

# Pas2JS: Communicating with the webserver

Michaël Van Canneyt

December 21, 2021

## Abstract

In a previous article we showed how to get started with pas2js, and how to compile a simple program that interacts with the HTML of the webpage. In this article, we show how to interact with an application server using JSON-RPC.

## 1 Introduction

A webpage almost invariably communicates with services hosted on a webserver. This can go from downloading a simple file to exchanging data with an application server. As explained in the previous article about real-world programming with pas2js, there are several communication protocols possible: SOAP, REST, JSON-RPC. The communication can happen over HTTP(s) or using websockets. Free Pascal supports all of these with several frameworks – FPC can be used to write a HTTP server or WebSocket server – or even both at the same time.

In this article, we'll explain how to use JSON-RPC on the server and in Pas2JS. The previous article laid the foundations for a login page, and we will now expand on this foundation to demonstrate how to let a pas2js program communicate with a server.

For this, we'll implement a `Users` service with 3 calls:

**Login** The login call to let a user log in using a username and password.

**Logout** The logout call.

**CreateUser** A call to create a new user in the user database.

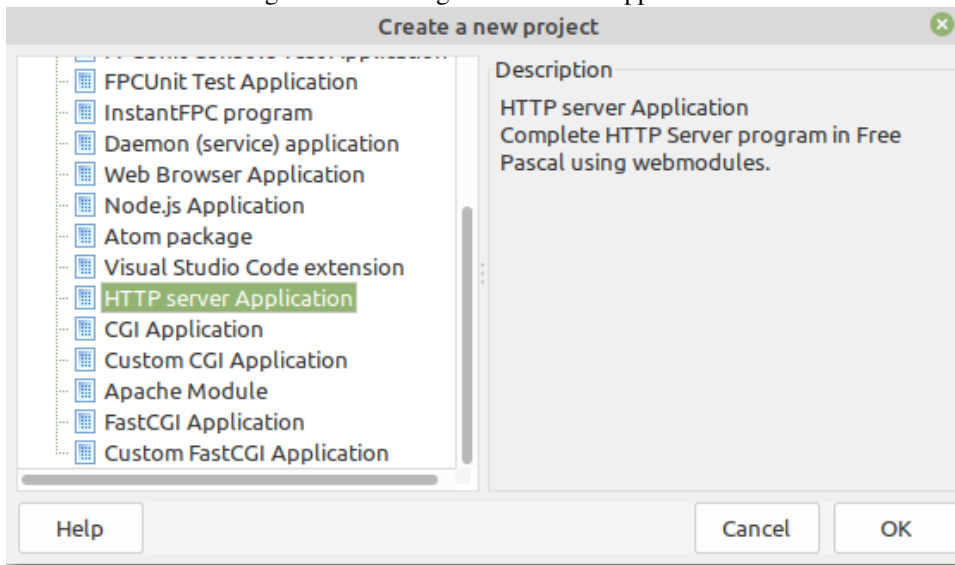
To make our application more secure, we'll also implement 2-factor authentication (2FA) using the Google Authenticator application: Free Pascal has a unit that can generate a time-based token which can be used with the Google Authenticator application.

This means the login page presented in the previous article needs to be expanded, so we can ask the user for the 2FA code. At the same time, we'll expand the HTML page a little, so it contains a menu bar in which we will add login and logout buttons as well as a place to show the user name.

## 2 The application server

To be able to create the application server, the `WebLaz` package must be installed in the IDE. If this is not yet the case, you can install it in the same way as the `Pas2jsDsgn` package had to be installed for Pas2JS support, using the Packages - Install packages menu.

Figure 1: Choosing HTTP Server Application



Once the package is installed you can make several kinds of webservice applications: CGI, FastCGI, standalone HTTP server or an apache module.

The HTTP server application (see figure 1 on page 2) needs the least setup, and is easiest to debug, so we'll take that. For a production environment, it may be better to use FastCGI or even an apache module - but this can be easily changed later during development. Once you choose this project type, the new project wizard will then present you with some options, as seen in figure 2 on page 3.

The 'Port to listen for requests' is the TCP/IP port on which the server will listen. Any port can be entered, but take care that the port is not yet in use on your system, and that your user is allowed to use this port: on Linux, port numbers below 1024 are reserved for the root user.

If the 'Register location to serve files from' option is checked, the wizard will insert code to let the HTTP server automatically serve files. No special code will need to be written for that, so this is very convenient. In the 'Directory' edit box, the directory from which to serve files can be specified: Subdirectories will be handled, but the program will refuse to handle files outside that directory. For most cases, the base directory will need to be set correctly in code anyway.

In the 'location' edit you can enter the start of the URL the server needs to get to serve files. In the configuration as shown in figure 2 on page 3, the URL

```
http://localhost:3000/files/css/login.css
```

will be mapped to the following filename on disk:

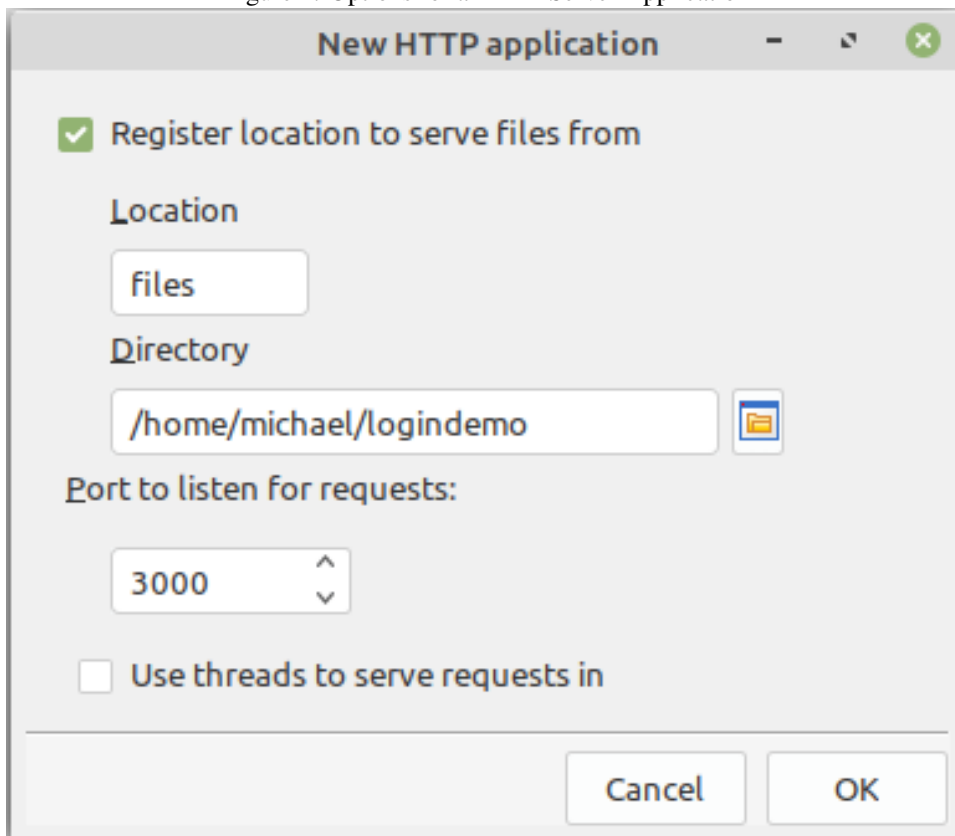
```
/home/michael/logindemo/css/login.css
```

The 'Threaded' checkbox tells the wizard to generate a program that will use threads to serve requests in. Special care must be taken when handling database access when you use threads, so for the moment we'll leave this unchecked.

