D11.1

CONTENT

ARTICLES

ADVERTISERS

Pascal is an imperative and procedural programming language, which Niklaus Wirth designed (left below) in 1968–69 and published in 1970, as a small, efficient language intended to encourage good programming practices using structured programming and data structuring. A derivative known as Object Pascal designed for object-oriented programming was developed in 1985. The language name was chosen to honour the Mathematician, Inventor of the first calculator: Blaise Pascal (see top right).

Niklaus Wirth

Contributors

Stephen Ball
`http://delphiaball.co.uk`
`@DelphiABall`

Dmitry Boyarintsev
`dmitry.living @ gmail.com`

Michaël Van Canneyt,
`michael @ freepascal.org`

Marco Cantù
`www.marcocantu.com`
`marco.cantu @ gmail.com`

David Dirkse
`www.davdata.nl`
`E-mail: David @ davdata.nl`

Benno Evers
`b.evers @ everscustomtechnology.nl`

Bruno Fierens
`www.tmssoftware.com`
`bruno.fierens @ tmssoftware.com`

Holger Flick
`holger @ flixments.com`

Mattias Gärtner
`nc-gaertnma@netcologne.de`

Max Kleiner
`www.softwareschule.ch`
`max @ kleiner.com`

John Kuiper
`john_kuiper @ kpnmail.nl`

Wagner R. Landgraf
`wagner @ tmssoftware.com`

Vsevolod Leonov
`vsevolod.leonov@mail.ru`

Andrea Magni `www.andreamagni.eu`
`andrea.magni @ gmail.com`
`www.andreamagni.eu/wp`

Paul Nauta PLM Solution `Architect`
`CyberNautics`
`paul.nauta @ cybernautics.nl`

Kim Madsen
`www.component4developers.com`

Boian Mitov
`mitov @ mitov.com`

Jeremy North
`jeremy.north @ gmail.com`

Detlef Overbeek - Editor in Chief
`www.blaisepascal.eu`
`editor @ blaisepascal.eu`

Howard Page Clark
`hdpc @ talktalk.net`

Heiko Rompel
`info @ rompelsoft.de`

Wim Van Ingen Schenau -Editor
`wisone @ xs4all.nl`

Rik Smit
`rik @ blaisepascal.eu`

Bob Swart
`www.eBob42.com`
`Bob @ eBob42.com`

B.J. Rao
`contact @ intricad.com`

Daniele Teti
`www.danieleteti.it`
`d.teti @ bittime.it`

Anton Vogelaar
`ajv @ vogelaar-electronics.com`

Danny Wind
`dwind @ delphicompany.nl`

Jos Wegman / Corrector / Analyst

Siegfried Zuhr
`siegfried @ zuhr.nl`

Trademarks All trademarks used are acknowledged as the property of their respective owners.
Caveat Whilst we endeavour to ensure that what is published in the magazine is correct, we cannot accept responsibility for any errors or omissions.
If you notice something which may be incorrect, please contact the Editor and we will publish a correction where relevant.
Subscriptions (2019 prices)

	Internat. excl. VAT	Internat. incl. 9% VAT	Shipment
Printed Issue ±60 pages	€ 155,96	€ 250	€ 80,00
Electronic Download Issue 60 pages	€ 64,20	€ 70	———
Printed Issue inside Holland (Netherlands) 60 pages	———	€ 250,00	€ 70,00

Member and donator of WIKIPEDIA
Member of the **Royal Dutch Library**
KB

Subscriptions can be taken out online at **www.blaisepascal.eu** or by written order, or by sending an email to **office@blaisepascal.eu**
Subscriptions can start at any date. All issues published in the calendar year of the subscription will be sent as well.
Subscriptions run 365 days. Subscriptions will not be prolonged without notice. Receipt of payment will be sent by email.
Subscriptions can be paid by sending the payment to:
ABN AMRO Bank Account no. 44 19 60 863 or by credit card or Paypal
Name: Pro Pascal Foundation-Foundation for Supporting the Pascal Programming Language (Stichting Ondersteuning Programeertaal Pascal)
IBAN no.: NL82 ABNA 0441960863 BIC ABNANL2A VAT no.: 81 42 54 147 (Stichting Programmeertaal Pascal)
Subscription department
Edelstenenbaan 21 / 3402 XA IJsselstein, The Netherlands
Mobile: + 31 (0) 6 21.23.62.68 **office@blaisepascal.eu**

From your editor

Hello dear readers,
the next issue is finally available and now we
need to explain why certain things happen.
In my country (Holland) it is springtime.
Now because of the rising temperatures it is very
nice to go out again and even Corona seems to
vanish.

This is exactly what I had in mind when I tried
to create an event in the Hortus Botanicus in
Leiden.
The plan was great: some of you were very
enthusiast, but there were to few of you.
Since I had to confirm the reservation within 2
weeks the risk was to large there wouldn't be
enough participants.
Now I advised my colleges from Barnsten to use
a later date for their event in May because than it
might be even warmer and no more real danger
of Corona. That will be on Thursday the 19th of
May in the Dutch "Brooklyn" Breukelen.

Alas that is not the only reason for being
cautious. Mr. Putin demonstrated he is a very
fearful men with a syndrome persecution
madness. This results in aggression and
oppressing his own people and even attacking
Ukraine.

To alleviate their fate of being driven from home
we need to help them. That's why we addressed
some advertisement for them to get free
subscriptions, a book and even a whole
programming tool from Componets4Developers.

I haven't seen many others but I urge them to do
something alike so these people will be able to
train for a better future. Lazarus itself is for free
for everybody and Delphi has a community tool.
Of course you can choose to help them in your
own way. I hope you will do so.

We all need to go on with our lives and one way is
to do programming.

Again, Michael van Canneyt (for Lazarus and FPC)
has been able to explain quite some things about
programming for the web and on the road to that
goal he created new abilities for Pas2JS.

Especially the explanation of how to use multiple
forms is very interesting. A MUST READ.

There is some extra news about Delphi 11.1 and
how to use it. The option to view the effect of
styles in design time is very helpful.
Because Artificial Intelligence is getting bigger
and better and thus more interesting I wanted
again to have an article on that subject.
There are some extra instalments to make and
Max Kleiner explains them, I try to make it easy to
install all the necessary tools : Python it self and
some extra's from Delphi.
Jim McKeeth is organising an event about Artificial
Intelligence on 30 of March. See the address
below.

For Lazarus is the planning to do this as well, not
through Python but with a direct tool which will
be developed soon. Its not we don't want Python,
but it is an extra step and that is not necessary in
FPC.
At the end of the article list I explain things about
Electron, how it works and what you can use it
for.

Pleas take a look at Jims online event
(*see the address below*):
 Thank you,

Detlef

"Dad, let's play hide and go seek. I'll hide your data, and you seek it. And when you can't find it, you pay me to return it."

AUTHOR: MAX KLEINER

People lie, numbers don't. — unknown.

starter   expert
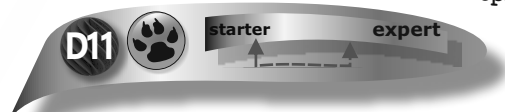
ABSTRACT:

In the last few Articles we have seen that **P4D** is a set of free components that wrap up the **Python** packages into **Delphi** and **Lazarus (FPC).** This time we go(t) the other way round: How can we show the **Python** User to profit from the **VCL** Components. To be able to use this article you should read the Articles from **Issue Nr: 96 Page 9 / Nr: 97 Page 9 / Nr: 98 Page 9 /** Its all available on your LIB stick.
You need to install Python first. In a separate Article: **INSTALLING PYTHON** in this issue is a guide for making the necessary installments.

INTRODUCTION

We create **Python** extension modules from **Delphi** classes, records or functions.
It can be the beginning of a long journey to provide **Delphi's VCL** library as a certain **Python module** to build powerful Windows GUI out from a Script.

The **Python** module we take a look at is called: `DelphiVCL.pyd`
It can be simply installed from the shell via `pip`:

```
pip install delphivcl
```

```
python.exe -m pip install delphivcl
```

in case you want to install the **32bit** version with the **32bit executable**.

On **Win,** the standard **Python** installer already associates the `.py` extension with a file type `(Python.File)` and gives that file type an open command that runs the interpreter `(G:\Program Files\Python\python.exe "%1" %*)`.
This is enough to make scripts executable from the `command prompt`.
We can use the **python-dll** as we use a **windows dll**.
Therefore `*.pyd` files are **dll-libraries**, but there are a few differences:
So far you have to know 3 different file types you can import from, after installed a known package like **Delphi VCL:**

❶ `*.py:`   The norm input source code that we had written.
❷ `*.pyc:`   The compiled bytecode. If you import a module, Py will build a *.pyc file that contains bytecode to make importing it again later easier and faster.
❸ `*.pyd:`   The mentioned windows dll file for Python.

```
import importlib.machinery, importlib.util
def new_import(ext_file):
    loader = importlib.machinery.ExtensionFileLoader("DelphiVCL",ext_file)
    spec   = importlib.util.spec_from_file_location("DelphiVCL",ext_file,
        loader = loader, submodule_search_locations=None)
#print("spec", spec, spec.loader, modulefullpath, __file__)
```

https://github.com/maxkleiner/DelphiVCL4Python/blob/main/tests/__init__.py

The project which we introduce is in the subdirectory **Delphi** and generates a **Python** extension module (*a DLL with extension "pyd" in Windows*) that allows you to create a user interface using **Delphi** from within **Python.**
A part of the **VCL** or **LCL** (almost and maybe) is wrapped with a few lines of code!

The small demo `TestApp.py` gives you a flavour of what is possible.
The machinery by which this is achieved is the `WrapDelphi` unit.
The subdirectory `DemoModule` demonstrates how to create **Python** extension modules using **Delphi,** that allow you to use in Python, functions defined in **Delphi.**
Compile the project and run `test.py` from the `command prompt` (e.g. py test.py).

The generated `pyd` file should be in the same directory as the **Python** file.
This project should be easily adapted to use with **Lazarus** and **FPC.**
After compiled to the `DelphiVCL.pyd` we want to use it in a **Python** script, which is the main topic of this article:

```
from delphivcl import *
```

Python code in one module gains access to code in another module, by the process of importing it. The import statement is the most common way of invoking the import machinery, but it is not the only way.

First we check our **Python** installation.
Python 3.* provides for all user and current user installations.
All user installations place the `Py-dll` in the **Windows System directory** and write the registry info to `HKEY_LOCAL_MACHINE` .
Current user installations place the `dll` in the install path and the registry info in `HKEY_CURRENT_USER` version **< PY 3.5.**

So, for current user installations we need to try and find the install path or package path since it may not be on the system path as an environment var.
In our case we set a const to demonstrate:

```
Const PYHOME    ='C:\Users\max\AppData\Local\Programs\Python\Python36-32\';
      VCLHOME ='r"C:\users\max\appdata\local\programs\python\python36-32\lib\site-packages\delphivcl\win32\delphivcl.pyd"';
```

So next we load the dll (with or without import statement possible),
call the **VCL** class and start the main procedure:

```
eg:= TPythonEngine.Create(Nil);
try
  eg.pythonhome:= PYHOME;
  eg.loadDLL;

  ...

  eg.execStr(LoadPy_VCLClass);
  eg.execStr(STARTMAIN);
  eg.execStr('main()');
```
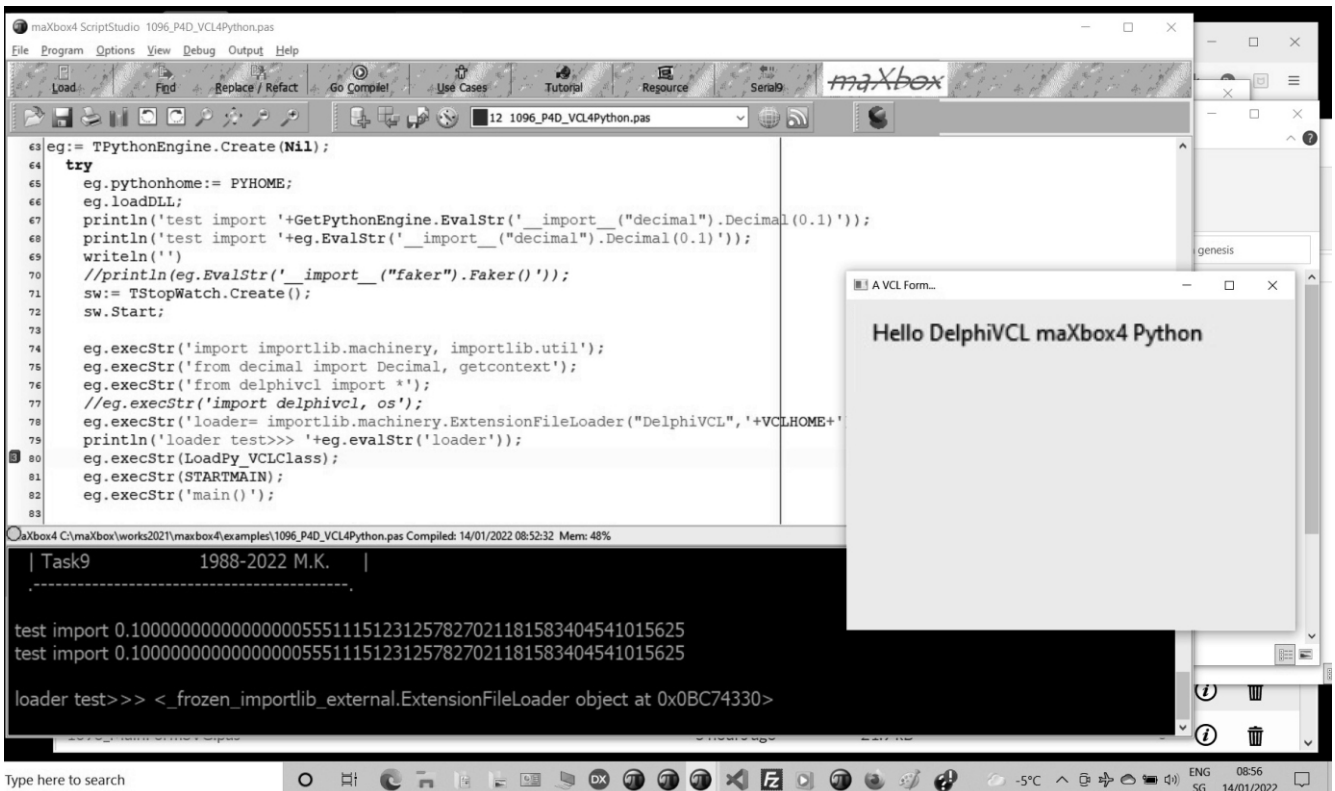

We can see the simple VCL-form as it says "Hello":

```
https://github.com/maxkleiner/DelphiVCL4Python/tree/main/samples/HelloWorld
```

As a special proof of
concept I run the `hello world sample` with **P4D** in a
maXbox script to show the compatibility between the two type- and memory layout
systems. But of course normally the script runs in a shell or with **PyScripter.** As a caveat I
can run this "test toggle workaround" only once, could be that a finalizer,
dispose or destructor is missing.

```
STARTMAIN =
  'def main():                              '+LF+
  ' Application.Initialize()                '+LF+
  ' Application.Title = "Hello Python"      '+LF+
  ' Main = MainForm(Application)            '+LF+
  ' Main.Show()                             '+LF+
  ' FreeConsole()                           '+LF+
  ' Application.Run()                        '+LF+
  , Main.Destroy()                          ';
```

We pass with the
`MainForm()` call our initialized Application to a **Python** class defined
in **LoadPy_VCLClass** which has the class name '`class`
`MainForm(Form)`: with two method-functions (def in class):

```
def __init__(self, owner):
def __on_form_close(self, sender,
action):
```

Imagine on the **VCL-form** from **Python**
is a `SynEdit-control` which enables to script in **Pascal** and **Python**
together, fascinating it:

Python Chats to Svg TestDemo2

| matplotlib | seaborn | free code |

```
1 #from 2delphi_module import svg_image
2 # single line tester - one liner only!
3 [n for n in range(300) if n % 2 !=0]
4
5
6
7
```

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99, 101, 103, 105, 107, 109, 111, 113, 115, 117, 119, 121, 123, 125, 127, 129, 131, 133, 135, 137, 139, 141, 143, 145, 147, 149, 151, 153, 155, 157, 159, 161, 163, 165, 167, 169, 171, 173, 175, 177, 179, 181, 183, 185, 187, 189, 191, 193, 195, 197, 199, 201, 203, 205, 207, 209, 211, 213, 215, 217, 219, 221, 223, 225, 227, 229, 231, 233, 235, 237, 239, 241, 243, 245, 247, 249, 251, 253, 255, 257, 259, 261, 263, 265, 267, 269, 271, 273, 275, 277, 279, 281, 283, 285, 287, 289, 291, 293, 295, 297, 299]

RunPy

10

In **P4D** you do have the mentioned memo with ExecStrings:

```
procedure TForm1.Button1Click(Sender: Tobject);
  begin PythonEngine1.ExecStrings( Memo1.Lines );
end;
```

This explains best the code behind, to evaluate an internal **Python** expression or statement. You are responsible for creating one and only one **TPythonEngine** instance.

CONCLUSION

The **VCL/LCL** is a mature **Windows/Linux** native **GUI** framework with a huge library of included visual components and a robust collection of 3rd party components and classes. It is the finest framework for native **Windows** applications, and we can use it with **Python!**
Python has only one type of module object, and all modules are of this type, regardless of whether the module is implemented in **Python, Delphi, FreePascal, C**, or something else.

