Creating Atom and VS Code plugins in Pascal

Michaël Van Canneyt

January 18, 2021

Abstract

The Atom and VS Code editors are among the most popular programmer editors. These editor are extensible for anyone that can create Javascript. Object Pascal programmers can also create Javascript, so logically they can also create VS Code and Atom extensions. In this article we show how.

1 Introduction

Web applications are cross-platform. Any platform that has a browser can run a web application. Less known is that you can run web applications on a desktop: Electron is an environment that uses the browser and Node.js to allow you to create desktop applications that are written in Javascript, and which run in a browser - sort of. It uses the chromium engine to render HTML - the HTML is the GUI of the application. The application logic is written in Javascript.

With the appearance of Electron, 2 powerful programming editors were created that use Electron: Atom and VS Code. Since they are built with Electron, they are cross-platform. And because Electron uses a Javascript engine (the same as Node.js), it is possible to load and execute arbitrary Javascript code in the editor. This Electron ability is used to allow people to extend the editor: you can extend the editor in whatever way you see fit. The only requirement is that the extension is written in Javascript.

Naturally, there are numerous plugins for both editors: in Atom they are called Packages, in VS Code, they are called Extensions. Both editors have plugins to facilitate programming in object pascal.

But can you also write pascal to extend these editors ? Fortunately, now you can. Pas2JS and the TMS Web Compile convert Object Pascal to Javascript. Javascript is a simple text file, and it should be possible to include the output of the pas2js or TMS Web Compiler in the editor in a format it accepts.

Note that the Lazarus IDE support requires the trunk version of the pas2JS compiler: it uses a `-ja` option to append a small piece of Javascript to the output. If you use a released version of the compiler, you can append this little piece of code manually.

2 Editor API

The Atom and VS Code editors are themselves written in Javascript. Because they are well-designed, they make an API available to any Javascript programmer that wishes to extend the editor. How can we make this API available to the pascal programmer ? In the exact same manner as the Browser API was made available to the Pascal programmer: by

writing external class definitions that 'translate' the Javascript APIs of the editors for the Object Pascal compiler.

This task has been accomplished: the APIs of both editors has been translated to Pascal: the units that contain these definitions have been created and made available in the subversion repository. The units with the editor APIs are called libatom and libvscode.

The APIs made available to you by these editors are huge. They contain hundreds of classes. They match the browser APIs for size, so needless to say that an in-depth study of these APIs is outside the scope of an article.

Although they offer the same kind of functionalities, the APIs of both editors are wildly different: code designed to run in 1 editor will not run in the other. Conceivably, a kind of unifying API can be made on top of these APIS, so as to allow a programmer to create a plugin that works for both editors;

3 Plugin architecture

Both editors have more or less the same architecture for a plugin: a plugin is similar to a library: it is a javascript file that must expose a number of functions, plus a manifest file describing the plugin.

The manifest file is a JSON file such as it is found in many Node.js packages: package.json, which must have some entries for the Editor to be able to load your plugin: the location of the Javascript file (a module) with the plugin code.

In the case of VS Code the javascript code must export 2 procedures:

activate This function is called when the plugin is loaded: In this function, you must install the necessary commands, hooks and keyboard shortcuts. For this purpose, the editor passes a context object as the sole argument to the procedure. The context contains an instance of the global editor object.

deactivate This function is called when the plugin is unloaded.

In the case of Atom, the javascript code must export the above 2 functions (although they have different arguments), and can export 2 additional procedures:

initialize This function is called exactly once before activating the package.

serialize This function is called when the package is being unloaded, so it can preserve state. The saved state is passed to the package `activate` function when the editor loads it.

The module concept of javascript translates almost directly to Libraries. Support for libraries is currently being implemented, but is not yet in the current release.

Because the pas2js compiler (2.0) does not yet support compiling libraries, it is easiest to use a little Javascript wrapper that will load the code generated by Pas2js. This little Javascript wrapper will start the pas2js rtl, and will call a function defined in the main program.

For Atom, this wrapper looks like this:

```
'use babel';

import { CompositeDisposable } from 'atom';
import { pas, rtl } from './pas2jsdemopackage.js';
```

```
export default {
  activate(state) {
    rtl.run();
    this.subscriptions = new CompositeDisposable();
    this.atomEnv = {
      atomGlobal : atom,
      subscriptions : this.subscriptions,
      initialState : state
    }
    this.atomHandler = {
      onDeactivate : function (a) {},
      onSerialize : function (a,o) {}
    }
    pas.program.InitAtom(this.atomEnv,this.atomHandler);
  },

  deactivate() {
    if (this.atomHandler.onDeactivate) {
      this.atomHandler.onDeactivate(this.atomEnv)
    }
    this.subscriptions.dispose();
  },

  serialize() {
    var obj = {};
    if (this.atomHandler.onSerialize) {
       this.atomHandler.onSerialize(this.atomEnv,obj)
    }
    return obj;
  }
};
```

You don't need to be a Javascript specialist to understand that this code exports the 3 functions mentioned abov. What is important for the pascal programmer, are 3 lines of code.

The first important line imports the symbols that can be found in any pas2js generated program:

```
import { pas, rtl } from './pas2jsdemopackage.js';
```

The `pas` object contains the code from all the units and the main program. The `rtl` object contains the pascal run-time code.

The second important line of code initializes the pascal runtime:

```
rtl.run();
```

This is the same code that can be found in a HTML page <script> tag to start a pascal-generated program.