When you confirm the settings, the following program source code is generated:

```
program loginserver;
```

Figure 2: Options for a HTTP Server Application



The image shows a dialog box titled "New HTTP application" with standard window controls (minimize, maximize, close) in the top right corner. The dialog contains the following elements:

- A checked checkbox labeled "Register location to serve files from".
- A label "Location" above a text input field containing the text "files".
- A label "Directory" above a text input field containing the path "/home/michael/logindemo". To the right of this field is a small folder icon.
- A label "Port to listen for requests:" above a spin box containing the number "3000".
- An unchecked checkbox labeled "Use threads to serve requests in".
- At the bottom right, there are two buttons: "Cancel" and "OK".

```

{$mode objfpc}{$H+}

uses
  sysutils, fpwebfile, fphttpapp, unit1;

begin
  RegisterFileLocation(' files', '/home/michael/logindemo/')
  Application.Title:='httpproject1';
  Application.Port:=3000;
  Application.Initialize;
  Application.Run;
end.

```

**But we will change this to the following:**

```

program loginserver;

{$mode objfpc}{$H+}

uses
  sysutils, fpwebfile, fpmimetypes, fphttpapp, unit1;

Var
  aDir : string;

begin
  MimeTypes.LoadKnownTypes;
  Application.Title:='Pas2JS demo server';
  Application.Port:=3000;
  Application.Initialize;
  if Application.HasOption('d','directory') then
    aDir:=Application.GetOptionValue('d','directory')
  else
    aDir:=ExtractFilePath(ParamStr(0))+ '../webwidget/';
  TSimpleFileModule.BaseDir:=ExpandFileName(aDir);
  TSimpleFileModule.RegisterDefaultRoute;
  Application.Run;
end.

```

**As you can see, the RegisterFileLocation call has been removed. It has been replaced by the following lines:**

```

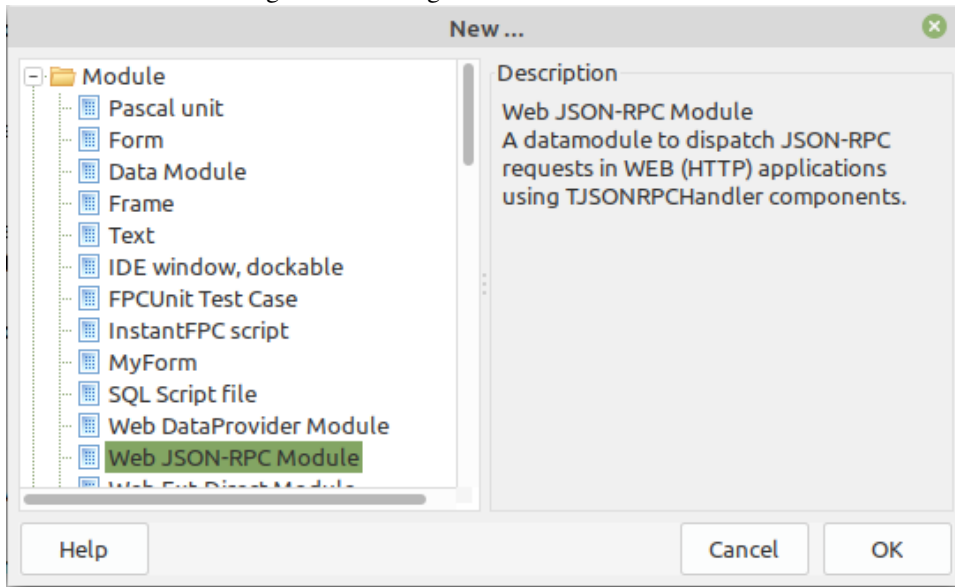
if Application.HasOption('d','directory') then
  aDir:=Application.GetOptionValue('d','directory')
else
  aDir:=ExtractFilePath(ParamStr(0))+ '../webwidget/';
TSimpleFileModule.BaseDir:=ExpandFileName(aDir);
TSimpleFileModule.RegisterDefaultRoute;

```

**The reason for this change is 2-fold:**

1. The requirement to use the '/files/' prefix in all URLs to serve files is not very convenient. It would be better not to have to type this prefix.

Figure 3: Creating a Web JSON-RPC module



Instead, it is easier to let the HTTP server try to serve as a file any URL it does not recognize as special.

This is what the call to the `TSimpleFileModule.RegisterDefaultRoute` class method does: it will register the `TSimpleFileModule` class (this class is a HTTP route handler made available by FPC) as the default route handler of the server: any non-recognized route will be treated as a file.

2. We set the `TSimpleFileModule.BaseDir` class variable to the directory where the `TSimpleFileModule` must look for files. The location can be set with the `-d` command-line option. Because of the location of the login page client project, a default of `../webwidget/` relative to the server project directory is used.

Note that in the trunk version of Lazarus, the 'New HTTP application' wizard has been improved, so the above changes do not have to be made: the wizard now can be used to configure the `TSimpleFileModule` for you.

### 3 The JSON-RPC service

The "New HTTP application" wizard has generated a first `WebModule` (a `TFPWebModule` descendent) in `unit1`. We don't need this webmodule, so we remove `unit1` and the webmodule from the project and save the resulting project as 'loginserver'.

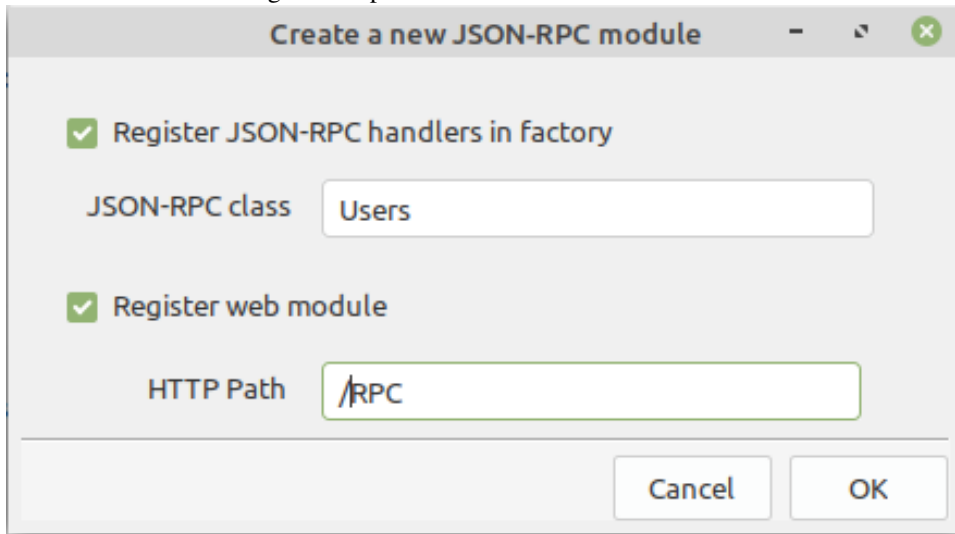
Instead, we use the File-New menu dialog to create a `Web JSON-RPC module` (see figure 3 on page 5). This module is the basis for the RPC server.

The RPC server in FPC is currently implemented using 3 components:

**TJSONRPCModule** This is a `WebModule` descendent that will serve JSON-RPC requests. You need at least 1 `TJSONRPCModule` in your application.

