "b": {
            "type": "string"
          },
          "c": {
            "type": "integer"
          }
        }
      }
    }
  }
}
```

How can we turn this in a client application that connects to this REST service ?

FPC comes with a command-line tool which you can use to generate classes to access the endpoints of a REST API: The tool is called `openapi2pas`. It has the customary `--help` command-line option:

```
> ./openapi2pas -h
Usage: openapi2pas [options]
Where options is one or more of:
-a --async             Generate asynchronous service calls.
-b --abstract          Split server in abstract handler and implementation modules (and units
-c --client            Generate client-side service.
-C --config=FILE       Read config file with converter settings.
-d --delphi            Generate delphi code for DTO/Serializer/Service definitions.
-e --enumerated        Use enumerateds (default is to keep strings).
-h --help              This message.
-i --input=FILE        OpenAPI JSON File to use. Required.
-n --no-implementation Skip generation of server service module (only useful when -b is used)
-o --output=FILE       Base filename for output.
-q --quiet             Be less verbose.
-r --server            Generate a HTTP server module.
-s --service-map=FILE  Read service and method name mapping from file.
-u --uuid-map=FILE     Read (and write) a file with UUIDs for interfaces.
-v --verbose-header    Add OpenAPI description to unit header.
-w --write-config=FILE Write a configuration file with current settings and exit.
```

Looking at this help message, we can deduce that the simplest command to create Pascal code from an openapi description is:

```
> openapi2pas -i simpleservice.json -o simple
etInfo : a needs serialize: False, deserialize: True
etWarning : No mapping for list: (Tag= ""), Generated: list=SimpleService.list
etInfo : Map list on SimpleService.List
etInfo : Found 1 Dto types
etInfo : Created 1 services
etInfo : Writing Dto definitions to file "simple.Dto.pas"
etInfo : Generating type Ta
etInfo : Writing serialize helpers to file "simple.Serializer.pas"
etInfo : Generating serialization helper type TaSerializer for Dto Ta
```

The output of the command tells us that the tool has created 2 units. The first `simple.Dto.pas`, contains the types defined in our json file:

```
unit simple.Dto;

interface

uses types;

Type
  Ta = Class(TObject)
    b : string;
    c : integer;
  end;

implementation

end.
```

That's as simple as it gets. The `.Dto` suffix in the filename stands for `Data Transfer Object`. This is just a default name, the tool has support for a configuration file in which you can specify the names of all generated units.

The `simple.Serializer.pas` unit which contains the serialization and deserialization code is bigger:

```
unit simple.Serializer;

interface

uses
  fpJSON,
  simple.Dto;

Type
  TaSerializer = class helper for Ta
    class function Deserialize(aJSON : TJSONObject) : Ta; overload; static;
    class function Deserialize(aJSON : String) : Ta; overload; static;
  end;
```

Note that the serializer unit uses the `simple.Dto` unit. The implementation is not shown here. Again, simple code.

Of course, this is only the data returned by our service. In order to create a client, we need also something which describes the various endpoints. To get that file, we must specify the '-c' option to the tool. The below is a shortened version of the output:

```
openapi2pas -i simpleservice.json -o simple -c
[snip]
etInfo : Writing service interface to file "simple.Service.Intf.pas"
etInfo : Generating service interface SimpleService (UUID: {65F57AF1-9184-481F-B55E-E0DF82848
etInfo : Writing service implementation to file "simple.Service.Impl.pas"
etInfo : Generating class TSimpleServiceProxy to implement service interface SimpleService
etInfo : Generating implementation for class TSimpleServiceProxy
```

This will create the above `simple.Dto.pas` and `simple.Serializer.pas` units, but as you can see in the output, it will also create 2 additional units. The first is `simple.Service.Intf.pas`:

```
unit simple.Service.Intf;

interface

uses
    fpopenapiclient, simple.Dto;

Type
  // Service result types
  TaServiceResult = specialize TServiceResult<Ta>;

  // Service ISimpleService
  ISimpleService = interface  ['{65F57AF1-9184-481F-B55E-E0DF8284821A}']
    Function List() : TaServiceResult;
  end;
```

We can see that one method exists, corresponding to our endpoint `operationId`: `List`. The result of this call is a small wrapper record, using the generic `TServiceResult` type:

```
generic TServiceResult<T> = record
  RequestID : TServiceRequestID;
  ErrorCode : Integer;
  ErrorText : String;
  Value : T;
  constructor create(aServiceResponse : TServiceResponse);
  function Success : Boolean;
end;
```

It is defined in the `fpopenapiclient` unit. As you can see, it contains `ErrorCode` and `ErrorText` fields. If the server reports an error, these fields will contain the HTTP return code and an error message.

The last unit is the `simple.Service.Impl` unit. It contains a proxy class that implements the above `ISimpleService` interface. Calling the methods of the proxy

class will process all the arguments to the function, it will construct and execute a HTTP request and process (deserialize) the result.

Here is the generated proxy code:

