Push notification support in Free Pascal

Michaël Van Canneyt

March 23, 2024

Abstract

Sending a message to a smartphone or a web application is part of many applications. Usually these messages are sent by a background service. Free Pascal contains units with which you can send Push notifications. A closer look.

1 Introduction

There are times when you may wish to send a message to people in your community, or your users, to notify them of an interesting or relevant event or a piece of news.

For the browser, there is a standard called 'WebPush' which allows you to send messages to a browser. The browser keeps a background process (so-called service workers) running which listens for such messages, and displays them when they arrive.

Smartphones have their own version of the messaging protocol: depending on the OS (Android or iOS), the API is different.

Regardless of the platform, the mode of operation is the same: The process starts by asking the user permission to show him or her notifications. Once the permission is granted, the OS or the browser can subscribe to the push message service, which returns a unique token that can be used to send messages to the device on which the user granted permission. The token is valid till the user retracts his permission.

Google offers a service which allows you to use such a token to send a message, regardless of the platform which issued the token: Firebase Cloud Messaging (FCM). It allows you to send messages, and provides you with statistics on how the users reacted on your messages.

This service can be accessed using a simple REST API. Thanks to some generous sponsoring, Free Pascal contains a unit which allows you to use this API without the need to worry about the details of the API.

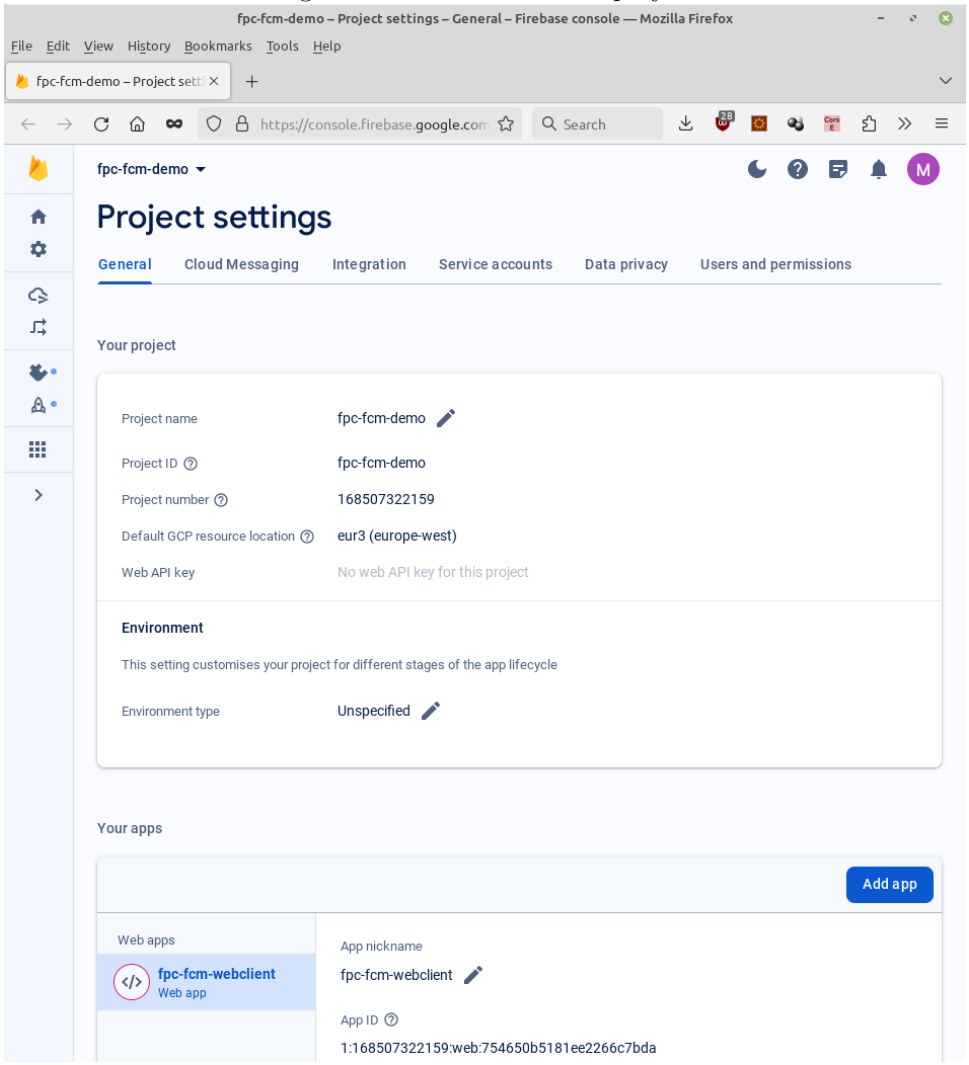
In what follows, we'll examine this unit.

2 Prerequisites

Before we dive into the API, some preparations must be made. FCM is not a free service, and if you plan to send a lot of messages, you will need to pay. It should not come as a surprise that Google requires you to register the application that you wish to use to send messages.

If you have not done this before, then you need to start by creating a Firebase project in the Google Firebase console:

Figure 1: A finished firebase project



`https://console.firebase.google.com/`

You will of course need to have a google account to log in to this service. Any valid Google account can be used.