**TJSONRPCHandler** This is a component which will handle exactly 1 JSON-RPC method. For each method you want to create in your JSON-RPC server, you must drop 1 `TJSONRPCHandler` component on a `TJSONRPCModule` descendent or a `TDatamodule`.

Figure 4: Options for the JSON-RPC module



**TJSONRPCDispatcher** This is a component that can will dispatch a JSON-RPC call to the correct `TJSONRPCHandler` component. The `TJSONRPCModule WebModule` will automatically create an instance of `TJSONRPCDispatcher` if you didn't specify one in its `Dispatcher` property.

Again, several options can (or must) be set to control the generated code, they are shown in figure 4 on page 6. The 'Register JSON-RPC handlers in factory' option can be left unchecked.

If checked, it will insert code that registers all `TJSONRPCHandler` components on the module in the JSON-RPC registry (the factory pattern is used to find `TJSONRPCHandler` instances). When doing so, it will use the 'JSON-RPC class' as the class name for the RPC-JSON registration: This is an extension in FPC which allows to have various 'classes' with methods in the JSON-RPC service.

The 'Register web module' will associate this web module with a HTTP route. The value 'RPC' shown in figure 4 on page 6 means that the following URL:

```
http://localhost:3000/RPC
```

will be considered the entry point for the JSON-RPC service: all requests to this URL will be handled by the `TJSONRPCModule`.

If you create multiple `TJSONRPCModule` modules, it is important you only register 1 of them as the handler for the RPC route, unless you want to create multiple routes for RPC calls. For the other `TJSONRPCModule` modules, it is sufficient to use the 'Register JSON-RPC handlers in factory' option.

When you click OK in the new JSON-RPC module wizard and save the new unit as `dmRPC`, the following code will be generated:

```
unit dmRPC;
{$mode ObjFPC}{$H+}

interface

uses
```

```

Classes, SysUtils, HTTPDefs, websession, fpHTTP, fpWeb,
fpjsonrpc, webjsonrpc;

type

  { TJSONRPCModule1 }

TJSONRPCModule1 = class(TJSONRPCModule)
private
public
end;

implementation

initialization
  RegisterHTTPModule('RPC', TJSONRPCModule1);
  JSONRPCHandlerManager.RegisterDatamodule(TJSONRPCModule1, 'RPC', );
end.

```

Unfortunately, a bug in the released version of Lazarus caused this wrong code to be generated in the initialization section. This bug has already been fixed in the trunk version of Lazarus.

Start by renaming the webmodule to TUsersModule, and then the initialization section code must be changed to the following:

```

initialization
  RegisterHTTPModule('RPC', TUsersModule);
end.

```

To add methods to our JSON-RPC server, we must drop a TJSONRPCHandler component on the datamodule: one component per method.

The TJSONRPCHandler class has the following important properties:

**Name** The component name serves also as the method name. The most recent version of the component in FPC allows you to specify an alternate name.

**Options** There are several options that can be set here:

**jroCheckParams** The type and number of incoming parameters is checked against the parameter definitions in `ParamDefs`.

**jroObjectParams** The parameters must be specified as a JSON object.

**jroArrayParams** The parameters must be specified as a JSON array.

**jroIgnoreExtraFields** If the call has extra parameters on top of the parameter definitions in `ParamDefs` they are ignored.

**ParamDefs** This collection property serves 2 purposes: It is used when generating the description of the full JSON-RPC API. This collection has 1 item for each expected parameter to the method, in the order that they should be passed to the method. Every item in the collection has 3 properties: `Name`, `DataType` (one of the valid JSON types) and `Required`.

And the following event handlers exist:

**OnExecute** This is the most important event handler: this event handler is called when the JSON-RPC method must be executed. It will get passed the parameters received from the client, and must return a `JSONData` value that is the result parameter.

**BeforeExecute** This is an event handler that is called before actually executing the method. Here you can implement authentication or logging.

**AfterExecute** This is an event handler that is called after the method was executed.

**OnParamError** This event handler is called when the `jrCheckParams` option is specified and there is a parameter mismatch in the received parameters.

For most applications, it is sufficient to set the `OnExecute` event handler.

For our application, we need to implement a login call and a call to create a new user. For this, we'll store the allowed users in the database. For simplicity we will use a Firebird database, with the following definition for the users table:

```
CREATE SEQUENCE GEN_USERS;

CREATE TABLE USERS (
  U_ID BIGINT NOT NULL ,
  U_NAME VARCHAR(50) NOT NULL,
  U_PASSWORD VARCHAR(50) NOT NULL,
  U_2FASEED VARCHAR(32) NOT NULL,
  CONSTRAINT PK_USERS PRIMARY KEY (U_ID)
);

CREATE UNIQUE INDEX UDX_USERS ON USERS (U_NAME);

set term ^;

CREATE TRIGGER TR_INSERTID FOR USERS
BEFORE INSERT
AS
BEGIN
  IF (NEW.U_ID IS NULL) THEN
    NEW.U_ID=GEN_ID (GEN_USERS, 1);
END^
```

The `U_2FASEED` field serves to store a shared secret for 2-factor authentication.

The login call needs 3 parameters: username, password and the 2-factor authentication code. These can be entered in the `ParamDefs` property, as shown in figure 5 on page 9

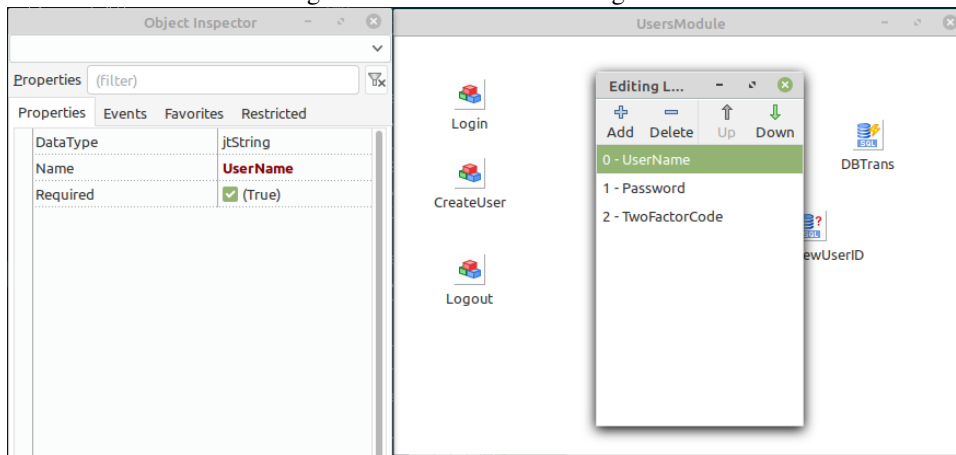
Now we can actually implement the `Login` call. For this we assign the `OnExecute` handler, and enter the following code:

```
procedure TUsersModule.LoginExecute(Sender: TObject;
                                     const Params: TJSONData;
                                     out Res: TJSONData);

Var
  A : TJSONArray absolute Params;
  aUserName,aPassword : String;
  aTwofactorCode : Integer;
```



Figure 5: Parameters for the login call



```

    OK : Boolean;

begin
    aUserName:=A.Strings[0];
    aPassword:=A.Strings[1];
    aTwoFactorCode:=A.Integers[2];
    OK:=CheckUser (aUserName, aPassword, aTwoFactorCode);
    Res:=TJSONBoolean.Create (OK);
    if OK then
        Session.Variables['User']:=aUserName;
    end;
end;

```

The first 3 lines simply save the values of the parameters, and in the 4th line the call to `CheckUser` will actually check the passed parameters with the contents of the database. If the call returns `True`, we store the username in the browser session: this way we can verify whether the browser user is authenticated or not in future calls.