The last piece of code transfers control to the `InitAtom` function in the pascal main program:

```
pas.program.InitAtom(this.atomEnv,this.atomHandler);
```

This last line is a design choice, a convention to handle the transfer of code to pascal; it is only necessary for the time being, because pas2js does not yet support libraries. When pas2js will support libraries, the above wrapper will of course no longer be necessary.

Note that a set of callbacks is passed to the `InitAtom` call. This is done so only 1 function needs to be exposed: when the `InitAtom` function returns, the handlers in the `atomHandler` object will be set and can be used to transfer control to the pascal code in the other 2 exposed functions.

The VS Code wrapper is entirely similar, except the name of the Pascal initialization function:

```
const vscode = require('vscode');
const pascalRuntime = require('./pas2jsdemoextension.js');

var callbacks = {
    onDeactivate: function (a) { }
}

function activate(context) {
    pascalRuntime.rtl.run();
    var vscodeEnv = {
        vscodeGlobal: vscode,
        extensionContext: context
    }
  pascalRuntime.pas.program.InitVSCode(vscodeEnv,callbacks);
}

function deactivate() {
  if (callbacks.onDeactivate) {
    callbacks.onDeactivate();
  }
}

module.exports = {
  activate,
  deactivate
}
```

4 Object Pascal Application

In the above, we've seen the Javascript code that will be used to kickstart the pascal code for our plugin.

To make it easier for you to create an object pascal program that can be used as a VS Code or Atom plugin, a unit was created as part of the pas2js package, which contains an 'Application' object.

The application object is much like the Delphi `TApplication` class, as it descends from `TCustomApplication` - a standard class in the Free Pascal runtime, which serves as the ancestor for all kinds of application classes - native or browser-based: console applications, GUI applications (in Lazarus) and Node.js or Browser based applications in pas2js.

Because the VS Code and Atom APIs are different, the application objects for both environments are of course also different. So the Pas2js distribution now has 2 units called atomapp and vscodeapp.

They each define an application object for use in the Atom and VS Code editor: Each object has a property that contains an instance of the global VSCode and Atom editor environment: these objects are made available by the editors: we'll see how to use this later on.

```
TAtomApplication = class(TCustomApplication)
 Protected
   procedure DoActivate(aState : TJSObject); virtual;
   procedure DoDeactivate(); virtual;
   procedure DoSerialize(aState : TJSObject); virtual;
 Public
   procedure SaveAtomEnvironment(aEnv : TAtomEnvironment;
                                 aCallBacks : TAtomPackageCallBacks);
   property Subscriptions : TAtomCompositeDisposable;
   Property Atom : TAtom;
end;
```

This is a simple application object (although some methods have been left out for clarity). It exposes 2 Javascript objects: `SubScriptions` and `Atom`, which will contain the Atom global editor object that exposes the complete Atom API for you, and a subscriptions object.

The subscriptions object is an Atom object that owns all resources you will allocate during the lifetime of your plugin. When the plugin is freed, the subscription will free all objects it owns.

To create your Atom plugin, the package program code must define a descendent of this class, and override the 3 protected virtual methods:

```
  TMyAtomApplication = Class(TAtomApplication)
  Protected
    procedure DoActivate(aState : TJSObject); override;
    procedure DoDeactivate; override;
    procedure DoSerialize(aState : TJSObject); override;
  end;
```

These methods obviously correspond to the 3 exported functions of the Javascript wrapper and will be called when the plugin is loaded and unloaded.

The Javascript wrapper code calls a function `InitAtom` (this method name is case sensitive). In this function, you can instantiate the atom application class, and call `SaveAtomEnvironment` to save the atom object and set the callbacks needed by the wrapper:

```
Procedure InitAtom(aAtom : TAtomEnvironment;
                   aCallBacks : TAtomPackageCallBacks);

begin
  If Application=Nil then
    Application:=TMyAtomApplication.Create(Nil);
  Application.SaveAtomEnvironment(aAtom,aCallBacks);
end;
```

This will also call the application object's `DoActivate` method.

The VS Code application object looks very similar:

```
TVSCodeApplication = class(TCustomApplication)
```

```
Protected
  procedure DoActivate; virtual;
  procedure DoDeactivate(); virtual;
Public
  procedure SaveVSCodeEnvironment(aEnv : TVSCodeEnvironment;
                                  aCallBacks : TVSCodeExtensionCallBacks);
  Property VSCode : TVSCode;
  Property ExtensionContext : TVSExtensionContext;
end;
```

Likewise, a VS Code extension program must contain a descendent of this class, and it must instantiate it in the `InitVSCode` function that is called by the Javascript wrapper.

```
Procedure InitVSCode(aVSCode : TVSCodeEnvironment;
                     aCallBacks : TVSCodeExtensionCallBacks);

begin
  If Application=Nil then
    Application:=TMyVSCodeExtension.Create(Nil);
  Application.SaveVSCodeEnvironment(aVSCode,aCallBacks);
end;
```

Again, the `DoActivate` method of the application class will be called by this process.

In the `DocActivate` method, you must place all code that will hook into the editor API: add commands, keyboard shortcuts etc.

5 Lazarus integration

The above code is the start of a VS Code or Atom plugin: It is possible to create an Atom or VS Code plugin using Atom or VS Code themselves to edit the pascal code, and use the above as a starting point: in that case you must manually add the wrapper code, package.json program file etc. to your project.