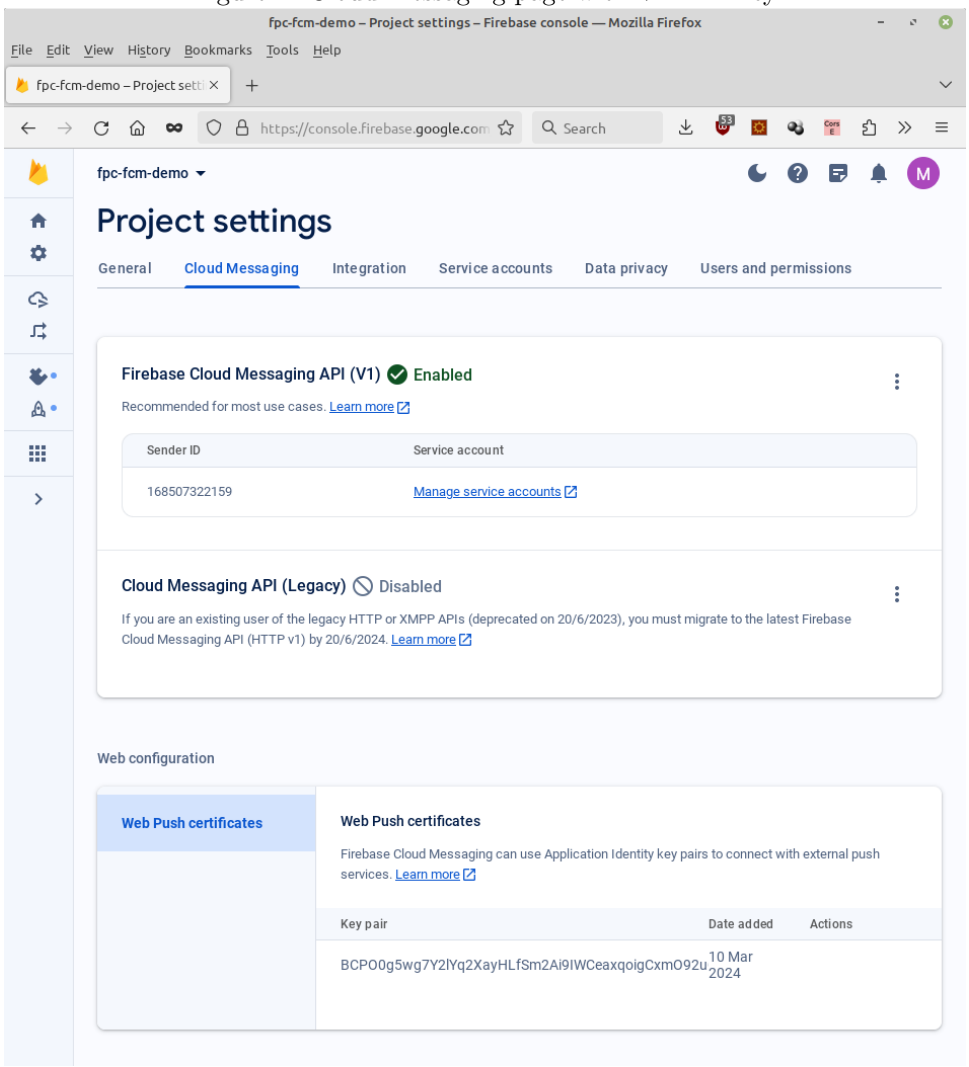
The project dropdown contains a menu item to create a new project. A complete walkthrough of the creation of a project is outside the scope of this article, but the process is simple and well-documented elsewhere (although the details of the screens change regularly).

When done, you should end up with something like figure 1 on page 2.

The next step is to register the application that will access the Google Firebase service. Multiple appplications can use the same Firebase Cloud Messaging project, and you should register them separately: On the page shown in figure 1 on page 2 you can see a button with which you can create a new application.

The application definition is important: when the application is created, you can download a JSON application configuration file which you need to use when accessing the Google FCM APIs: among other things it contains a unique project

Figure 2: Cloud Messaging page with VAPID Key



identifier. The file contents will look something like this:

```
{  apiKey: "XYZ",
   authDomain: "fpc-fcm-demo.firebaseapp.com",
   projectId: "fpc-fcm-demo",
   storageBucket: "fpc-fcm-demo.appspot.com",
   messagingSenderId: "123",
   appId: "1:123" }
```

If you wish to send messages to users in the browser, you will need a VAPID key. This is a special key identifying your browser application, and is required by the WebPush protocol that is used by Firebase to send messages to the browser.

When your Firebase project is created, you can get a VAPID key from the project page on the 'Cloud Messaging' page (see figure 2 on page 3), below 'Web Push Certificates'. The public key needs to be copied, as it will have to be used in the browser.

The last step is to create a Service Account: The service account is needed to actu-

ally send messages. This account is used to get an access token for authenticating the requests to the Firebase HTTP send REST APIs. You can get the service account private key on the 'Service accounts' page from your project, shown in figure 3 on page 5. Just like the application info configuration file, you need to download the service account info file, and save it somewhere on your harddisk. It should look something like the following:

```
{ "type": "service_account",
  "project_id": "fpc-fcm-demo",
  "private_key_id": "123456789",
  "private_key": "XXXYYYZZZ",
  "client_email": "firebase-adminsdk@gserviceaccount.com",
  "client_id": "987654321",
  "auth_uri": "https://accounts.google.com/o/oauth2/auth",
  "token_uri": "https://oauth2.googleapis.com/token",
  "auth_provider_x509_cert_url":
      "https://www.googleapis.com/oauth2/v1/certs",
  "client_x509_cert_url": "https://www.googleapis.com/",
  "universe_domain": "googleapis.com" }
```

When all these steps are completed, you have all information to get started.