The `CheckUser` method is actually pretty standard code to run a query and compare the contents of the password with the password stored in the database: The routine starts by connecting to the database, using the `ConnectDB` call. The details of this method we will not describe here; the interested user can consult the source code.

```

function TUsersModule.CheckUser(const aName, aPassword: String;
                                ACode: Integer): Boolean;

Var
    DBPassword,
    Secret: string;
begin
    ConnectDB;
    With QGetUser do
        begin
            ParamByName('NAME').AsString:=aName;
            Open;
            try
                Result:=Not IsEmpty;
                if Result then
                    begin

```

```

        DBPassword:=FieldName('U_PASSWORD').AsString;
        Secret:=FieldName('U_2FASEED').AsString;
        Result:= (aPassWord=DBPassword);
        If Result Then
            Result:=Check2FA(Secret, aCode);
        end;
    finally
        Close;
    end;
end;
end;
end;

```

The `QGetUser` is a `TSQLQuery` component which was dropped on the `RPCModule`. We enter the following SQL command in its SQL property:

```

SELECT
    U_ID, U_PASSWORD, U_2FASEED
FROM
    USERS
WHERE
    (U_NAME=:NAME)

```

If the query does not return a result, we know the username is not known and authentication should fail. If the query returns a result, we verify the password in code. To be really safe, it would of course be better to save the password in hashed form and compare the hashed form of the incoming password with the hash stored in the database.

If the username was correct and the password matched the password stored in the database, the 2FA shared secret stored in the database is used to check the 2-factor authentication code:

```

function TUsersModule.Check2FA(const aSeed : String;
                               aCode : Integer) : Boolean;

Var
    O : Integer;

begin
    Result:=TOTPValidate(aSeed, aCode, 1, O);
end;

```

The `TOTPValidate` function is implemented in the `onetimepass` unit, part of `FPC`.

```

function TOTPValidate(const aSecret: AnsiString;
                     const Token: LongInt;
                     const WindowSize: LongInt;
                     var Counter: LongInt) : Boolean;

```

It requires a secret, the token to verify code, a maximum allowed deviation (the window-size, measured in seconds). It will return a counter – the counter can be used to implement a counter-based verification token, but we will not use that and stick to a time-based token.

The logout call is very simple. It does not need any parameters at all, and the code is quite simple.

```

procedure TUsersModule.LogoutExecute(Sender: TObject);

```

```

                                const Params: TJSONData;
                                out Res: TJSONData);
begin
    Session.Variables['User']:=aUserName;
    Res:=TJSONNull.Create;
end;

```

It is important to always set the `res` result, even if it is `Nil`: failing to do so will lead to unpredictable behaviour.

To create a user, we require a username and password, and simply insert a record in the database. This can be done again with very little code. We start by verifying whether the current user is allowed to do so.

For the current demonstration application, we check that the user is the `Admin` user, and we raise an exception when the user is not the administrator user. The exception will be caught by FPC's JSON-RPC implementation, and translated to a valid JSON-RPC error response:

```

procedure TUsersModule.CreateUserExecute(Sender: TObject;
    const Params: TJSONData; out Res: TJSONData);

```

```

Var
    A : TJSONArray absolute Params;
    aUserName, aPassword : String;
    aID : Int64;

```

```

begin
    if Session.Variables['User']<>'Admin' then
        Raise Exception.Create('Only admins can create users');
    aUserName:=A.Strings[0];
    aPassword:=A.Strings[1];
    aID:=DoCreateUser(aUserName, aPassword);
    Res:=TJSONInt64Number.Create(aID);
end;

```

After collecting the username and password from the passed parameters, the `DoCreateUser` method is called to actually create the user in the database.

The response of the `DoCreateUser` call is the ID of the new user record in the database; For Firebird (or other databases that support sequences), this ID can be fetched separately, this is implemented in the `GetNewUserID` method using a simple query.

```

function TUsersModule.DoCreateUser(const aName,
    aPassword: String): Int64;

```

```

Var
    aSeed : String;

```

```

begin
    aSeed:=TOTPSharedSecret();
    Result:=GetNewUserID;
    ConnectDB;
    With QInsertUser do
        begin
            ParamByName('ID').AsLargeInt:=Result;

```

```

    ParamByName('NAME').AsString:=aName;
    ParamByName('PASSWORD').AsString:=aPassword;
    ParamByName('SEED').AsString:=aSeed;
    ExecSQL;
end;
end;

```

The `TOTPSharedSecret` call (also implemented in the `onetimepass` unit) creates a new (random) shared secret usable in the Google Authenticator. It is stored in the database together with the new user record.

## 4 2FA and the Google Authenticator app

The Google authenticator works using a shared key: the application that wishes to authenticate a user using 2FA generates a shared secret and stores this somewhere, associated with the user.

The user registers this shared key in the google authenticator. The app uses this secret to generate a secret code every 30 seconds. When asked for it, the user enters this code in the web application. The web application server also generates the secret code using the shared key associated with the user, and compares it with the code given by the user: if they match, it confirms the identity of the user.

Obviously, the shared secret for 2-Factor authorization must be communicated by some safe means to the user when the new user record is inserted in the database. The application as it is now does not provide any method to communicate this secret – it would lead too far to discuss that. Converting it to a QR code is one way, sending the code by text or some other means is another.

To use the shared secret, the Google Authenticator application must be installed on a device (smartphone, tablet) owned by the user: this can be an IOS or Android device.

In the Google authenticator app, the user must add a new key using the 'Add' button and selecting either 'Scan a QR code' or 'Enter key', after which a description and the shared secret can be entered as in figure 6 on page 13.

When it is time to authenticate, the application asks for the authentication code, and the user has 30 seconds to enter the code displayed in the Google authenticator app in the website, see figure 7 on page 14. after 30 seconds, a new code is generated.

## 5 Modifying the client

Now that we have the server programmed, it is time to enhance the client application. In the previous article 2 applications were presented: one programmed with plain HTML classes, one with WebWidget components. Here we will only enhance the application programmed with WebWidget components, but the sample client application written using plain HTML classes can be adapted in much the same way. We'll start by adding a menu (a navbar in web parlance, using the `<nav>` tag). The HTML for the navbar can be found in the code accompanying this article, but the resulting nav bar is shown in figure 8 on page 15.

The navbar CSS in Bulma is quite simple, is responsive, and features a hamburger menu – the three little lines that appear when the CSS hides the menu on small screens. When clicking the hamburger menu, the menu itself must be shown or hidden in code.