VCL4Python topics
- •https://learndelphi.org/python-native-windows-gui-with-delphi-vcl/
- •http://www.softwareschule.ch/examples/weatherbox.txt
- •http://www.softwareschule.ch/examples/pydemo37.htm
- •https://github.com/maxkleiner/DelphiVCL4Python
- •https://t.co/lNhgxqNr7B

AUTHOR: DETLEF OVERBEEK

starter — expert

INSTALLATION
PROCEDURES FOR PYTHON AND DELPHI4PYTHON

If you follow the guidelines of this article it will be fairly easy for you to handle.
For **Lazarus** we will have an other implementation because we will make **Lazarus** directly approachable for the use of special Libraries we need to connect to for **Artificial Intelligence**. That will probably become available in the next Issue (103). That said we will also create a future possibility to have **Python for Lazarus.**

There might be a confusion about several titles for the **Delphi** Projects, there is **Delphi4Python** and there is **Python4Delphi** as well of course **Python** itself.First of all yo need to install **Python.**
You need to have the program because all other subjects use it.

The **"Delphi for Python"** way means you will install a group of components for direct use and get about 34 example programs. Its nice to install them, because otherwise you will need to get the Python4Delphi which is much harder and confusing to install as well get it organized.
So I will give a short explanation.
Installing **Python** (in this case for **Windows**)
To download it go to: `https://www.python.org/`
(there is of course a list of beta versions if you want to use those, but I advise you to chose a stable version).

Python	PSF	Docs	PyPI	Jobs	Community

python™

Donate | Search | GO | Socialize

About | Downloads | Documentation | Community | Success Stories | News | Events

All releases
Source code
Windows
macOS
Other Platforms
License
Alternative Implementations

Download for Windows

Python 3.10.2

Note that Python 3.9+ *cannot* be used on Windows 7 or earlier.
Not the OS you are looking for? Python can be used on many operating systems and environments.
View the full list of downloads.

```
# For loop on
>>> numbers =
>>> product =
>>> for number
...     product
...
>>> print('The
The product is
```

Python is a programming language that lets you work quickly and integrate systems more effectively. ⟫ Learn More

⏻ **Get Started**
Whether you're new to programming or an experienced developer, it's easy to learn and use Python.
Start with our Beginner's Guide

⬇ **Download**
Python source code and installers are available for download for all versions!
Latest: Python 3.10.2

📄 **Docs**
Documentation for Python's standard library, along with tutorials and guides, are available online.
docs.python.org

💼 **Jobs**
Looking for work or have a Python related position that you're trying to hire for? Our **relaunched community-run job board** is the place to go.
jobs.python.org

https://www.python.org/downloads/

About IDLE

IDLE Help
Python Docs   F1
Turtle Demo

IDLE Shell 3.10.2   —   □   ×

File   Edit   Shell   Debug   Options   Window   Help

```
Python 3.10.2 (tags/v3.10.2:a56ebcc, Jan 17 2022, 14:12:15) [MSC v.1929 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

About IDLE 3.10.2 (64 bit)   ×

IDLE

Python's Integrated Development
and Learning Environment

email: idle-dev@python.org
https://docs.python.org/3.10/library/idle.html

Python version: 3.10.2   Tk version: 8.6.12

| License | Copyright | Credits |

IDLE version: 3.10.2

| README | NEWS | Credits |

Close

Pandoc   ⌄

PuTTY (64-bit)   ⌄

Python 3.10   ⌃

IDLE (Python 3.10 64-bit)

Python 3.10 (64-bit)

Python 3.10 Manuals (64-bit)

Python 3.10 Module Docs (64-bit)

There are two different apps
combined: The **IDLE** Shell witch is
"Pythons Integrated Development and
Learning Environment".
The next one shows the **Python Index** of
Modules: *see page 7 of this article.*
There are quite a lot of possibilities:
you will have to try them your self

Ln: 3   Col: 0

```
◨ Command Prompt

Microsoft Windows [Version 10.0.19044.1526]
(c) Microsoft Corporation. All rights reserved.

C:\Users\edito>pip install delphi vcl
Collecting delphi
  Downloading delphi-2.0.1-py3-none-any.whl (7.4 kB)
ERROR: Could not find a version that satisfies the requirement vcl (from
ERROR: No matching distribution found for vcl
WARNING: You are using pip version 21.2.4; however, version 22.0.3 is av
You should consider upgrading via the 'C:\Users\edito\AppData\Local\Prog
--upgrade pip' command.

C:\Users\edito>_
```

I have inverted the **Command Prompt** which is shown here, so they will be better readable. Please read the text and it will help you during installation. You can set the PDF file to show two pages side by side, so can read the whole texst.

```
◨ Command Prompt

  Downloading delphi-2.0.1-py3-none-any.whl (7.4 kB)
ERROR: Could not find a version that satisfies the requirement vcl (from ve
ERROR: No matching distribution found for vcl
WARNING: You are using pip version 21.2.4; however, version 22.0.3 is avail
You should consider upgrading via the 'C:\Users\edito\AppData\Local\Program

C:\Users\edito>-m pip install--upgrade pip
'-m' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\edito>pip install--upgrade pip
ERROR: unknown command "install--upgrade" - maybe you meant "install"

C:\Users\edito>-m pip install --upgrade pip
'-m' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\edito>python.exe -m pip install --upgrade pip
Requirement already satisfied: pip in c:\users\edito\appdata\local\programs
Collecting pip
  Downloading pip-22.0.3-py3-none-any.whl (2.1 MB)
     |████████████████████████████████| 2.1 MB ...
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 21.2.4
    Uninstalling pip-21.2.4:
      Successfully uninstalled pip-21.2.4
Successfully installed pip-22.0.3

C:\Users\edito>
```
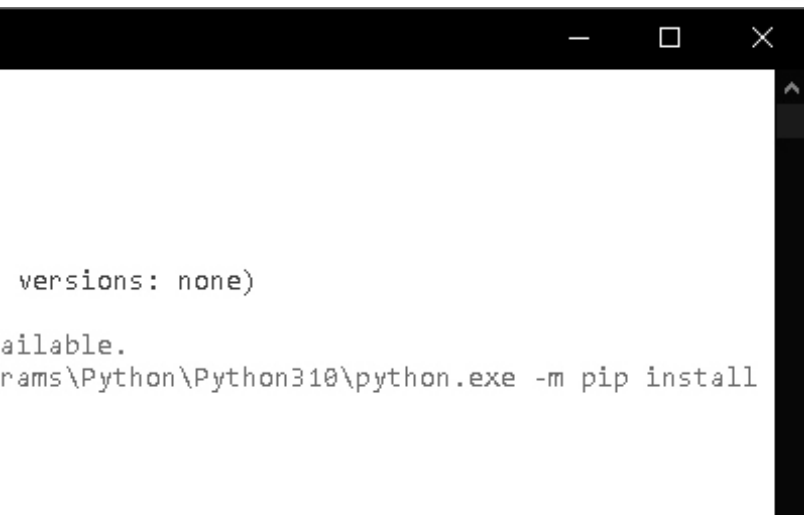
versions: none)

ailable.
rams\Python\Python310\python.exe -m pip install

I have inverted the **Command Prompt** which is shown here,
so they will be better readable.
Please read the text and it will help you during installation.
You can set the PDF file to show two pages side by side, so can read the whole texst.

ersions: none)

able.
ns\Python\Python310\python.exe -m pip install --upgrade pip' command.

:\python\python310\lib\site-packages (21.2.4)

15

Python4Delphi 1.0 by Dietmar Budelsky, Morgan Martinet, Kiriakos Vlahos

Python for Delphi (P4D) is a set of free components that wrap up the Python DLL into Delphi. They let you easily execute Python scripts, create new Python modules and new Python types. You can create Python extensions as DLLs and much more.

1 Oct 2021
MIT license

⊕ Install

Demo02.dproj - Projects ⊹ ×

Demos_01_to_34
∨ Demo01.exe
  > Build Configurations (Debug)
  > Target Platforms (Windows 64-bit)
  ∨ Unit1.pas
    Unit1.dfm
∨ **Demo02.exe**
  > Build Configurations (Debug)
  > Target Platforms (Windows 64-bit)
  ∨ Unit1.pas
    Unit1.dfm
> Demo03.exe
> Demo04.exe
> Demo05.exe
> Demo06.exe
> Demo07.exe
> Demo08.exe
> Demo09.exe
> demodll.dll
> Demo10.exe
> ThrdDemo.exe
> Demo16a.exe
> Demo16b.exe
> Demo17.exe
> Demo21.exe
> Demo22.exe
> Demo23.exe
> VarPythUnitTest.exe
> Demo26.exe
> Demo27.exe
> Demo28.exe
> Demo29.exe
> Demo30.exe
> Demo31.exe
> Demo32.exe
∨ ThrdDemo2.exe
  > Build Configurations (Debug)
  > Target Platforms (Windows 64-bit)
    SortThds.pas
  > ThSort.pas
> Demo34.exe

RAD Install

Completed!

✓ Adding environment path "Source\fmx"
✓ Compiling project "Python.dpk"...
✓ Compiling project "PythonVcl.dpk"...
✓ Compiling project "PythonFmx.dpk"...
✓ Compiling project "dclPython.dpk"...
✓ Compiling project "dclPythonVcl.dpk"...
✓ Compiling project "dclPythonFmx.dpk".
✓ Installing package "dclPython280.bpl"...
✓ Installing package "dclPythonVcl280.bpl
✓ Installing package "dclPythonFmx280.b
✓ Opening project "Demos_01_to_34.grou
✓ Executing command "start "" "C:\Users\
ℹ Process completed successfully.

Installation of the components of Delphi for Python works inside Delphi and the easiest way is to handle it is through the GetIt Package manager.
Search for "Pyt" (*See at the top*) and the click install.
The demo apps and components will be installed automatically.

Some are shown on the
next page 6 of this article

Welcome Page	ThSort	fmMain ×	Unit1	Unit1

VarPyth unit tests   — □ ×

PyPythonGUIInputOutput1

```
class XYZ(object):
  pass

class Foo:
  def __init__(Self, Value=0):
    Self.Value = Value
  def __del__(Self):
    print ("delete", Self)
  def __add__(self, other):
    return Foo(self.Value + other.Value)
  def Inc(Self, AValue = 1):
```

VarPyth unit tests   — □ ×   Dates | Test Objects

```
delete <__main__.Bar object at 0x0000000006220F70>
Sequence test was Ok.
Mapping test was Ok.
Dates test was Ok.
delete <__main__.Foo object at 0x0000000006057D60>
Created <__main__.Foo object at 0x0000000062202E0>
delete <__main__.Bar object at 0x000000000620AE90>
Instanciate class Foo: <__main__.Foo object at 0x00000000061B5FC0>
Test -> a, b, c : 10, 5, 15
Objects test was Ok.
delete <__main__.Foo object at 0x00000000061B5FC0>
delete <__main__.Foo object at 0x0000000062202E0>
delete <__main__.Foo object at 0x0000000006222530>
delete <__main__.Foo object at 0x0000000006223190>
delete <__main__.Foo object at 0x00000000062231F0>
```

☑ Included   ☑ Included

imes

```
class XYZ(object):
  pass

class Foo:
  def __init__(Self, Value=0):
    Self.Value = Value
  def __del__(Self):
    print ("delete", Self)
  def __add__(self, other):
    return Foo(self.Value + other.Value)
  def Inc(Self, AValue = 1):
    Self.Value = Self.Value + AValue
  def GetSelf(Self):
    return Self
  def GetValue(Self):
    return Self.Value
  def SetABC(Self, A, B, C):
    Self.A = A
    Self.B = B
    Self.C = C
  def Add(Self, AFooInst):
    Self.Value = Self.Value + AFooInst.Value
class Bar(Foo):
  def Inc(Self, AValue = 1):
    Self.Value = Self.Value - AValue
def Add(a, b):
  return a + b
def MakeList(a, b, c, d):
  return [a, b, c, d]