3 The message object

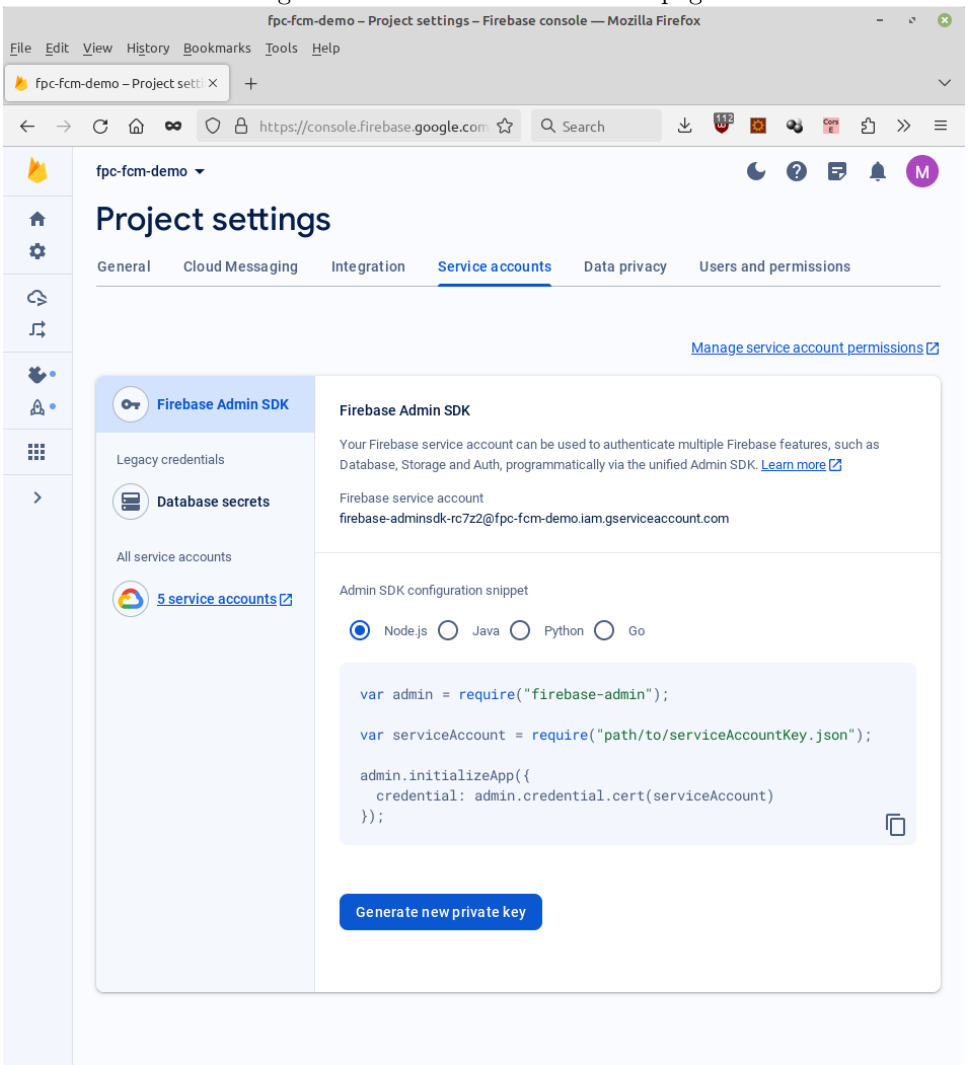
To send a message to the Firebase Cloud Messaging APIs means to send a JSON object. The following page describes the message that can be sent:

```
https://firebase.google.com/docs/reference/fcm/
   rest/v1/projects.messages#resource:-message
```

The `fpfcmtypes` unit contains an Object Pascal class definition that offers you all the fields present in the specification. This has the advantage that you can use code completion.

```
TNotificationMessage = class(TJSONPersist)
public
  procedure ToJSON(aObj : TJSONObject);
  function Encode : string;
  procedure Clear;
  // toplevel properties, valid for all platforms.
  // Notification data.
  Property Recipient : String;
  Property RecipientType : TRecipientType;
  property Data: TStrings;
  property Title: string;
  property Body: string;
  property Image: string;
  // available in Apple and Android, not in web.
  property Options: TMessageOptions;
  Property SendOptions : TNotificationSendOptions;
  // Apple specific
  property AppleConfig : TAppleConfig;
  // Android specific
```

Figure 3: The service account info page



```
  property AndroidConfig : TAndroidConfig;
  // Web specific
  Property WebPushConfig : TWebPushConfig;
  // FCM options
  Property FCMOptions : TFCMOptions;
end;
```

The meaning of the `Recipient`, `Title` , `body` and `Image` properties should be intuitively clear. The `RecipientType` property is of type `TRecipientType` and can be one of `rtToken`, `rtTopic`, `rtCondition`: you can send messages to one person, or multiple persons.

The `AppleConfig`, `AndroidConfig`, `WebPushConfig` and `FCMOptions` contain specific configuration options which FCM will use when sending the message to one of these platforms. For day-to-day use you will probably not need to set these, but they are all detailed in the webpage above: the properties of these objects reflect the properties specified in the specification.

The `SendOptions` property is a set which tells the message object which of these optional configuration parts it should include in the JSON message.

Once you have filled the object properties, the `Encode` function will return a string that contains the JSON message to send to the FCM server. The `Clear` method will clear all properties.

So, to send the same message to multiple users, you fill the message properties, and in a loop set the recipient, retrieve the JSON and send the JSON to the FCM server.

