Executing programs on the server in Pas2JS

Michaël Van Canneyt

September 8, 2023

Abstract

In this article we show how to give the user of a browser-based program feedback from long-running processes on the server, using 2 components: one in pas2js, one in Free Pascal/Lazarus.

1 Introduction

When using a web-based program, not everything can be done in the browser. Often, tasks are executed through some RPC (Remote Procedure Call) mechanism on the webserver. This can be a simple task such as executing an SQL statement on a database and returning a result. Or it can be a more complicated and time-consuming task such as making a backup of a database, indexing PDF files, compiling a software project and running a test suite, or even installing software on the server. Ideally, the output of these remote programs should also be presented to the user.

To keep programs scalable, these tasks should be short-lived. A return time of 1 second for a HTTP request is already a long time, so executing a time-consuming task and waiting for the return using a single HTTP request is not a good idea: the HTTP server is occupied with the request, the browser or any proxy servers between the HTTP server and the browser may decide to time-out your request.

Much better is to start the process using a HTTP request, and use a mechanism to poll the status of the executed process. In this article we present one such mechanism.

2 Architecture

The solution we present here consists of 2 components. One component which is used on the server, and which can be used to start a process, capture its output and poll for the status of the process. The other component takes care of the polling process on the client.

These components are ignorant of the communication mechanism between browser and server, this means that they do not implement the actual RPC calls used to start the process: There are many possible mechanisms, and some may be more suitable for your purpose than others.

The components are called `TProcessCapture` for the server part and `TProcessCapturePoller` for the client (pas2js) part. The server part takes care of executing a program and redirecting the output to a file, the client part implements the polling mechanism and some callbacks to handle the actual server calls and the result.

We'll demonstrate both components with a simple set of programs:

- A test program to be executed. It is used for demonstration purposes only.
- A HTTP server program that allows to serve HTML files and that offers an RPC mechanism to start the test program and handle status requests.
- a Simple pas2js program that will run in the browser and which will remotely execute the test program. It will show the output of the test program in the browser.

We'll start with the test program.

3 The test program

To demonstrate the workings, the test program needs to do 3 things:

1. It must run for some time, several seconds at least. This is done with a simple loop and a call to `sleep`.
2. needs to show that it receives command-line arguments: we will simply output the program parameters.
3. It must demonstrate that it is run in a specific directory. We'll just print the working directory.
4. It needs to produce some output.

All this is easily accomplished with a trivial program:

```
uses sysutils;

var
  i : integer;
  D : TDateTime;

begin
  Writeln('Current dir: ',GetCurrentDir);
  Write('Args:');
  For I:=1 to ParamCount do
    Write(' ',ParamStr(i));
  Writeln();
  D:=Now;
  For I:=1 to 150 do
    begin
    Sleep(100);
    Writeln('Tick ',i);
    Flush(output);
    end;
  Writeln(SecondsBetween(Now,D), ' seconds elapsed');
  flush(output);
end;
```

The only noteworthy thing about this program is that it flushes standard output after writing a line: By default, Free Pascal buffers output of writeln statements if it detectes that it is not writing to a console. Since our program will be run with the output redirected, the buffering will be activated, and so, in order to send the output faster to the brouwser, we flush standard output manually.

4 The server component

Before explaining the server component, it is a good idea to explain why a new component is needed. After all, Free Pascal ships since ages with the `TProcess` component, which can be used to start a process and read its output using a stream. So why not simply use that ? This component is not really suitable for our task, for several reasons:

- A web server process (e.g. cgi, fastcgi) can be ended before the process has finished. All information about the executed process would be lost.
- The component cannot be used to redirect output to a file. It would require reading all data from the file in a separate thread, save it somewhere etc. This complicates matters considerably, and if the HTTP program ends, all further input/output would stop.
- similarly, no input file can be specified, it would require similar handling as the output file.
- Item since the `TProcess` component is confined to a single process, there is no way to scale your web application.

In essence, the `TProcess` component is stateful, and we need a stateless component in order to work in a web environment.

So, a new component is needed.

The server component `TProcessCapture` has the following declaration:

```
TProcessCapture = Class(TComponent)
Public
  Function Execute(Exe : String; Args: Array of string) : string;
  Function Execute(Exe : String; Args: TStrings) : string;
  Function CleanupProcess(Const AProcess : String) : Boolean;
  Function GetOutputFile(Const AProcess : String) : String;
  Function GetPidFile(Const AProcess : String) : String;
  Function GetStatusFile(Const AProcess : String) : String;
  Function GetProcessID(Const AProcess : String) : Integer;
  Function IsProcessRunning(Const AProcess : String) : Boolean;
  Function GetProcessExitStatus(Const AProcess : String) : Integer;
  Function GetProcessOutput(Const AProcess : String; Var AOffSet : Integer) : RawByteString;
Published
  Property LogDir : String Read FLogDir Write FLogDir;
  Property InputFile : String Read FInputFile Write FInputFile;
  Property WorkingDir : String Read FWorkingDir Write FWorkingDir;
Property OutputCodePage : TSystemCodePage Read FOutputCodePage Write FOutputCodePage;
end;
```

The main methods are almost self-explanatory:

Execute Executes the program `Exe`, passing it the arguments `Args`, which can be given as an array of strings, or a stringlist. The return of this function is a process identifier.

CleanupProcess will clean up the output and status files for the process identified by `AProcess`. You should call this only after the process has exited.

IsProcessRunning returns `True` if the process identified by `AProcess` is still running.

GetProcessExitStatus returns the exit status of the process identified by `AProcess`. if the process is still running, -1 is returned.

GetProcessOutput returns the output of the process identified by `AProcess`, starting at byte offset `AOffSet` (zero based) till the end of available output.

There are some auxilary methods that you do not need under ordinary circumstances:

GetOutputFile Returns the name of the output file associated with the process `aProcess`.

GetPidFile Returns the name of the process ID file associated with the process `aProcess`. This file will be created as soon as the process starts.

GetStatusFile Returns the name of the status file associated with the process `aProcess`. This file will only exist after the program has exited.