But at the moment of writing, the Lazarus code editor is still much more suitable for writing Object Pascal than VS Code or Atom: The Lazarus code tools provide much more possibilities than either of these general-purpose editors do.

That is why the Lazarus Pas2Js support has been extended with 2 project types to create an Atom or VS Code plugin. These wizards will create a skeleton project for you, which can be compiled using pas2js and which is ready to be installed in the Atom or VS Code editor.

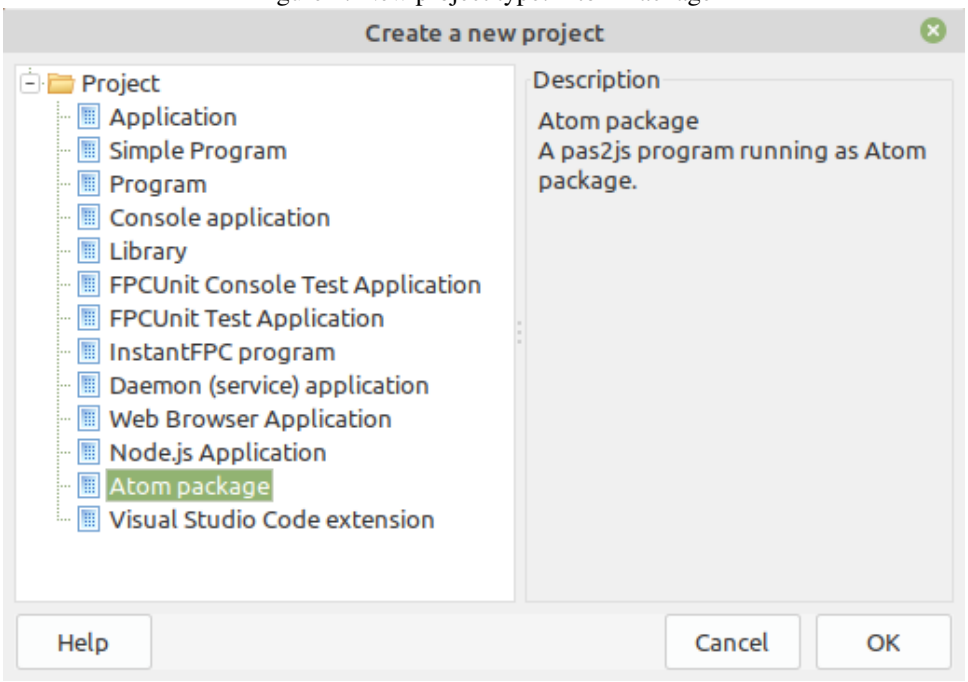
We'll demonstrate the use of these wizards in the following sections.

6 A sample Atom package

As a Pascal and SQL programmer, the author of this article is used to not to have to care about the case of keywords and identifiers. However, not everyone shares this view. In Delphi and Lazarus, the code formatter can be used to remedy this sloppiness: the IDE code formatter will happily correct casing for you.

Unfortunately, this disregard for casing extends to project documentation, SQL statements, resulting in documentation with for example SQL keywords written in a wide variety of casings.

Figure 1: New project type: Atom Package



The author writes documentation primarily using Markdown, in Atom. So, since we can now write plugins in Pascal, a good first attempt at a plugin is a plugin to fix casing of some SQL keywords: we uppercase a selection of SQL keywords, thus having a more unified look for all SQL.

Before starting to code this, it is a good idea to look at the APIs made available to you in the Atom flight manual:

`https://flight-manual.atom.io/api/v1.54.0/AtomEnvironment/`

The Atom flight manual also contains a good anatomy of an Atom package, it describes all files that Atom expects to find.

To get started, we invoke the Lazarus wizard to create an Atom plugin, as shown in figure figure 1 on page 7.

When selected, a small dialog appears to set some options, as shown in figure 2 on page 8: The various items that can be entered here serve to generate a skeleton project:

Directory Every Atom plugin lives in its own directory. Here you specify the directory for the new plugin.

Description A textual description of your package, it goes in the manifest file (package.json).

