More OpenAPI support in FPC and Lazarus

Michaël Van Canneyt

January 4, 2025

Abstract

In a previous article, we introduced the OpenAPI support of Free Pascal and Lazarus. In this article, we introduce some additions and improvements which make the use of OpenAPI in Lazarus/FPC even easier.

1 Introduction

Since the previous article on OpenAPI, there have been some improvements to the Free Pascal and Lazarus support for OpenAPI. In this article we'll show some of these improvements and treat some points that were not treated in the previous article:

- Making sure GUIDs do not change when regenerating the API units.
- Providing custom names for APIs.
- New support for YAML files.
- New support for a server proxy datamodule.
- Lazarus 'new OpenAPI project' wizard.

We'll start with the existing things before proceeding to the improved things.

2 Immitable GUIDs

. For each service in the OpenAPI description file, an interface definition is generated. An interface definition in Delphi normally has a GUID associated with it, to uniquely determine the interface on the system.

In the previous article, the example file resulted in an interface definition as follows:

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

Note the GUID in the interface definition. The OpenAPI specification does not have the notion of a GUID for an API, but the interface support in Object Pascal does need this GUID. So the openapi converter generates a GUID for you: this is a randomly generated GUID. Because it is random, this also means that every time you invoke the converter tool, a new GUID is generated.

This is not very practical: it is better if a code generater is idempotent: A generated file remains exactly the same if it is regenerated. If the GUID changes every time the file is regenerated, it becomes more difficult to tell the actual changes. What is more, if the GUID is used in the code that makes use of the generated units, the code may simply stop working.

To remedy this, the OpenAPI to pascal conversion tool keeps a mapping to map service names to GUIDS. This mapping can be loaded from file and can be saved to file when done.

In the command-line utility you can specify the file name using the `-u` or `--uuid-map` command line argument: this can be used to specify the filename to load the map from and to save the map to when done, for example like this:

```
openapi2pas -i simpleservice.json -o simple -u guid.map
```

If the map file does not exist yet, it will not be read, but it will be created with all generated GUIDs when the converter is done. If the map file exists, it will be read before generating code, and it will be updated when the code is generated: if new interfaces were added to the API, the GUIDs that were generated will be added to the file.

The map file can be edited, it is a simple `key=value` format:

```
ActivitiesService={1528B206-7C98-458A-8A80-E1DF06834B78}
ContactsService={647E5197-7462-41CA-B39C-BF0D014FAC28}
DevicesService={3501BB33-B0DD-4E3C-AE3E-0078E15C364A}
LoginService={4296BFFC-978B-4465-9A9E-0AAC750F7644}
CalendarService={5826D547-3973-4869-B813-D187968E460B}
SettingsService={3161DE5B-D92A-49E5-A696-9D14F17CECCD}
SignupService={0B1220DD-A3D8-4F07-A848-9F9AEB08CED1}
UsersService={84F369FB-4FC0-4BED-9110-8CFDE1890683}
```

In the Lazarus OpenAPI code generation wizard, you have the option of editing the GUIDs manuall, on the 'GUID map' tab page of the wizard, see figure 1 on page 3. The editor allows you to set the mapping if you so desire: this is useful when a service name changes.