GetProcessID Returns the process ID of the process `AProcess`.

Lastly, there are some properties:

LogDir The directory where all log and status files are created. The directory will be created if it does not exist.

InputFile A file with prepared input for the process. Note that this does not allow you to interact with the process. This property is only used when starting the program.

WorkingDir The working directory for the started program. This property is only used when starting the program.

OutputCodePage The codepage in which the program writes its output.

To work with this component, you will typically perform the following steps:

1. Set appropriate values for `LogDir`, `InputFile`, `WorkingDir` and `OutputCodePage`. They contain sensible defaults, but it is better to be explicit.
2. Start the program using the `Execute` method, and save the resulting `ProcessID` string.
3. initialize an offset variable to zero.
4. Check if the process is still running with `IsProcessRunning`, passing it `ProcessID`.
5. Get the output of the process using `GetProcessOutput`, passing it `ProcessID` and the current offset. Update the offset.

6. Repeat the last 2 steps till the program exits.

It should be noted that you can free the `TProcessCapture` after every step and recreate it before performing a call: it is stateless. This is necessary if the component is to work in a web environment where the different steps will be performed as part of different HTTP requests: the steps may be performed by different instances of the application server. To work correctly, the `LogDir` and `OutputCodePage` properties must be set to the same values between invocations.

It also means that the same component can be used to control different processes. Although this is not recommended if you use threads: the component is not reentrant.

To do its work, the `TProcessCapture` component executes a small helper program called `taskhelper`: this program does the work of launching the actual program that needs to be executed with redirected in and output. It also takes care of registering the exit status of the program. On Unix platforms, it is possible to do without this program, but on Windows, the mechanism to start a new process `CreateProcess` necessitates the use of an extra program. To make the behaviour across platforms consistent, the `taskhelper` program is used everywhere. Its sources are distributed with the trunk version of FPC, but the source has been included in the sources of this article.

5 The server program

To demonstrate the working of the component, we'll make a small HTTP server that executes the test program when it receives a `StartProcess` command from the browser through JSON-RPC, and which has a `GetStatus` command to get the status of the process. The process will also serve the files for the client application. To do this, in the 'New project' dialog we select 'HTTP Server application', and in the wizard that is shown we select 'Server files from default location' and under 'Web module to create' we select 'Web JSON-RPC Module', as shown in figure 1 on page 6. In the next dialog which creates the module to JSON-RPC Module, we only need to register the web module (we have only 1 module in the server application), (see figure 2 on page 7) and we'll use the `/RPC` URL path to serve JSON-RPC requests from.

Once that is done, we need 2 JSONRPCHandler components from the `FPWeb` tab in the component palette, one for each request:

StartProcess this call takes 2 arguments: 2 strings, which we must define in the `Params` property of the component. We'll give them the names `A` and `B`. The call will return the process identifier to the client application.

GetStatus this call also takes 2 arguments: a string (the ProcessID), and an Int64 number (an offset), we also define them in the `Params` property. The call will return the process exit code (-1 if the process is still running), the available output starting at the given offset identifier to the client application. It also returns the new offset.

These 2 RPC calls are the API we expose to the browser to control our process. The actual work is done by the `TProcessCapture` component.

The `TProcessCapture` component is not (yet) on the component palette of Lazarus, so we create it in code in the `OnCreate` handler of the datamodule, and destroy it in the `OnDestroy` handler:

Figure 1: The start of the server application

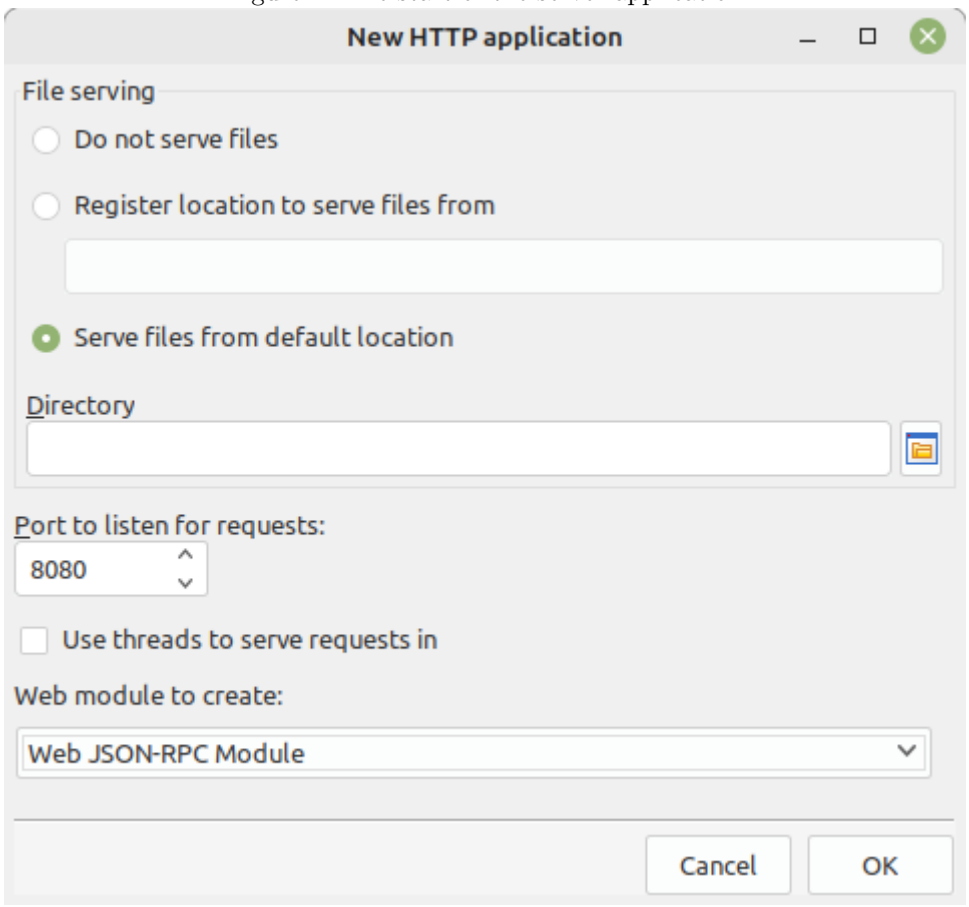
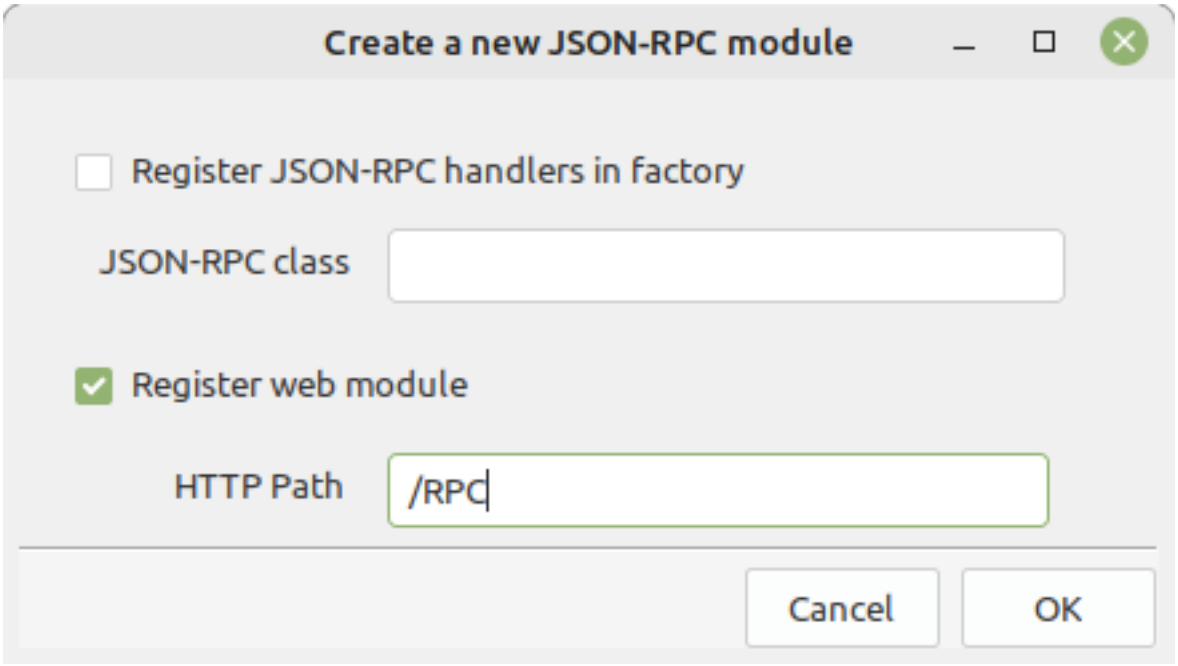


Figure 2: Creating the JSON-RPC webmodule




```
procedure Tprocesscontrol.DataModuleCreate(Sender: TObject);
begin
  Capture:=TProcessCapture.Create(Self);
end;

procedure Tprocesscontrol.DataModuleDestroy(Sender: TObject);
begin
  FreeAndNil(Capture);
end;
```

The latter is strictly speaking not necessary since the component is owned by the datamodule and will be destroyed when the datamodule is destroyed, but for clarity we destroy it manually anyway.

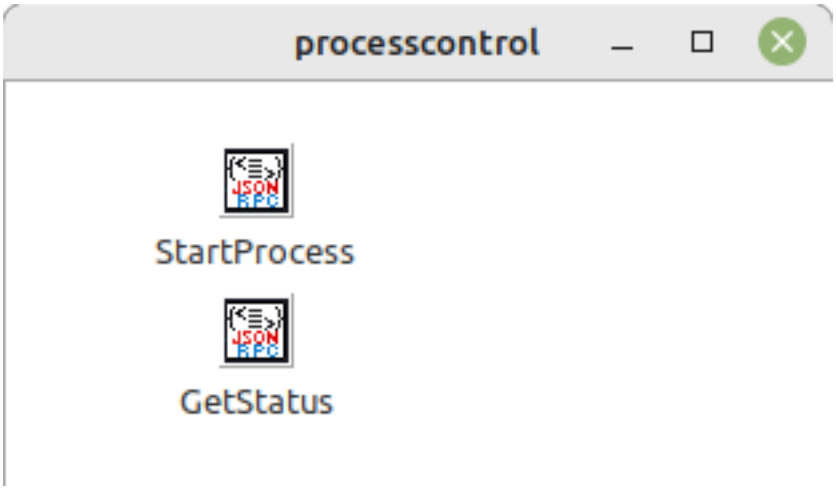
In the `OnExecute` event of the `StartProcess` handler, we collect the 2 arguments `A` and `B` and start the test program:

```
const
  LongProcess = 'longprocess' {$ifdef windows} + '.exe' {$endif} ;

var
  arr : TJSONArray absolute Params;
  a, b, Exe, PID : string;

begin
  Res:=Nil;
  a:=Arr.Strings[0];
  b:=Arr.Strings[1];
```

Figure 3: The finished JSON-RPC webmodule




```
  Exe:=ExtractFilePath(ParamStr(0))+longprocess;
  PID:=Capture.Execute(Exe,[a,b]);
  Res:=TJSONString.Create(PID);
```

As you can see in this code, we use the `Execute` method of the `TProcessCapture` class to start the process.

For the `GetStatus` call, the code is a little longer, but not so much.

The code starts by getting the arguments, and checking the whether the process is still running. If the process is no longer running, then the exit status is retrieved.

```
procedure Tprocesscontrol.GetStatusExecute(Sender: TObject;
   const Params: TJSONData; out Res: TJSONData);
var
  arr : TJSONArray absolute Params;
  PID,aOutput : string;
  Offset,Status : Integer;
begin
  Res:=Nil;
  PID:=Arr.Strings[0];
  OffSet:=Arr.Int64s[1];
  if Capture.IsProcessRunning(PID) then
    Status:=-1
  else
    Status:=Capture.GetProcessExitStatus(PID);
  aOutput:=Capture.GetProcessOutput(PID,Offset);
  Res:=TJSONObject.Create(['status',Status,'output',aOutput,'offset',offset]);
end;
```

Regardless of whether the process was still running or not, finally the available output is retrieved and all 3 elements (status, output, new offset) are returned to the client in a JSON object.