Contrary to CSS frameworks such as Bootstrap, Bulma does not offer some standard Javascript file to perform this task, but the task is programmed easily enough: We assign an

Figure 6: Adding the shared secret

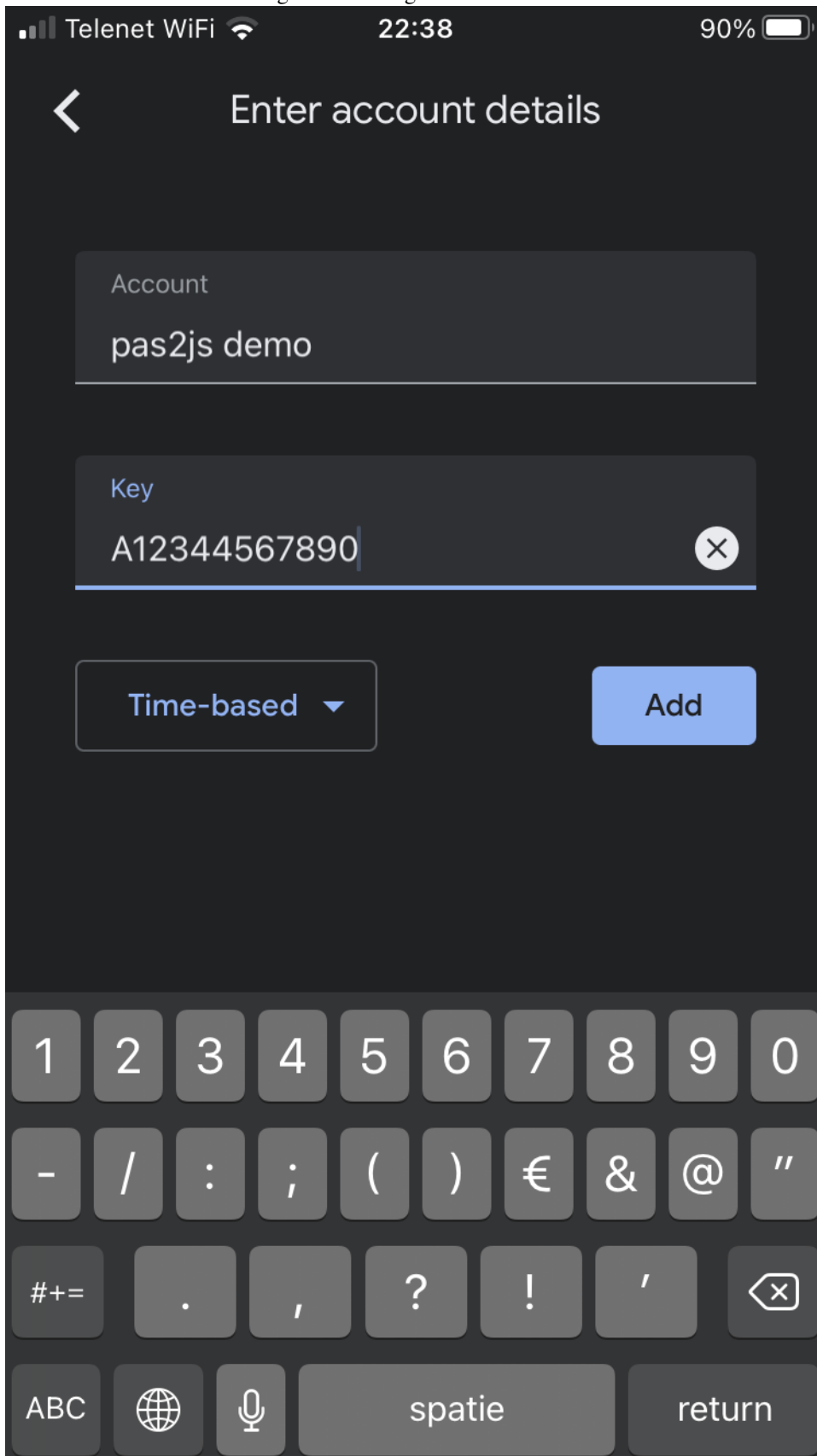


Figure 7: Adding the shared secret

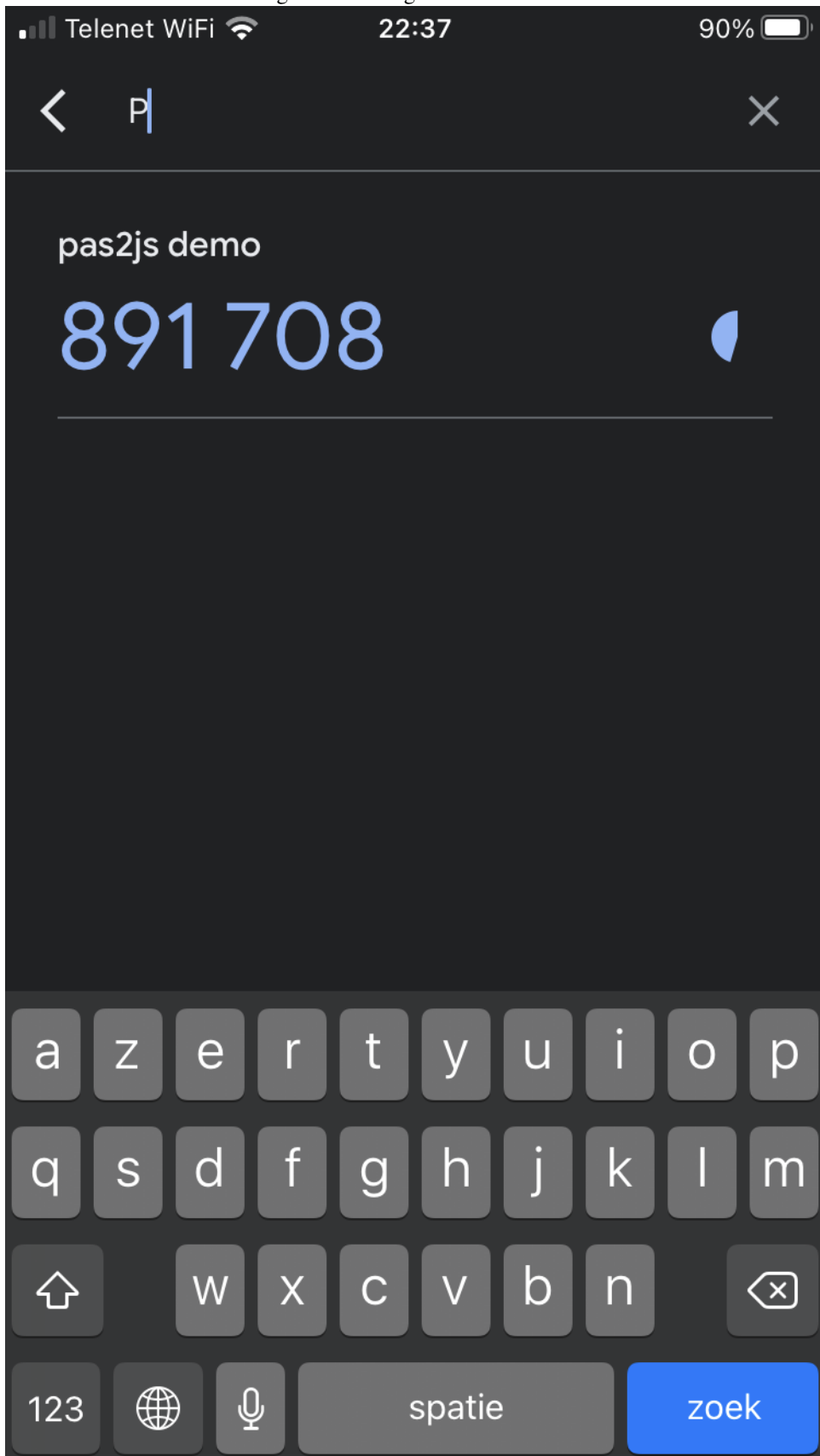
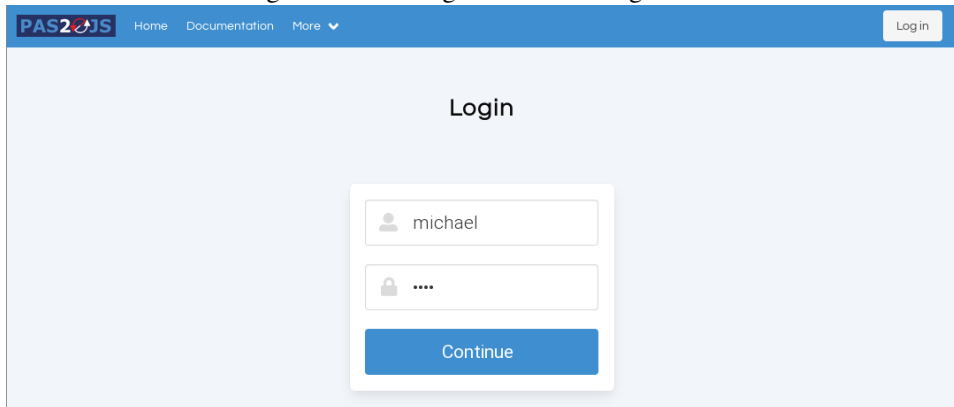