4 The client object

The `fpfcmclient` unit contains the `TFCMClient` object. This class is the client used to communicate with the FCM server. It takes care of authentication and sending the message. The class does not have a lot of methods or properties:

```
TFCMClient = class(TComponent)
Public
  Procedure InitServiceAccount(const aFileName: string;
                               aRoot: TJSONStringType);
  Procedure InitServiceAccount(const aJSON : TJSONObject);
  function Send(aMsg : TNotificationMessage;
                aRecipient : UTF8String) : Boolean;
  function Send(aMsg : TNotificationMessage;
                aRecipients : Array of UTF8String) : Boolean;
  Property WebClient : TAbstractWebClient;
  property BearerToken: TBearerToken;
  property ServiceAccount: TServiceAccountData;
  Property LogFile: String;
  property OnError: TFCMErrorEvent;
  property OnNewBearerToken : TFCMBearerTokenEvent;
  property OnResponse: TFCMResponseEvent;
end;
```

The properties are quite simple:

WebClient This can be set to an instance of an abstract `TWebClient` class. This

class abstracts away the details of the HTTP protocol, we'll show how to use this. If you don't set it, then the `TFCMClient` class will create a default instance.

BearerToken The bearer Authentication token to be used in the next HTTP request. You can set this if you stored it somewhere. If none was set or the token is expired, a bearer token will be fetched as needed using the service account details.

ServiceAccount This property provides read-only access to the Service account data. This property must be initialized with the `InitServiceAccount` method.

LogFile Set this property to the name of a file if you want a log of the HTTP communications with google FCM servers: all headers and bodies of request and response will be written to this file.

The following events are fully optional:

OnError You can set this event if you wish to handle errors yourself. If not set, an exception is raised on error.

OnNewBearerToken as mentioned before, a bearer token will be fetched as needed using the service account details. This event is called when a new access token was received. You can use it to store it for the next call.

OnResponse this event is called with the send response: you can store the response in a database if you so desire.

The class has only 2 public methods:

InitServiceAccount You can call this with the name of a file and an optional path to a root element. This will initialize the `ServiceAccount` data. The file to provide is the service account configuration file you downloaded when you set up the service account. You can also obtain the JSON data by some other means and directly provide the JSON data.

Send This will send the message to the the FCM server using the HTTP REST Api. You can specify one or multiple recipients. The `Recipient` field of the message will be filled with each of the specified recipients, and the message will be sent. The function returns `True` if the message was sent successfully (if you don't handle errors using `OnError`, an exception is raised when something goes wrong.)

Armed with this class, how can we send a push notification ? Quite simply. To demonstrate this, We create a console application based on `TCustomApplication`. In its `DoRun` method, we enter the following code:

```
Recip:=GetOptionValue('r','recipient');
Msg:=nil;
Client:=TFCMClient.Create(Self);
Try
  ConfigureClient(Client);
  Msg:=TNotificationMessage.Create;
  ConfigureMessage(Msg);
  Client.Send(Msg,Recip);
Finally
```

```
  Msg.Free;
  Client.Free;
end;
```

The `Recip` call The most work happens in the `ConfigureClient` and `ConfigureMessage` calls:

To configure the client, we need several items, all of which will be provided through the command-line options of our program. The first thing is to initialize the service account. The file with the service account information can be specified using the `-s` command-line option, but a default filename is used if it was not specified:

```
procedure TFCMApplication.ConfigureClient(aClient : TFCMClient);

const
  Err =  'No service account configuration file found: %s';

var
  CfgFile : string;

begin
  // Service account info
  CfgFile:=GetOptionValue('s','service-account');
  if CfgFile='' then
    CfgFile:=ChangeFileExt(ParamStr(0),'-service-account.json');
  if not FileExists(CfgFile) then
    Raise EInOutError.CreateFmt(Err,[CfgFile]);
  aClient.InitServiceAccount(CfgFile,'');
  // Access token reuse
  if HasOption('a','access-token') then
    begin
    FAccessTokenFile:=GetOptionValue('a','access-token');
    // Load initial token
    if FileExists(FAccessTokenFile) then
      aClient.BearerToken.LoadFromFile(FAccessTokenFile);
    // Set handler so we save the token when it was fetched.
    aClient.OnNewBearerToken:=@DoHandleNewToken;
    end;
  // Log file
  if HasOption('l','log') then
    aClient.LogFile:=GetOptionValue('l','log');
end;
```

Using the `-a` option you can specify the bearer token: the bearer token record has a method to load the token from file, or to save it to file. you can use these methods to reuse the same token. Normally a token has a life time of about 1 hour, after which a new token must be fetched. Lastly the log file can be set using the `-l` option.

The `DoHandleNewToken` method is called when a new token is requested. In this method, the token can be saved to file:

```
procedure TFCMApplication.DoHandleNewToken(Sender: TObject;
                                           const aToken: TBearerToken);
begin
```

```
  aToken.SaveToFile(FAccessTokenFile);
end;
```

The `ConfigureMessage` method sets the properties of the notification message. You can specify a JSON file to load the message from (using the `-m` option). Or you can specify the message body, title and image with the `-b`, `-t` `-i` options, respectively.

```
procedure TFCMApplication.ConfigureMessage(Msg : TNotificationMessage);

begin
  if HasOption('m','message') then
    LoadMessageFromFile(Msg,GetOptionValue('m','message'));
  if HasOption('t','title') then
    Msg.Title:=GetoptionValue('t','title');
  if HasOption('b','body') then
    Msg.Body:=GetoptionValue('b','body');
  if HasOption('i','image') then
    Msg.Body:=GetoptionValue('i','image');
end;
```

The `LoadMessageFromFile` is again quite simple:

```
procedure TFCMApplication.LoadMessageFromFile(
     Msg : TNotificationMessage;
     const aFileName : string);

Var
  F : TFileStream;
  D : TJSONData;
  Obj : TJSONObject absolute D;

begin
  D:=Nil;
  F:=TFileStream.Create(aFileName,fmOpenRead or fmShareDenyWrite);
  try
    D:=GetJSON(F);
    if not (D is TJSONObject) then
      Raise EFCM.CreateFmt('Invalid JSON data in message file %s',[aFileName]);
    Msg.Title:=Obj.Get('title',Msg.Title);
    Msg.Body:=Obj.Get('body',Msg.Body);
    Msg.Image:=Obj.Get('image',Msg.Image);
  finally
    D.Free;
    F.Free;
  end;
end;
```

As you can see, the message file is a simple JSON file with 3 keys: `title`, `body` and `image`

With that, the client is almost ready to be used. There is one small detail to take care of: the `WebClient` property is not set anywhere. The `TFCMClient` class will then create a default webclient. The default webclient needs to be configured, and this can be done by adding the following units to the uses clause:

fphttpwebclient this sets the default web client to a class based on `TFPHTTPClient`;

opensslsockets this enables support for HTTPS for `TFPHTTPClient`. HTTPS is needed to communicate with the FCM services.