f = Foo()
print ("Created", f)
f.Inc()
f.Inc(2)
b = Bar()
b.Inc()
b.Inc(2)
```

Python 3.10.2 documentation

Hide | Locate | Back | Forward | Home | Font | Print | Options

Contents | Index | Search | Favourites

- 3.10.2 Documentation
- Python Module Index
- What's New in Python
- The Python Tutorial
- Python Setup and Usage
- The Python Language Reference
- The Python Standard Library
- Extending and Embedding the P...
- Python/C API Reference Manual
- Distributing Python Modules
- Installing Python Modules
- Python HOWTOs
- Python Frequently Asked Questi...
- Glossary
- About these documents
- Dealing with Bugs
- Copyright
- History and License

Python » 3.10.2 Documentation »

Python Documentation contents

- What's New in Python
  - What's New In Python 3.10
    - Summary – Release highlights
    - New Features
      - Parenthesized context managers
      - Better error messages
        - SyntaxErrors
        - IndentationErrors
        - AttributeErrors

Execute Script	Test Integers	Test Floats	Test Strings	Test Sequences	Test Mappings	Test Dates	Test Objects
	☑ Included	☑ Included	☑ Included	☑ Included	☑ Included	☑ Included	☑ Included
	Run selected tests once			Run selected tests n times		1000 times	

Pydoc: Index of Modules   ×   +

← → C ⌂ ⓘ localhost:63083   ⊕ ⤴ ☆ ⌂ ⊘ ✱ ⬤ ⋮

Python 3.10.2 [tags/v3.10.2:a58ebcc, MSC v.1929 64 bit (AMD64)]   Module Index : Topics : Keywords
Windows-10   [   ] Get   [   ] Search

Index of Modules

Built-in Modules

_abc	_imp	_stat	errno
_ast	_io	_statistics	faulthandler
_bisect	_json	_string	gc
_blake2	_locale	_struct	itertools
_codecs	_lsprof	_symtable	marshal
_codecs_cn	_md5	_thread	math
_codecs_hk	_multibytecodec	_tracemalloc	mmap
_codecs_iso2022	_opcode	_warnings	msvcrt
_codecs_jp	_operator	_weakref	nt
_codecs_kr	_pickle	_winapi	sys
_codecs_tw	_random	_xxsubinterpreters	time
_collections	_sha1	array	winreg
_contextvars	_sha256	atexit	xxsubtype
_csv	_sha3	audioop	zlib
_datetime	_sha512	binascii	
_functools	_signal	builtins	
_heapq	_sre	cmath	

C:\Users\edito\AppData\Local\Programs\Python\Python310

C:\Users\edito\AppData\Local\Programs\Python\Python310\python310.zip

C:\Users\edito\AppData\Local\Programs\Python\Python310\DLLs

_asyncio	_lzma	_ssl	_tkinter
_bz2	_msi	_testbuffer	_uuid
_ctypes	_multiprocessing	_testcapi	_zoneinfo
_ctypes_test	_overlapped	_testconsole	pyexpat
_decimal	_queue	_testimportmultiple	select
_elementtree	_socket	_testinternalcapi	unicodedata
_hashlib	_sqlite3	_testmultiphase	winsound

C:\Users\edito\AppData\Local\Programs\Python\Python310\lib

__future__	dataclasses	multiprocessing (package)	sre_constants
_aix_support	datetime	netrc	sre_parse
_bootsubprocess	dbm (package)	nntplib	ssl
_collections_abc	decimal	ntpath	stat
_compat_pickle	difflib	nturl2path	statistics
_compression	dis	numbers	string
_markupbase	distutils (package)	opcode	stringprep
_osx_support	doctest	operator	struct
_py_abc	email (package)	optparse	subprocess
_pydecimal	encodings (package)	os	sunau
_pyio	ensurepip (package)	pathlib	symtable
_sitebuiltins	enum	pdb	sysconfig
_strptime	filecmp	pickle	tabnanny
_threading_local	fileinput	pickletools	tarfile
_weakrefset	fnmatch	pipes	telnetlib
abc	fractions	pkgutil	tempfile
aifc	ftplib	platform	test (package)
antigravity	functools	plistlib	textwrap
argparse	genericpath	poplib	this
ast	getopt	posixpath	threading
asynchat	getpass	pprint	timeit
asyncio (package)	gettext	profile	tkinter (package)
asyncore	glob	pstats	token
base64	graphlib	pty	tokenize
bdb	gzip	py_compile	trace
binhex	hashlib	pyclbr	traceback
bisect	heapq	pydoc	tracemalloc
bz2	hmac	pydoc_data (package)	tty
cProfile	html (package)	queue	turtle
calendar	http (package)	quopri	turtledemo (package)

barnsten
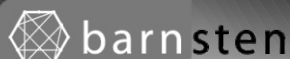
Kom naar de Delphi Dag 2022!
Wat hebben we hiernaar uitgekeken! Een LIVE Delphi dag met interessante sessies verzorgd door Delphi Experts.

Wat kunt u verwachten?

Maak kennis met ervaren Delphi experts die u tijdens hun boeiende sessies meer vertellen over de door hen gebruikte technieken, nieuwe technologieën, Delphi innovaties zoals bijvoorbeeld het gebruik van Delphi in combinatie met Mendix (Low-Code en Web) en Python (AI).

Bent u of gaat u bestaande applicaties migreren? Onze sprekers kunnen u daar zeker bij helpen! Of leer meer over Duster, de ideale tool om u te ondersteunen bij de migratie van uw Delphi code naar de nieuwste versie. Buiten de sessies om heeft u alle gelegenheid om contact te hebben met de sprekers en collega-ontwikkelaars.

Wie zijn de sprekers?

De sprekers vandaag zijn Delphi MVP's Bob Swart (Bob Swart Training & Consultancy), Danny Wind (The Delphi Company) en Marco Geuze (GDK Software). Ook zal Laurens van Run van het bedrijf Mendrix een interessante presentatie verzorgen.

Alle sessies zijn nederlands gesproken.

Dagprogramma

09:00	- 09:30	-	Welkom met koffie/thee en lekkers
09:30	- 09:45	-	Opening door Barnsten
09:45	- 10:45	-	Gebruik Delphi met Python in Artificial Intelligence Neural Networks - door Danny Wind
10:45	- 11:15	-	Pauze
11:15	- 12:15	-	Delphi en Mendix - een mooi duo - door Marco Geuze en Kees de Kraker
12:15	- 13:00	-	Lunch
13:00	- 14:00	-	High quality and maintainable code in Delphi - door Laurens van Run van Mendrix
14:10	- 5:10	-	Duster migratie tool - door GDK Software
15:10	- 15:30	-	Pauze
15:30	- 16:30	-	Delphi (Automatisch Testbaar) Web Development met IntraWeb - door Bob Swart
16:30	- 16:45	-	Q&A

Kosten en voorwaarden:
De toegangsprijs is € 99,-- excl. BTW en incl. toegang tot alle sessies, koffie/thee/frisdranken, snack en lunch. De tickets zijn niet te annuleren, maar kunnen wel worden overgedragen aan een collega. Sessies en datum kunnen wijzigen in geval van onvoorziene omstandigheden.

https://www.eventbrite.nl/e/tickets-delphi-dag-2022-290410775447

DELPHI 11.1 THE LATEST VERSION
UPDATE OR UPGRADE?

D11.1

The SplashScreen:The New Logo

ABSTRACT

UPDATE: A change in information, a modification of existing or known data.
An update is a concept that in itself means that there is an improvement.
UPGRADE: Upgrading is the process of replacing a product with a newer version
of the same product. An upgrade is a concept that in itself means that there is
an increase in value.
I felt it necessary to clarify for myself the difference, because among other
things I wanted to know what is the real nature of the latest version.
According to this is **Delphi 11.1** an **Upgrade,** and that means we need to
expect it as a new version, which in consequence means
you have to replace the older version 11.

INSTALLING
MIGRATION

Because of that you need to (make a backup)
create a Migration file before you install the new
version:
Installing will help you to first uninstall and
then install. After the install you can set your
original migration file to recreate your
settings as much as possible to overcome
the endlessly and irritating reinstalling of
all kinds of components. This means the
creation of a migration file. Let us start
with that. In windows choose pen the
program overview by clicking the
(probably - left bottom) Windows
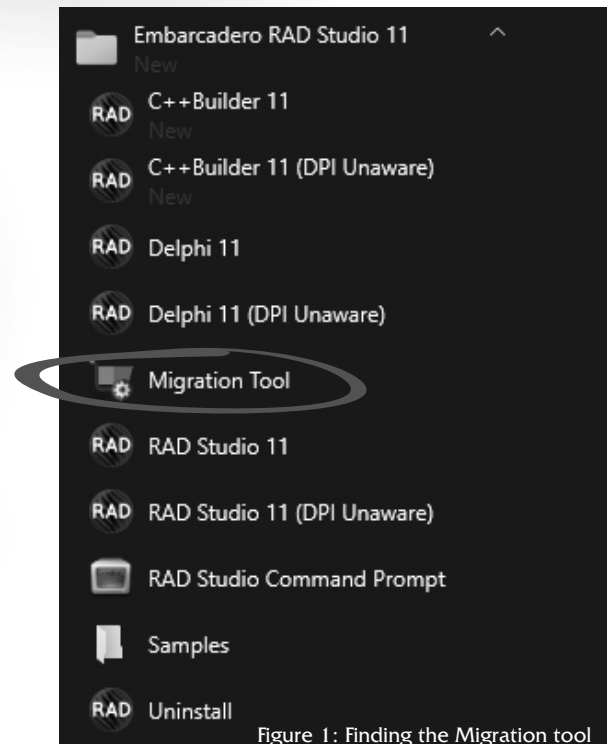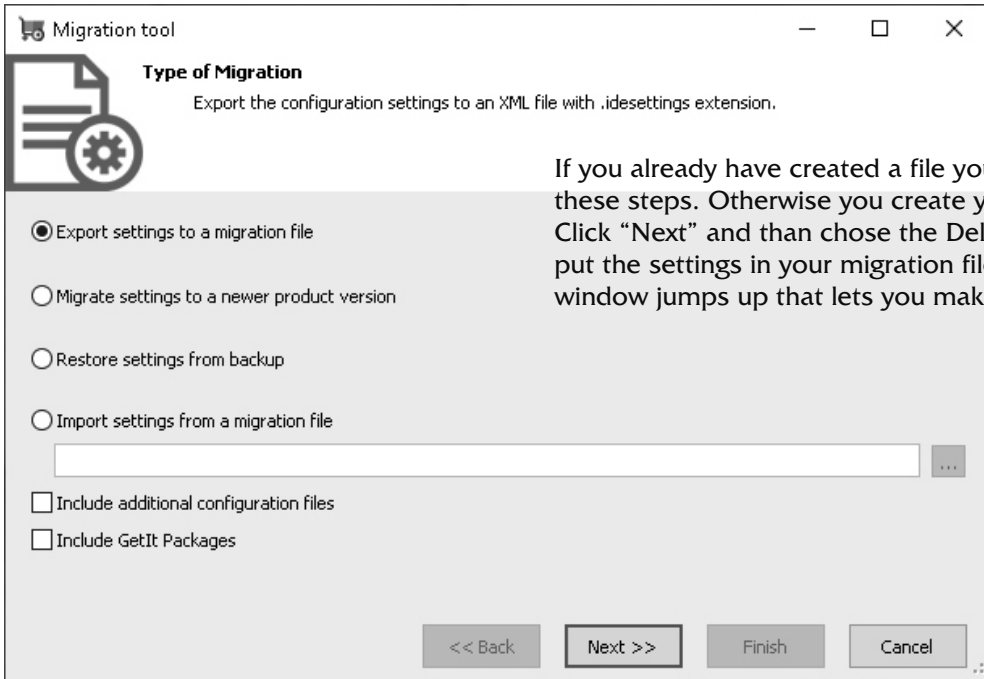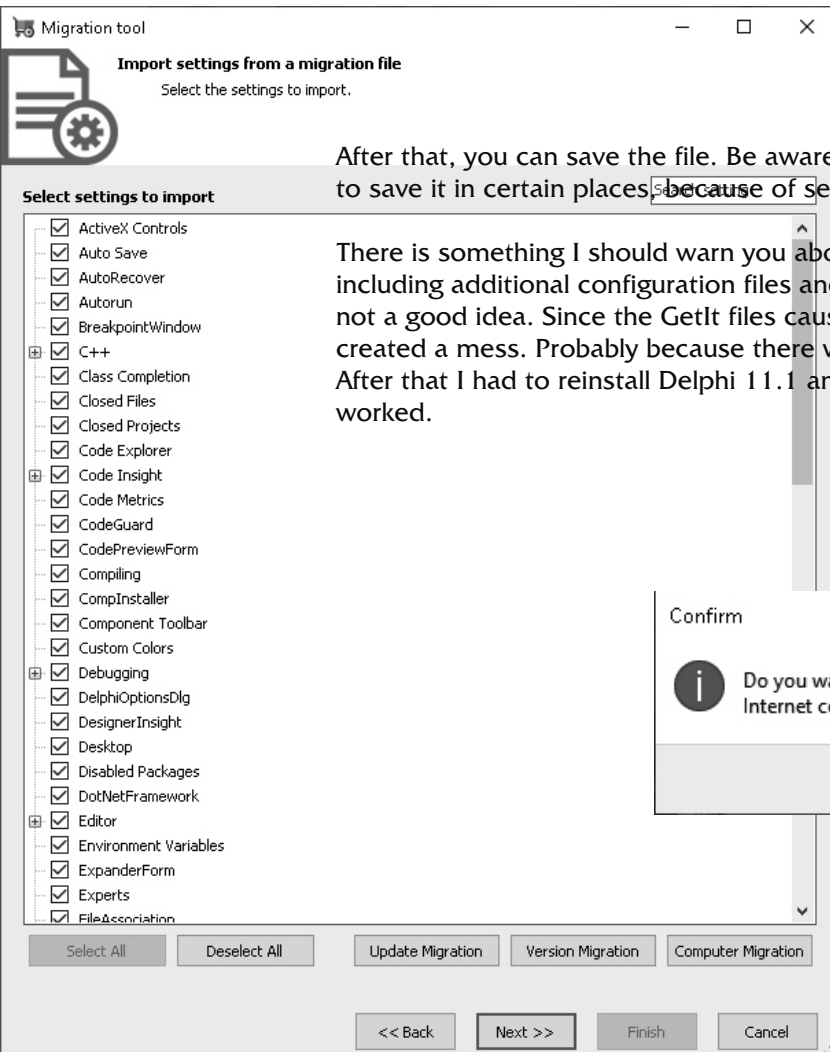Symbol. Here you search for
Embarcadero and will see
Figure 1.

Figure 1: Finding the Migration tool

Migration tool

Type of Migration

Export the configuration settings to an XML file with .idesettings extension.

⦿ Export settings to a migration file

◯ Migrate settings to a newer product version

◯ Restore settings from backup

◯ Import settings from a migration file

☐ Include additional configuration files
☐ Include GetIt Packages

[<< Back] [Next >>] [Finish] [Cancel]