Figure 8: The navigation bar in a large screen



ID to the hamburger menu tag, and to the menu itself. This allows us to create webwidgets that reference these tags, and we can attach a `OnClick` handler to the hamburger `<div>` tag. As shown in the previous article, this is done in the `BindElements` routine:

```
divMenuHamburger:=TTagWidget.Create(Self);
divMenuHamburger.elementID:='navbar-burger';
divMenuHamburger.Refresh;
divMenuHamburger.OnClick:=@ClickNavBar;
divMenu:=TTagWidget.Create(Self);
divMenu.elementID:='navbar-main';
divMenu.Refresh;
```

The `ClickNavBar` method is then simply:

```
procedure TMyApplication.ClickNavBar(Sender: TObject;
                                     Event: TJSEvent);
begin
  if Pos('is-active',divMenu.Classes)<>0 then
    DivMenu.RemoveClasses('is-active')
  else
    DivMenu.AddClasses('is-active');
end;
```

The `AddClasses` and `RemoveClasses` methods add or remove CSS classes to the HTML tag of a widget.

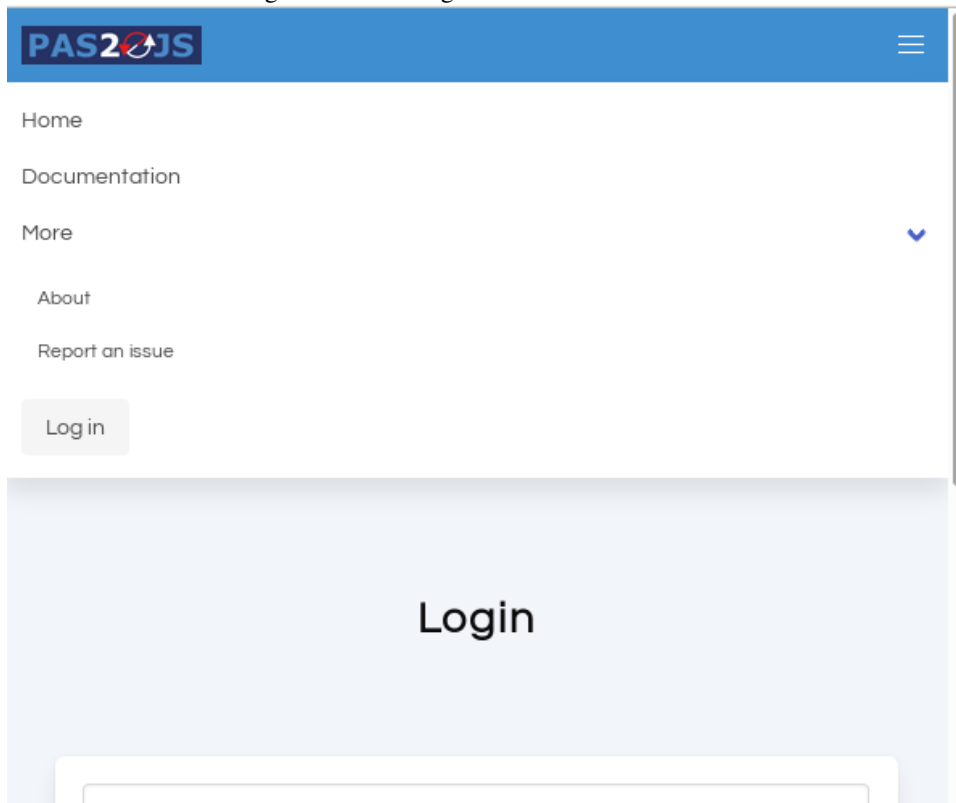
Applying the CSS class `is-active` shows the menu, if it is absent, the menu is hidden. The result can be seen in figure 9 on page 16

In the navbar, we add some menu items (not functional at this time), and also buttons to log in or log out and a small section to display the user name once the user is logged in. The log in button will display the login dialog, and the log out button will log out the user and then displays the login dialog.

For each of these elements, a `TTagWidget` is made and associated with the corresponding HTML tag using the ID in the `BindElements` call:

```
divMenuLogin:=TTagWidget.Create(Self);
divMenuLogin.elementID:='mnuLogin';
divMenuLogin.OnClick:=@DoLoginMenuClick;
```

Figure 9: The navigation bar in a small screen



```
divMenuLogin.Refresh;
divMenuLogout:=TTagWidget.Create(Self);
divMenuLogout.elementID:='mnuLogout';
divMenuLogout.Refresh;
divMenuLogout.OnClick:=@DoLogoutClick;

procedure TMyApplication.DoLoginMenuClick(Sender: TObject;
                                           Event: TJSEvent);
begin
    divDlgLogin.RemoveClasses('is-hidden');
    ShowLogin('');
end;

procedure TMyApplication.DoLogoutClick(Sender: TObject;
                                        Event: TJSEvent);
begin
    ShowLogout;
    DoLogout;
end;
```

The `DoLogout` method will do the actual logout call to the server.

The `showLogin` and `ShowLogout` methods simply hide or show various HTML tags:

```
procedure TMyApplication.ShowLogout;
begin
```



```

    divDlgLogin.RemoveClasses('is-hidden');
    ShowLogin('');
    divMenuUser.AddClasses('is-hidden');
    divMenuLogin.RemoveClasses('is-hidden');
    divMenuLogout.AddClasses('is-hidden');
end;

```

The `is-hidden` CSS class is provided by Bulma and will hide the element to which it is applied.

The `ShowLogin` does a little more work. Because we wish to have 2-Factor Authorization, the login dialog is split in 2 parts: the first part asks for the user name and password, and the second part asks for the 2FA code. Each part is contained in a Bulma "box" tag, having an ID of `div2FA` and `Login`. Again each tag will be represented by a webwidget and bound to the HTML tag in the `BindElements` method.

The `ShowLogin` method shows the Login box, but hides the 2FA box. Additionally, `ShowLogin` will be called after a failed login attempt, to allow the user to start over: in that case an error message is passed in the `aError` parameter. If it is non-empty, an Error div is shown or hidden beneath the username entry.

```

procedure TMyApplication.ShowLogin(const aError : String);

begin
    div2FA.AddClasses('is-hidden');
    divLogin.RemoveClasses('is-hidden');
    if aError<>'' then
        begin
            divError.RemoveClasses('is-hidden');
            divError.TextContent:=aError;
        end
    else
        begin
            divError.AddClasses('is-hidden');
            divError.TextContent:='';
        end;
end;

```

As can be seen in figure 8 on page 15, the login button from the first version of the application has been changed so it displays the `Continue` text. When pressed, it hides the login box, and displays the 2FA Box:

```

procedure TMyApplication.doContinueClick(sender : TObject;
                                         event : TJSEvent);


begin
    divLogin.AddClasses('is-hidden');
    div2FA.RemoveClasses('is-hidden');
end;

```

The result of this is that the user sees the following part of the login dialog, where he must enter the 2FA authentication code, as shown in figure 10 on page 18. After that, the login actually can be performed.