Additionally, in the program start code, we need to add the following line:

```
DefaultWebClientClass:=TFPHTTPWebClient;
```

This will instruct the `TFCMClient` class to use the `TFPHTTPWebClient` class when creating a `TWebClient` instance.

With this code in place (plus some helper code to show a usage message and some checks for the command-line options), the program can be executed from the command-line as follows:

```
sendmsg -m message.json -s service.json -a token.json -r TOKEN
```

Here `TOKEN` needs to be replaced with a token obtained by asking the user for permission to send him (or her) a message. The service acount data will be loaded from file `service.json` (which you should have downloaded using the FCM project console).

The message is specified in the file `message.json`:

```
{
  "title"  : "A nice message",
  "body" : "With a nice body",
  "image" : "https://www.freepascal.org/favicon.png"
}
```

If you specify the command several times with different recipient tokens, you'll see that the token is saved in the `token.json` file and reused for the next calls.

5 Obtaining a token using a website

In the above, we have not shown how to get a token for sending a message to a user, we assumed it was available. In practice, this token must be obtained from the user by asking his permission to send him messages. This is a function that is available in mobile operating systems and in the browser.

We'll use pas2js to demonstrate how to get such a token and use it to send a message to the browser. The browser has a simple interface to show notifications, which is - unsurprisingly - called `Notification` and which is detailed here:

```
https://developer.mozilla.org/en-US/docs/Web/API/notification
```

The method to request permission to show notifications is - can it be more simple? - called `requestPermission`.

When called, the browser will pop up a message asking for your permission to show you you notifications. The return value is a Promise, which will be fulfilled when the user has granted (or denied) the permission. After obtaining permission, you can get the token that you can use to send messages.

To do this, the `Push` manager can be used:

https://developer.mozilla.org/en-US/docs/Web/API/PushManager

The subscribe call will result in a subscription, which can be converted to a token.

The developers of Firebase have created a little API layer around this call, and we will follow their guidelines to get and use a token. This is a simple precaution: in fact it is not clear from the Firebase documentation whether Firebase simply uses the raw token obtained from the browser or adds some information to it which it needs to deliver the message.

The Firebase Cloud Messaging API has been made available in the firebaseapp unit. We will use it to get a token, and we'll send this token to a small HTTP server application, which will use it to send a Push notification using the TFCMClient component presented above.

So, to this end, we create a new "Web Browser Application" project in Lazarus. This will create a HTML page and a program file. In the HTML page, some files need to be added to be able to use the firebase API:

```
<script src="firebase-app-compat.js"></script>
<script src="firebase-messaging-compat.js"></script>
```

These files can be downloaded from the firebase site, as described under:

https://firebase.google.com/docs/web/alt-setup

We're using the compatibility layer (because that is what is described in the firebaseapp unit), but it would also be possible to use the modular API.

To initialize a Firebase application, the application configuration file which you created and downloaded when you defined your project in the Firebase console, needs to be used. You can include the JSON object definition in the application code, but you can also put it in a file on your HTTP server:

```
var firebaseConfig = {
  authDomain: "fpc-fcm-demo.firebaseapp.com",
  projectId: "fpc-fcm-demo",
  storageBucket: "fpc-fcm-demo.appspot.com",
  messagingSenderId: "123",
  appId: "1:123"
};
```

You can then include this file (we'll name it config.js) in the main HTML File of your project together with the include of your project file:

```
<script src="config.js"></script>
<script src="webclient.js"></script>
```

The rest of the HTML is quite simple: we add 1 edit control (edtMessage) and 2 buttons (btnSend and btnRegister) to the HTML, plus some DIV tags to display the token and some output of the program.

```
<div class="container">
  <div class="box">
    <h3 class="title is-3">FCM Push notification demo</h3>
    <div class="field">
      <label class="label">Message to send:</label>
```

```
      <div class="control">
        <input id="edtMessage" class="input" type="text" placeholder="Message to send">
      </div>
    </div>  <!-- .field -->

    <div class="field is-grouped">
      <div class="control">
        <button id="btnSend" class="button is-link" disabled>Send</button>
      </div>
      <div class="control">
        <button id="btnRegister" class="button is-link is-light">Register</button>
      </div>
    </div> <!-- .field -->
  </div> <!-- .box -->
  <div id="pnlToken" class="box is-hidden">
    <p>Your token: <span id="lblToken">?</span> <p>
  </div> <!-- .box -->
</div> <!-- .container -->
<script>
rtl.run();
</script>
<div id="pasjsconsole"></div>
```

The `pasjsconsole` element is where `WriteLn` feedback will be displayed.