If you already have created a file you don't need to follow these steps. Otherwise you create your First Migration file. Click "Next" and than chose the Delphi version you want to put the settings in your migration file. Click"Next" again and a window jumps up that lets you make choices.

Figure 2: Export settings

Migration tool

Import settings from a migration file

Select the settings to import.

Select settings to import

- ☑ ActiveX Controls
- ☑ Auto Save
- ☑ AutoRecover
- ☑ Autorun
- ☑ BreakpointWindow
- ☑ C++
- ☑ Class Completion
- ☑ Closed Files
- ☑ Closed Projects
- ☑ Code Explorer
- ☑ Code Insight
- ☑ Code Metrics
- ☑ CodeGuard
- ☑ CodePreviewForm
- ☑ Compiling
- ☑ CompInstaller
- ☑ Component Toolbar
- ☑ Custom Colors
- ☑ Debugging
- ☑ DelphiOptionsDlg
- ☑ DesignerInsight
- ☑ Desktop
- ☑ Disabled Packages
- ☑ DotNetFramework
- ☑ Editor
- ☑ Environment Variables
- ☑ ExpanderForm
- ☑ Experts
- ☑ FileAssociation

[Select All] [Deselect All] [Update Migration] [Version Migration] [Computer Migration]

[<< Back] [Next >>] [Finish] [Cancel]

After that, you can save the file. Be aware that Windows does only allow you to save it in certain places, because of security reasons.

There is something I should warn you about: I tried to do a complex export by including additional configuration files and Include GetIt packages. That was not a good idea. Since the GetIt files caused quite a lot of work and finally created a mess. Probably because there were newer files.
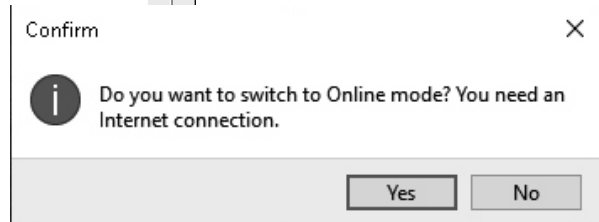After that I had to reinstall Delphi 11.1 and after simply not doing extras it all worked.

Confirm

Do you want to switch to Online mode? You need an Internet connection.

[Yes] [No]

Figure 3: You'd better be sure
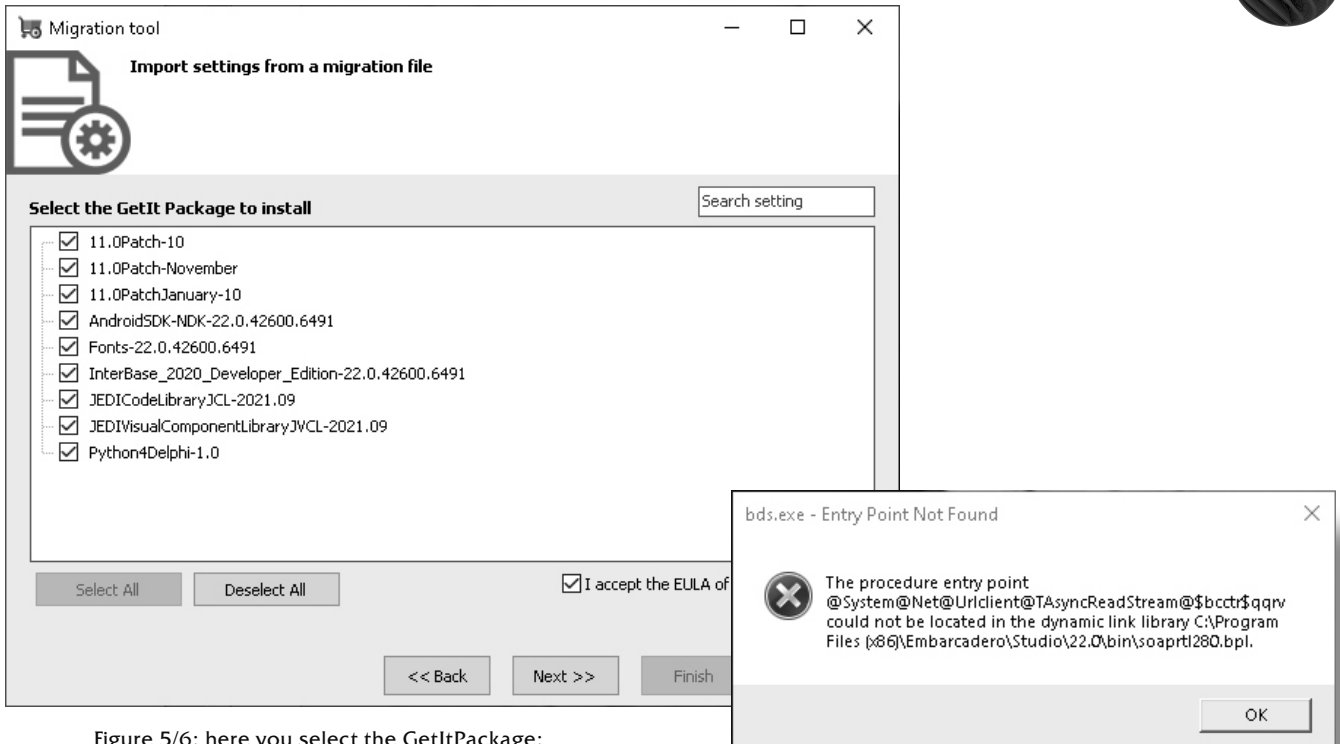
Figure 4: Settings for import

D11.1

Figure 5/6: here you select the GetItPackage:
the problem is that if you choose to do so your new version of **GeIt** will
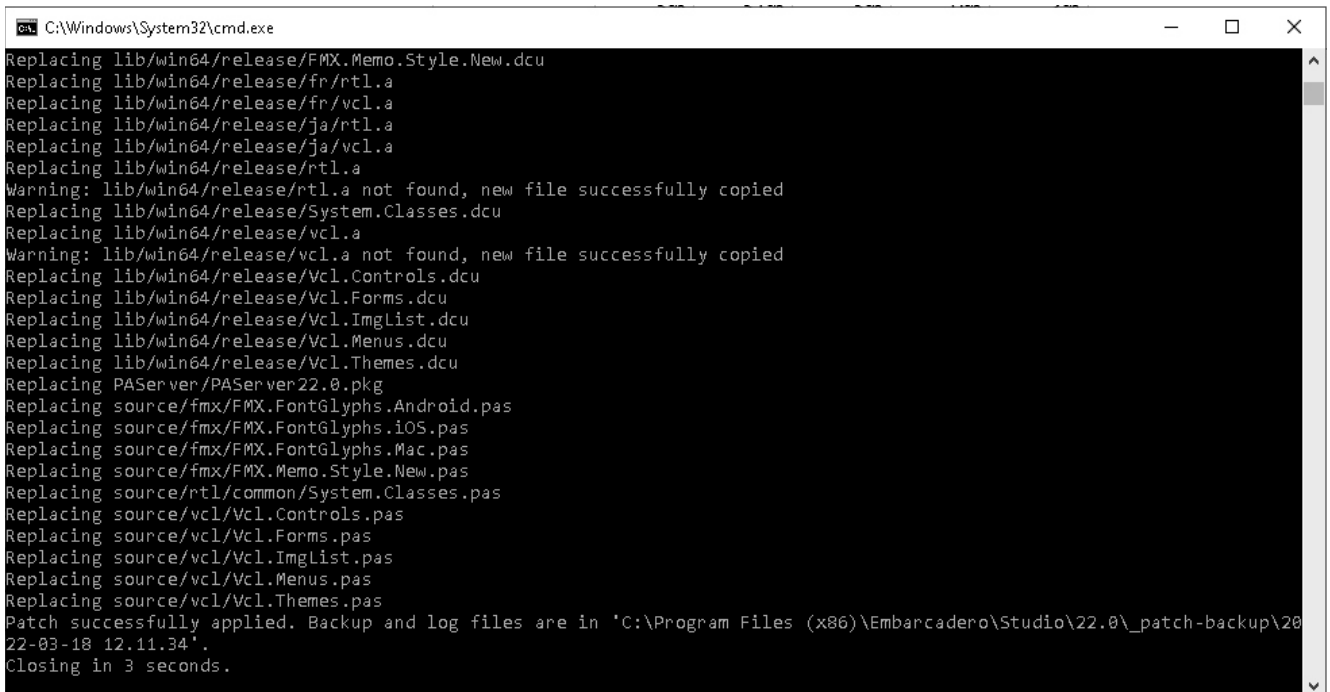become a problem.

Figure 7: Not knowing this I executed it and than a lot of
time was used to install all the old Getit packages.

I was not able to start Delphi again.
The easiest solution I found was reinstalling.
Since I am using the `.iso` file (`RADStudio_11_1_esd_10_8973a.iso`)
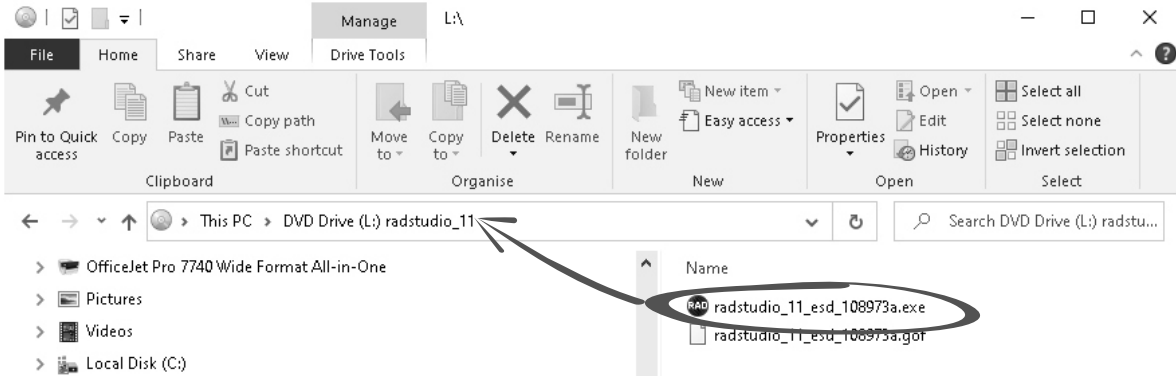it does not need that much time to do it all over. Much better than it ever was....

Figure 7:
For installing you will find if you use the .iso
a new virtual DVD player.

Unpacked to a virtual DVD
drive the number of the
Delphi version is 11. But do
not worry. its correct.

DELPHI 11.1 THE LATEST VERSION
UPDATE OR UPGRADE?

After creating
the Migration setting file you can start.
We need to find out what are the essential new things for
this version, and that's exactly what I want to do in this article...

If you have a version of **Delphi** 11 it will ask you if you want it to be uninstalled.
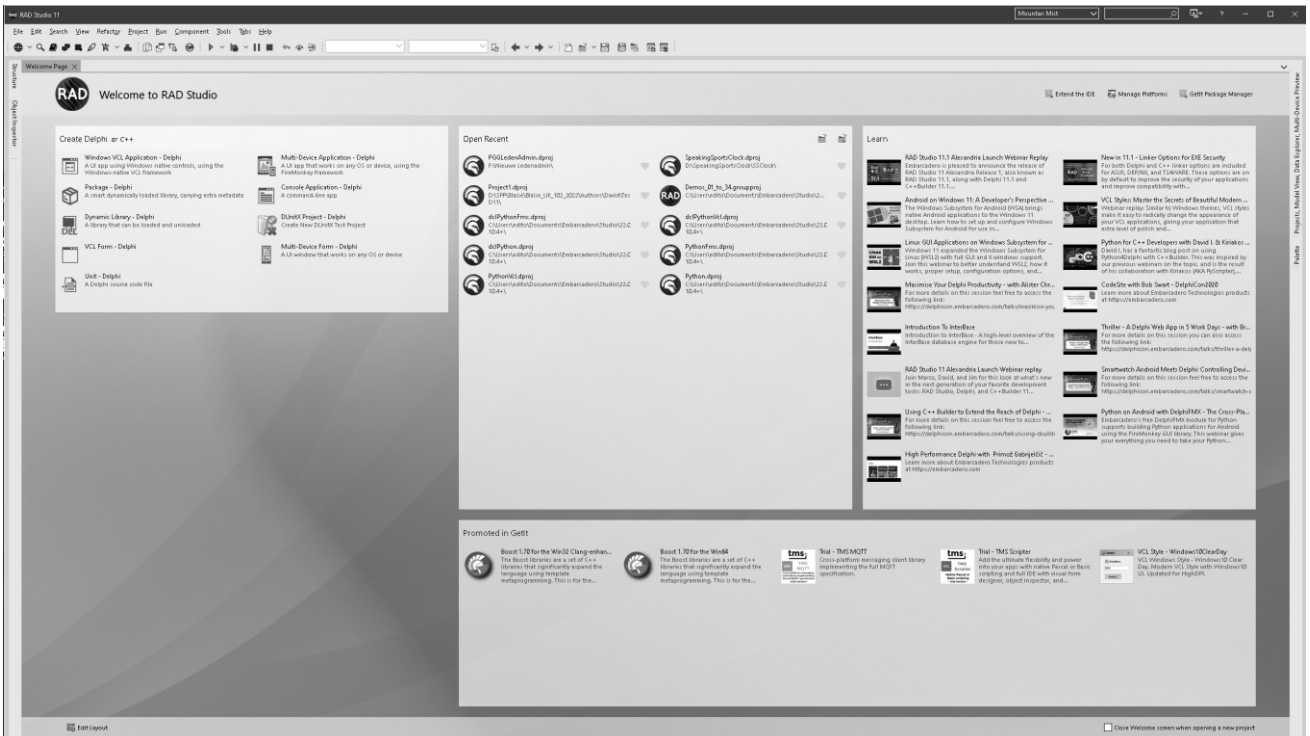Alas that needs to be done.

Figure 9: The new version progress

Here I show what is new and updated in the 11.1 version:

❶ IMPROVED IDE

Figure 10: The new version is shown for the first time

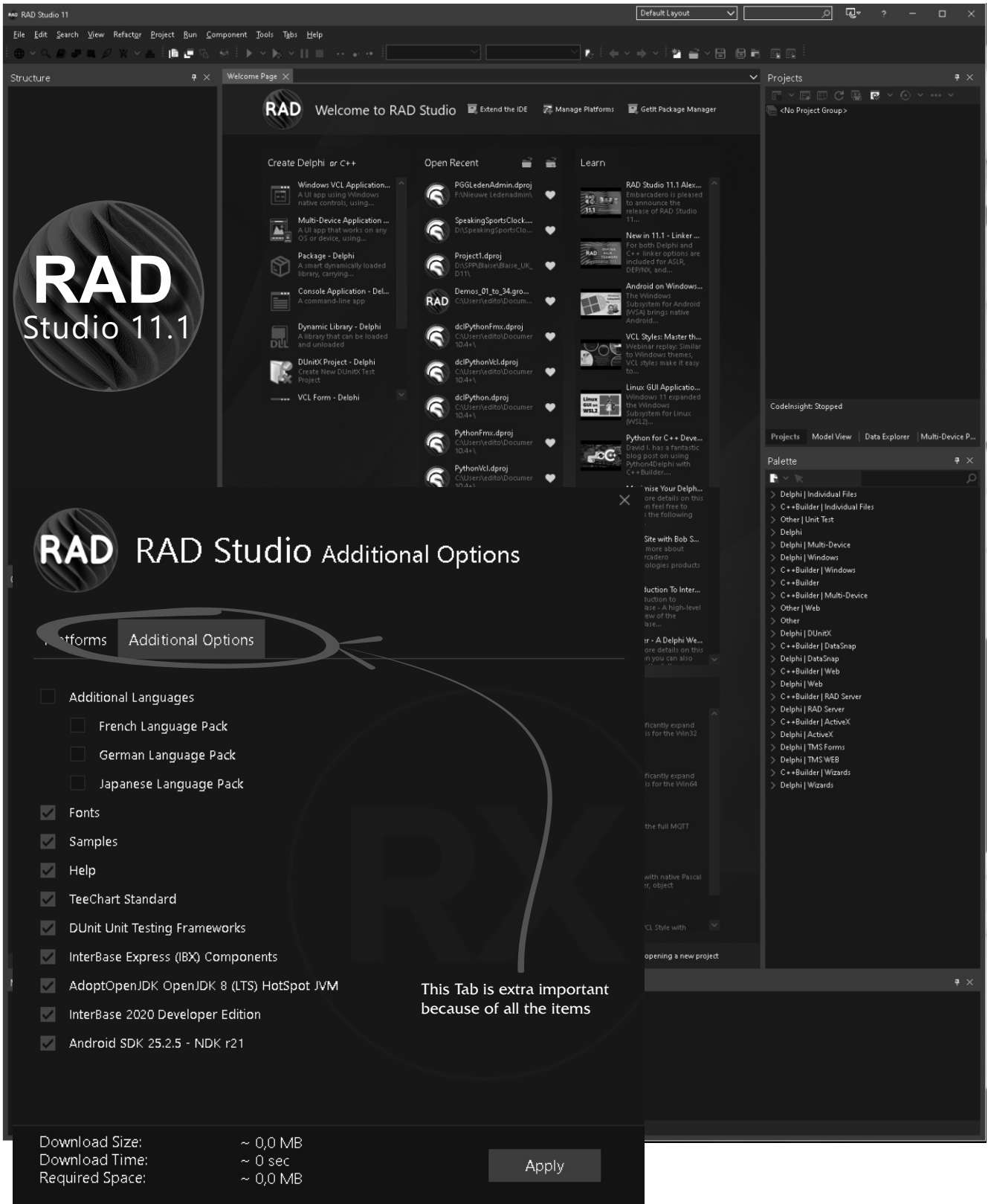Figure 11 : Make sure you have Checked these three options.

Figure12 : The dark mode and the menu showing Additional options

CODE INSIGHT

Code Insight for **Delphi,** is improved. The **Delphi LSP (Language Server Protocol architecture)** engine is greatly improved with most projects loading and **UPDATING ERROR INSIGHT** between 5 and 30 times faster. Type parameters are now visible when completing a class declaration, including **T** in a generic declaration, and showing set types.
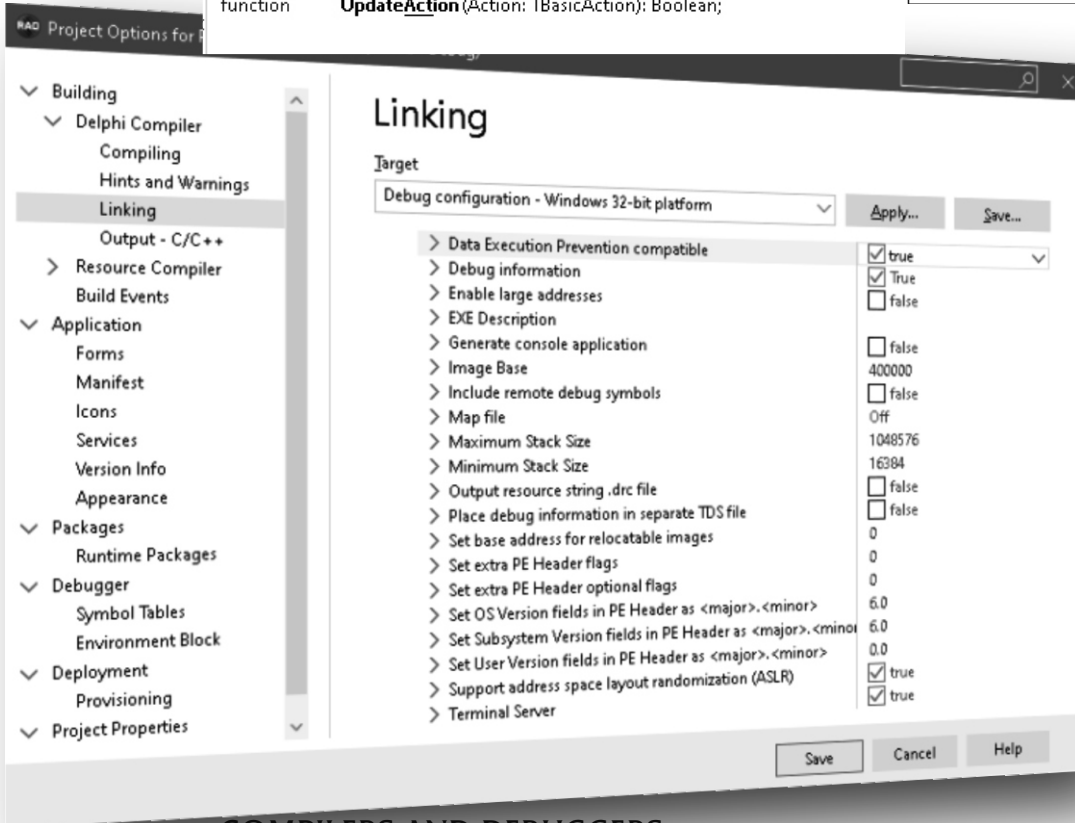
```
procedure TfrmLedenAll.ClrBtn_Expired_ListClick(Sender: TObject);
Var I: Integer;
begin {1}
  ClrBtn_Expired_List.act|
```

property	**Action**: TBasicAction;
function	**IsCustomStyleActive**: Boolean;
procedure	**InitiateAction**;
property	**ScaleFactor**: Single;
function	**ExecuteAction** (Action: TBasicAction): Boolean;
function	**UpdateAction** (Action: TBasicAction): Boolean;

ExecuteAction function
Parameters
Action
TBasicAction

Project Options for

Linking

Target

Debug configuration - Windows 32-bit platform   Apply...   Save...

> Data Execution Prevention compatible	☑ true
> Debug information	☑ True
> Enable large addresses	☐ false
> EXE Description	
> Generate console application	☐ false
> Image Base	400000
> Include remote debug symbols	☐ false
> Map file	Off
> Maximum Stack Size	1048576
> Minimum Stack Size	16384
> Output resource string .drc file	☐ false
> Place debug information in separate TDS file	☐ false
> Set base address for relocatable images	0
> Set extra PE Header flags	0
> Set extra PE Header optional flags	0
> Set OS Version fields in PE Header as <major>.<minor>	6.0
> Set Subsystem Version fields in PE Header as <mino|	6.0
> Set User Version fields in PE Header as <major>.<minor>	0.0
> Support address space layout randomization (ASLR)	☑ true
> Terminal Server	☑ true

Building
  Delphi Compiler
    Compiling
    Hints and Warnings
    Linking
    Output - C/C++
  Resource Compiler
  Build Events
Application
  Forms
  Manifest
  Icons
  Services
  Version Info
  Appearance
Packages
  Runtime Packages
Debugger
  Symbol Tables
  Environment Block
Deployment
  Provisioning
Project Properties

Save   Cancel   Help

COMPILERS AND DEBUGGERS

Improved stability and performance of **Delphi** compilers for various platforms. The **Delphi macOS 64-bit ARM** and **Android 64**-bit debuggers are now based on the **LLDB** debugger architecture, which was already in use for the **Delphi iOS** 64-bit debugger.

As a result, **Delphi** debuggers are unified on this technology for most of the supported platforms, as a way to deliver increasingly better quality over time. Moreover, there are quality improvements for a better **Delphi RTL** integration,

The LLDB Debugger (LLDB) is the debugger component of the LLVM project. It is built as a set of reusable components which extensively use existing libraries from LLVM, such as the Clang expression parser and LLVM disassembler. LLDB is free and open-source software under the University of Illinois/ NCSA Open Source License, a BSD-style permissive software license. Since v9.0.0, it was relicensed to the Apache License 2.0 with LLVM Exceptions.

RTL, UI AND DATABASE LIBRARIES

◘ There are new Optimizations and quality improvements to the core Delphi RTL in the 11.1 Release.

◘ New `TURLStream` class, a `TStream` descendant with support for `async operations`.

◘ Support for **Windows 11** and **Server 2022** in the `TOSVersion` data structure. VCL enhancements to `TTreeView`, `TRichEdit`, `TEdgeBrowser`, `TLabelledEdit` and `TNumberBox`, `flickering` and `DoubleBuffering`, **VCL** high-DPI and scaling.

◘ New Demo through **GetIt** showing the use of **WinUI 3** library in Delphi.

◘ **FireMonkey** quality improvements to **TListView,** improved **Android SDK** integration, `TWebBrowser`, `Windows` high-DPI-related issues, and performance.

◘ **FireDAC** adds Structure View integration and offers support for **MariaDB 10.6, SQLite Encrypted Edition** (SEE), and **Firebird 4** new data types.

◘ **RAD Studio 11.1** also improves **DataSnap** quality and the ability to deploy a WebBroker application on Android.

◘ **RAD Server** adds **SysAdmin** endpoints including logs handling, backups management, and database validations, plus integrated deployment for **RSLite.**
You can find some details about **RSLite** at
`https://ashleyit.com/` or a demo
`https://ashleyit.com/rs/rslite/`
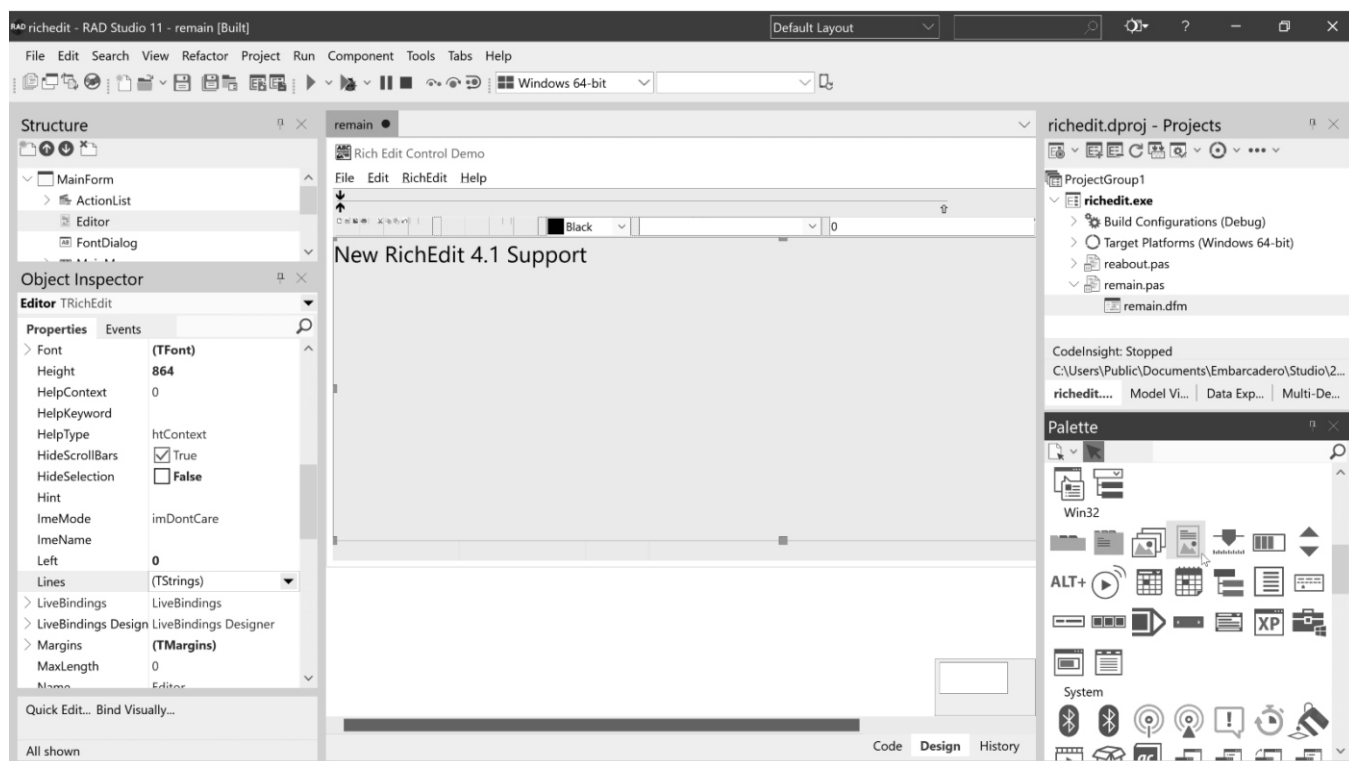
INTEGRATION / NEW PLATFORM TARGETING

Starting the **11 Alexandria** release, **Enterprise** and **Architects** users get a preview of the new **AWS (Amazon Web Service) SDK** for **Delphi** (*licensed from Appercept*), and in the future new releases are expected.