Figure 10: The 2-Factor authentication code

# Login



## 6 Actually talking to the server: The RPC Client

Our server can handle the JSON-RPC protocol. By definition, this means of course we must send JSON to the server. To send JSON to a server is easily done in a browser: the required JSON is easily constructed, and the `XMLHttpRequest` class or the `Fetch` call can be used for sending it to the server and handling the response. `JQuery` offers a `ajax` method. All these can be used in `pas2js`.

But `Pas2js` comes with a `TRPCClient` class which is geared specially towards JSON-RPC: it will automatically create the correct envelope, it assigns the required keys such as `id` and `jsonrpc`, and takes care of error responses: all kind of things that one would expect. But it also offers batching: you can batch calls explicitly, or let the client perform automatic batching; calls are batched, and after a configurable time, the batch is sent to the server. This can be used to improve performance: by sending several method calls in 1 HTTP request, the time spent on network communication is reduced.

This class is defined as follows, with only the relevant methods and properties:

```
TRPCClient = class(TComponent)
Public
  Constructor Create(aOwner : TComponent); override;
  Destructor Destroy; override;
  Function CreateRequestParamsBuilder : TRPCRequestParamsBuilder;
  Function ExecuteRequest(const aClassName, aMethodName : String;
                        aParams : TJSArray;
                        aOnSuccess : TRPCResultCallBack = Nil;
                        aOnFailure: TRPCFailureCallBack = nil) : NativeInt;
  Function ExecuteRequest(const aClassName, aMethodName : String;
                        aParams : TJSObject;
                        aOnSuccess : TRPCResultCallBack = Nil;
                        aOnFailure: TRPCFailureCallBack = nil) : NativeInt;

  Procedure CloseBatch;
Published
  Property URL : String;
  Property Options : TRPCOptions;
  Property BatchTimeout: Integer;
  Property JSONRPCversion : String;
  Property CustomHeaders : TStrings;
  Property OnConfigRequest : TRPCConfigRequest;
  Property OnCustomHeaders : TRPCHeadersRequest;
  Property OnUnexpectedError : TRPCUnexpectedErrorCallback;
end;
```

The 2 `ExecuteRequest` methods will be treated below. The `URL` property is the URL for the RPC server. For our application, this will be something like

```
/RPC
```

Or, equivalently

```
http://localhost:3000/RPC
```

The `Options` property is a combination of the following values:

**roParamsAsObject** create a parameter builder (see below) that creates the parameters as an object.

**roFullMethodName** Combine classname and method name into a single name, using a dot as the separator. The effect of this option is that the name of the RPC method becomes 'classname.methodname'. The default is to send the classname in a separate 'class' key.

**roUseBatch** ExecuteRequest will not send the JSON-RPC request to the server at once. Instead, calls are batched and sent when CloseBatch is called.

**roAutoBatch** If roUseBatch is specified together with this value, the first call to ExecuteRequest that starts a batch sets a timer: when the timer expires, CloseBatch is called automatically.

**roForceArray** In case only 1 RPC method call is sent, force use of an array. By default, if only a single method is executed, only the object describing the call is sent. With this option enabled, the object is wrapped in an array. You can use this option if the JSON-RPC server is only capable of receiving arrays.

To execute methods on the server, the following 2 calls are important:

```
// Execute a request. Params can be passed as object or array
Function ExecuteRequest(const aClassName, aMethodName : String;
                        aParams : TJSArray;
                        aOnSuccess : TRPCResultCallBack = Nil;
                        aOnFailure: TRPCFailureCallBack = nil) : NativeInt;
Function ExecuteRequest(const aClassName, aMethodName : String;
                        aParams : TJSObject;
                        aOnSuccess : TRPCResultCallBack = Nil;
                        aOnFailure: TRPCFailureCallBack = nil) : NativeInt;
```

The aClassName, aMethodName parameters are used to select the method to execute: they map directly to the classname, method name used on the server. The parameters to the calls can be passed as an Javascript array or object.

The last 2 parameters are callbacks (event handlers) which will be called in case of success or failure to execute the call. Since every request to the server is asynchronous, a callback mechanism is needed (a second mechanism using Javascript promises is also in the works).

The result of these functions is the id of the method call in the JSON-RPC protocol.

For example, to execute the Login call, we could create the following code:

```
var
  Params : TJSArray;
begin
  Params:=TJSArray.New('Michael', 'Secret', 123);
  RPCClient.ExecuteRequest('Users', 'Login', Params, @DoOK, @DoFail);
end;
```

In this code, we assume that the RPCClient is set up appropriately elsewhere.

The DoOK and DoFail methods could look like this:

```
procedure DoOK(aResult: JSValue);
begin
  if not Boolean(aResult) then
    ShowLogin('Invalid combination of username/password')
  else
    StartLogin(aUser);
```

```

end;

procedure DoFail(Sender: TObject; const aError: TRPCError);
begin
  ShowLogin('Error during login: '+aError.Message);
end;

```

The `aUser` variable contains the user name. Note that the `DoOK` call uses a `JSValue` parameter (equivalent to a `Variant`).

In the above code you can see the use of the `ShowLogin` method introduced earlier, and also how an error is reported by the `TRPCClient` class: a `TRPCError` record is used.

```

TRPCError = record
  ID : NativeInt;
  Code : NativeInt;
  Message : String;
  ErrorClass : String;
end;

```

This structure is used both for server errors as for communication errors. The meaning of the fields should be clear from their names: the `Code` and `Message` are taken from the JSON-RPC protocol. In case of HTTP protocol errors, they will contain the HTTP status code and text. If an exception class name is available, it is reported in the `ErrorClass` field. `ID` contains the ID from the call.

The `StartLogin` method simply displays the user name in the navbar and hides the login dialog and login button, and shows the logout button instead, a matter of adding or removing the `is-hidden` CSS class:

```

procedure TMyApplication.StartLogin(Const aUser : String);
begin
  divdlgLogin.AddClasses('is-hidden');
  divMenuLogout.RemoveClasses('is-hidden');
  divMenuLogin.AddClasses('is-hidden');
  divMenuUser.RemoveClasses('is-hidden');
  divLblUser.TextContent:=aUser;
  FLoggedInUser:=aUser;
end;

```

## 7 Using a service class

As can be seen from the above code sample, a login call is not difficult to code, but this method does have some drawbacks:

- It is not type safe. Both parameters and return value are not checked.
- If you must do the same call in different places in the application, it would be better to have a single, typed call that can be reused.

These 2 problems can be solved easily. The `TRPCCustomService` class (part of the `fprpcclient` unit) is meant to act as a base class with which a proxy class can be

constructed for the 'Classes' defined by the JSON-RPC server. It introduces some auxiliary methods that help in building the parameters for the ExecuteRequest class.