```
unit simple.Service.Impl;

{$mode objfpc}
{$h+}

interface

uses
  fpopenapiclient
  , simple.Service.Intf                      // Service definition
  , simple.Dto;

Type
  // Service ISimpleService
  TSimpleServiceProxy = Class (TFPOpenAPIServiceClient,ISimpleService)
    Function List() : TaServiceResult;
  end;
```

The `TFPOpenAPIServiceClient` class is also implemented in the `fpopenapiclient` unit. It has methods to construct the service URL and execute a service request.

The methods of that class are used to implement the service proxy:

```
implementation
uses
  SysUtils
  , simple.Serializer;

Function TSimpleServiceProxy.List() : TaServiceResult;

const
  lMethodURL = '/simple';

var
  lURL : String;
  lResponse : TServiceResponse;

begin
  Result:=Default(TaServiceResult);
  lURL:=BuildEndPointURL(lMethodURL);
  lResponse:=ExecuteRequest('get',lURL,'');
  Result:=TaServiceResult.Create(lResponse);
  if Result.Success then
    Result.Value:=Ta.Deserialize(lResponse.Content);
end;

end.
```

As you can see, this is again quite simple code.

With these 4 units, we can now create a program to connect to our REST api server. We start by creating a serice proxy, setting the URL on which the REST API server is listening, and we specify the HTTP transport client to use.

After that, all is ready to connect to the server. We call the `List` method, and print the result if everything was OK.

```
uses
  fphttpwebclient, jsonparser,
  simple.Dto, simple.Service.Intf, simple.Service.Impl,

var
  lService : TSimpleServiceProxy;
  Res : TaServiceResult;

begin
  lService:=TSimpleServiceProxy.Create(Nil);
  lService.BaseURL:='http://localhost:8080';
  lService.WebClient:=TFPHTTPWebClient.Create(lService);
  Res:=lService.List();
  if Res.Success then
    Writeln('Got result: b: ',Res.Value.b,', c: ',Res.Value.c)
  else
    Writeln('Error : ',Res.ErrorCode,' - ',Res.ErrorText);
  Res.Value.Free;
  lService.Free;
end.
```

As you can see, writing 20 lines of code are sufficient to create a client for the REST API.

The `TFPOpenAPIServiceClient` class, of which `TSimpleServiceProxy` is a descendant, uses a `TAbstractWebClient` class to handle HTTP requests. This abstract class has a descendant called `TFPHTTPWebClient`, which uses `TFPHTTPClient` to execute the HTTP request: it is this class that is created and assigned to the `WebClient` property of our `TSimpleServiceProxy` class. A `TAbstractWebClient` descendant also exists for Synapse, if you prefer to use synapse, that is also possible. It should not be hard to write a descendant for the Indy http client as well.

5 Creating an OpenAPI server

In the above, we showed how to create a REST API service client. This will allow you to connect to any REST API server described by an OpenAPI specification document.

Of course, you may also want to create a REST API Service yourself. Either for production use, or for having a local server that implements the same API as some remote service API: this can be useful for testing implementations locally, for example to mock the REST API in unit test programs.

Naturally, the `openapi2pas` program supports creating a server which implements the REST API specified in an OpenAPI document. Generating the server is done with the `-r` option (only relevant output is shown in the below command output):

```
openapi2pas -i simpleservice.json -o simple -r
```

```
[snip]
etInfo : Writing server HTTP handler module implementation to file "simple.Module.Impl.pas"
```

The `simple.Module.Impl.pas` contains a descendent of the `TFPOpenAPIModule` class. This in turn is a descendent of the `TDatamodule` class, allowing you to open it in the lazarus form editor.

The unit looks as follows:

```
unit simple.Module.Impl;

interface

uses
  fpopenapimodule, httpprotocol, httpdefs, fphttpapp, httproute, simple.Dto;

Type
  TSimpleServiceModule = class(TFPOpenAPIModule)
  Public
    class Procedure RegisterAPIRoutes(aBaseURL : String; aUseStreaming : Boolean = False); ov
    Procedure HandleListRequest(aRequest : TRequest; aResponse : TResponse); virtual;

    function List() : Ta; virtual;
  end;
```

Quite simple. There are 3 methods. The first method, `RegisterAPIRoutes`, serves to register the endpoints in the HTTP router.