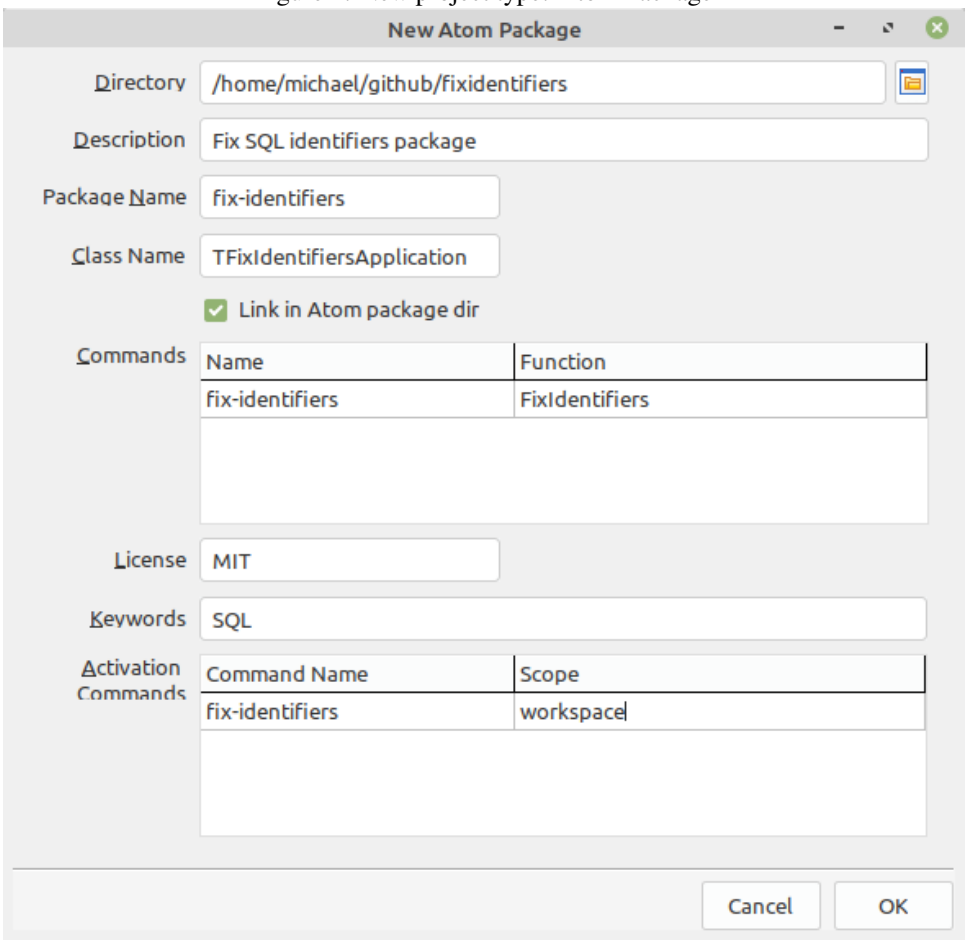
Package Name A (unique) name for your package, it goes in the manifest file.

Class Name The Pascal class name for the application class.

Link in Atom package dir When you check this flag, the IDE will create (on unix and MacOS) a link in the Atom package directory to the directory where you create your project. When you next start Atom, it will then load your plugin.

Commands The commands that your package will provide to the editor. For each command you must specify a unique name, and the name of a pascal function that will

Figure 2: New project type: Atom Package



be called when the command is invoked. The command names are entered in the created menu.json file but also in the pascal code, to register the callback for the command. An empty function will be generated for each function you specify here.

License The license for your package, it goes in the manifest file.

Keywords Some keywords (space separated) for your package, it goes in the manifest file.

Activation Commands The commands that will cause your package to be loaded by the editor. The scope is a valid Atom scope identifier such as `atom-workspace`. You can leave this list empty.

Once you confirm your choices, the IDE will create a project with several files (see figure 3 on page 10):

fix_identifiers.lpr The program with generated code: it can be compiled.

menu.json Atom menu entries: the menu entries here will appear in Atom under the 'Packages' menu.

package.json The package manifest file.

keymaps.json The keymaps offered by your package: It is necessary to edit the generated file, and assign a unique key combinations to each command.

package.less CSS for your package if your code needs it. This will be loaded into the editor.

packageglue.js This file contains the Atom package wrapper Javascript code shown above. You may edit this to your liking. If you change the name (or output file) of the Pascal project, you must manually change the name of the imported project file here.

The most interesting is of course the program code. The class declaration is much as we expected:

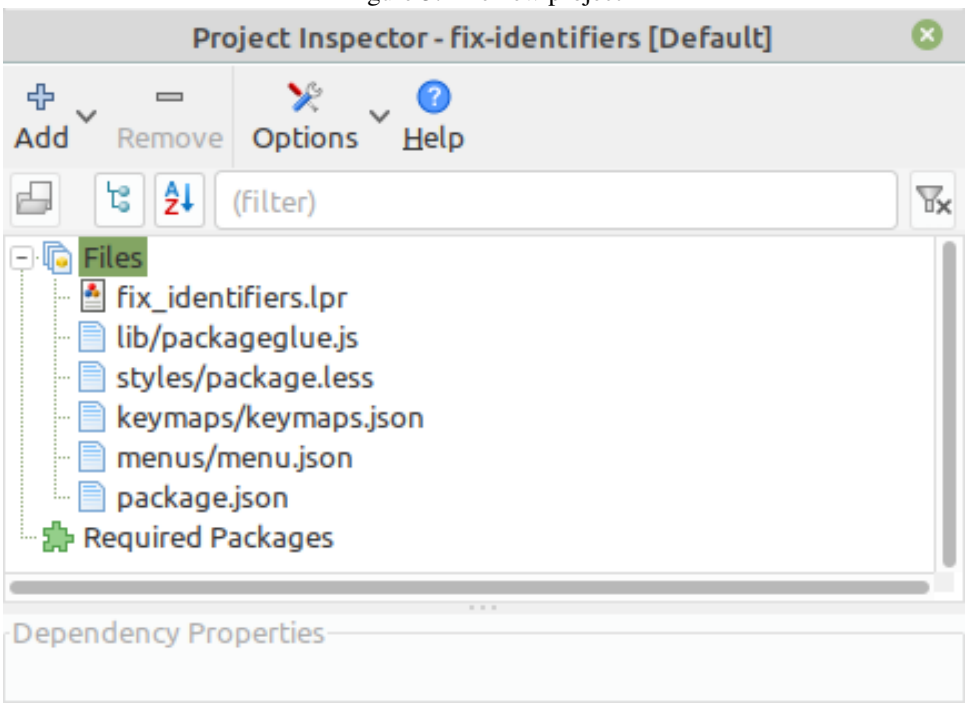
```
TAtomFixIdentifiersApplication = Class(TAtomApplication)
Protected
  procedure DoActivate(aState : TJSObject); override;
  procedure DoDeactivate; override;
  procedure DoSerialize(aState : TJSObject); override;
Public
  Procedure FixIdentifiers;
end;
```