3 Providing custom names for APIs

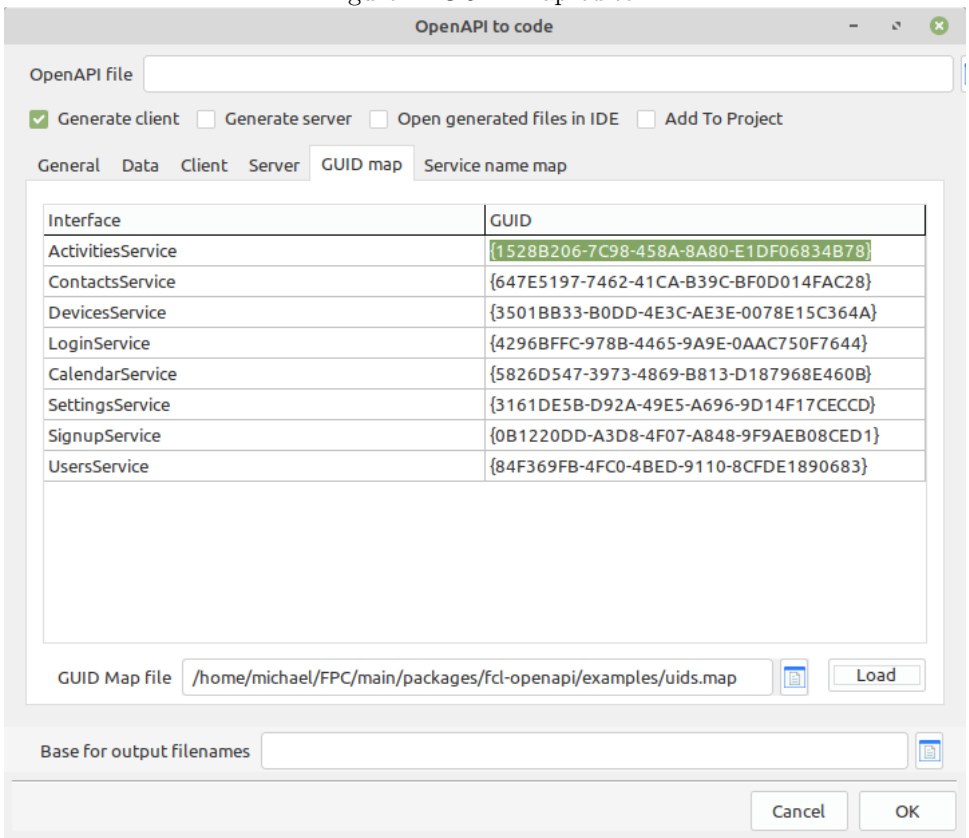
In the previous article, we described the algorithm that the code converter uses to create services and method names. If you don't like the mechanism that is used, or you wish to assure yourself that the interface and method names remain the same, you can specify a service map. This service map maps a service operation to a service (or interface) name and method.

As explained in the previous article on OpenAPI support, a service operation is identified in one of 2 ways, just as the code generator identifies methods:

OperationId the `OperationId` can be specified in the various operations in the OpenAPI file:

```
"paths": {
  "/simple": {
    "get": {
```

Figure 1: GUID map editor



```
          "operationId": "list",
        }
      }
    }
```

HttpVerb:URL the operation can be identified using the verb and URL: in the above example `get:simple`.

The map file is a simple `key=value` file, where the key is one of the above, and the value is the combination of service name and method, separated by a dot. The following is an example:

```
deleteActivity=SimpleActivitiesService.DeleteActivity
getAllActivities=SimpleActivitiesService.GetAllActivities
delete:/activities=SimpleActivitiesService.DeleteAllActivities
patch:/activities=SimpleActivitiesService.PatchActivities
get:/activity/info=SimpleActivitiesService.GetActivityInfo
post:/contact.accept=SimpleContactsService.AcceptContact
get:/contacts=SimpleContactsService.GetAllContacts
post:/contact=SimpleContactsService.AddContact
delete:/contact=SimpleContactsService.DeleteContact
```

The first two lines in this example show how to use the operation ID, the others identify an operation using the `HTTPVerb:URL` pair as a key.

The map can also be useful to move methods from one service to another, allowing you for example to regroup the methods in services that make more sense to you.

The command-line utility allows you to specify a filename for the service map using the `-s` or `--service-map` command-line options, for example like this:

```
openapi2pas -i simpleservice.json -o simple -s servicenames.map
```

Note that in difference with the GUID file, the specified file must exist. Also the code generator will not modify the file. It is only used to read the mapping. If no mapping is present for a given operation, the default mechanism is used to determine the service name and method name.