```
class Procedure TSimpleServiceModule.RegisterAPIRoutes(aBaseURL : String; aUseStreaming : Boo
begin
  RegisterOpenAPIRoute(aBaseURL,'/simple',@HandleListRequest,aUseStreaming);
end;
```

The HTTP router is part of the Free Pascal HTTP server functionality, and maps URLs to a functions that handles the URL. In the above case, `RegisterOpenAPIRoute` will map the URL

```
http://localhost/simple
```

To function `HandleListRequest`. Whenever the above URL is requested, the `HandleListRequest` will be executed, and the response is sent back to the client.

The implementation of this function is also generated by the `openapi2pas` tool. The `PrepareRequest` call will set the response type to `application/json` and will then execute an optional event handler, allowing you to modify anything you want in the request/response objects. It then executes the actual service method (in our case 'List') and captures the result in a local result variable (`lResult`):

```
Procedure TSimpleServiceModule.HandleListRequest(aRequest : TRequest; aResponse : TResponse);

var
  lResult : Ta;

begin
  lResult:=Default(Ta);
```

```
  try
    if PrepareRequest(aRequest,aResponse) then
      begin
      lResult:=List();
      try
        aResponse.Content:=lResult.Serialize;
      finally
        FreeAndNil(lResult);
      end;
      end;
    ProcessResponse(aRequest,aResponse);
  except
    on E : Exception do
      HandleRequestError(E,aRequest,aResponse);
  end;
end;
```

The handler then serializes the result (i.e. creates JSON from it) and returns the result to the client in `ProcessResponse`.

Lastly, the tool has also generated a `List` method, which you must fill with your code:

```
function TSimpleServiceModule.List() : Ta;

begin
  Result:=Default(Ta);
end;
```

We can change this method for example to:

```
function TSimpleServiceModule.List() : Ta;

begin
  Result:=Ta.Create;
  Result.b:='Hello';
  Result.c:=42;
end;
```

And with this we've created our service code.

If your OpenAPI document changes, then regenerating the code will of course destroy your service code. It would be better if the `HandleListRequest` and the actual `List` method were in 2 different units. The `openapi2pas` tool caters for this possibility. You can specify the `-b` or `--abstract` option. In that case, the tool will generate 2 units:

```
> openapi2pas -i simpleservice.json -o simple -r -b
[snip]
etInfo : Writing server HTTP handler module implementation to file "simple.Module.Handler.pas
etInfo : Writing server HTTP module implementation to file "simple.Module.Impl.pas"
```

The first unit `simple.Module.Handler.pas` contains the `HandleListRequest` call, but declares the `List` call as abstract:

```
unit simple.Module.Handler;
```

```
interface

uses
  fpopenapimodule, httpprotocol, httpdefs, fphttpapp, httproute, simple.Dto;

Type
  TAbstractSimpleServiceModule = class(TFPOpenAPIModule)
  Public
    class Procedure RegisterAPIRoutes(aBaseURL : String; aUseStreaming : Boolean = False); ov
    Procedure HandleListRequest(aRequest : TRequest; aResponse : TResponse); virtual;

    function List() : Ta; virtual; abstract;
  end;
```

The second unit (`simple.Module.Impl`) now contains only the `List` method:

```
unit simple.Module.Impl;

interface

uses
  simple.Module.Handler, simple.Dto, SysUtils, simple.Serializer;

Type
  TSimpleServiceModule = class(TAbstractSimpleServiceModule)
  Public
    function List() : Ta; override;
  end;

implementation

function TSimpleServiceModule.List() : Ta;

begin
  Result:=Default(Ta);
end;
```

Regenating the handler unit will now not overwrite your implementation: if you specify the `-n` or `--no-implementation` option in addition to the `-b` option, the `openapi2pas` tool will only generate the handler unit.

When the service changes, you can simply add new methods or change the methods as you see fit. New methods can be added easily with the Lazarus IDE's 'Abstract methods' refactoring wizard.

So, all that is left to do is to create the program. This again is simplicity itself:

```
program openapiserver;

uses fphttpapp, simple.Module.Impl;

begin
  TSimpleServiceModule.RegisterAPIRoutes('/');
  Application.Port:=8080;
```

```
  Application.Initialize;
  Application.Run;
end.
```

With these 4 lines of code, the REST API server is coded. It suffices to execute it, and the server is ready to answer requests at port 8080. So, we run our testclient:

```
> testclient
Got result: b: hello, c: 42
```

Which is what we were hoping to see.

The openapi2pas tool has many more options, which we did not yet treat, but we'll discuss them in more detail in a future article.

6 SQLDBRestBridge support for OpenAPI

Since some years, FPC comes with a set of components to expose the contents of a database using a REST API: SQLDBRestBridge. This technology was described in an earlier article; basically, it allows you to define a set of REST endpoints, each of which will return the result of a SQL query. The most basic query is simply `select * from mytable`. It also allows you to execute POST/PUT/PATCH/DELETE requests to modify the data.

SQLDBRestBridge has its own format to describe the endpoints it offers: the `/metadata` endpoint. This is used in the Lazarus IDE to offer advanced features such as defining datasets using the object inspector, etc.

With the appearance of the OpenAPI support in FPC, the SQLDBRestBridge has been enhanced to emit an OpenAPI document that describes all endpoints it offers.

To enable this feature, it suffices to include the `sqldbrestopenapi` unit in your project, and to set the `rdoOpenAPI` option in the `TSQLDBRestDispatcher`'s `Dispatchoptions` property. All the rest is automatic. If your SQLDBRestBridge dispatcher is configured to handle requests at

```
http://localhost:3000/REST
```

Then the OpenAPI document describing all endpoints will be available at

```
http://localhost:3000/REST/_openAPI
```

The exact location is configurable. As with all SQLDBRestBridge HTTP requests, readable response can be retrieved by adding a parameter to the request:

```
http://localhost:3000/REST/_openAPI?humanreadable=1
```

When you execute this request in the browser, it will look somewhat like figure 2 on page 18. With this OpenAPI specification document, you can now create a client that uses objects instead of datasets to access the data from the SQLDBRestBridge server. You can also test the OpenAPI specification in an openAPI editor such as swagger, as can be seen in figure 3 on page 18

Figure 2: SQLDBRest emitting an OpenAPI document

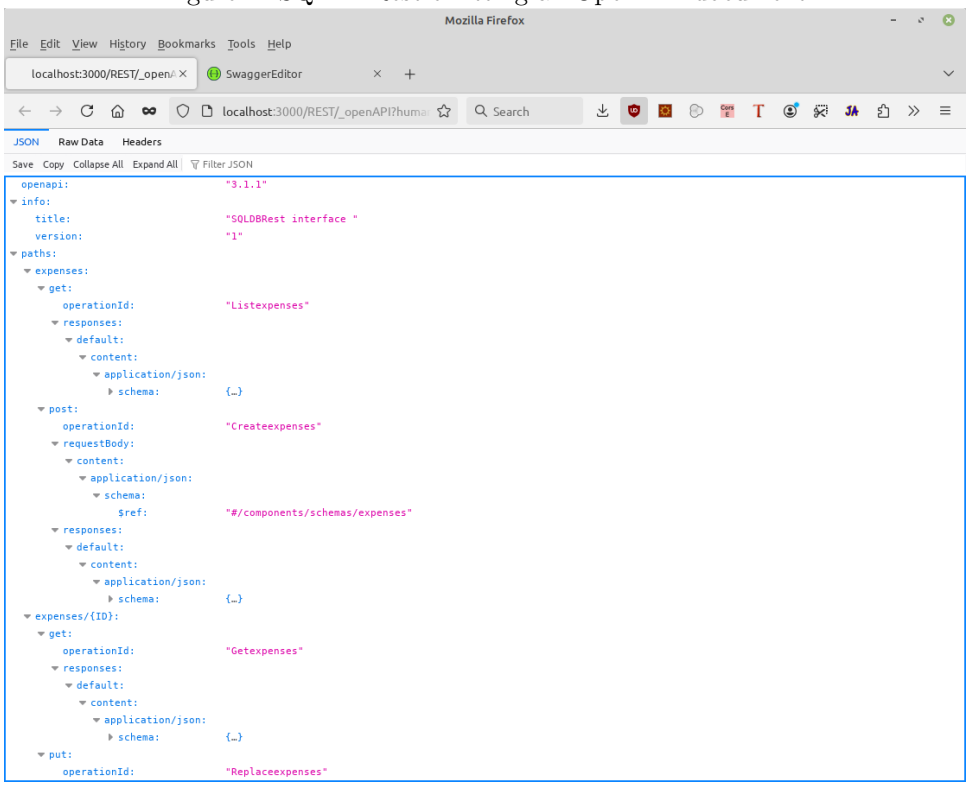
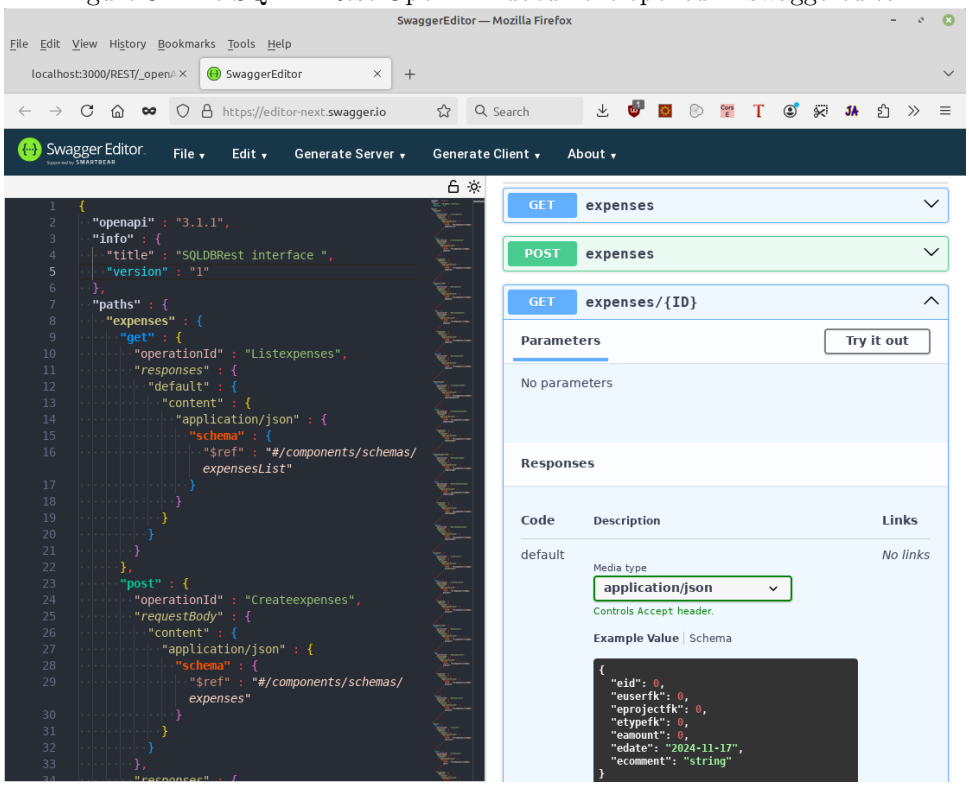


Figure 3: The SQLDBRest OpenAPI document opened in swaggereditor



7 Lazarus support

If the command-line is not your thing, then you are in luck: A wizard has been added to the Lazarus IDE. It offers an easy-to-use interface, offering the same possitibilities as the command-line tool.

To use it, you must install the 'lazopenapi' package. If you compile lazarus with the development version of FPC (3.3.1), then you can compile and install it in the usual manner: all the openapi and json-schema units are available.

If you compile with the released version of FPC (3.2.2), these units are not available in your installation. However, they are coded in such a way that they will compile with version 3.2.2 (in fact, they are developed using FPC 3.2.2). So, to compile the 'lazopenapi' package, you must first copy the files with JSON-Schema and OpenAPI support to the `ver_3_2_2` subdirectory of the packages' `lazarus/components/openapi` directory.

The following files must be copied from FPC's main branch in gitlab:

- All files under `packages/fcl-jsonschema/src`, which you can find at

  `https://gitlab.com/freepascal.org/fpc/source/-/tree/main/packages/fcl-jsonschema/src`

- All files under `packages/fcl-openapi/src`, which you can find at

  `https://gitlab.com/freepascal.org/fpc/source/-/tree/main/packages/fcl-openapi/src`