The data module will look like figure 3 on page 8.

Before the program can be used, there are two last things to be done when using the

release version of FPC on linux. The HTTP connection on which requests arrive is passed to the task helper, and as a consequence the connection is not closed when the `StartProcess` call returns, causing the browser to wait till the process exits. This of course defeats the purpose of the whole exercise. To remedy this, we must set the Close-On-Exec flag on the socket handle. This can be done easily by handling the `OnAllowConnect` handler of the HTTP server.

To do so, we add the following to the project file:

```
THTTPApplication = Class(fphttpapp.THTTPApplication)
 constructor Create(aOwner : TComponent); override;
private
 procedure DoConnect(Sender: TObject; ASocket: Longint; var Allow: Boolean);
end;

{ THTTPApplication }

constructor THTTPApplication.Create(aOwner: TComponent);
begin
 inherited Create(aOwner);
 OnAllowConnect:=@DoConnect;
end;

procedure THTTPApplication.DoConnect(Sender: TObject; ASocket: Longint; var Allow: Boolean);
{$IFDEF UNIX}
const
 FD_CLOEXEC = 1;
{$ENDIF}
begin
 {$IFDEF UNIX}
 FpFcntl(aSocket, F_SETFD, FD_CLOEXEC);
 {$ENDIF}
 Allow:=True;
end;
```

Lastly, to serve the files of the client program, we set the base directory for the file serving module to the directory with the client program files:

```
Function GetBaseDir : String;

begin
 Result:=ExtractFilePath(ParamStr(0));
 Result:=Result+'..'+PathDelim+'client';
 Result:=ExpandFileName(Result);
end;
```

(this code assumes there are 2 directories: one for the server, one for the client.)

Finally, we load all known mime types, and create our own HTTP application:

```
Var
 Application:THTTPApplication;

begin
 MimeTypes.LoadKnownTypes;
```

```
 TSimpleFileModule.BaseDir:=GetBaseDir;
 TSimpleFileModule.RegisterDefaultRoute;
 Application:=THTTPApplication.Create(Nil);
 Application.Title:='Process server';
 Application.Port:=8060;
 Application.Initialize;
 Application.Run;
 Application.Free;
end;
```

Note that we set the HTTP port to port 8060.

6 The browser client-side component

In the browser the `TProcessCapturePoller` component is used to help working with the `TProcessCapture` component on the server. It does not start the actual process, it just takes care of polling the server for the status of the started process, and triggers a series of events based on results. It also handles the state of the output offset parameter. There are properties to control how often and how long the polling mechanism must try, and how many errors can be tolerated before the polling is abandoned.

To be agnostic of the actual RPC mechanism used, the actual poll is also achieved using an event. It is the responsability of the programmer to implement this event, and to use the `ReportProgress` mechanism to comunicate the server results to the component.

This component has the following declaration:

```
Type
  TProcessStatus = (psRunning, // Process still running
                    psExited,  // Process has exited
                    psError    // Too many errors
                    );

  TOnGetProcessStatusEvent = Procedure (Sender : TObject; aProcessID : String; aOffset : Nati
  TOnProcessDoneEvent = Procedure (Sender : TObject; aStatus : TProcessStatus; aExitCode : In
  TOnProcessOutputEvent = Procedure (Sender : TObject; aOutput : String) of object;
  TOnStatusFailEvent = Procedure (Sender : TObject; aError : String) of object;

TProcessCapturePoller = class(TComponent)
Public
  Procedure Start;
  Procedure Cancel;
  Procedure ReportProgress(aStatus : TProcessStatus;
                           aOutput : String;
                           aExitCode : Integer;
                           aOffSet : NativeInt);
  Procedure ReportProgressFail(const aMessage : string);
  Property Canceled : Boolean ;
  Property FailCount : Integer;
  Property StatusCheckCount : Integer;
  Property OutputOffset : NativeInt;
```

```
Published
  Property ProcessID : String;
  Property OnGetProcessStatus : TOnGetProcessStatusEvent;
  Property OnProcessDone : TOnProcessDoneEvent;
  Property OnProcessOutput : TOnProcessOutputEvent;
  Property OnStatusFail : TOnStatusFailEvent;
  Property LinebasedOutput : Boolean;
  Property PollInterval : Integer;
  Property MaxFailCount : Integer;
  Property MaxCheckCount : Integer;
end;
```

The methods perform the following tasks

Start this starts the polling process.

Cancel this cancels the polling process.

ReportProgress This method must be used when the `OnGetProcessStatus` event handler received the status of the process from the server. The `aStatus` parameter is one of the available statuses, `aOutput` is the output of the process. Parameter `aExitCode` is the exit code (in case status is `psExited`) and `aOffset` is the new offset (as reported by the server).

ReportProgressFail this method must be used when the server call to get the process status failed. The `aMessage` status parameter can be used to indicate what exactly failed.

The following events can be handled:

OnGetProcessStatus This is the only event that must be implemented. It is triggered at regular intervals, when the poller needs to inquire the status of the server process. The poller will pass the process ID and current output offset to the event, so the user does not need to track the state of these parameters.

OnProcessDone This is called when the process has exited or the polling was cancelled. It reports the status (psError in case of error) and the exit code of the process.

OnProcessOutput This is called when output of the process was received: The `aOutput` parameter contains the reported output. This event will be called multiple times.

OnStatusFail This is called when the `ReportProgressFail` was called to signal a failure of the call to get the status of the process. It can be called multiple times, depending on the value of `MaxFailCount`.