```
https://blogs.embarcadero.com/appercepts-new-aws-sdk-for-delphi-
available-with-rad-studio-and-delphi-enterprise-and-architect/
```

Customers have access to free **Delphi UI** libraries for **Python** developers, and can also use **Python** libraries in **RAD Studio** applications. **RAD Studio 11.1** delivers official support to operating systems released after 11.0 shipped: **Windows 11, macOS 12 Monterey, iOS 15, and Android 12! (***Python for Delphi will be an article in the next issue***)**

integration

targeting
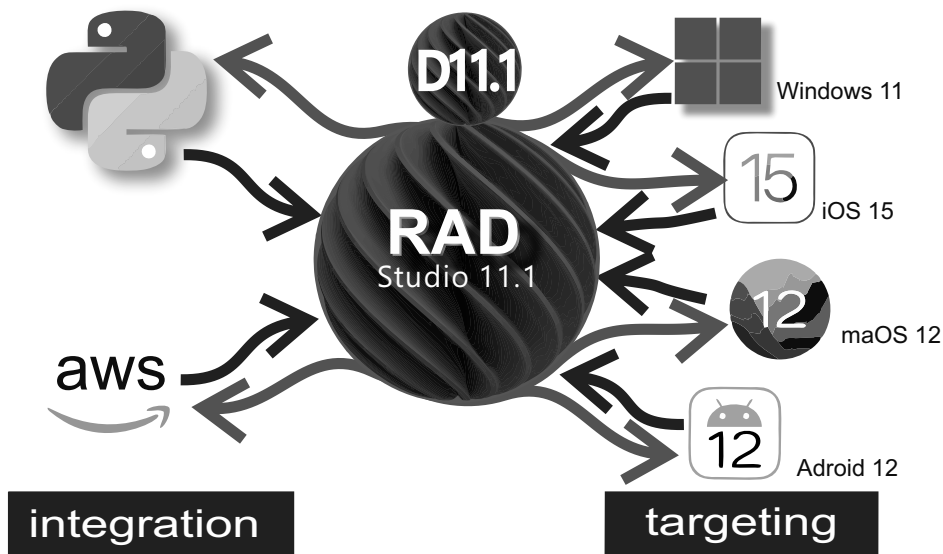
RAD ON 4K+ SCREENS!

RAD Studio 11 adds high-DPI support to the IDE, enabling
developers to work on larger, high-resolution screens. Full support for
the latest 4k+ high-resolution monitors improves daily developer activities
with cleaner, sharper fonts and icons, and high-resolution support throughout
the IDE windows, including in the VCL and FMX form designers and code editor.
FireMonkey for Windows now uses the same DP model (rather than Pixel model) of all
platforms, offering a significant enhancements of the apps rendering on Windows HighDPI
and 4K monitors.
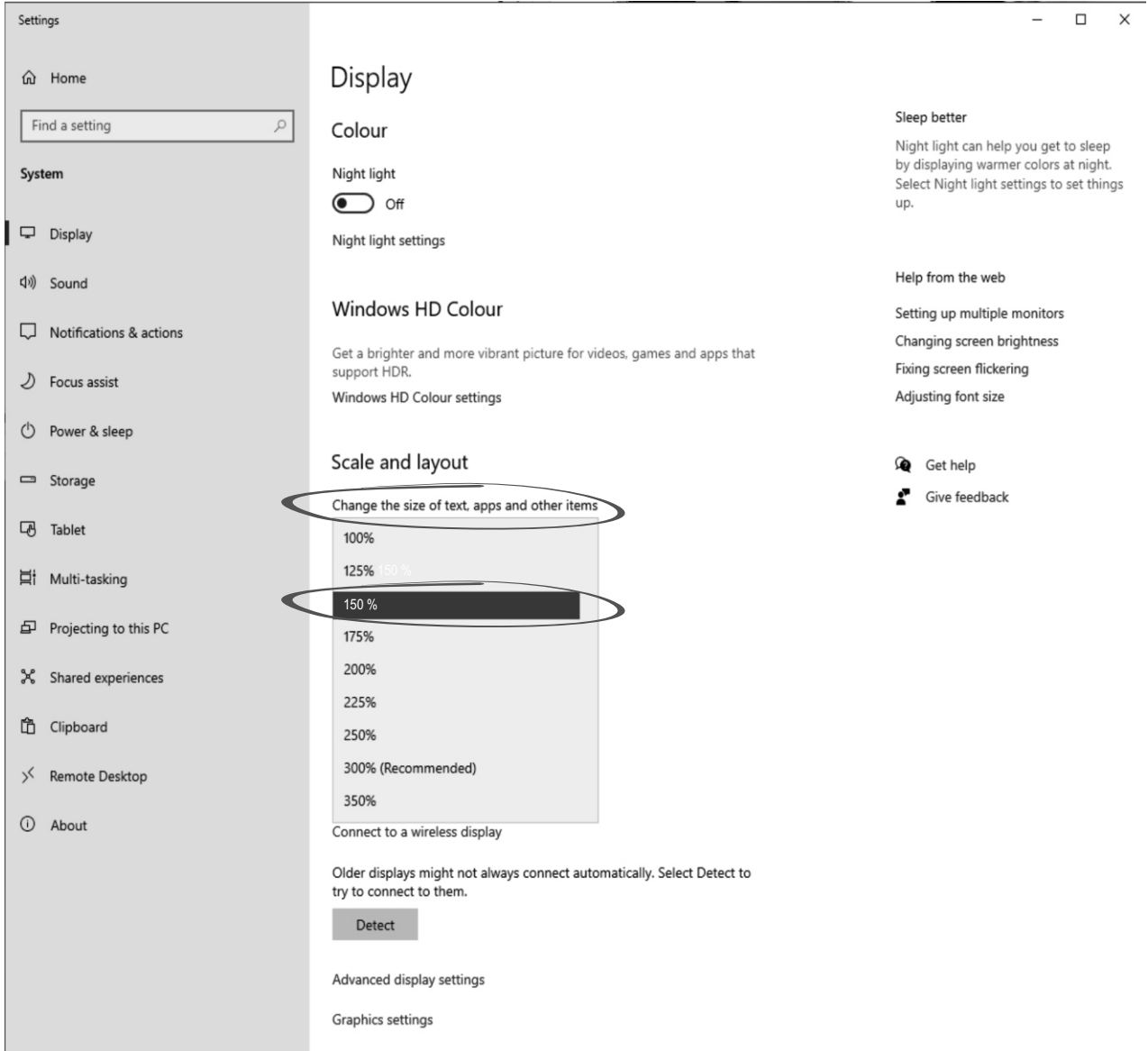
VCL STYLES AT DESIGN TIME

VCL Styles now provides design-time support:

Prototype stylish **UIs** even faster by seeing immediately at design-time how your styled forms and controls will look when running.

Viewing at design time how styles will impact the **UI** at runtime improves the design and testing process for modern **UIs.** Creating better **UIs** faster is **especially useful when working with per-control styles.**

DPI IDE creates a very clear look and design by showing during designtime wat the final product might look like.
This happens through the **VCL** styles design-time support.
You must of course not forget to enable the **VCL STYLES**

COMPILE POSSIBLE FOR ANDROID API 30
Android API and Libraries updated - **API 30, Google Play V3, Android X.**

This includes the latest billing **API** (*required by Google Play Store*). Enhanced **Delphi RTL** for **Android,** supporting for **Android API** level 30.
Support for the new "AndroidX" libraries.
In-app **purchase component** to help monetize your applications.
You can of course do without. There is an investigation from the **EEC** if this is allowed.

Android solution **Google** Play Billing Library Version 4.
Enhanced **FireMonkey Application** Platform for creating native **Android ARMv7** applications for **Android 11, 10, Pie (9.0), Oreo (8.1)**

DEVELOPMENT FOR
M-SERIES APPLE SILICON

Compile for **macOS** and eventually use the new
universal package for **AppStore** submission.
You can now compile for both existing **Intel** and new **M-series**
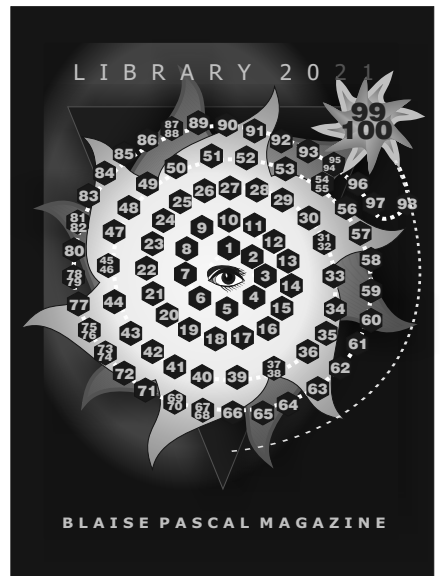**macOS** processors (**Apple Silicon**).
Compiling for the newest processor versions enables the fastest
performance across all platforms, and supports universal packaging for the
macOS app store.

With RAD Studio 11 it is possible to compile binaries for **macOS ARM.**
Since the new M1 processor is incredibly fast it is more than important to create
native apps for it.

By Michaël Van Canneyt

ABSTRACT
Lazarus evolves continuously.
Because it is an open source project, you don't need to wait for a release to be able to use the latest features. In this article we show how to compile and use the latest development version of the **Lazarus IDE.**

❶ INTRODUCTION

The **Lazarus team** keeps on developing the **Lazarus IDE** and the **LCL** (*the Lazarus Component Library*). If you are eager to use one of the new features, it is not necessary to wait for the official release of a new version of **Lazarus.**

Because **Lazarus** is an open source project, you can perfectly install the latest sources and build **Lazarus** for yourself.

The sources of **Lazarus** are available publicly on **Gitlab:**
`https://gitlab.com/freepascal.org/lazarus/lazarus`
In order to build **Lazarus** yourself, you need 2 things:

◼ AN EXISTING LAZARUS INSTALLATION.
At the moment of writing, this is version 2.2.0, sing **Free Pascal** compiler 3.2.2. In this text we assume Lazarus is installed in its default location: `C:\Lazarus`

◼ A GIT CLIENT.
This is not really a necessity, but makes life easier if you want to update **Lazarus** on a regular basis. The **Lazarus** installation has everything to build a new version of **Lazarus.** This should not come as a surprise, because the **Lazarus IDE** rebuilds itself as soon as you install a new package in the IDE. You can make do without **git,** as it is always possible to download lazarus sources in a **zip** file:
`https://gitlab.com/freepascal.or g/lazarus/lazarus/-/archive /main/lazarus-main.zip`
This URL gives you a ZIP file with the latest version of Lazarus.

SOME PRELIMINARIES

Building **Lazarus** requires you to enter some commands on the command-line:
Lazarus is built using the **GNU Make tool,** which is a command-line tool.
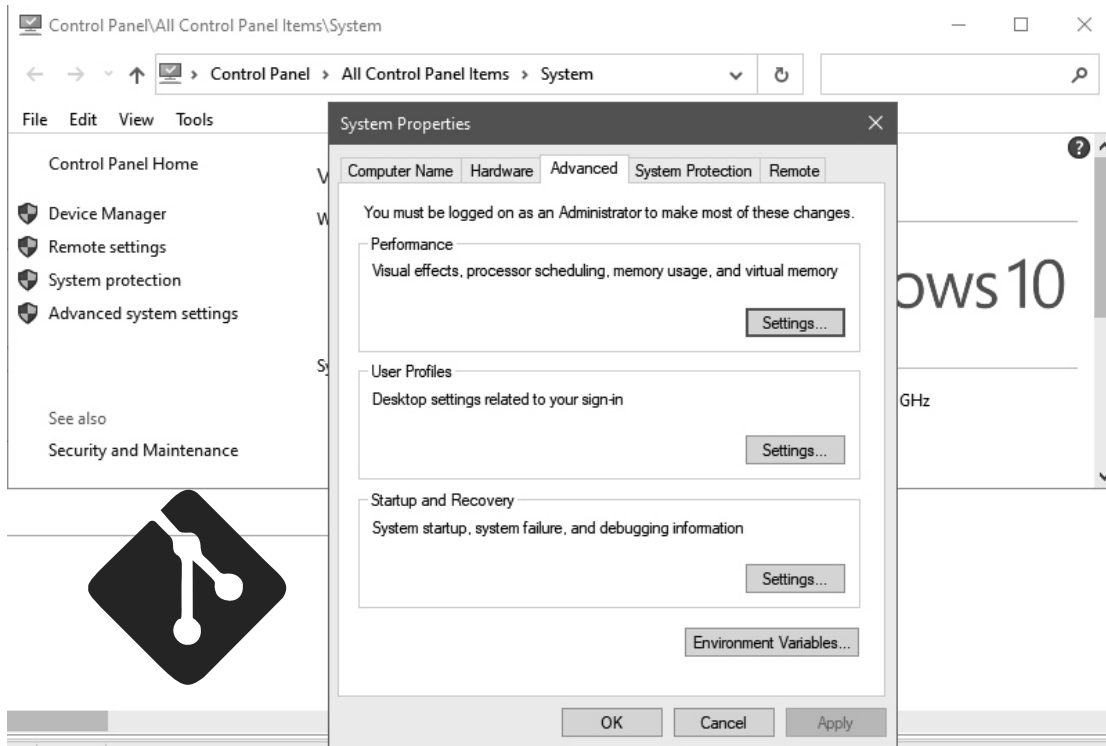
Figure 1: System control panel page with advanced settings

The tool is called `make`, and is installed together with
Free Pascal on **Windows. Linux** or **Mac** installations have a `make` tool installed by default.
To be able to use the make tool, it must be in a directory that is included in the `PATH`
environment variable. So, you must make sure this is the case, Again, on **Linux** and **Mac**
this is normally the case.

If you are on **Windows,** and have **Delphi** installed, you will also have the **Delphi make** tool
installed. It serves the same purpose as the **GNU make** tool, but has much less features.
It is therefore important that when you enter the `make` command on the command-line,
that the correct version of make is used.

During its installation procedure, **Delphi** changes the `PATH` environment variable to include
the directory with the **Delphi** version of make (*as well as the other delphi tools*).
So, it is imperative that the `PATH` environment variable must be set in such a way that the
directory with the **FPC** version of make comes before the one with the **Delphi** version of
make. **Delphi** no longer uses its make tool, so changing this will not damage the **Delphi**
installation.

To set the `PATH` variable, in the **Windows Control Panel**, choose 'System'.
In this dialog, the 'Advanced system settings' link must be used , in which case you
will see a dialog pop up which resembles *figure 1 on page 2*.

The 'Environment variables...' button in the bottom-right of that dialog allows you to set the
environment variables of Windows. There are 2 sets of variables: user-specific variables (*at the top*)
or system variables. Both will contain a `PATH` variable.
In the command-line window, both `PATH` variables will be used. The directories in the system `PATH`
variable take precedence over the ones in the user-specific `PATH` variable.
If you have **Delphi** installed, it is therefore best to change the system `PATH` variable. Select the
'`PATH`' variable, and press the 'Edit...' button. A special dialog will pop up in which the contents
of the `PATH` variable have been split into lines: one per directory, *see figure 2 on page 3.*
In this dialog the 'New' button can be used to add a new directory to the `PATH`.
The directory to add is:
`C:\lazarus\fpc\3.2.2\bin\x86_64-win64`
If you have the **Win32** version of lazzrus installed, the directory to use is:
`C:\lazarus\fpc\3.2.2\bin\i386-win32`

if you have another version of **Lazarus** (or **Free Pascal**), you may need to adapt the path.
You can use the 'Move up' and 'Move down' buttons to move the new directory before the
entry of the **Delphi IDE** (*as visible in figure 2 on page 3*).
After you confirm the new `PATH` settings with the 'OK' button, you can check that the correct
version of make is called, by entering the following command on the command-prompt:
`make -v`
The output will be something like this:
```
c:\Development\lazarus>make -v
GNU Make 3.80
Copyright (C) 2002 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty;
not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```
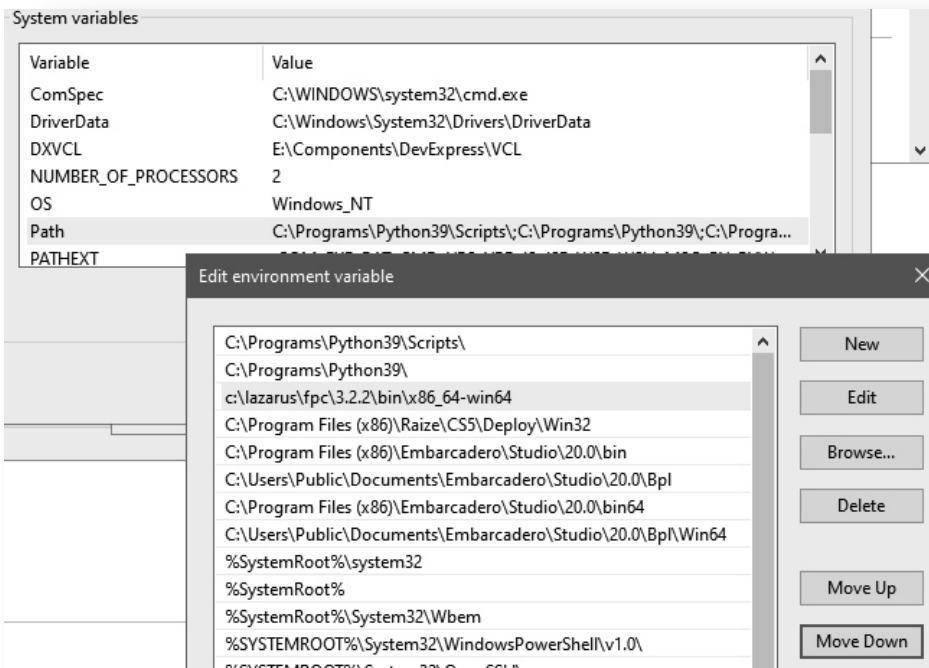