The interesting function is of course `DoActivate`, this is where we start the ball rolling. The new project wizard has already filled it with code:

```
procedure TAtomFixIdentifiersApplication.DoActivate(aState: TJSObject);

Var
  cmds : TJSObject;
begin
  inherited DoActivate(aState);
  cmds:=TJSObject.New;
  cmds['fix-identifiers:activate']:=@FixIdentifiers;
  subscriptions.add(Atom.Commands.Add('atom-workspace', cmds));
end;
```

Figure 3: The new project



The `Atom.Commands` object controls all the commands of the Atom editor. It has a method `Add` which needs a scope, and a Javascript object which has a set of properties: each property has the name of a command, and the property value is the function that must be called when the command is activated. In the code above, the command name is `fix-identifiers:activate` and the function is `FixIdentifiers`.

The result of the `Add` command is an Atom disposable: We add it to the `Subscriptions` so that when the package is unloaded, the disposable is freed.

An empty `FixIdentifiers` procedure has been generated by the wizard, we just need to fill it with code. To do what we want, we need to find a reference to the currently active editor, and the text buffer that it is actually editing. When we have the buffer, when can simply tell Atom to do a series of search and replaces for a set of words that we wish to correct the case for: The buffer API has a method for this.

The following is a possible implementation, with a limited list of keywords to replace:

```
Procedure TAtomFixIdentifiersApplication.FixIdentifiers;

Const
  ToUpperCase : Array of string
              = ('bigint','smallint','int','varchar',
                 'char','not null default','not null');
  SErrNoEditor = 'Cannot fix identifiers. No editor is active!';
  SErrNoBuffer = 'Cannot fix identifiers. No buffer available!';

Var
  Ed : TAtomTextEditor;
  Buf : TAtomTextBuffer;
  S : String;
  P : Integer;
```

```
begin
  Ed:=Atom.WorkSpace.getActiveTextEditor;
  if not Assigned(Ed) then
    begin
    Atom.notifications.addWarning(SErrNoEditor);
    Exit;
    end;
  Buf:=Ed.getBuffer;
  if not Assigned(Buf) then
    begin
    Atom.notifications.addWarning(SErrNoBuffer);
    Exit;
    end;
  For S in ToUpperCase do
    DoUpperCase(Buf,S);
end;
```

We first get a reference to the active text editor: The `WorkSpace` object of the Atom editor manages the editors, and the `getActiveTextEditor` method of this object returns the currently active editor. This can of course be empty, and we display a nice notification if this is the case.

Once we have the editor, we get the underlying text buffer with `getBuffer`: because multiple editors can be editing the same buffer at the same time, the underlying buffer is a separate object of the editor. Normally the buffer cannot be empty, but for safety's sake we check for this and display a message if no buffer is found.

Once the buffer is found, we loop through our list of identifiers we wish to uppercase, and call `DoUppercase`, passing it the buffer and the keyword we wish to uppercase.

The `TAtomTextBuffer` object has search (and replace) methods: `Scan`, `BackwardsScan` and `replace` which allow us to do what we want.

Unfortunately, the `replace` method does not allow to use placeholders in the replacement text, so we opt for the `BackwardsScan` method. Using the backwards scan instead of the forward scan, this avoids the danger that the search algorithm gets stuck in an infinite loop, because the replacement text will also match the search expression.

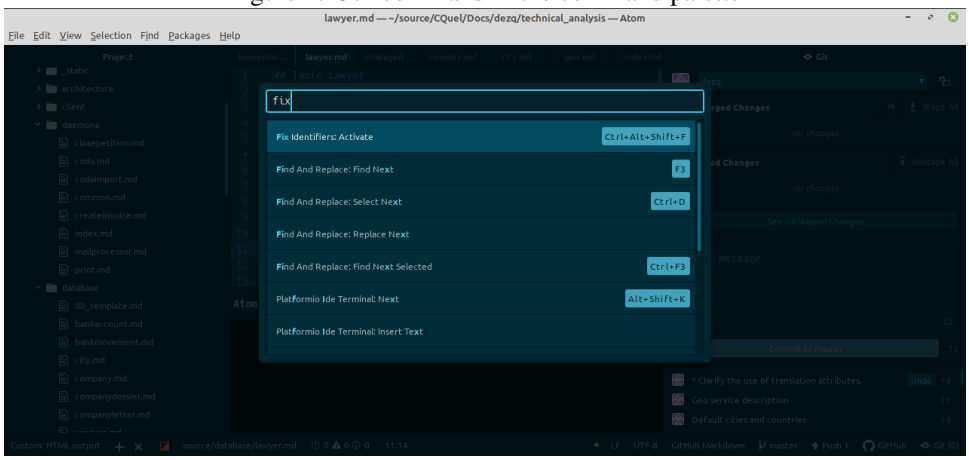
The routine starts with creating a regular expression that will only match whole-word forms of the keyword, and a lowercase and uppercase version of the search term.

```
Procedure TAtomFixIdentifiersApplication.DoUppercase(
          aBuf : TAtomTextBuffer;
          aWord : String);

Var
  S,ARegex,aLower,aUpper : String;
  P : Integer;

begin
  aRegex:='(^|\W*)'+aWord+'(\W|$)';
  aLower:=LowerCase(aWord);
  aUpper:=UpperCase(aWord);
  aBuf.BackwardsScan(TJSRegexp.New(aRegex,'ig'),
    procedure(aMatch : TAtomBufferScanMatch)
    begin
```

Figure 4: Our command in the command palette




```
    s:=aMatch.matchText;
    P:=Pos(aLower,LowerCase(S));
    S:=Copy(S,1,P-1)+aUpper+Copy(S,P+Length(aWord),Length(S)-P);
    aMatch.replace(S);
    end);
end;
```

Then it invokes the `BackwardsScan` option with a regular expression object (defined in the `JS` unit) and a callback: the callback is invoked for each matched item.