The behaviour of the component is controlled by the following properties:

LinebasedOutput If set to `True` the component will split the received output in lines, and will call `OnProcessOutput` for each line instead of reporting the whole received output in one call (if set to `False`)

PollInterval the time period (in milliseconds) after which `OnGetProcessStatus` event is triggered. Default is 500ms. Note that the event is only retriggered after the result (success or failure) of the previous event has been reported. This is done in order to avoid overlapping getstatus calls.

MaxFailCount The maximum number of failures that may be reported before polling is abandoned. Default is 1.

MaxCheckCount The maximum number of times the component will poll before reporting a timeout.

Finally, the following properties can be used to get some information about the polling process:

Canceled The polling process was canceled.

FailCount The number of failures since the polling was started.

StatusCheckCount The number of times the status will still be checked.

OutputOffset The current output offset.

It may seem strange to have the `OnProcessDone`, `OnStatusFail` and `OnProcessOutput` events if the fetching of the process status must be implemented in an event: surely the event handler can display the output, decide when the process has ended etc. The reason is twofold: first of all, the state logic for the output can be handled by the component, but more importantly: by having these events available, the component can easily be used as a parent for descendents that incorporate the polling RPC mechanism in the component. (as will be demonstrated below).

7 The client program

Armed with this component, we can now start the client side program. In the 'Project - New project' dialog we select the 'Web browser program' item, and enter the correct settings, as shown in figure 4 on page 13. The html file is best saved as `index.html`.

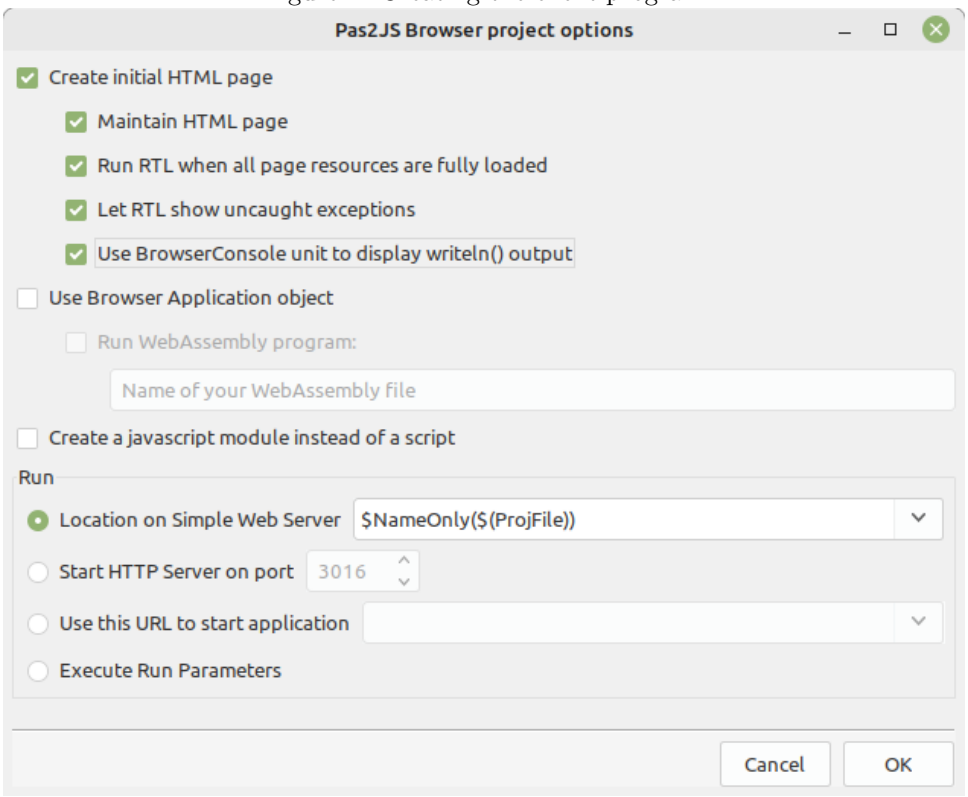
The HTML needs 5 elements:

1. A button to start the process.
2. A button to cancel the polling process.
3. An edit for parameter A for the started program.
4. An edit for parameter B for the started program.
5. An HTML element in which the output of the program will be shown. We will use the `browserconsole` unit output mechanism for this: a simple `Writeln` statement will result in the appending of the output to this element.

The following simple HTML (using Bulma CSS) will do the job just fine:

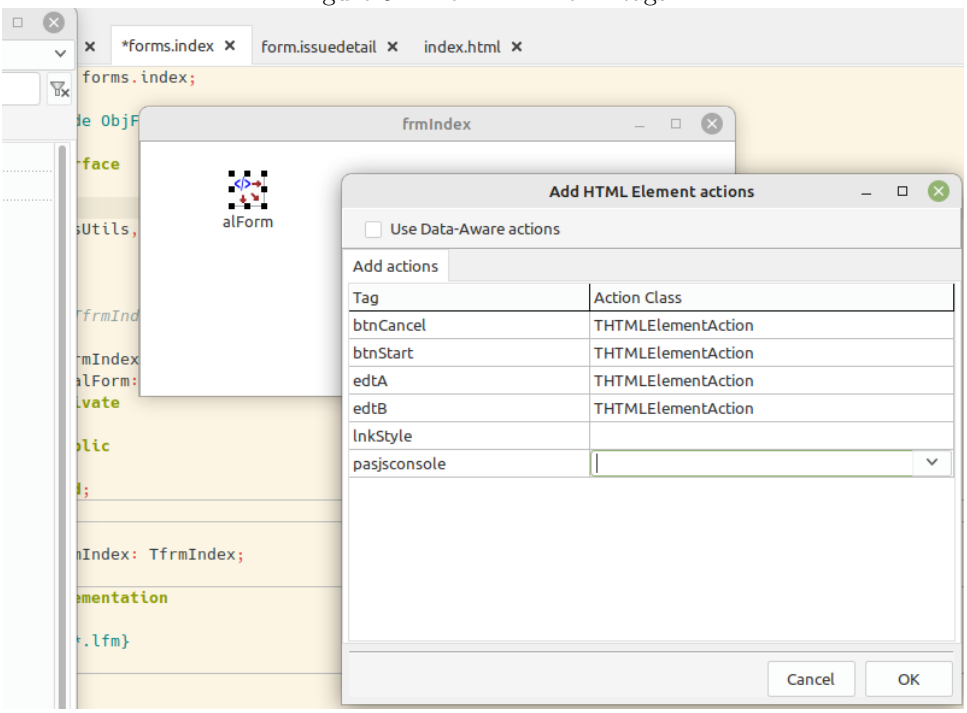
```
<h3 class="title is-3">Process output demo</h3>
<div class="box">
  <h4 class="title is-4">Start parameters</h4>
  <div class="field">
    <label class="label">Argument A</label>
    <div class="control">
      <input id="edtA" type="text" class="input"
             placeholder="Enter argument A">
```