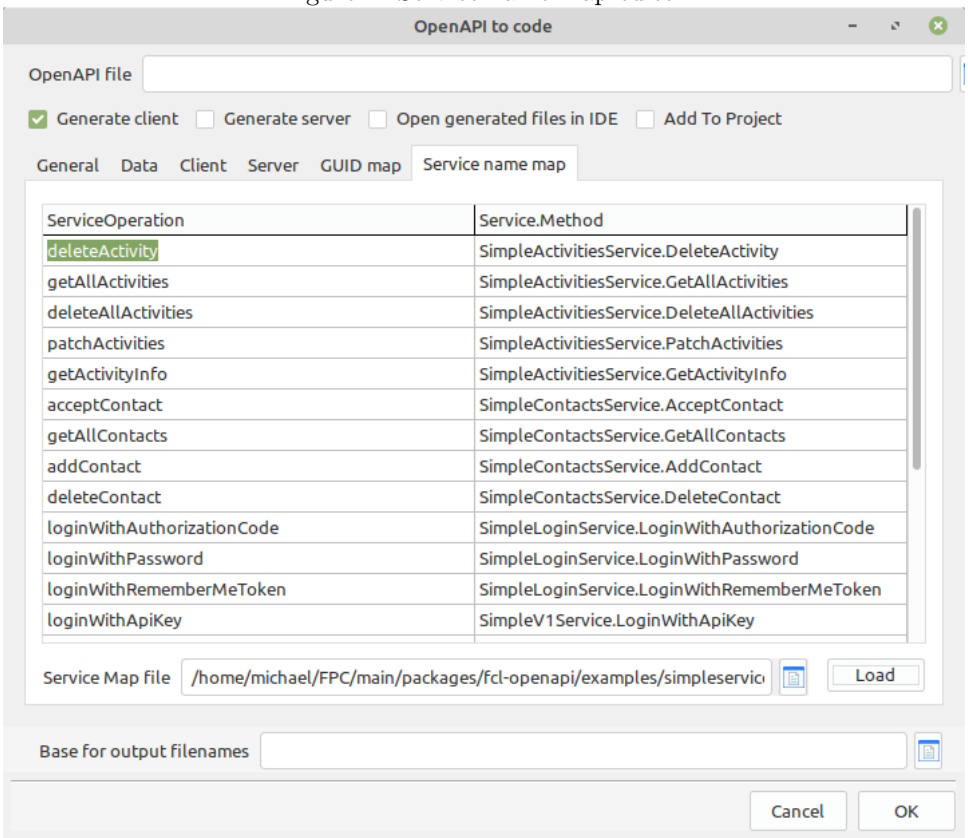
In the lazarus IDE, the `service name map` tab of the OpenAPI wizard allows you to load and edit the service map, as shown in figure 2 on page 5.

4 YAML support

The OpenAPI specification document uses the YAML format as the format for descibing an API using OpenAPI. YAML is a format that was introduced as an easier to read and write version of JSON. Where JSON is written with delimiter characters for fields, sequence items and objects, such as

```
{
  "openapi": "3.1.0",
  "info": {
    "title": "Tic Tac Toe",
    "version": "1.0.0"
  },
  servers : [
```

Figure 2: Service name map editor



```
      { "url": "http://localhost:8089/" },
      { "url": https://localhost:443/" }
    ]
  }
```

YAML does away with all these delimiters. It relies on dashes (-) and colon (:) characters to indicate sequences and values, and relies on spaces to indicate nested structures and end-of-lines to indicate the end of values. The above would be written as

```
openapi: 3.1.0
info:
  title: Tic Tac Toe
  version: 1.0.0
servers:
- url: http://localhost:8089/
- url: https://localhost:443/
```

Which is indeed easier on the eye as JSON. However, this apparent simplicity comes also with some drawbacks and ambiguities, aggravated by the fact that YAML has 2 modes: "Flow" mode (which is actually JSON) and block mode, which can be nested. Whether the original goal of having something simpler than JSON has been accomplished, is therefore debatable, but as a consequence, any JSON document is 'automatically' a valid YAML file.

Whatever your preference, YAML or JSON, FPC now has you covered: a YAML parser has been added to FPC, and the tool to read an OpenAPI description can now also handle OpenAPI specifications written using the YAML file format. There is therefore no need anymore to use an external tool to convert the YAML to JSON.

There is no class that specifically translates the YAML file to an OpenAPI document. Instead, the fpyaml.json unit contains a routine that converts YAML to JSON. So if you want to read an OpenAPI description from a YAML file, this is what you must do:

First, convert the YAML to JSON:

```
uses fpyaml.parser, fpyaml.data, fpyaml.json, fpopenapi.reader, fpopenapi.objects;

Function GetJSONFromYAML(const aOpenAPIFile : string) : TJSONStringType;

var
  lParser : TYAMLParser;
  lYAML : TYAMLStream;
  lJSON : TJSONData;

begin
  lYAML:=Nil;
  lJSON:=Nil;
  lParser:=TYAMLParser.Create(aOpenAPIFile);
  try
    lYAML:=lParser.Parse;
    lJSON:=YamlToJSON(lYAML);
    Result:=lJSON.FormatJSON();
  finally
    lYAML.Free;
```

```
    lJSON.Free;
    lParser.Free;
  end;
end;
```