Figure 2: Environment variables dialog

DOWNLOAD USING GIT

In an earlier series of articles in
Blaise Pascal Magazine,
the installation and use of **Git** has been covered in depth. In this article we will therefore
limit the instructions to the download of Lazarus sources.
The repository can be cloned from:
`https://gitlab.com/freepascal.org/lazarus/lazarus.git`
or, if you prefer to use **SSH:**
`git@gitlab.com:freepascal.org/lazarus/lazarus.git`


```
◆ MINGW64:/c/development

Filip@DESKTOP-N3UHPTO MINGW64 /c/development
$ cd /c/development

Filip@DESKTOP-N3UHPTO MINGW64 /c/development
$ git clone https://gitlab.com/freepascal.org/lazarus/lazarus.git
Cloning into 'lazarus'...
remote: Enumerating objects: 516055, done.
remote: Counting objects: 100% (82/82), done.
remote: Compressing objects: 100% (63/63), done.
remote: Total 516055 (delta 26), reused 66 (delta 19), pack-reused 515973
Receiving objects: 100% (516055/516055), 166.50 MiB | 26.75 MiB/s, done.
Resolving deltas: 100% (426518/426518), done.
Updating files: 100% (12821/12821), done.

Filip@DESKTOP-N3UHPTO MINGW64 /c/development
$
```

Figure 3: Git clone on the command-line

We'll install the lazarus sources below a directory `C:\Development\`
Obviously, you're free to choose whatever directory you want.
If you have **Git** for **Windows** installed, then you can clone the sources with the following
command in the **Git** bash window:
`cd /c/Development`
`git clone https://gitlab.com/freepascal.org/lazarus/lazarus.git`
This is also the command you can give on **Mac** or **Linux,** and the output will look like
figure 3 on page 4. If you are using **TortoiseGit,** then you can use the context menu of the
Windows file explorer:
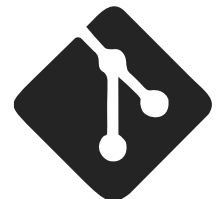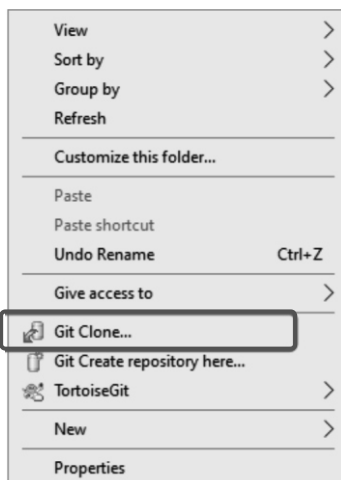
Doing so,
will show the Git clone dialog, shown in *figure 4 on
page 5*, where you can enter the URL mentioned
above. See the first article of the Using Git series
(BPM issue 97, 98 and 99/100). After the initial
clone operation, you can always update the
sources with the git pull command.

BUILDING LAZARUS

When the git clone operation is complete, **Lazarus** can be built. For this, the windows command-line windows must be used. Do not attempt to use the bash shell from your **Git** for windows installation: this build environment is not supported.

Building the **Lazarus IDE** is a matter of 2 commands:

```
cd c:\Development\Lazarus
make bigide
```

The `make bigide` will actually build **Lazarus,** together with some commonly used packages. Building **Lazarus** takes some time. The make command will also build `Startlazarus.exe` and some other tools.

When make stops running, please take a look at the output of the make command - in particular, check whether errors are displayed or not.

If not, all went well, and a `lazarus`, `startlazarus` and `lazbuild` command will be present in the build directory.
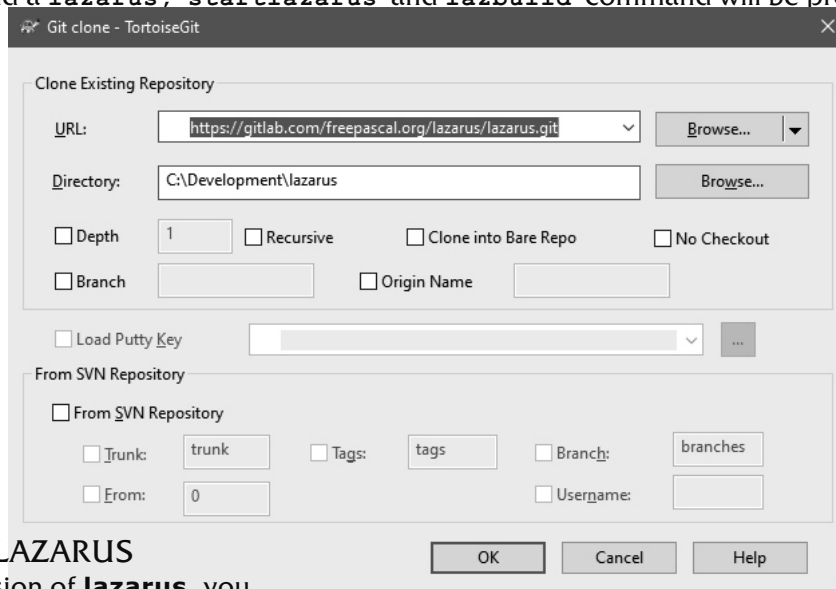
CONFIGURING LAZARUS

To start your new version of **lazarus,** you must use the newly created application binary.

You can start it in the **Explorer,** but it is of course easier to create a shortcut on the desktop: in the **File explorer**, simply drag the lazarus executable to the desktop while keeping the Alt key pressed. (*Or use the context menu 'New - shortcut' in the fle explorer*).

When you first start the new **Lazarus**, you may get some dialogs in which **Lazarus** tells you that the settings have changed: *see figure 5 on page 6 and figure 6 on page 6.*
If you wish to use two separate configurations for your installed lazarus and the newly compiled **Lazarus,** you should cancel here, and adapt the shortcut so it contains the commandline option -pcp indicated in *figure 6 on page* 6, for example:

```
--pcp=C:\test_lazarus\configs
```

You can of course choose any directory you want for the configuration.
When you did all this, you will probably still get the Lazarus installation check-up dialog shown in *figure 7 on page 7.* In particular, the **GDB (gnu debugger)** location will be missing. You can reuse the one from the original lazarus installation:

```
C:\lazarus\mingw\x86_64-win64\bin\gdb.exe
```

To ensure that you are now really working with the latest lazarus, you can check the **Help - About Lazarus dialog.** It should display the latest version number, which is 2.3.0 at the time of writing of this article, as can be seen in *figure 8 on page 7.*

Local Disk (C:) › Development › lazarus

Name	Date modified	Type	Size
languages	1		
lazarus.app	1		
lcl	1		
packager	1		
startlazarus.app	1		
test	1		
tools	1		
units	1		
.gitattributes	1		
.gitignore	1		
COPYING.GPL.txt	1		
COPYING.LGPL.txt	1		
COPYING.modifiedLGPL.txt	1		
COPYING.txt	1		
fpmake.pp	1		
fpmake_add.inc	1		
fpmake_proc.inc	1		
lazarus.exe	1		
lazbuild.exe	1		
localize.bat	1		
localize.sh	1		
Makefile	1		
Makefile.fpc	1		
README.md	1		
startlazarus.exe	1		

Incorrect configuration directory found ✕

⚠ Welcome to Lazarus.
The IDE configuration found was previously used by another installation of Lazarus.
If you have two or more separate installations of Lazarus, they should not share the same configuration. This may lead to conflicts and your Lazarus installations may become unusable.

If you have only one installation and copied or moved the Lazarus executable, then you may upgrade this configuration.
If you want to use two different Lazarus versions you must start the second Lazarus with the command line parameter primary-config-path or pcp.

For example:
C:\Development\lazarus\startlazarus.exe --pcp=C:\test_lazarus\configs

Choose:

* Update info: Use this configuration and update it for being used with this Lazarus in future. The old installation will no longer use this.
* Ignore: Use this configuration but keep the warning. This may lead to conflicts with the other installation.
* Abort: Exit now. You can then fix the problem by starting this Lazarus with the correct configuration.

Additional information:
This configuration is at: C:\Users\Filip\AppData\Local\lazarus
It belongs to the Lazarus installation at: C:\lazarus\lazarus.exe
The current IDE was started from: C:\Development\lazarus\lazarus.exe

[Update info]   [Ignore]   [Abort]

Figure 5: Starting a new Lazarus version for the first time

Upgrade configuration ✕

❓ Welcome to Lazarus 2.3.0

There is already a configuration from version 2.2.0 in
C:\Users\Filip\AppData\Local\lazarus
The old configuration will be upgraded.

If you want to use two different Lazarus versions you must start the second Lazarus with the command line parameter primary-config-path or pcp.

For example:
C:\Development\lazarus\startlazarus.exe
--pcp=C:\test_lazarus\configs

[Upgrade]   [Abort]
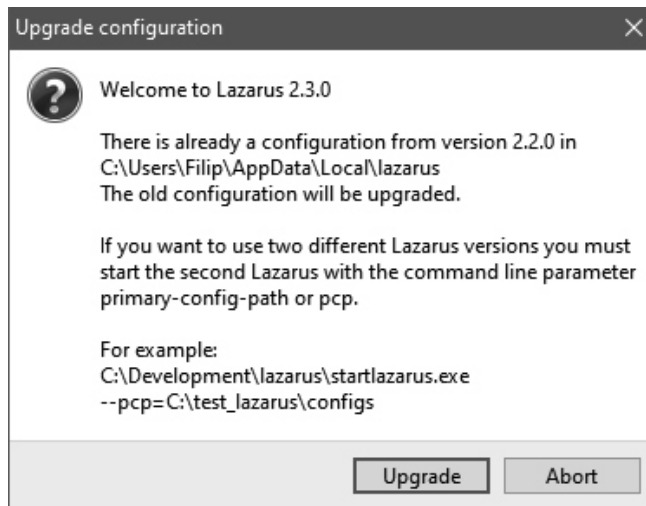
Figure 6: Creating a new confiuration or not
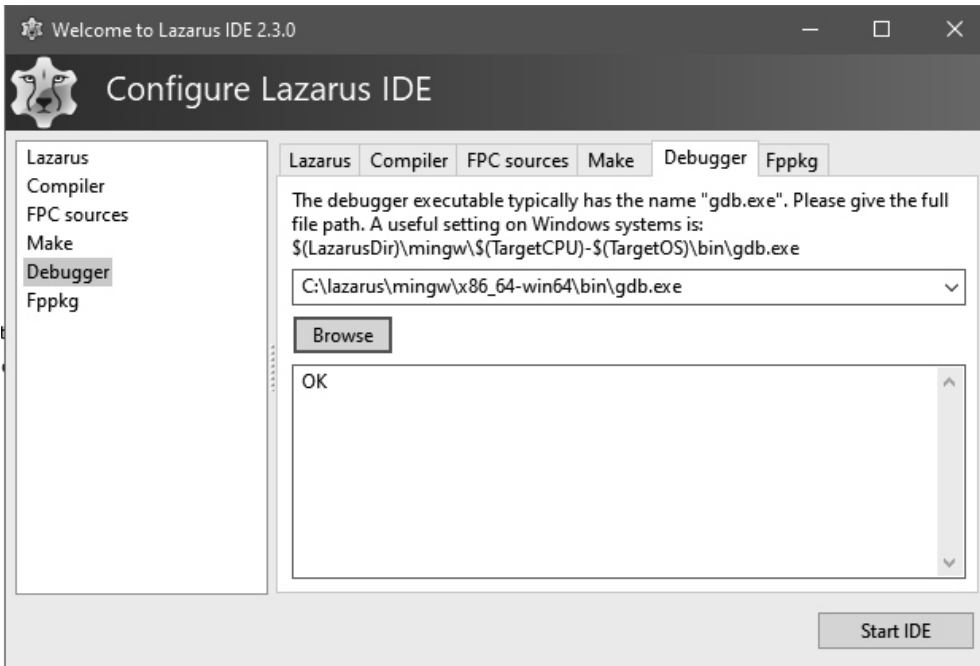
Figure 7: Lazarus start-up check-up result

Figure 8: 'About Lazarus' version check

CONCLUSION

Lazarus is an open source tool. This means you do not have to wait for the latest version to
be released. Instead, in this article we have demonstrated how you can build your own
version of Lazarus:
this should be within reach for every Object Pascal developer, regardless of the level of expertise. . .

41

By David Dirkse

Figure 1: The Cats Eye
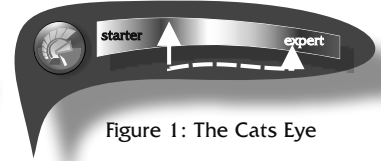
INTRODUCTION

David has created the so called **Droste** effect program:
Just for fun! The **Droste** effect is nothing but a picture in a
picture in a picture etc. Its nice to how this is handled and as an
extra you get some nice extra controls for free. So he created the
Droste-Effect-App. So thank David Dirkse and if your interested buy
his book about **Computer, Math & Games and Graphics**.
Available at:

`https://www.blaisepascalmagazine.eu/`
`product/books-computer-graphics-math-games-download-pdf/`

SOME SHORT STORY ABOUT THE "DROSTE EFFECT"

The **Droste effect**, known in art as an example of **mise en abyme**

(*ranslation placement at the escutcheon's center: depiction of the escutcheon itself*

within an „escutcheon": image within an image : story within a story.),

is the effect of a picture recursively appearing within itself,
in a place where a similar picture would realistically be expected to
appear.
This produces a loop which mathematically could go on forever,
but in practice only continues as far as the image's resolution
allows. The effect is named after a Dutch brand of cocoa, with an
image designed in 1904.
It has since been used in the packaging of a variety of products.
Apart from advertising, the **Droste effect** is displayed on the
tins and boxes of **Droste** cocoa powder which displayed a
nurse carrying a serving tray with a cup of hot chocolate and
a box with the same image, designed by **Jan Misset**. The
effect has been a motif, too, for the cover of many comic
books, where it was especially popular in the 1940s.

Mathematics

The appearance is recursive: the smaller version
contains an even smaller version of the picture,
and so on. Only in theory could this go on
forever, as fractals do; practically, it continues
only as long as the resolution of the picture
allows, which is relatively short, since each
iteration geometrically reduces the
picture's size.

load/save file to/from disc

load image from disc in PIP area

clear image

frame around image

reload image

copy image to PIP area

select PIP area

reset PIP dimensions

PIP (Picture in Picture) project
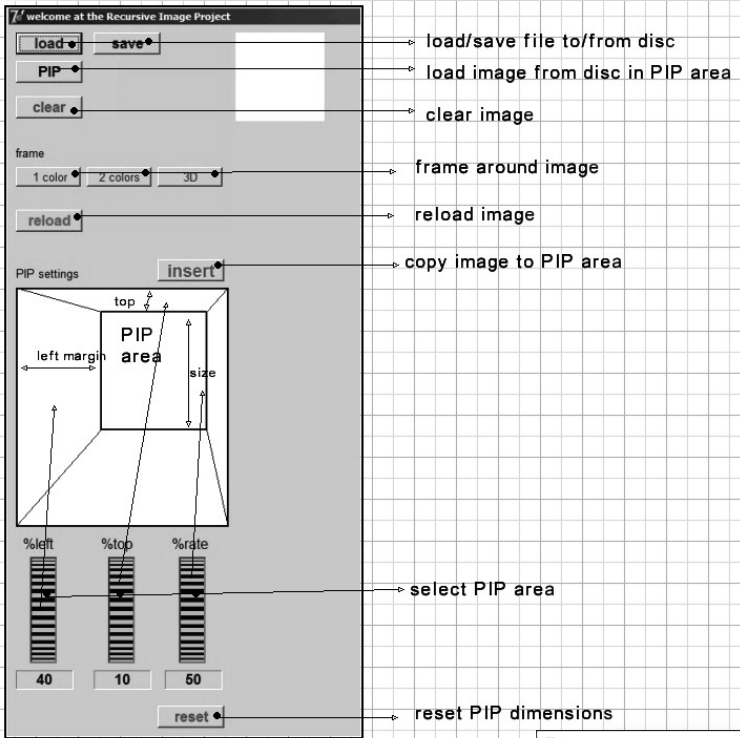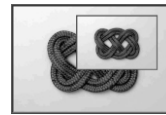
This is all create in Delphi 7.
Take a look and see the
 difference on the next page.