The callback receives an object that describes the match, and that contains a `replace` method to actually replace the found term in the text buffer: we must use it to replace the search term with the uppercase keyword, taking care that we match any non-letters before and after the keyword.

That's it. Our plugin is ready. All that is left to do is assign a key combination to our command in the keymaps.json file:

```
{
  "atom-workspace": {
    "ctrl-alt-shift-f": "fix-identifiers:activate"
  }
}
```

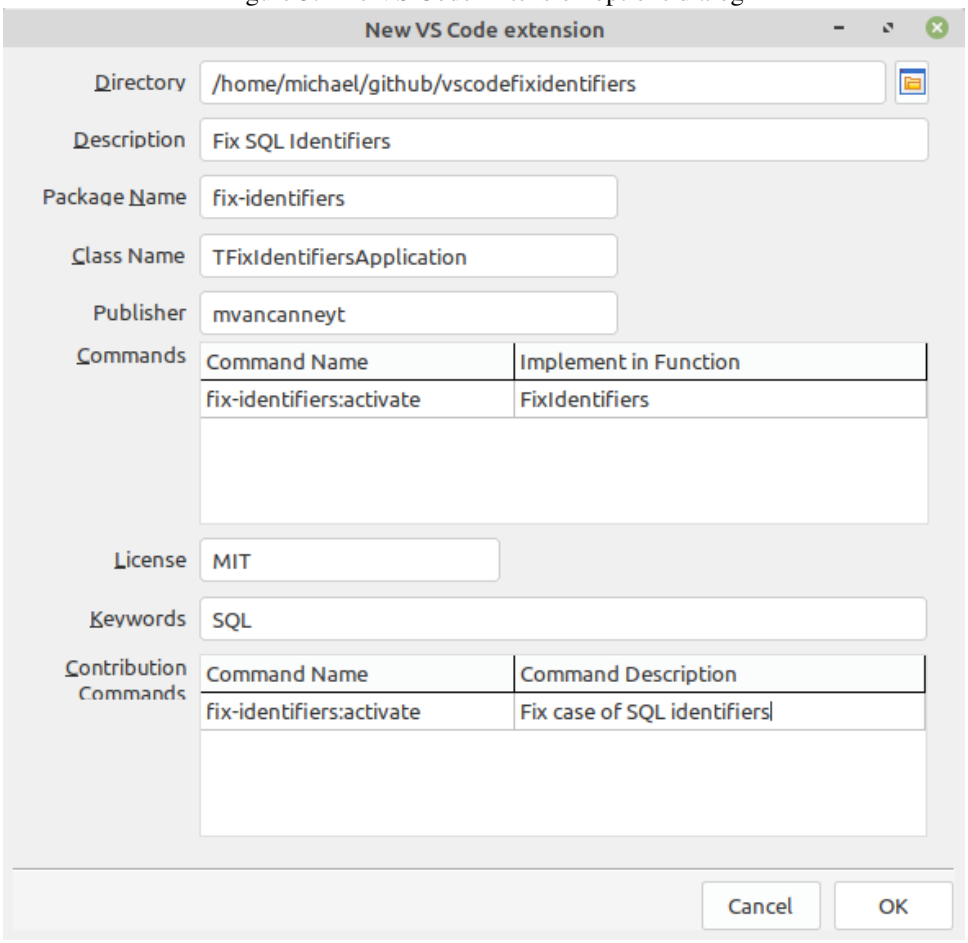
We compile the lazarus project, and restart Atom. When we press ctrl-shift-P to invoke the command palette, we start typing the command name, we can see our command as in figure 4 on page 12.

You can easily verify that the command actually changes the casing of the SQL keywords.

If you wish to debug the package, you must start the Atom editor with the '–dev' command-line option. When you do so, then you can show the Chromium 'Developer tools' using the 'View - Developer - Toggle Developer tools' menu: this will present you with the sources of your plugin - in pascal - and you can debug the Atom package.

To distribute your package, all you need to do is create a .zip file from the directory with the code, or push it onto a github repository.

Figure 5: The VS Code Extension options dialog



7 A sample VS Code package

The functionality that was made for the Atom editor can of course also be implemented for VS Code. To do so, we can start the VS Code Extension in the Lazarus IDE's 'Project-New Project' dialog. figure 5 on page 13

The various items that can be entered for VS Code are - not surprisingly - very similar to the one in the Atom package dialog:

Directory Every VS Code extension lives in its own directory. Here you specify the directory for the new extension.

Description A textual description of your package, it goes in the manifest file (package.json).

Package Name A (unique) name for your package, it goes in the manifest file.

Class Name The Pascal class name for the application class.

Publisher If you want to publish your package in the online VS Code extension repository, here you must enter your developer name.