Figure 4: Creating the client program




```
    </div>
  </div>
  <div class="field">
    <label class="label">Argument B</label>
    <div class="control">
      <input id="edtB" type="text" class="input"
        placeholder="Enter argument B">
    </div>
  </div>
  <div class="field is-grouped">
    <div class="control">
      <button id="btnStart" class="button is-primary">
        Start process
      </button>
    </div>
    <div class="control">
      <button id="btnCancel" class="button is-warning is-light">
        Cancel
      </button>
    </div>
  </div>
</div>
<div class="box">
  <h4 class="title is-4">Process output</h4>
  <div id="pasjsconsole">
```

Figure 5: The HTML form tags



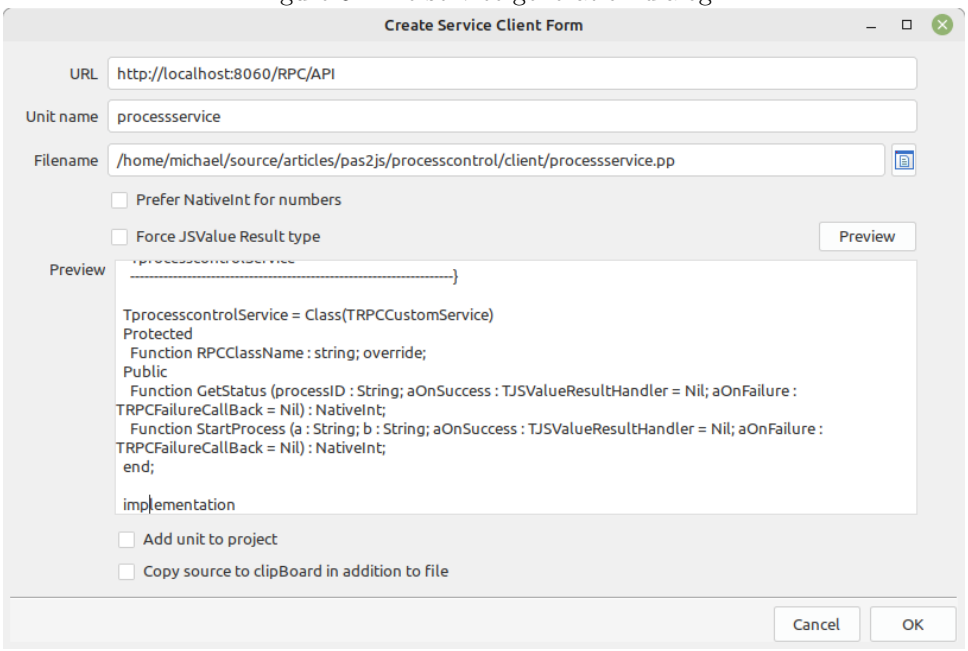

```
  </div>
</div>
```

To interact with this HTML, we first create a `HTML Fragment module` using the 'File - new' dialog. We name it 'frmIndex' and set the 'UseProjectHTML' property to true. On this module, we drop a `THTMLElementActionList` component from the component palette. Using the component context menu 'Create actions for HTML tags', we can create actions for all tags in the above HTML, as shown in figure 5 on page 14. We need a `TPas2jsRPCClient` from the `Pas2JS` tab in the component palette: this component will handle the RPC requests, and we'll name it `RPC` for short. The component can only do its work correctly if it knows where the server is: We need to enter the `URL` property. As shown in an earlier article, we can now generate a service proxy: this is a class which has correct method definitions, reflecting the methods defined in our RPC server. Calling these service methods will actually execute the methods on the server. Right-clicking on the `RPC` component and selecting 'Create Service Client component' shows the service generation dialog as shown in figure 6 on page 15. We name the unit 'processservice' and tell the IDE to add it to the project.

Now we can start coding the application. We will create the `TProcessCapturePoller` and service client in the `OnCreate` event of our index form module:

```
procedure TfrmIndex.DataModuleCreate(Sender: TObject);
begin
  Service:=TprocesscontrolService.Create(Self);
  Service.RPCClient:=RPC;
  FPoller:=TProcessCapturePoller.Create(Self);
  FPoller.OnProcessOutput:=@DoDoutput;
  FPoller.OnGetProcessStatus:=@DoGetStatus;
```

Figure 6: The service generation dialog




```
  FPoller.OnProcessDone:=@DoProcessDone;
  FPoller.OnStatusFail:=@DoStatusFail;
end;
```

Note that we assign the `RPC` client to our service definition, and that we assign events to all event handlers of the poller component.

To start the process, we add an `OnClick` event handler to the `actbtnStart` action. In it, we collect the values for the `A` and `B` parameters from the respective input boxes, and use these to call `StartProcess` on our `Service` component.

We take care to handle the `OnSuccess` and `OnFail` handlers of this method - remember, the calls to the server are asynchronous:

```
procedure TfrmIndex.actbtnStartExecute(Sender: TObject; Event: TJSEvent);

  procedure DoStartFail(Sender: TObject; const aError: TRPCError);
  begin
    Writeln('Failed to start process : ',aError.Message);
  end;

  procedure DoStartOK(aResult: JSValue);

  begin
    FJobID:=String(aResult);
    FPoller.ProcessID:=FJobID;
    FPoller.Start;
  end;

var
  a,b : string;
```

```
begin
  a:=actedtA.Value;
  b:=actedtB.Value;
  Service.StartProcess(A,B,@DoStartOK,@DoStartFail);
end;
```

If the start call fails, we simply log the fact. If the start call succeeds, we record the result (a process ID) in the poller `ProcessID` property and start the poller.