and then use this to read the OpenAPI from the JSON data. The following routine will detect the file format from the filename, and do the necessary conversion if needed:

```
Procedure ReadOpenAPI(const aOpenAPIFile : string) : TOpenAPI;

var
  Loader : TOpenAPIReader;
  API : TOpenAPI;
  lJSON : String;

begin
  Loader:=Nil;
  Result:=TOpenAPI.Create;
  try
    Loader:=TOpenAPIReader.Create(Nil);
    if TYAMLParser.IsYamlFileName(aOpenAPIFile) then
      begin
      lJSON:=GetJSONFromYAML(aOpenAPIFile);
      Loader.ReadFromString(ResultAPI,lJSON);
      end
    else
      // Assume JSON
      Loader.ReadFromFile(Result,aOpenAPIFile);
  finally
    Loader.Free;
  end;
end;
```

The `TYAMLParser.IsYAMLFileName` will return `True` if the file name extension is `.yaml` or `yml`.

Note that not all YAML can be converted to JSON, as a YAML file can contain multiple YAML documents, and YAML allows sequences and objects to be used as keys in an object: a concept not translatable to JSON.

5 Creating a server proxy datamodule

An OpenAPI description can contain multiple services offered by a server. Each service is an interface, and each operation is a method of that service. In order to execute a service method, you need to create a service implementation instance, configure it with a web client to offer HTTP Services, and then call the method you want. If there are lots of services, then this becomes tedious.

The code generator has been enhanced to generate a 'server proxy': this is a datamodule which has a property for each service offered by the server. Each property is an interface as generated by the OpenAPI code generator. It will also create a web client, and offers a `BaseURL` property, which, when set, will set the property of the same name on each of the services. If so desired, the code generator will generate a

`.lfm` form file, so you can inherit visually from this module and add additonal code to it. By default, this datamodule descendant is called `TProxyServer`, but you can change this in the configuration file.

For the example service we showed in the previous article, this is the code that is generated:

```
Type
  TServerProxy = class(TDataModule)
  private
    FWebClient : TAbstractWebClient;
    FSimpleService : TSimpleServiceProxy;
    FBaseURL : String;
    function GetSimpleService : ISimpleService;
    Procedure SetBaseURL(const aValue : string);
  protected
    Procedure CreateServices; virtual;
  public
    constructor Create(aOwner : TComponent); override;
    Property SimpleService : ISimpleService read GetSimpleService;
    Property BaseURL : String Read FBaseURL Write SetBaseURL;
    Property WebClient : TAbstractWebClient Read FWebClient;
  end;

var ServerProxy : TServerProxy;
```

The `ServerProxy` variable is only generated if you ask for a form file.

To use this module, you can simply create it, set the base URL, and you're ready to execute methods:

```
var
  proxy : TServerProxy;

begin
  Proxy:=TServerProxy.Create(Nil);
  Proxy.BaseURL:='http://localhost:8089/REST';
  Proxy.SimpleService.List;
end;
```