The application code is in a `TDemoApp` class, a descendent of the `TBrowserApplication` class that comes standard with Pas2JS. When the application starts, the `DoRun` method is called. In it, the following code is executed to initialize some fields representing the various `div` HTML elements and the buttons. For the latter it also sets the `click` callbacks:

```
var
  config : TJSObject; external name 'firebaseConfig';

procedure TDemoApp.DoRun;

begin
  RPCModule:=TRPCModule.Create(Self);
  pnlToken:=GetHTMLElement('pnlToken');
  lblToken:=GetHTMLElement('lblToken');
  edtMessage:=TJSHTMLInputElement(GetHTMLElement('edtMessage'));
  btnSend:=TJSHTMLButtonElement(GetHTMLElement('btnSend'));
  btnSend.addEventListener('click',@HandleSend);
  btnRegister:=TJSHTMLButtonElement(GetHTMLElement('btnRegister'));
  btnRegister.addEventListener('click',@HandleRegister);
  Writeln('Initializing application...');
  App:=Firebase.initializeApp(config);
  App.messaging.onMessage(@HandleReceivedMessage);
  RegisterServiceWorker;
end;
```

The last lines initialize the Firebase API using the `config` object from our `config.js` file, and sets the `OnMessage` event handler of the firebird messaging API. The `OnMessage` event can be used to react to messages. Here we'll just display the notification message by creating a `TJSNotification` instance:

```
procedure TDemoApp.HandleReceivedMessage(aMessage: TJSObject);
var
  Notif : TJSObject;
  Opts : TJSNotificationOptions;

begin
  if assigned(aMessage) then
    console.debug('Message received: ',aMessage);
  Notif:=TJSObject(aMessage['notification']);
  Opts:=TJSNotificationOptions.new;
  Opts.body:=string(Notif['body']);
  Opts.image:=string(Notif['image']);
  TJSNotification.new(string(Notif['title']),opts);
end;
```

We can do this because we requested permission from the user to display notifications. Note that when the webpage is not loaded and focused, the service worker will display the message (also using the `Notification` API of the browser).

The last step when initializing the application is to register a service worker script:

```
procedure TDemoApp.RegisterServiceWorker;

begin
  Window.Navigator.serviceWorker.register('firebase-messaging-sw.js').
    _then(function (js : JSValue) :JSValue
    begin
    reg:=weborworker.TJSServiceWorkerRegistration(js);
    if assigned(Reg) then
      Writeln('Registered service worker...')
    end,function (js : JSValue) :JSValue
    begin
      Writeln('Unable to register service worker')
    end);
end;
```

in the above code, the service worker script is called "`firebase-messaging-sw.js`", it can be downloaded from the Firebase documentation pages. The `register` method returns a promise, which resolves to a service worker registration object. Here we just save the registration in the `reg` field for later use, and display a message to show whether the service worker was succesfully registered or not.

For more elaborate web pages, more things can be done once the service worker is registered, such as establishing a message channel between the service worker and the main web page. For our current example, this is not needed.

A service worker is a small service, maintained by the browser: It will run in the background, and one thing it can do is to listen for incoming messages - exactly what it needs to do for our demo.

The service worker script does not do much, except initializing the firebase messaging application:

```
importScripts('https://www.gstatic.com/firebasejs/9.2.0/firebase-app-compat.js');
importScripts('https://www.gstatic.com/firebasejs/9.2.0/firebase-messaging-compat.js');
importScripts('config.js');
```

```
firebase.initializeApp(firebaseConfig);
```

With all this, the browser application is ready to receive push messages.

To get a Firebase messaging token, the user must click the 'Register' button. In the 'click' handler of the register button (`HandleRegister`), the service worker registration which we saved earlier is passed on to the `getToken` call of the Firebase messaging API to get a Firebase messaging token.

```
TFirebaseMessaging = class external name 'firebase.messaging.Messaging' (TJSObject)
  function getToken (options : TMessagingGetTokenOptions): string; async;
end;
```

The `TMessagingGetTokenOptions` object has a `serviceworkerRegistration` field: When set, the firebase API knows which service worker will be handling the receipt of messages. Additionally, the `vapidkey` field must be set to the VAPID key you created when the Firebase project was created. The `TheVAPIDKey` constant (not shown here) contains this key.

```
procedure TDemoApp.HandleRegister(event: TJSEvent);

var
  Token : string;
  opt : TMessagingGetTokenOptions;

begin
  opt:=TMessagingGetTokenOptions.New;
  opt.serviceworkerRegistration:=self.Reg;
  opt.vapidKey:=TheVAPIDKey;
  Token:=Await(App.messaging.getToken(opt));
  if (token='') then
    RequestPermission
  else
    HaveToken(token);
end;
```

The `GetToken` call is an `Async` call, so it returns a promise. Using the `Await` built-in, we can transform it to an actual token string. If the token string is empty, the firebase API does not yet have a token, and we must request the user his or her permission to show notifications. If a non-empty token is returned, we can send it to the server. This is done using the `RequestPermission` and `HaveToken` calls, respectively:

```
procedure TDemoApp.requestPermission;

 function onpermission (permission : jsvalue) : jsvalue;

 var
   token : string;

 begin
   if (permission='granted') then
     begin
     writeln('Notification permission granted.');
     handleregister(nil);
```

```
    end;
 end;

begin
  Writeln('Requesting permission...');
  TJSNotification.requestPermission()._then(@OnPermission)
end;
```

The `RequestPermission` class method of `TJSNotification` is a method of the browser. Since the request for permission can take a while, the result of the call is a promise: When the user reacted to the prompt for permission to send messages, the promise resolves to a string that contains the decision of the user: permission is granted or denied.