The onclick handler for the 'Cancel' button is much simpler: We just need to cancel the poller.

```
  procedure TfrmIndex.actbtnCancelExecute(Sender: TObject; Event: TJSEvent);
begin
  Writeln('Canceled wait for process.');
  FPoller.Cancel;
end;
```

All that remains to do is to handle the 4 events of the `TProcessCapturePoller` component.

We'll start with the simple ones, the `OnProcessOutput` and `OnStatusFail` events. In it, we just need to output the messages that are passed to the event handler:

```
 procedure TfrmIndex.DoStatusFail(Sender: TObject; aError: String);
begin
  Writeln('Error getting status: ',aError);
end;

procedure TfrmIndex.DoDoutput(Sender: TObject; aOutput: String);
begin
  Writeln(aOutput);
end;
```

The `OnProcessDone` event handler is equally simple, we print the status and exit code (if there is one)

```
procedure TfrmIndex.DoProcessDone(Sender: TObject;
                                  aStatus: TProcessStatus;
                                  aExitCode: Integer);

Const
  Exits : Array[TProcessStatus] of string
        = ('Running','Exited','Error');

begin
  Write('Process ',Exits[aStatus]);
  if aStatus=psExited then
    Writeln(' with exit code ',aExitCode)
  else
    Writeln();
end;
```

Last but not least, we must handle the `OnGetProcessStatus` event. This simply calls the `GetStatus` procedure from our service component, and handles the result handlers: in each handler the appropriate method of the `TProcessCapturePoller` component is called with the received result:

```
  procedure TfrmIndex.DoGetStatus(Sender: TObject;
                                  aProcessID: String;
                                  aOffset: NativeInt);


  procedure DoStatusFail(Sender: TObject; const aError: TRPCError);
  begin
    FPoller.ReportProgressFail(aError.Message);
  end;

  procedure DoStatusOK(aResult: JSValue);

  const statuses : array[Boolean] of TProcessStatus
                 =  (psError,psRunning);

  Var
    D : TJSObject absolute aResult;
    aExitCode : Integer;
    aNewOffset : NativeInt;
    aOutput : string;
    aStatus : TProcessStatus;

  begin
    aOutput:=String(D['output']);
    aExitCode:=NativeInt(D['status']);
    aNewOffset:=NativeInt(D['offset']);
    aStatus:=Statuses[aExitCode=-1];
    FPoller.ReportProgress(aStatus,aOutput,aExitCode,aNewOffset)
  end;

begin
  Service.GetStatus(FJobID,aOffset,@DoStatusOK,@DoStatusFail);
end;
```

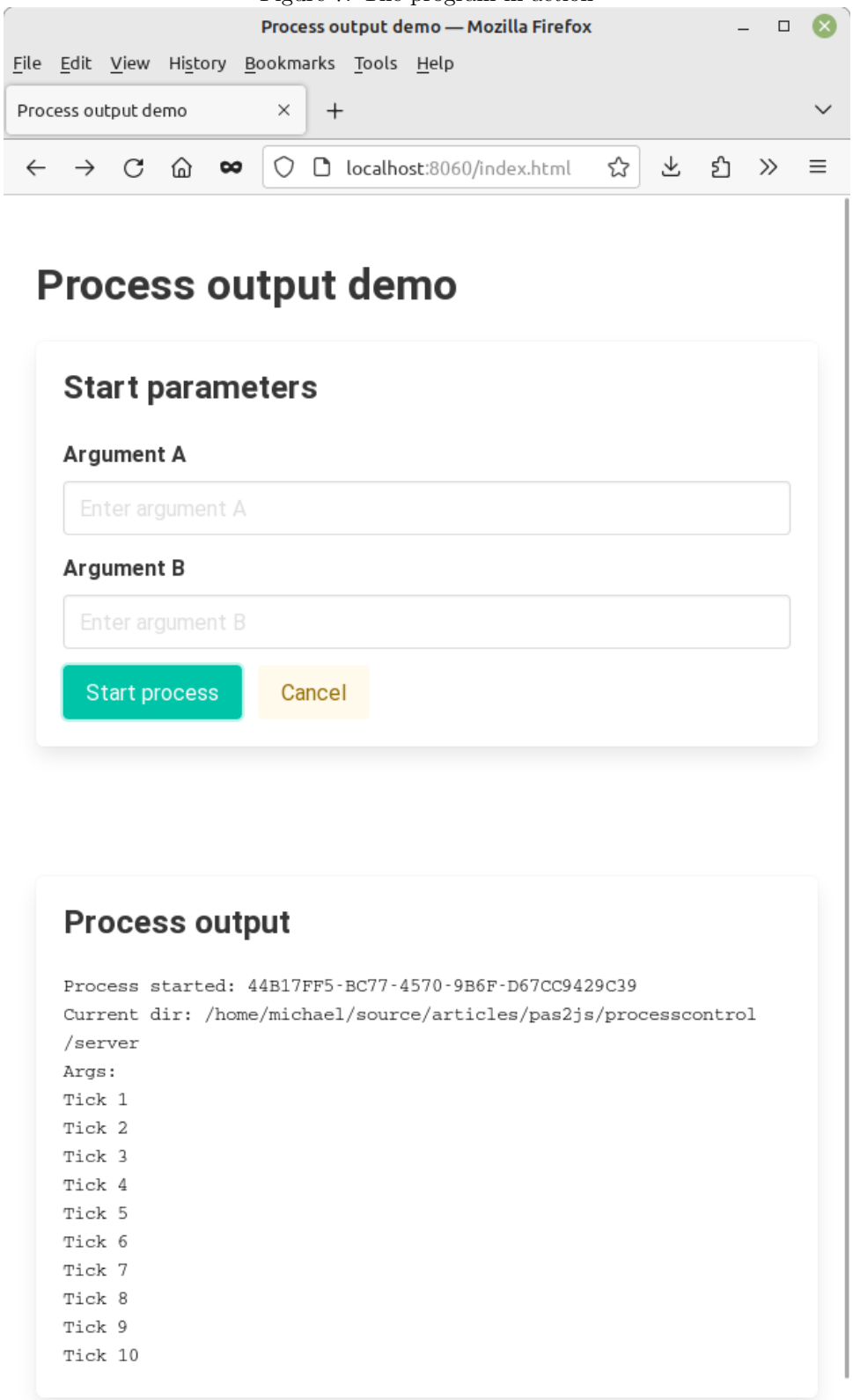
With this, the logic of our application is ready. Remains to write the main program routine, which is very short indeed: All we need to do is create our module and call `Show`:

```
var
  frm : TfrmIndex;

begin
  MaxConsoleLines:=15;
  frm:=TfrmIndex.Create(Nil);
  Frm.Show;
```