- From `packages/fcl-json/src`, you also need the `jsonwriter` file, you can find it at

  `https://gitlab.com/freepascal.org/fpc/source/-/blob/main/packages/fcl-json/src/jsonwrite`

Once you copied all these files to the `ver_3_2_2` subdirectory of the package, the 'lazopenapi' package can be compiled and installed in the IDE.

The package will install a menu item 'OpenAPI code generator...' under the 'Tools' menu. Clicking it will show a dialog where you can select the OpenAPI description file, and set the available options. At the bottom you can set the base filename. It serves as the base for all generated filenames.

The options are divided over various tabs, where the first one has some general options, figure 4 on page 20. The following options are applied globally:

Delphi code generate code for Dto/Serializer that can be compiled with Delphi.

verbose header When checked, adds the used command-line options to the header of the unit.

use enumerated types When checked, enums will be used instead of strings when the OpenAPI indicates there is an enumeration.

Unit name suffix template This is the template that is used to generate the unit names if they are not explicitly specified: the {`kind`} is replaced with the kind of unit (Dto, Serialize, Service.Intf etc.). The result is then appended to the base output name.

Unit name extension The file extension for the generated units.

Service name suffix The service names will have this suffix appended.

Figure 4: The general options

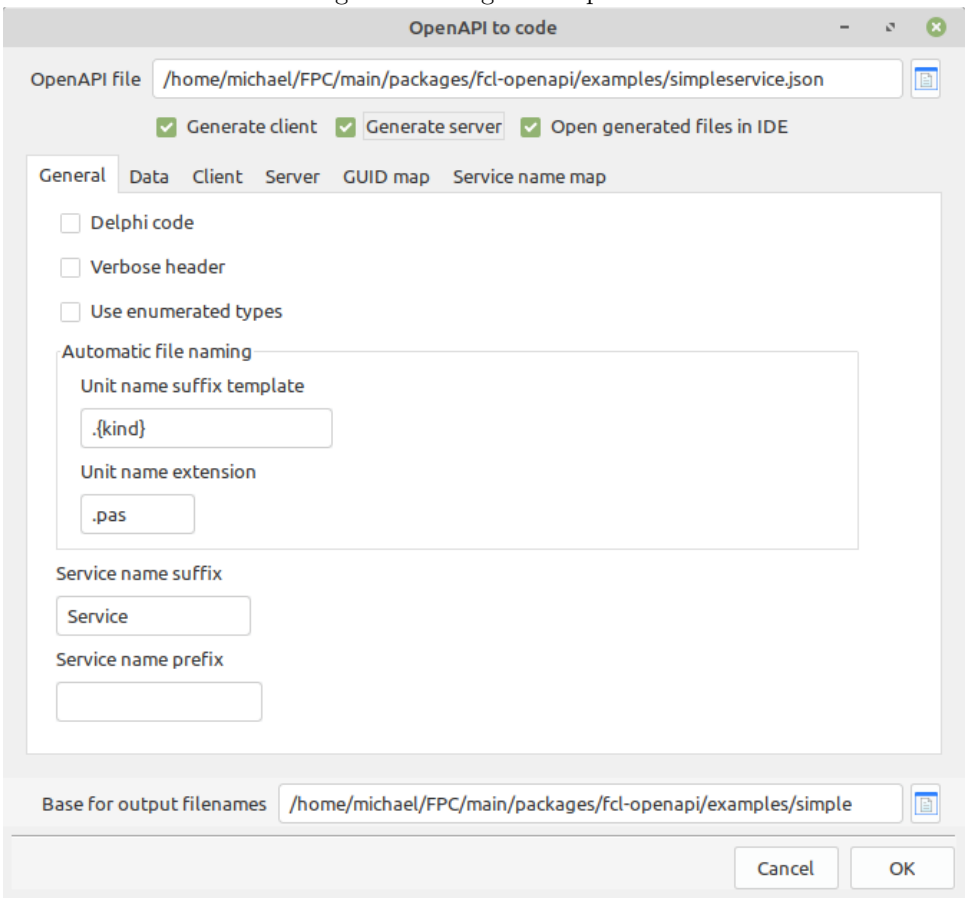
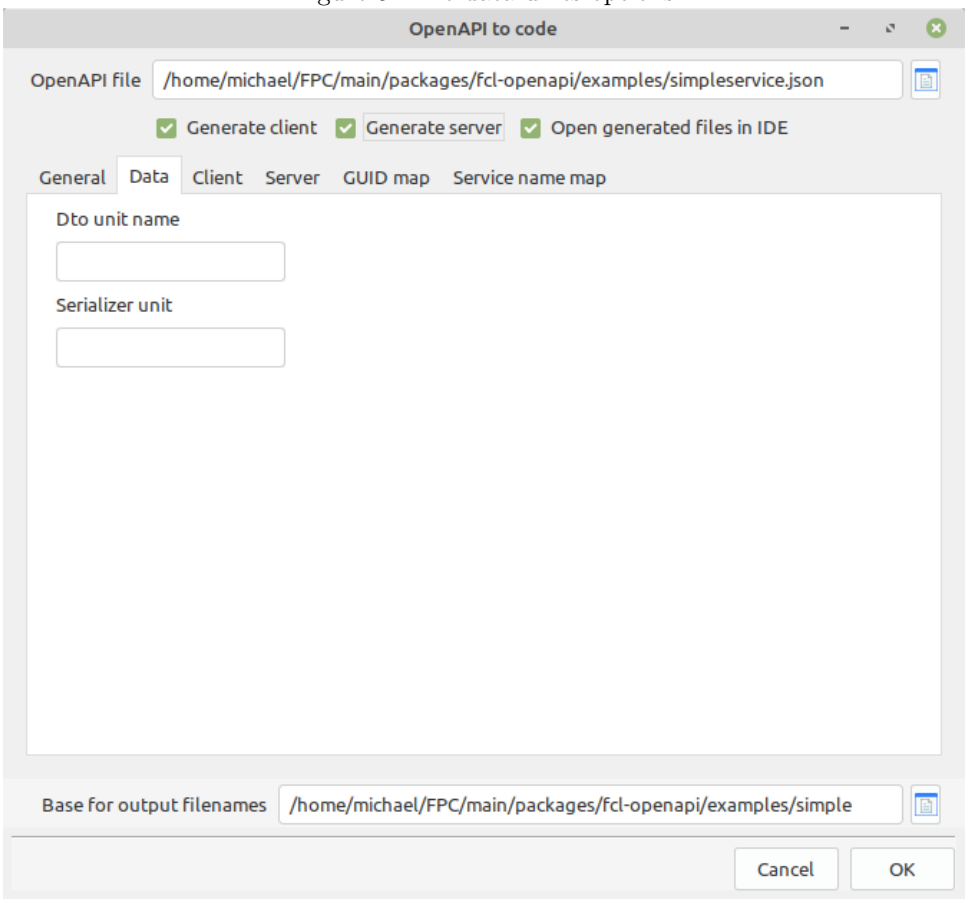


Figure 5: The data units options



Service name prefix The service names will have this prefix prepended.

If you don't like the default unit naming scheme, you can set the unit names for the `Dto` and `Serializer` units on the `Data` tab figure 5 on page 21

The names and options for the client service can be set on the third "Client" tab (see figure 6 on page 22). There are several options that can be set:

Asynchrone service calls Generate calls that use a callback to return the result. You can use this if your HTTP requests are run in another thread, or in pas2js.

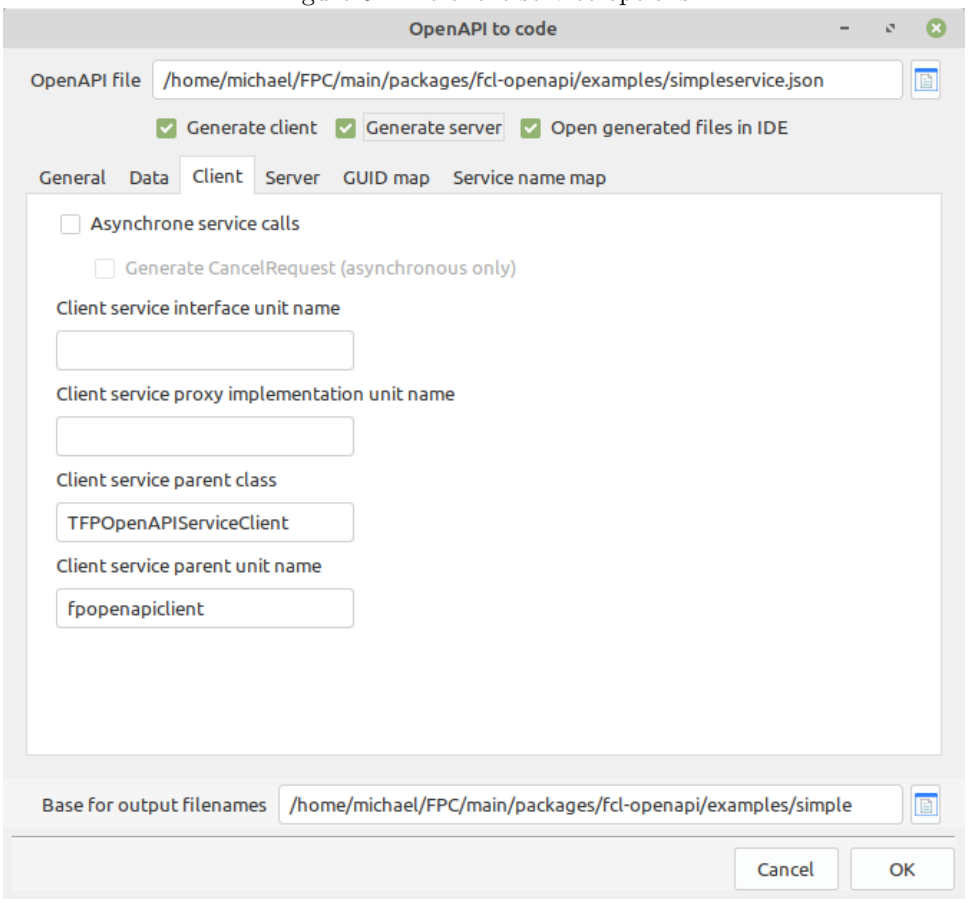
Generate `CancelRequest` When checked, adds a 'CancelRequest': the calls return a unique ID that can be ued to cancel the HTTP request.

Client service parent class The parent class for the service proxy classes: you can put the name of any descendant of `TFPOpenAPIServiceClient` here, or any other class which implements the same methods as that class.

Client service parent unit The unit in which the parent class for the service proxy class is defined.

Client service interface unit name the unit name for the client service interface definition: by default this is the base name appended with `.Service.Intf`.

Figure 6: The client service options



Client service proxy implementation unit name the unit name for the client service proxy implementation: by default this is the base name appended with `.Service.Impl`.

Lastly, the names and options for the server module can be set on the fourth "Server" tab (see figure 7 on page 23). Here also, many options that can be set:

Generate abstract service calls in HTTP handler module All service calls will be marked 'abstract' in the handler module, and moves the generation to the

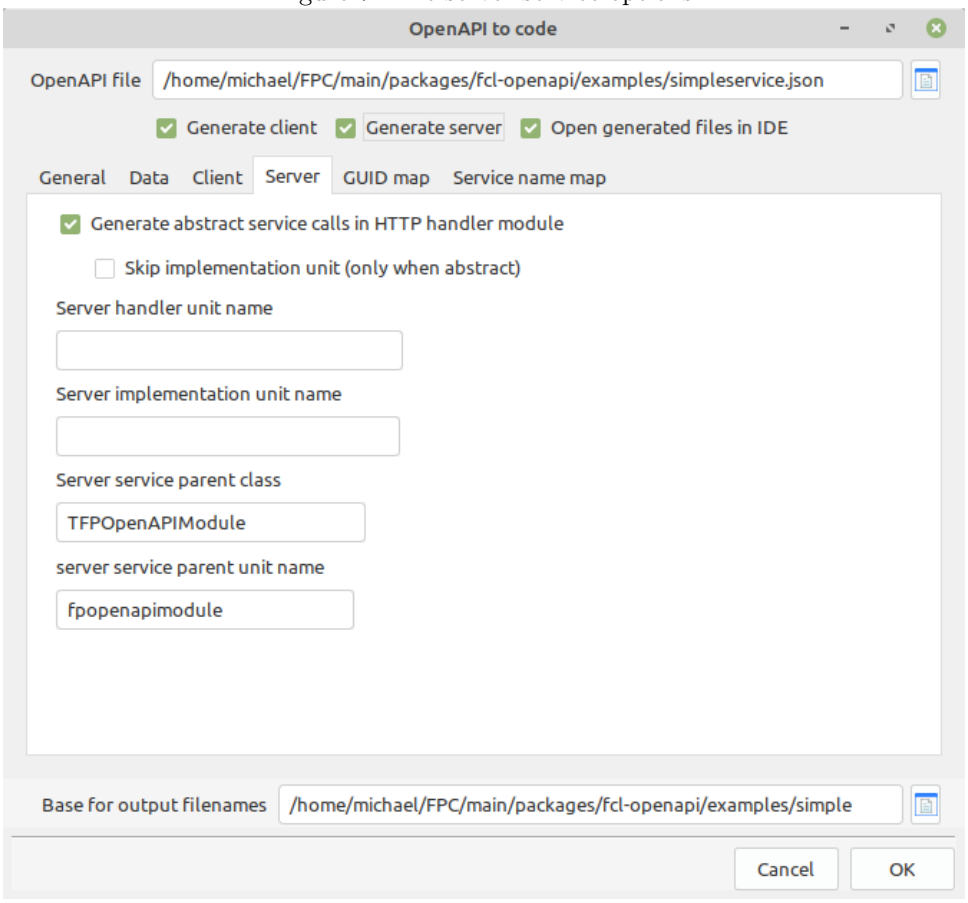
Skip implementation unit Check this if you are re-generating the definitions and do not wish to overwrite your implementation file.

Server service parent class The parent class for the service HTTP handler module classes: you can put the name of any descendant of `TFPOpenAPIModule` here, or any other class which implements the same methods as that class.

Server service parent unit The unit in which the parent class for the service HTTP handler module class is defined.