When permission is granted, the `HandleRegister` method is again called: in that case, the Firebase messaging API's `getToken` method will succeed and the token will be sent to the `HaveToken` call. This call shows the token in the HTML, and sends the token to our application server:

```
procedure TDemoApp.HaveToken(aToken : string);

begin
  Showtoken(aToken);
  Sendtoken(aToken);
  btnSend.disabled:=False;
  btnRegister.disabled:=False;
end;
```

The `ShowToken` is easy, it sets the inner text of the `DIV` element with id `lblToken`:

```
procedure TDemoApp.ShowToken(aToken : string);
begin
  pnlToken.classlist.remove('is-hidden');
  lblToken.innerText:=aToken;
  Writeln('Received token: ',aToken);
end;
```

The `SendToken` uses a JSON-RPC call `RegisterSubscription` to send the token to our application server :

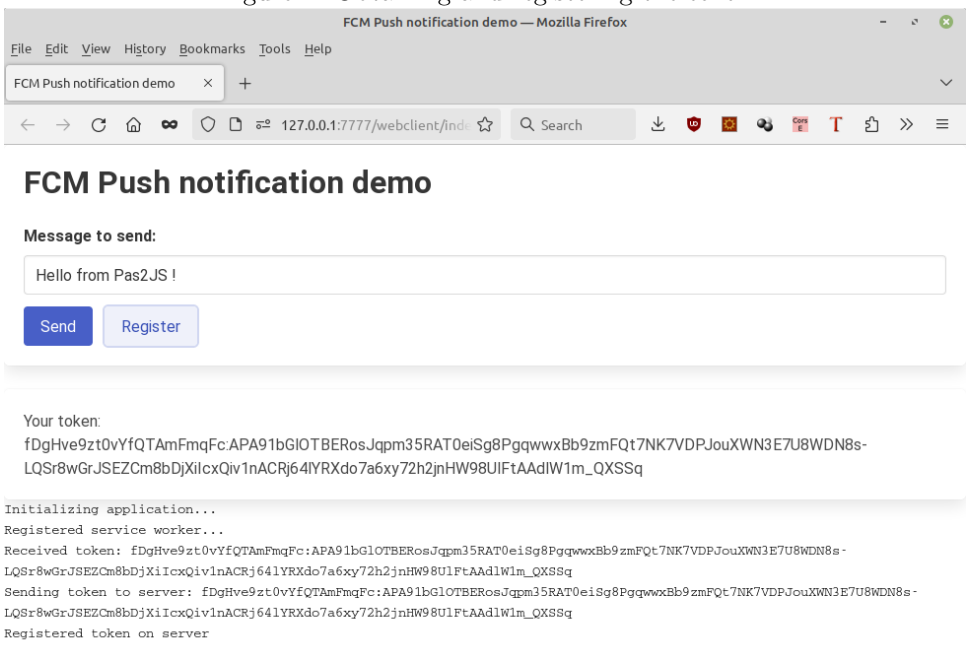
```
procedure TDemoApp.SendToken(aToken : string);

  procedure DoOK(aResult: JSValue);
  begin
    Writeln('Registered token on server');
  end;

  procedure DoFail(Sender: TObject; const aError: TRPCError);
  begin
    Writeln('Failed to register token on server: '+aError.Message);
  end;

begin
  Writeln('Sending token to server: ',aToken);
  RPCModule.Service.RegisterSubscription(aToken,@DoOK,@DoFail);
end;
```

Figure 4: Obtaining and registering the token



The result of this code can be seen in figure 4 on page 16.

Once the token has been obtained and was succesfully sent to the server, the user can use the `Send` button to send a message to himself. In a real application, the messages will of course be sent by the server in response to some external event.

The event handler of the `Send` button is `HandleSend`: This method constructs a small JSON object with the data for the message. Note that the format of the message object is the same as the one we used to send a message using the command-line utility presented earlier.

When the message object is constructed, the handler uses the `SendNotification` JSON-RPC call to send the message to our application server:

```
procedure TDemoApp.handlesend(event: TJSEvent);

  procedure DoOK(aResult: JSValue);
  begin
    Writeln('Message transferred to server for sending');
  end;

  procedure DoFail(Sender: TObject; const aError: TRPCError);
  begin
    Writeln('Failed to transfer message to server for sending: '+aError.Message);
  end;

var
  Msg : TJSObject;

begin
  Msg:=New([
    'title','Free Pascal FCM demo',
```

```
    'body',edtMessage.Value,
    'image','https://www.freepascal.org/favicon.png'
  ]);
  Writeln('Sending message : ',TJSJSON.stringify(Msg));
  RPCModule.Service.SendNotification(Msg,@DoOK,@DoFail);
end;
```

This concludes the interactive part of the application.

The application server exposes a small JSON-RPC service. As shown in previous articles about Pas2JS and its support for JSON-RPC, and a unit with the proxy object for this JSON-RPC server can be generated automatically (the unit is called `service.messgingserver`). The generated proxy object has the following declaration:

```
TMessagingService = Class(TRPCCustomService)
Protected
  Function RPCClassName : string; override;
Public
  Function SendNotification (Message : TJSObject;
                             aOnSuccess : TJSValueResultHandler = Nil;
                             aOnFailure : TRPCFailureCallBack = Nil) : NativeInt;
  Function RegisterSubscription (Token : String;
                                 aOnSuccess : TJSValueResultHandler = Nil;
                                 aOnFailure : TRPCFailureCallBack = Nil) : NativeInt;
end;
```

An instance of this proxy object is created on a RPCModule datamodule, which is implemented in the `module.messagingservice` unit:

```
TRPCModule = class(TDataModule)
  Client: TPas2jsRPCClient;
  procedure DataModuleCreate(Sender: TObject);
private
  FService: TMessagingService;
public
  Property Service : TMessagingService Read FService;
end;
```

The data module has a `TPas2JSRPCClient` component on it, and the proxy object is created in the `OnCreate` event of the RPCModule datamodule, and connected to the RPCClient object:

```
procedure TRPCModule.DataModuleCreate(Sender: TObject);
begin
  FService:=TMessagingService.Create(Self);
  FService.RPCClient:=Client;
end;
```

With that, our web application is finished.