Commands The commands that your package will provide to the editor. For each command you must specify a unique name, and the name of a pascal function that will

be called when the command is invoked. Again, an empty function will be generated for each function you specify here.

License The license for your package, it goes in the manifest file.

Keywords Some keywords (space separated) for your package, it goes in the manifest file.

Contribution Commands The commands that will cause your package to be loaded by the editor. The scope is a valid Atom scope identifier such as `atom-workspace`. This list goes in the manifest file. VS Code editor will use this list to present your commands in the command palette.

For the sample code, we use the same names and settings as in the Atom package.

When you click OK, the IDE will make a set of files that make up the extension (see figure figure 6 on page 15):

fix_identifiers.lpr The program with generated code: it can be compiled.

.vscode/tasks.json This file is used by VS Code to build your package: It contains the pas2js command-line needed to build your package; it is possible to edit and compile your code in VS Code.

.vscode/launch.json This file is used by VS Code to run and debug your package: It contains the necessary command-line options needed to start VS Code with your extension loaded.

package.json The package manifest file.

js/packageglue.js This file contains the VS Code extension Javascript wrapper code shown above. As in the case of the Atom package: If you decide to change the name (or output file) of the Pascal project, you must not forget to change the name of the imported project file here.

Again, the IDE has generated a project file that is ready to be compiled, you just need to create some code in the correct callbacks.

```
TFixIdentifiersApplication = Class(TVSCodeApplication)
Protected
  procedure DoActivate; override;
  procedure DoDeactivate; override;
Public
  function FixIdentifiers(args : TJSValueDynArray) : JSValue;
end;
```

Note that the generated `FixIdentifiers` function has some arguments and returns a value.

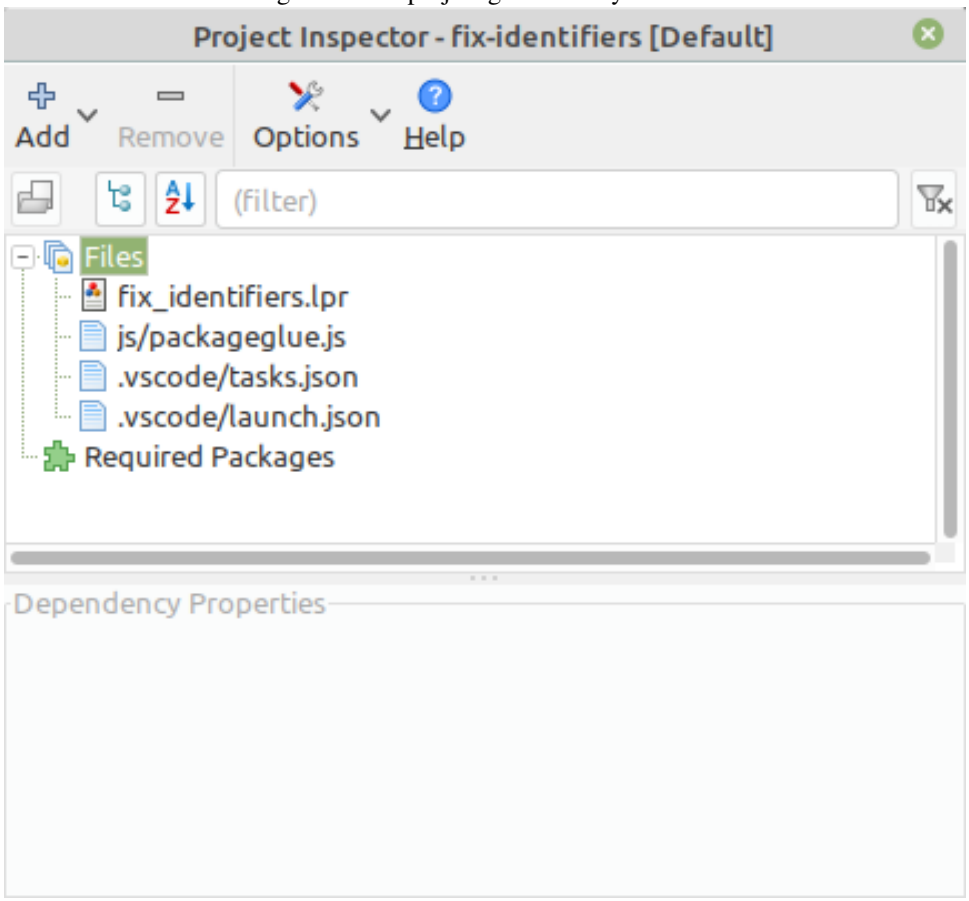
The `DoActivate` function contains the code to register our command:

```
procedure TFixIdentifiersApplication.DoActivate;

Var
  disp : TVSDisposable;

begin
  inherited DoActivate;
  disp:=VSCode.commands.registerCommand('fix-identifiers:activate',
```

Figure 6: The project generated by the IDE



```
                                                @FixIdentifiers);
  TJSArray(ExtensionContext.subscriptions).push(disp);
end;
```

This code looks very similar to the `DoActivate` code of the Atom package. The result of the `registerCommand` command is a VS Code disposable class: The `ExtensionContext` was passed by VS Code to the VS Code extension. It contains a `Subscription` array which you can push elements on: We push the disposable result of the `registerCommand` to the `Subscriptions` array.

The `FixIdentifiers` method has been generated by the Lazarus IDE wizard, and we must fill it with code to implement our plugin.