For our login RPC server, this proxy could be defined as follows:

```
TUserService = Class(TRPCCustomService)
Protected
  Function RPCClassName : string; override;
Public
  Function Login(Const aUserName,aPassword : String;
                aCode : Integer;
                aOnSuccess : TBooleanResultHandler = Nil;
                aOnFailure : TRPCFailureCallBack = Nil) : NativeInt;
  Function Logout(aOnSuccess : TEmptyResultHandler = Nil;
                 aOnFailure : TRPCFailureCallBack = Nil) : NativeInt;
  Function CreateUser(Const aUserName,aPassword : String;
                    aOnSuccess : TNativeIntResultHandler = Nil;
                    aOnFailure : TRPCFailureCallBack = Nil) : NativeInt;
end;
```

As you can see, the public methods here mimic exactly the definition of the methods defined in our JSON-RPC server. The success handler is also typed: instead of a JSValue return, a Boolean return is expected. The RPCClassName must return the name of the class on the server:

```
function TUserService.RPCClassName: string;
begin
  Result:='Users';
end;
```

And the implementation of the login call can be done as follows:

```
function TUserService.Login(const aUserName, aPassword: String;
                           aCode: Integer;
                           aOnSuccess: TBooleanResultHandler;
                           aOnFailure: TRPCFailureCallBack
                           ): NativeInt;

  Procedure DoSuccess(Sender : TObject; const aResult : JSValue);

  begin
    If Assigned(aOnSuccess) then
      aOnSuccess (Boolean(aResult));
  end;

Var
  _Params : JSValue;

begin
  StartParams;
  AddParam('UserName', aUserName);
  AddParam('Password', aPassword);
  AddParam('Code', aCode);
  _Params:=EndParams;
  Result:=ExecuteRequest (RPCClassName, 'Login', _Params,
```

```

                                @DoSuccess, aOnFailure);
end;

```

The `StartParams` and `EndParams` methods of the `TRPCCustomService` class respectively create an instance of the `TRPCRequestParamsBuilder` class, and return the final parameters for use in the `ExecuteRequest` call. The `TRPCRequestParamsBuilder` class will create the parameters for the call as required by the settings of the `TRPCClient` class: as a JSON array or a JSON object.

The `AddParam` method of the `TRPCCustomService` class adds a parameter to the list of parameters: an overloaded version of this call exists for every supported JSON type. Finally, `ExecuteRequest` is called and the success callback is re-routed through a local method, which will typecast then result to the appropriate type for the `Login` success callback (in this case, a boolean).

The result of all this is that now you can instantiate an instance of `TUserService` and do:

```

procedure TMyApplication.doServerLogin(const aUser, aPassword: String;
aCode : Integer);

    procedure DoOK(aResult: Boolean);
    begin
        if not aResult then
            ShowLogin('Invalid combination of username/password')
        else
            StartLogin(aUser);
        end;
    end;

    procedure DoFail(Sender: TObject; const aError: TRPCError);
    begin
        ShowLogin('Error during login: '+aError.Message);
    end;

begin
    FUserService.Login(aUser, aPassword, aCode, @DoOK, @DoFail);
end;

```

The `FUserService` variable is an instance of the `TUserService` class. This code is reusable and type-safe.

The setup of the service and RPC client is very simple:

```

procedure TMyApplication.SetupServices;

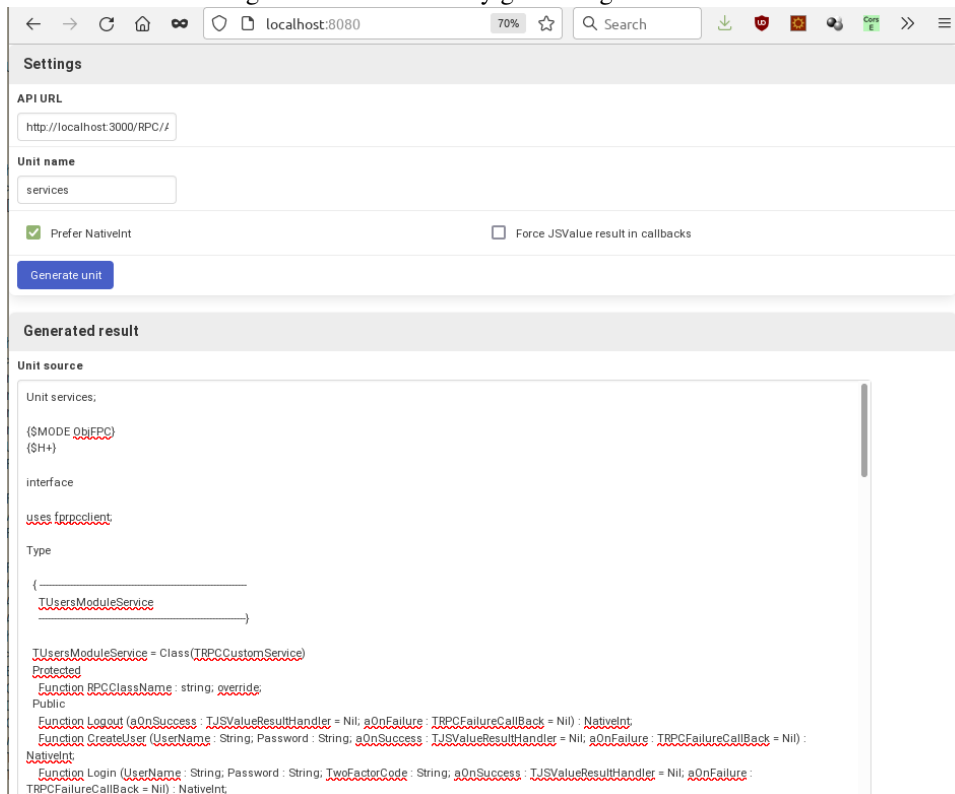
begin
    FRPCClient:=TRPCClient.Create(Self);
    FRPCClient.URL:='/RPC';
    FUserService:=TUserService.Create(Self);
    FUserService.RPCClient:=FRPCClient;
end;

```

This code can be called for example in the constructor of the application object.

It is possible to code the `TUserService` class manually. But even this is not necessary: The FPC JSON-FPC server can generate a description of the available services and method calls in JSON format: an extension of the `Ext.Direct` format used by `ExtJS`.

Figure 11: Automatically generating service code



Using the class `TAPIClientCodeGen` from the `fprpccodegen` unit (available in native FPC and in `pas2js`) the JSON description can be consumed and a unit with the above service code can be automatically generated. The generated unit will contain a service class for every class exposed by the FPC JSON-RPC server.

The `pas2js` distribution contains a demo project (`apiclient`) that uses this unit and allows you to generate the service classes exposed by a server, 100% automatically. All that is required is the URL where the FPC JSON-RPC server is listening for requests. It is shown in figure 11 on page 24.

Better yet, the trunk version of the FPC JSON-RPC server code can generate this code automatically, you can get it by entering the following URL in the browser:

`http://localhost:3000/RPC/API?format=pascal&unitname=services`

This way, your service description can be regenerated at any moment, and will always reflect exactly what the server is expecting as input and what data it is returning.

Since the JSON RPC server only supports JSON types, the generated code can only use the generic JSON types when generating code. An extension is planned where type hints can be given and for example a record type can be specified instead of a generic `TJSONObject` class, or a `TDateTime` instead of a string.

## 8 Conclusion

In this article we have shown how to construct a RPC server using a click-and-point mechanism. We've also shown how to call the RPC server and how to generate a service



description. The GUI of our application has been expanded, and when you look at the `BindElements` method, you'll see that this has become quite large. In the next article, we'll show how to generate this code automatically, and how to load the HTML for the dialogs dynamically.