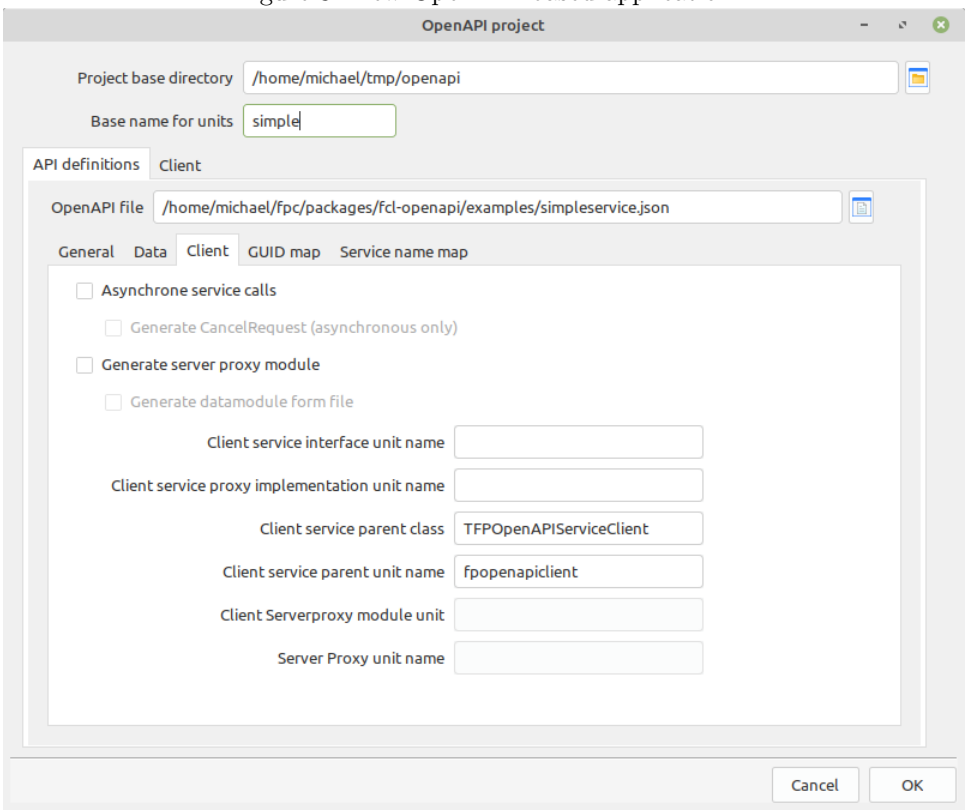
6 OpenAPI support for Lazarus projects

In the previous article on OpenAPI, we introduced a tool in the Lazarus 'Tools' menu. This allowed you to generate a set of units from an OpenAPI api description.

Meanwhile the OpenAPI support for lazarus has been improved, now you can create a complete project based on an OpenAPI description. 3 items have been added to the 'New project' dialog in the IDE:

- OpenAPI client application.
- OpenAPI server application.
- OpenAPI client and server application.

Figure 3: New OpenAPI based application



The first two of these items start a client or server application, generates the OpenAPI units according to the options as specified in the dialog (presented below) and then adds the units to the project. The last item creates both projects, and puts them in a project group.

All three items start with the same dialog, shown in figure 3 on page 9.

In this dialog, you specify the same information as in the wizard shown in the previous article, and some more:

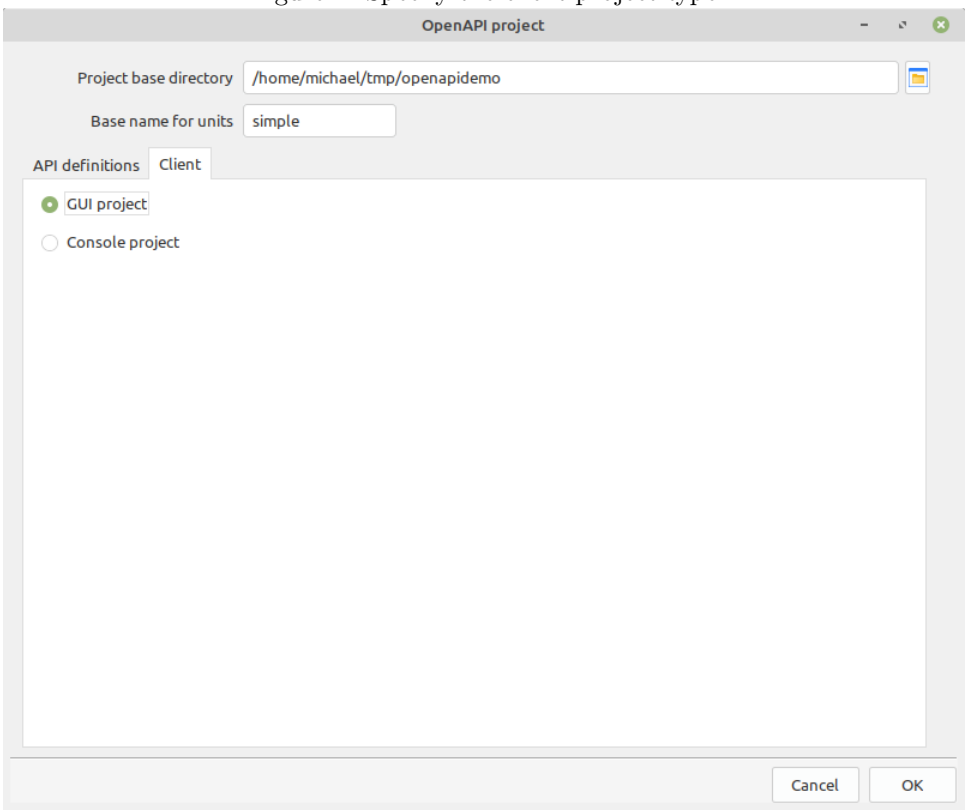
Project base directory This is the directory in which all files will be created. If you create only a client or a server project, then all files will be in this directory. If you create both projects, then 3 subdirectories will be made: `client`, `server` and `common`. The former will contain the project-specific code, the `common` directory will contain the generated OpenAPI files.