At the top you see the functionality
of the app. Its all very easy.
Of course if you want to see the
code used for this: as a subscriber
you have the projects (D7+D11.1)
available.
It is originaly written in **Delphi 7,**
but I recreated it in **Delphi 11.1
(Alexandria).**
As you can see on page 3 of this
article I used the new
VCL style sheet and chose the
"Ruby Graphite". In the next issue
I' ll explain how it works.
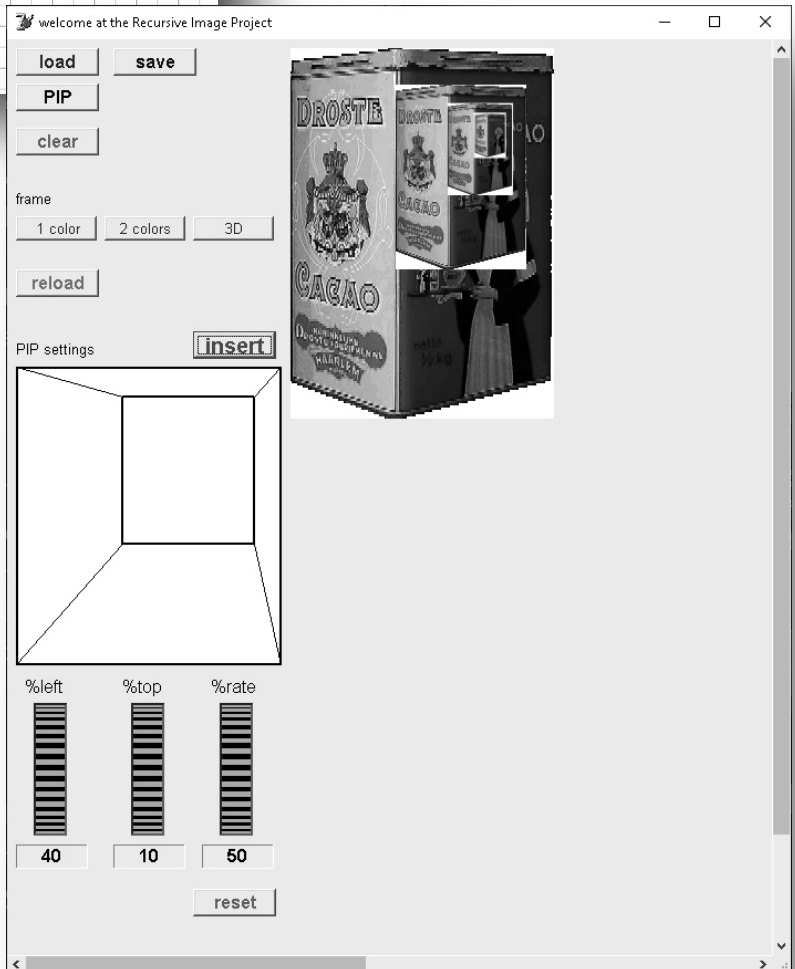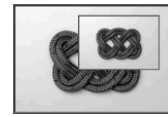
COMPONENTS TO INSTALL:
- **Rotation button** (separate component)
  and a group of 5 components:

- **Colormixer**
- **ColorPicker**
- **ElBox**
- **DavTimer**
- **Arraybtn**

The components are available for subscribers of
course and in an earlier version (Issue 65, page 20)
I explained how to install the components.
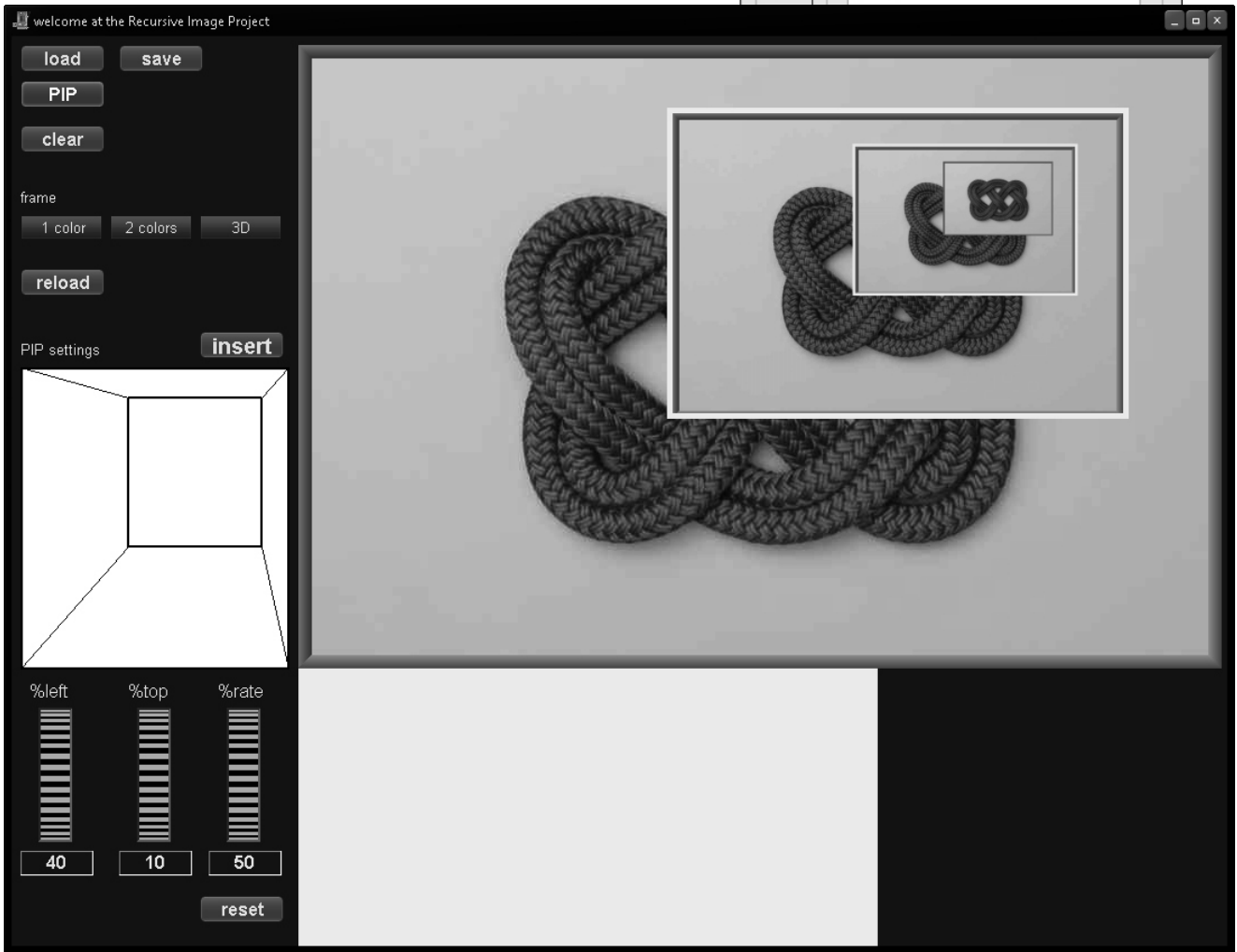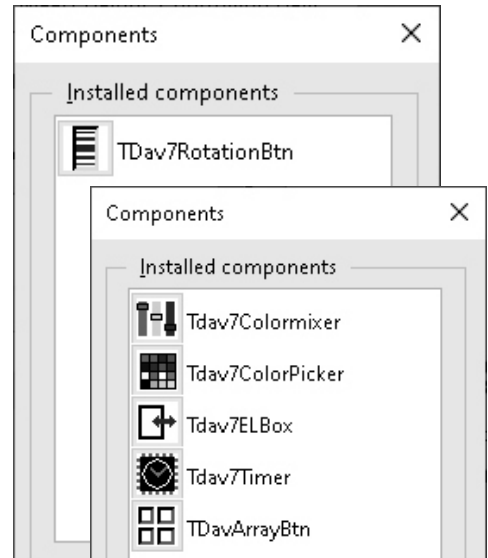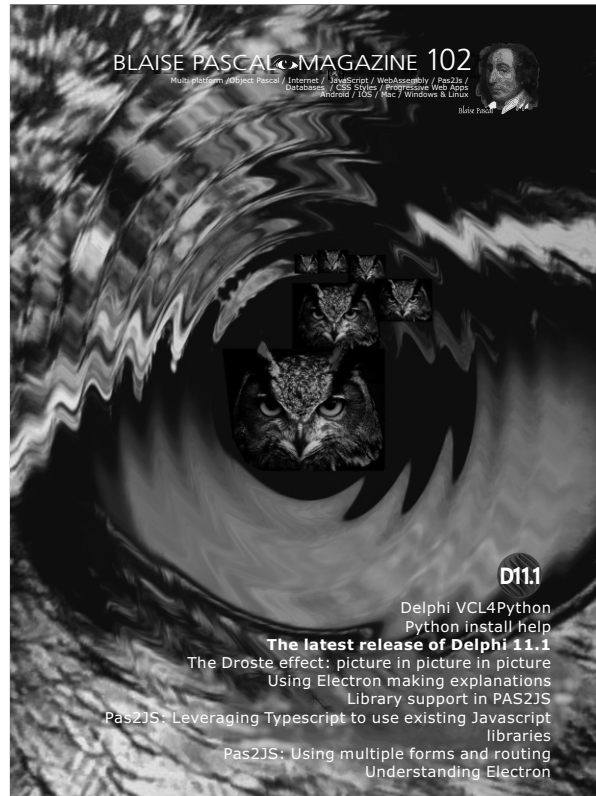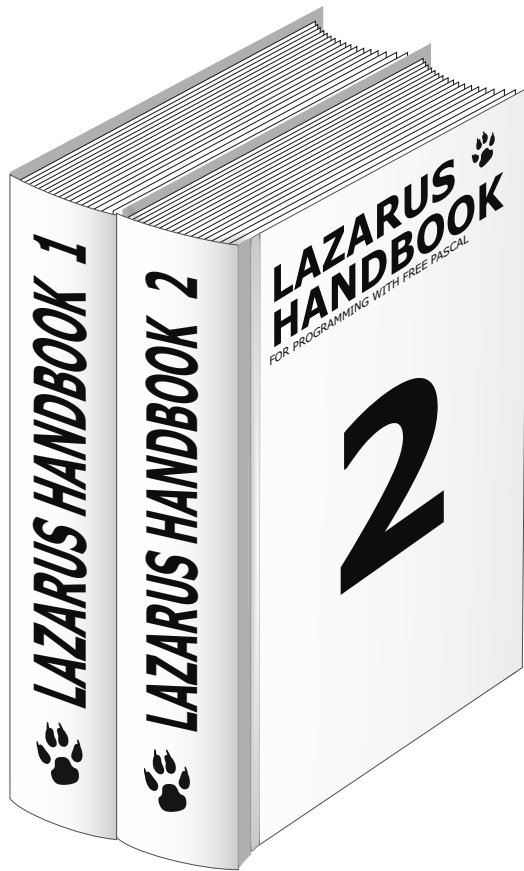
LEVERAGING TYPESCRIPT DECLARATIONS IN PAS2JS

By Michael van Canneyt

TS

❶ INTRODUCTION

To say that there are a lot of free **Javascript** libraries or frameworks out there is an understatement.

Normally, any **Javascript** class or function can be used in **PAS2JS:** By falling back on assembler blocks, any Javascript function can be called. The transpiler will happily insert any **Javascript** in your final transpiled code. But if the transpiler has external declarations for the **Javascript** classes or functions, the transpiler can and will check your code against the definitions it has.

Plain **Javascript** has a major drawaback: it is not typesafe. To remedy this, people at **Microsoft** created **TypeScript:** a type system for **Javascript.** It is a superset of **Javascript,** which is transpiled to **Javascript.**
(*One of the authors of TypeScript was also one of the creators of Delphi*)
This type system is made popular by **Angular** and other large **Javascript** frameworks. People writing **TypeScript** code face the same problem as **Pas2JS** users: how to make use of the many **Javascript** libraries, and still write **Typesafe** code?
The answer to this problem are declaration modules (*files with extension .d.ts*): these modules do not implement any functionality. They just describe the **API** offered by an external **Javascript** library. The **TypeScript** compiler reads this declaration and uses it to validate the **TypeScript** code that makes use of the **Javascript** library: It serves exactly the same purpose as a **PAS2JS** unit with external classes.
Many plain **Javascript** libraries offer such a **TypeScript** declaration module in their distribution.
But there are also a lot of libraries that do not offer such a declaration module.
Because there are a lot of **TypeScript** programmers, there is an ongoing effort to describe these **JAVASCRIPT** libraries: the DefinitelyType d repository on **Github.**

ABSTRACT

PAS2JS: contains a tool to convert **TypeScript** declaration modules to a pascal unit with external class definitions. This can be used to create import units for many **Javascript** libraries. In this article, we show how to use this tool.

It is available at:
`https://github.com/DefinitelyTyped/DefinitelyTyped/`
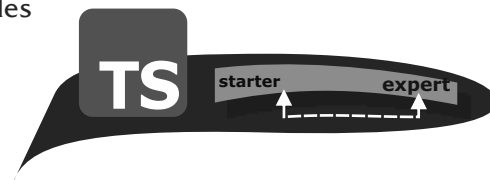
It contains many tens of thousands of declaration modules. **TypeScript** programmers that wish to use a **javascript** library can just check out this repository and use the declaration module of the package they wish to use in their project. PAS2JS could use a similar repository of import units. Indeed, ideally, the **TypeScript** declaration modules can just be re-used so all the hard work of all these **TypeScript** would benefit the **PAS2JS** users as well.

Fortunately, this is possible to a certain extent:
The upcoming version of pas2js comes with a tool that converts a **TypeScript** declaration module to a pascal unit with external definitions: `dtstopas`.
Better yet, an online service exists which makes this possible today.
Last but not least, the tool and the webservice have been integrated in the **Lazarus IDE.**
You can create an import unit directly in your project from within the **Lazarus IDE:**
Simply use the File-New menu item.
We'll discuss each of these possibilities in turn.

LEVERAGING TYPESCRIPT DECLARATIONS IN PAS2JS

TS

2 DTS2PAS

The **dts2pas** tool is a small command-line tool which will transform a *.d.ts file to a
pascal unit. Running it without options (or option -h) gives the following output:

```
Usage: dts2pas [options]
Where options is one or mote of:
-a --alias=ALIAS           Define type aliases (option can be speficied multiple times)
                           where ALIAS is one of
                           a comma-separated list of Alias=TypeName values
                           a @FILE : list is read from FILENAME, one line per alias
-h --help                  Display this help text
-i --input=FILENAME        Parse .d.ts file FILENAME
-l --link=FILENAME         add {$linklib FILENAME} statement. (option can be specified -o -
-output=FILENAME           Output unit in file FILENAME
-s --setting=SETTINGS      Set options. SETTINGS is a comma-separated list of the following
                           coRaw
                           coGenericArrays
                           coUseNativeTypeAliases
                           coLocalArgumentTypes
                           coUntypedTuples
                           coDynamicTuples
                           coExternalConst
                           coExpandUnionTypeArgs
                           coaddOptionsToheader
                           coInterfaceAsClass (*)
                           coSkipImportStatements
                           Names marked with (*) are set in the default.
-u --unit=NAME             Set output unitname
-w --web                   Add web unit to uses, define type aliases for web unit
-x --extra-units=UNITLIST  Add units (comma-separated list of unit names) to uses
                           This option can be specified multiple times.
```

From this output we can see the minimal operation options are:
`dts2pas -i 7zip-min/index.d.ts -o 7zip.pp`
This will run the declaration conversion on the file `7zip-min/index.d.ts` and will write the
resulting pascal file to `7zip.pp`
This is what the declaration input file looks like:

```
export function unpack(pathToArchive: string,
                       whereToUnpack: string,
                       errorCallback: (err: any) => void): void;
export function unpack(pathToArchive: string,
                       errorCallback: (err: any) => void): void;
export function pack(pathToDirOrFile: string,
                     pathToArchive: string,
                     errorCallback: (err: any) => void): void;
export function    list(pathToArchive: string,
                          callback: (err: any, result: Result[]) => void): void;
export function        cmd(command: string[],
                       errorCallback: err: any) => void): void;
export interface Result {
      name: string;
      date: string;
      time: string;
      attr: string;
      size: string;
      compressed: string;
}
```

LEVERAGING TYPESCRIPT DECLARATIONS IN PAS2JS

TS

And this is what the tool produces as output
(*lines have been formatted for better readability*):

```
Unit _7zip;

{$MODE ObjFPC}
{$H+}
{$modeswitch externalclass}

interface

uses SysUtils, JS;

{$INTERFACES CORBA}
Type
   // Forward class definitions
   TResult = Class;
   Tunpack_errorCallback = Procedure (err : JSValue);
   // Ignoring duplicate type Tunpack_errorCallback (errorCallback)
   Tpack_errorCallback = Procedure (err : JSValue);
   Tlist_callback = Procedure (err : JSValue; result : array of TResult);
   Tcmd_errorCallback = Procedure (err : JSValue);
   TResult = class external name 'Object' (TJSObject)
       name : string;
       date : string;
       time : string;
       attr : string;
       size : string;
       compressed : string;
end;

Procedure cmd(command : array of string;
      errorCallback   : Tcmd_errorCallback);
      external name 'cmd';

Procedure list(pathToArchive : string;
      callback : Tlist_callback);
      external name 'list';

Procedure pack(pathToDirOrFile : string;
      pathToArchive : string;
      errorCallback : Tpack_errorCallback);
      external name 'pack';

Procedure unpack(pathToArchive : string;
      whereToUnpack : string;
      errorCallback : Tunpack_errorCallback);
      external name 'unpack'; overload;

Procedure unpack(pathToArchive : string;
      errorCallback : Tunpack_errorCallback);
      external name 'unpack'; overload;

implementation
end.
```

LEVERAGING TYPESCRIPT DECLARATIONS IN PAS2JS

Some things to note:
- types are prepended with T: **Javascript** is case sensitive, and often you will encounter variables with the same name as a type, but with different casing - a class name usually starts with a capital. To avoid name clashes, the tool prepends a T to type names.
- The tool correctly spots overloaded versions and marks them as such.
- The tool creates auxiliary types for complex function argument types.
- The special **any** type is replaced with **JSValue.**
- The **JS** unit is automatically used.