Server handler unit name the unit name for the service HTTP handler module: by default this is the base name appended with `.Module.Handler`. This unit is not generated when 'Generate abstract service calls' is checked.

Figure 7: The server service options



Server implementation unit name the unit name for the server actual service implementation: by default this is the base name appended with `.Module.Impl`. This unit is not generated when 'Generate abstract service calls' and 'Skip implementation unit' are checked.

Once you have filled in all the options, and clicked the 'OK' button, the units will be generated, and you can add them to your project: when you checked the 'Open generated files in IDE' they will be opened in the IDE and you'll be able to add them to the project with the project inspector with a few clicks.

8 Conclusion

This is the initial release of the OpenAPI support and code generator. It is already in use in production: while not all possibilities of OpenAPI are yet usable in the code generator, the generated code is sufficient to access REST API services that are "well-behaved". If you want to create your own APIs and stick to the constraints outlined in this document, you can create server and client implementatons, ready to go. The OpenAPI support is expected to be improved in several areas:

- Add support for multiple return types. In particular, error returns.
- Add the possibility to handle non-JSON request bodies and returns.

- Allow to use one file per service definitions and implementation, instead of one unit containing all services.
- Tighter integration with the Lazarus IDE, in particular the "New project" wizard, and the ability to automatically regenerate the files when the OpenAPI file changes.

We'll of course keep you updated on these changes. In addition, we've not yet covered all current possibilities, this also will be treated in a future article.