Base name for units this is the base name for the units, to which the usual suffices will be appended. The units will be generated in a directory based on the `Project base directory` setting.

Client project type Here you can specify the type of project to create for the client. You have the choice between a GUI or a Console project (see figure 4 on page 10).

Server project type Here you can specify the type of project to create for the server project. You have again the choice between a GUI and a Console project, but additionally you can choose a HTTP server project (see figure 5 on page 11).

Figure 4: Specify the client project type



The console application for the client is almost identical to the application we built in the previous article on OpenAPI. The 'HTTP server project' is identical to the server application presented in the previous article. The GUI application is a normal Lazarus application as you would get when you choose the 'Application' type in the 'New project' wizard.

For the client project, if you asked to create a server proxy module (as introduced in the previous section), code will be added to instantiate the module, and if a Server URL is present in the OpenAPI document it will be set as the value of its `BaseURL` property. So the module will be ready to use.

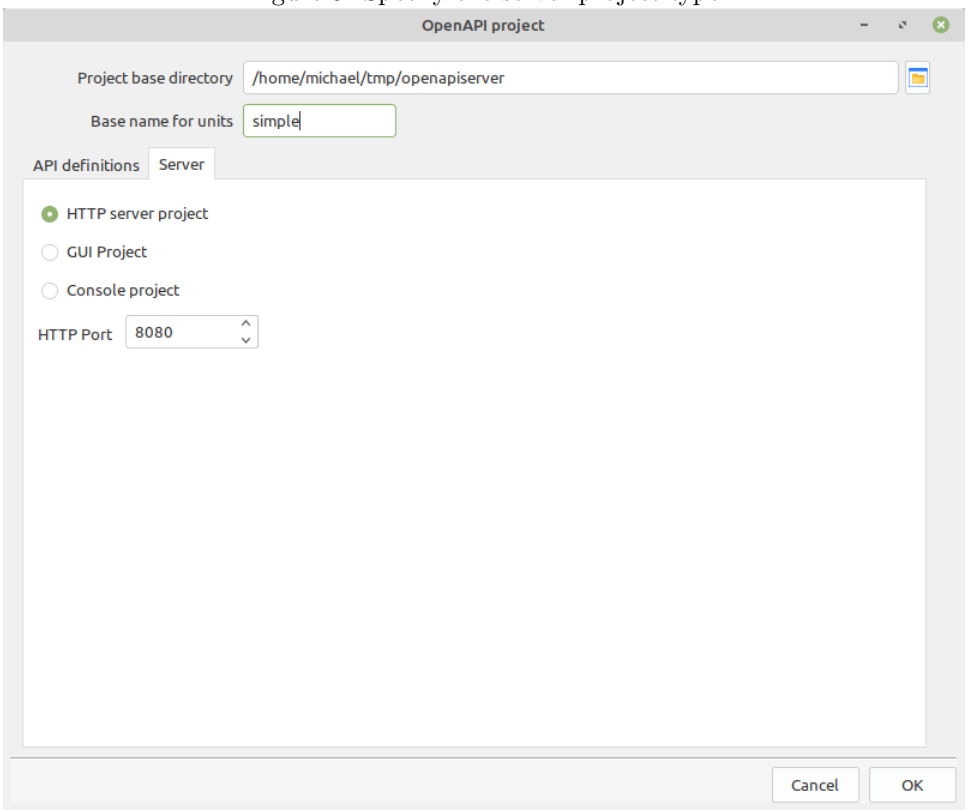
Once you have finished the wizard, the project wizard will create a `client.lpr` and/or a `server.lpr` project in the indicated directory. (in a later version, you'll be able to specify the names of the projects. For the moment, they are hard coded.).

All the information you entered in the new project wizard will be saved in a file next to the project. The file can be read by the `openapi2pas` command-line tool, but can also be used in the IDE:

In addition to the configuration file, the wizard will save some extra information in the project `.lpi` file(s). This extra information will allow you to easily regenerate the OpenAPI unit files using the project inspector. In the project inspector popup menu, there will be a 'Regenerate OpenAPI units' menu item (figure 6 on page 12), which will simply regenerate the files with the info given when the project was generated.