The API of VS Code is quite big, it is documented here:

`https://code.visualstudio.com/api/references/vscode-api`

Unfortunately, the VS Code editor does not offer a search and Replace API such as it exists in Atom. We must implement the search and replace ourselves.

To change the contents of a document, you must first obtain a reference to the document. We do this in a similar manner as in the Atom plugin: the `window.activeTextEditor` returns an active text editor: an object of class TVSTextEditor.

The `Document` property of a `TVSTextEditor` class is an in instance of `TVSTextDocument` which contains the actual document that the user is editing. This class does not offer methods to directly manipulate the text: every edit must be done by a `TVSTextEditorEdit` instance: the `TVSTextEditor` class has a `edit` method which creates such an edit and calls an event handler with the created instance (called `editBuilder` in the code below).

To make things easier, we will retrieve the whole text of the document (there is a method called `getText` for this), replace all identifiers in this text, and then use the `TVSTextEditorEdit`'s `replace` method to set the new text of the document. The `replace` method replaces a given range's text with a new supplied text.

The replacing of the text needs a Range (class `TVSRange`): this is a small object that contains 2 positions: the start and end position of a range of text. Since we will be replacing the whole text of the document, we create a range that starts at row 0 column 0, and which is 1 line too long: the `validateRange` method of the `TVSTextDocument` clips the range so it is valid, and we use that to correct the Range.

Putting all this together leads to the following code:

```
function TFixIdentifiersApplication.FixIdentifiers(
         args : TJSValueDynArray) : JSValue;

Const
  SErrNoEditor = 'Cannot replace identifiers: no editor';

Var
  Ed: TVSTextEditor;
  aText : String;
  R: TVSRange;

begin
  Result:=null;
  Ed:=VSCode.window.activeTextEditor;
  if not Assigned(Ed) then
    begin
```

```
    VSCode.window.showInformationMessage(SErrNoEditor);
    exit;
end;
  aText:=Ed.document.getText();
  aText:=DoUppercaseSQL(DoUpperCaseSQL(aText));
  R:=TVSRange.New(0,0,Ed.document.lineCount,0);
  R:=Ed.document.validateRange(R);
  Ed.edit(procedure (editBuilder: TVSTextEditorEdit)
    begin
    editBuilder.replace(R,aText);
    end
  );
end;
```

The `DoUppercaseSQL` does the actual search and replace on the text. It is a simple loop which uses the standard Javascript `String.replace` to do the search and replace.

```
Function TFixIdentifiersApplication.DoUpperCaseSQL(
                                    aText : String) : String;

Const
  ToUpperCase : Array of string
              = ('bigint','smallint','int','varchar',
                 'char','not null default','not null');

Var
  S,aRegex,aRepl : String;

begin
  Result:=aText;
  For S in ToUpperCase do
    begin
    aRegex:='(^|\W*)'+S+'(\W|$)';
    aRepl:='$1'+UpperCase(S)+'$2';
    Result:=TJSString(Result).replace(
               TJSRegexp.New(aRegex,'ig'),
               aRepl);
    end;
end;
```

This is maybe not the most efficient algorithm, but it will do nicely for demonstration purposes.

Compiling the program and debugging it in VS Code we can see that the project actually works:

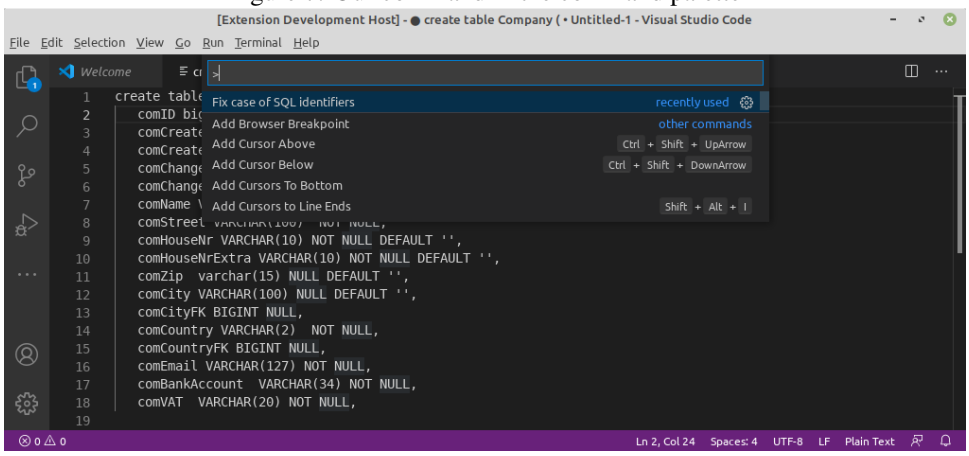
If you want to distribute your package, you need to build it. In order to do so you need to install the `vsce` (Visual Studio Code Extensions) npm package:

```
npm install -g vsce
```

This will install a vsce command on your system, which you can then use to create a .vsix file:

```
vsce package
```

Figure 7: Our command in the command palette



The packager may complain about a missing repository, but if all went well you will get a message that your file was created:

```
home:~/github/vscodefixidentifiers> vsce package
DONE  Packaged:
/home/michael/vscodedemo/fix-identifiers-0.0.1.vsix (17 files, 365.06KB)
```

8 Conclusion

VS Code and Atom plugins are normally created in Javascript. Thanks to Pas2JS and TMS Web core, Pascal programmers can now program plugins for these 2 popular engines in Pascal. The method shown here will become more simple in the future: when library support is finished, then the glue code will no longer be necessary.