6 Sending a message from an application server

In the above, we created a web page that is set up to receive and display push notifications. The actual notifications are sent by a HTTP application server:

The demonstration application server is a simple HTTP application. It exposes a JSON-RPC service, which is implemented in a unit `module.rpc`. The actual handling of the RPC calls is implemented in the `module.messaging` unit, by 2 `TJSONRPCHandler` objects called `RegisterSubscription` and `SendNotification`.

The `OnExecute` event handler of the `RegisterSubscription` object is quite simple: it extracts the token from the JSON data passed from the browser:

```
procedure TdmMessaging.RegisterSubscriptionExecute(Sender: TObject;
   const Params: TJSONData;
   out Res: TJSONData);

var
  Parms: TJSONArray absolute params;
  aToken : UTF8String;

begin
  If Parms.Count<>1 then
    Raise Exception.Create('Invalid param count');
  if Parms[0].JSONType<>JTString then
    Raise Exception.Create('Invalid param type for token');
  aToken:=Parms[0].AsString
  SaveToken(aToken);
  Res:=TJSONBoolean.Create(True);
end;
```

When the call to the `Savetoken` method returns, a result is sent back to the browser.

The `SaveToken` method uses a simple stringlist, which it loads from file, adds the token to and saves again in the file:

```
procedure TdmMessaging.SaveToken(const aToken : UTF8String);

var
  L : TStrings;
  FN : String;

begin
  FN:=DeviceTokensFileName;
  L:=TStringList.Create;
  try
    if FileExists(FN) then
      L.LoadFromFile(FN);
    L.Add(aToken);
    L.SaveToFile(FN);
  finally
    L.Free;
  end;
end;
```

The `DeviceTokensFileName` function returns the name of the file in which tokens must be saved, the details of this function are in the source code of this application.

The handling of the `SendNotification` message does something similar: it extracts the data from the JSON object passed from the client to fill the `TNotificationMessage` object:

```
procedure TdmMessaging.SendNotificationExecute(Sender: TObject;
                                               const Params: TJSONData;
                                               out Res: TJSONData);

var
  Parms: TJSONArray absolute params;
  Obj : TJSONObject;
  Msg : TNotificationMessage;

begin
  If Parms.Count<>1 then
    Raise Exception.Create('Invalid param count');
  if Parms[0].JSONType<>jtObject then
    Raise Exception.Create('Invalid notification');
  Obj:=Parms.Objects[0];
  Msg:=TNotificationMessage.Create;
  try
    Msg.Title:=Obj.Get('title',Msg.Title);
    Msg.Body:=Obj.Get('body',Msg.Body);
    Msg.Image:=Obj.Get('image',Msg.Image);
    SendMessage(Msg);
    Res:=TJSONBoolean.Create(True);
  finally
    Msg.Free;
  end;
end;
```

When the message object is filled with the data from the webpage, it is passed on to the `SendMessage` method, and a result is sent back to the browser.

The `SendMessage` method does the actual work of sending the push notification message with the `TFCMClient` class. It starts by loading the last token from the token file created by the `RegisterSubscription` call:

```
procedure TdmMessaging.SendMessage(Msg : TNotificationmessage);

var
  Sender : TFCMClient;
  aConfig, aToken : String;

begin
  aToken:=LoadLastToken;
  Sender:=TFCMClient.Create(Self);
  try
    aConfig:=GetServiceAccountFileName;
    Sender.LogFile:=GetLogFileName;
    Sender.InitServiceAccount(aConfig,'');
    Sender.OnNewBearerToken:=@HandleNewAccessToken;
    if FileExists(AccessTokenFile) then
      Sender.BearerToken.LoadFromFile(AccessTokenFile);
    Sender.Send(Msg,aToken);
  finally
    Sender.Free;
  end;
end;
```

The `HandleNewAccessToken` method (used to save the access token) is identical to the one in our command-line utility. All that remains to be done is to add the units that set the default `WebClient` class to use.

After that, our application is finished. It should be clear that other than saving and loading the token from a file, this application server code is not substantially different from the code in the command-line utility we constructed earlier.

To run this example, you must follow the following steps:

- Make sure the service account file is located next to the HTTP server program (`messageserver`) with the correct name: `messagserver-serviceaccount.json`.
- Start the messageserver HTTP server program.
- Make sure the `config.js` file is properly set up as described above, as well as the firebase Javascript files.
- Start the web-based client program in the browser. To do so, for example in Lazarus, just press F9 to run it, and it should open in the browser.
- In the Firefox browser, make sure the developer tools console is opened (press F12), and set the 'Enable service workers over HTTP (when toolbox is open)' in the toolbox settings. This step is only necessary for testing: if you host the page on a HTTPS website, then service workers will automatically be allowed.
- Press the register button in the website. You should see a confirmation message as in figure 4 on page 16.
- Type some nice message and press 'Send'.

The result should look like figure 5 on page 21. Note that due to the particular desktop manager of the author, the icon is not shown in the picture. If you noted the token, you should also be able to send a message to the same browser using the command-line tool.

7 Conclusion

In this article we showed that sending push notifications to a user is not so difficult to do. In fact, more time is spent on setting up all the necessary files and configuring the project on firebase, than on actually coding the application. The sending of messages made use of FCM - Firebase Cloud Messaging. It is possible to do the same without Firebase, by using the WebPush protocol built-in in the browser. We'll leave the discussion of that to a future contribution.

Figure 5: Receiving the message