If you somehow move or rename the project and the extra information is no longer correct, then you can correct the information in the 'Project Options' dialog (figure

Figure 5: Specify the server project type



7 on page 13). There is little enough information to be managed on this page:

OpenAPI file The full path to the OpenAPI file.

Generator configuration This is the name of the configuration file for the OpenAPI code generator, as created by the IDE when you start the project.

Base unit filename This is the base unit filename to which a suffix will be appended when generating all the individual units.

The majority of the information needed to regenerate the files correctly is in the generator configuration file.

Last but not least, the 'Tools' dialog presented in the previous article will pick up all this info and save it so you don't need to re-enter the information every time you start the dialog: it also allows you to change the mapping or change some other settings.

7 Conclusion

With all this in place, it becomes easy to maintain an application that serves or consumes a REST api described by an OpenAPI document. As described in the previous articles, there are still some enhancements planned for the OpenAPI generator. You'll be able to read on these enhancements in a future article.

Figure 6: Regenerate the OpenAPI units in the project inspector

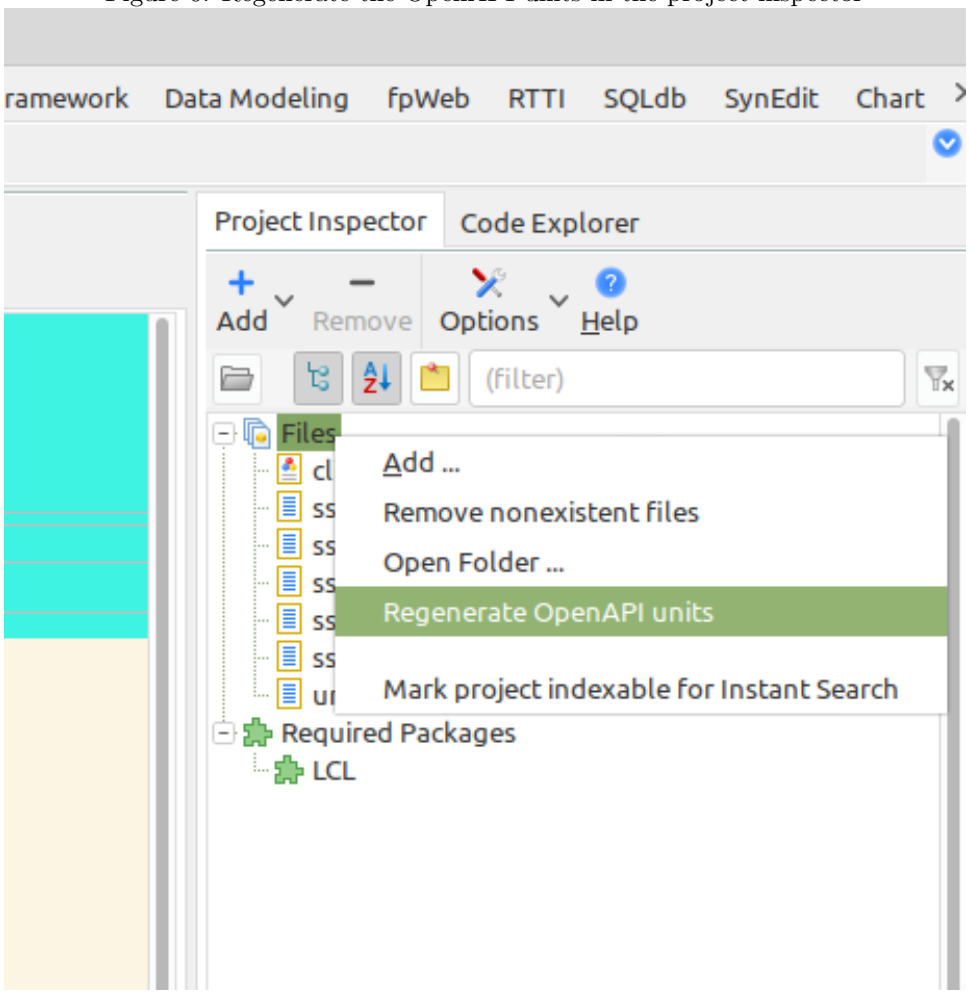


Figure 7: Editing the extra info for OpenAPI projects