Setting the `MaxConsoleLines` to 15 will make sure you can see the messages scroll over the screen as the output of the server process comes in. The result of this code is shown in figure 7 on page 18.

Figure 7: The program in action



8 Creating a server process execution component

Earlier in the article we mentioned that it could seem strange that there are events to report status and output when the actual call to get the status is executed in the form module: at that point you will already know the status, so why still report it to the component ?

Part of the answer is that what we have shown above is just one way to use the component. A second way is that you can also create a descendent of this component which handles the getting of the status all by itself. In that case, the events are the only way to get notifications of the status of the process. In the following we show how to make such a descendent.

The `TProcessCapturePoller` component is actually a simple descendent of the `TCustomProcessCapturePoller` component, which simply implements the method to get the status of the process using an event.

What we can do is create a descendent of the `TCustomProcessCapturePoller` component which has the `TprocesscontrolService` class built-in. This component will know all by itself how to execute a process on the server. This component would look as follows:

```
TRemoteExecutor = class(TCustomProcessCapturePoller)
Protected
  procedure DoStatusCheck; override;
Public
  Procedure Execute(a,b : String);
Published
  Property RPCClient : TRPCClient Read GetClient Write SetClient;
  Property OnProcessDone;
  Property OnProcessOutput;
  Property OnStatusFail;
  Property LinebasedOutput;
  Property PollInterval;
  Property MaxFailCount;
  Property MaxCheckCount;
end;
```

We left out the constructor and destructor, which simply create and destroy the `TprocesscontrolService`.

```
constructor TRemoteExecutor.Create(aOwner: TComponent);
begin
  inherited Create(aOwner);
  FService:=TprocesscontrolService.Create(Self);
end;
destructor TRemoteExecutor.Destroy;
begin
  FreeAndNil(FService);
  inherited Destroy;
end;
```

The `Service` field is used to get and set the `RPCClient` property:

```
function TRemoteExecutor.GetClient: TRPCClient;
begin
```

```
  Result:=FService.RPCClient;
end;

procedure TRemoteExecutor.SetClient(AValue: TRPCClient);
begin
  FService.RPCClient:=aValue;
end;
```

The Execute method takes the correct parameters, and in essence does what was done in the form in our original code:

```
  procedure TRemoteExecutor.Execute(a, b: String);

  procedure DoStartFail(Sender: TObject; const aError: TRPCError);
  begin
    SetFailCount(MaxFailCount);
    ReportProgressFail(aError.Message);
  end;

  procedure DoStartOK(aResult: JSValue);

  begin
    ProcessID:=String(aResult);
    Start;
  end;

begin
  Service.StartProcess(A,B,@DoStartOK,@DoStartFail);
end;
```

Note that if the process failed to start, the fail count is set to the maximum, this will cause the `ReportProgressFail` method not to schedule a new check.

The `DoStatusCheck` method contains simply the code that was present in the form in our first implementation:

```
procedure TRemoteExecutor.DoStatusCheck;

  procedure DoStatusFail(Sender: TObject; const aError: TRPCError);
  begin
    ReportProgressFail(aError.Message);
  end;

  procedure DoStatusOK(aResult: JSValue);

  const statuses : array[Boolean] of TProcessStatus
                 =  (psError,psRunning);

  Var
    D : TJSObject absolute aResult;
    aExitCode : Integer;
    aNewOffset : NativeInt;
    aOutput : string;
    aStatus : TProcessStatus;
```

```
  begin
    aOutput:=String(D['output']);
    aExitCode:=NativeInt(D['status']);
    aNewOffset:=NativeInt(D['offset']);
    aStatus:=Statuses[aExitCode=-1];
    DoReportProgress(aStatus,aOutput,aExitCode,aNewOffset)
  end;

begin
  service.GetStatus(ProcessID,OutputOffset,@DoStatusOK,@DoStatusFail);
end;
```

The form code is now much simpler. We only need to create the `TRemoteExecutor` component, and set its 3 events:

```
procedure TfrmIndex.DataModuleCreate(Sender: TObject);
begin
  FRemote:=TRemoteExecutor.Create(Self);
  FRemote.OnProcessOutput:=@DoDoutput;
  FRemote.OnProcessDone:=@DoProcessDone;
  FRemote.OnStatusFail:=@DoStatusFail;
end;
```

The event handler for the 'Start' button is now a simple one-liner:

```
procedure TfrmIndex.actbtnStartExecute(Sender: TObject; Event: TJSEvent);

begin
  FRemote.Execute(actedtA.Value,actedtB.Value);
end;
```

The event handler to get the status is no longer needed.

The functional working of the program is not different, but if you have a lot of locations in your program where you need to execute programs on the server, it makes sense to abstract away the remote execution in this manner.

9 Conclusion

In this article we've shown that executing programs on a HTTP Server from a Pas2JS program does not need to be difficult. The component to automate the process is independent of a RPC mechanism, and as such can be used as-is, or it can be used as the parent for a more elaborate component which handles all communication by itself.