The resulting file can be compiled as-is:

```
> pas2js 7zip.pp
Info: 11458 lines in 6 files compiled, 0.3 secs
```

The **dts2pas** tool has several options, we'll explain them here (*using the long version of each option*):

alias This can be used to define type aliases. Aliases can be specified in 2 ways:
❶ As a comma-separated list of **Name=Alias** pairs:
   `--alias=AType=MyType`
   This will replace every occurence of the **AType** in the declaration file with **MyType**

❷ Using a @ character, a filename to load a list of **Name=Alias** pairs (*one per line*):
   `--alias=@MyAliasFile.lst`
   This will read file **MyAliasFile.lst**. Each line of the file must contain a pair.

help   Display a help text
input   as seen, this is used to specify the input file to parse.
link   with an argument **FILENAME** will insert a linklib statement:

```
{$linklib FILENAME}
```

When using the resulting unit, this will insert an import statement in the final Javascript:

```
import FILENAME from "FILENAME";
```

output   with an argument **FILENAME** sets the output filename.

setting   with an argument **SETTINGS** sets various conversion options, they are discussed below. Names marked with (*) are set in the default

unit   with an argument **NAME** sets the output unitname to **NAME**. When not specified, it is deduced from the output filename. .

web   Adds the web unit to the uses clause and defines type aliases for all web unit classes: this unit is part of pas2js and contains definitions of all classes exposed by the browser.
extra-units with an argument **UNITLIST** will add the units in **UNITLIST** (*a comma separated list of unit names*) to the uses clause. Some TypeScript modules depend on other modules using import statements: the **dts2pas** tool will not recursively translate these other modules, but if you have translated them already, this option can be used to add the converted unit names

LEVERAGING TYPESCRIPT DECLARATIONS IN PAS2JS

TS

The **setting** argument accepts a comma-separated list of named flags that influence the conversion process and the generated code. When translating **TypeScript** to **Pascal,** sometimes choices must be made because some **TypeScript** structures to not translate oneon-one to pascal constructs. Many of these choices are controlled using the flags which you can specify in the settings option. Here is an overview:

`coRaw` This will not generate a unit header or implementation section. You can use this to generate an include file.

`coGenericArrays`
Instead of using array of Type for array types, the converter tool will write arrays as `TArray<Type>`. There is no functional difference in **PAS2JS** between the 2 declarations.

`coUseNativeTypeAliases`
This will translate some basic types such as long to Integer.

`coLocalArgumentTypes`
If auxiliary types are generated for methods, these will be generated in a Type section within the class, for example:

```
type
    TSomeClass = Class
    Public
       Type
           TMyMethod_B_Array = Array of integer;
       Function MyMethod(B : TMyMethod_B_Array) : Integer;
end;
```

The default behaviour is to generate a global type with the class name prepended:

```
type
    TSomeClass_MyMethod_B_Array = Array of integer;
    TSomeClass = Class
    Public
       Function MyMethod(B : TSomeClass_MyMethod_B_Array) : Integer;
    end;
```
`coUntypedTuples`
A tuple* in **TypeScript** is a fixed-length array of values. If the `dts2pas` tool can determine the type of the element, it will generate a typed array:
(In mathematics, a tuple is a finite ordered list (sequence) of elements.)

```
    Type
       TSomeTuple = array[1..3] of string;
```

If this flag is set, the array element will be untyped (type JSValue):

```
    Type
       TSomeTuple = array[1..3] of JSValue;
```

`coDynamicTuples`
A tuple in TypeScript is a fixed-length array of values. The `dts2pas` tool will declare the type with the same number of elements. However, javascript allows you to specify less elements than in the definition of the tuple. To accomodate for this, using this flag you can let the converter generate a dynamic array:
```
    Type
            TSomeTuple = array of string;
```

LEVERAGING TYPESCRIPT DECLARATIONS IN PAS2JS

coExternalConst
A constant in a TypeScript declaration will be translated literally. For example:

```
const myConst = "Hello, World";
```
Is translated as:
```
const
  myConst = 'Hello, World';
```

This means the constant is duplicated in the pascal code. Using the flag
coExternalConst, the constant is translated as a reference instead:

```
const
  myConst : String; external name 'myConst';
```

coExpandUnionTypeArgs
A variable of union type in **TypeScript** can have one of the
possible types in the union type. This cannot be expressed in Pascal, so the default
behaviour is to replace this with the **JSValue** catch-all type:

```
function func (a : string | number) : int;
```

is translated to **Pascal** as

```
function func (a : jsvalue) : integer;
```

With the **coExpandUnionTypeArgs** switch, for function or method arguments
with union type, the converter will create overloaded versions for each type. In the
above example this means the following declarations are produced:

```
function func (a : string) : integer; overload;
function func (a : double) : integer; overload;
```

coaddOptionsToheader
If this switch is present, the converter will insert a comment with
the used conversion options to the unit header. If the unit needs to be regenerated,
the options used to create the original are available.

coInterfaceAsClass
TypeScript knows interface definitions. The standard behaviour of the dts2pas tool is to
translate this to an interface definition. With this switch,
the interface will be declared as a **Pascal** class.

coSkipImportStatements
Any import statements in a **TypeScript** module are written to
the converted pasal file as comments. With this option, these comments are not
generated.

❸ THE WEB-BASED SERVICE

On the **Free Pascal** server, a **(cgi)** web service exists that can be used to translate any file from the **DefinitelyTyped** repository to a **Pascal** unit. The service is located at

```
https://www.freepascal.org/~michael/service/dts2pas.cgi
```

On the server, the **DefinitelyTyped** repository is checked out, and is updated daily. By specifying a file name (relative to the types directory in the repository), the service outputs the translated unit. Using the following URL

```
https://www.freepascal.org/~michael/service/dts2pas.cgi/
convert/?file=7zip-min/index.d.ts&unit=7zip
```

(*the line has been split for readability*) you will get the same file as in the result above. The following query variables are accepted, they have the same meaning as their commandline counterparts.

file	the file to convert, relative to the types directory in the **DefinitelyTyped** repository
unit	the unit name to use.
aliases	Aliases to to use, using the same format as the command-line tool.
extraunits	Extra units to add to the command-line tool.
prependlog	Insert conversion log as comments in the source.
flagname=1	Switch on any of the flags mentioned earlier.

You can get a list of files available for conversion, one per line:

```
https://www.freepascal.org/~michael/service/dts2pas.cgi/list?raw=1
```

By leaving out the `raw=1` the output is a **Javascript** array variable definition. The latter option is used in a small web page, shown in figure 1 on page 7:
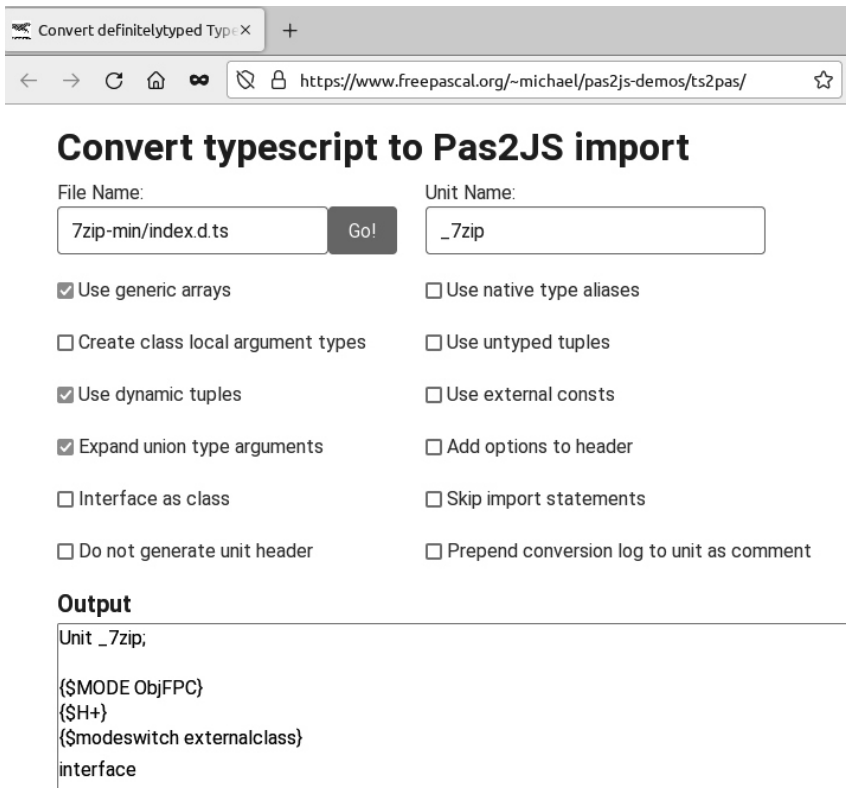
Figure 1: dts2pas web front-end

```
https://www.freepascal.org/~michael/pas2js-demos/ts2pas/
```

LEVERAGING TYPESCRIPT DECLARATIONS IN PAS2JS

TS

This page (*obviously written in pas2js*) is a simple front-end to the service. The service and front-end page will still be extended to provide more options, such as entering aliases or uploading a **TypeScript** file to convert.

❹ INTEGRATION IN THE LAZARUS IDE

Both the web-based service as the command-line tool have been integrated in the **Lazarus IDE:** using the File-New menu, you can directly convert a **TypeScript** file and make the resulting pascal file part of your program, *see figure 2 on page 8*.

When clicked, a small wizard pops up which allows you to select a description file from disk, or you can opt to use the web-based service:
enter the name of a module - a list of matching files will be presented as soon as you enter 2 characters: *see figure 3 on page 9*.
On the same tab, you can enter extra units, aliases and indicate that the web unit must be used - together with all known aliases: basically the same options as available in the web interface or command-line.

The second tab (*figure 4 on page 9 of the wizard page*) allows you to specify the conversion settings (*or flags*).
When done, you can click OK, and the IDE will create a new unit, part of your **PAS2JS** project, containing the converted **TypeScript** declaration module.
If all goes well, it is ready to use, as *seen in figure 5 on page 10*
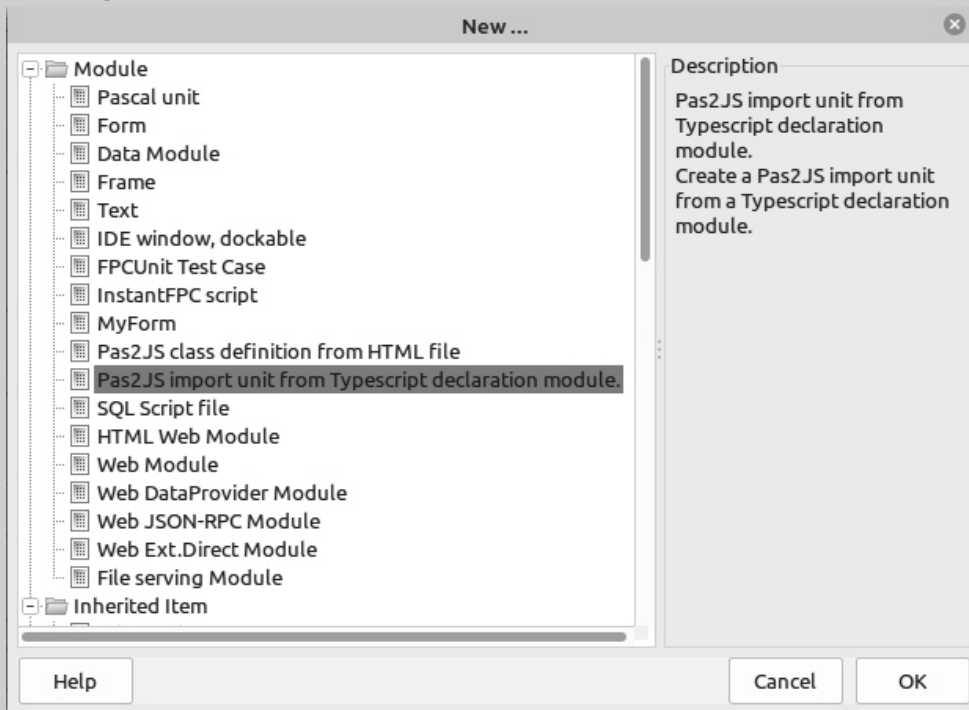
Figure 2: The File-New entry to import
a TypeScript descripton file

LEVERAGING TYPESCRIPT DECLARATIONS IN PAS2JS

Input | Code generation

○ Local file

Declaration File | Select a .d.ts file

⦿ FPC DefinitelyTyped Service

Module name | capture-c ▽

capture-console/index.d.ts

Extra units

web, console

☑ Use Web unit and aliases

Aliases

Class	Alias
Console	TJSConsole
_	TJSLODash

Cancel | OK
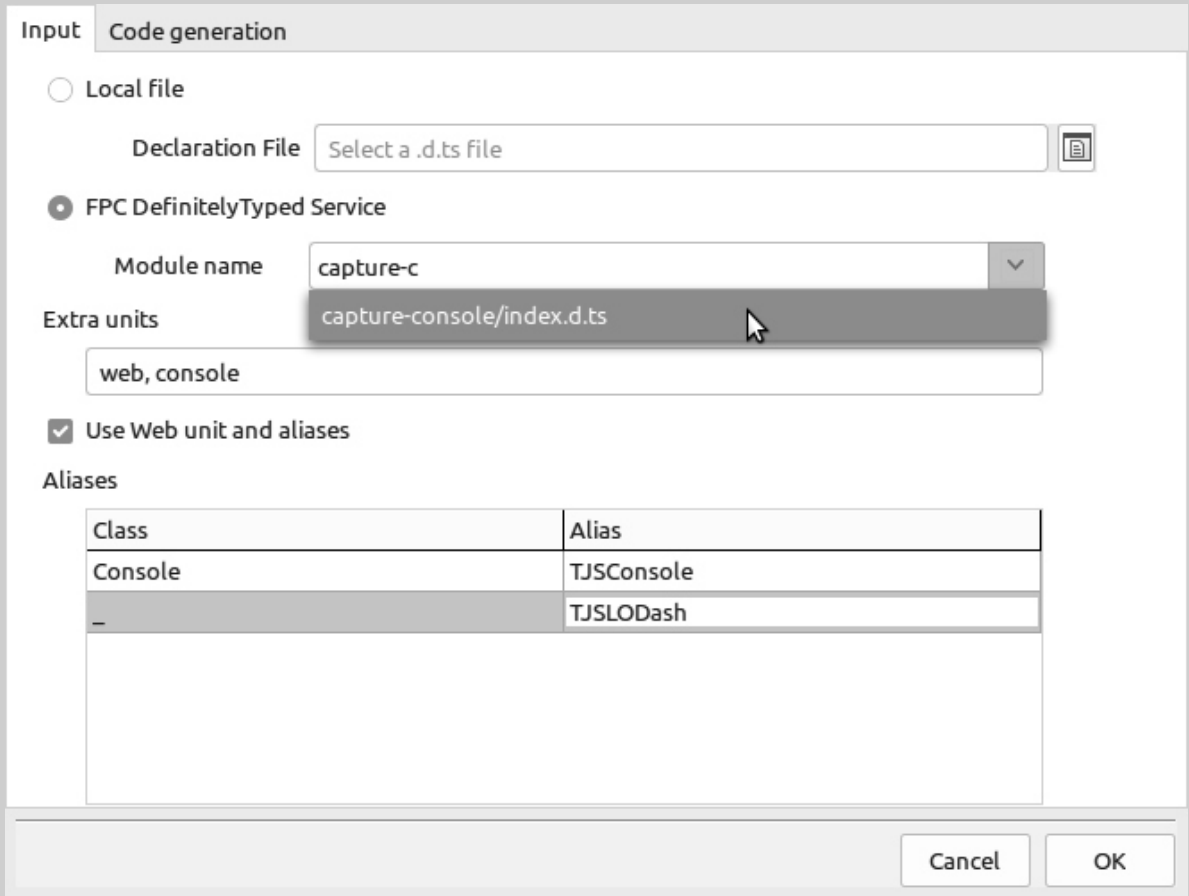
Figure 3: Selecting a TypeScript file or module

Convert typescript module declaration to pascal unit — ⤢ ✕

Input | Code generation

Options

☐ Raw code (no unit)   ☐ Generic Arrays

☑ Use native type aliases   ☐ Use class local argument types

☐ Untyped tuples   ☑ Dynamic tuples

☐ External consts   ☑ Expand union-typed arguments

☑ Add used options to header   ☑ Generate Interfaces as classes

☐ Skip Import statements

Cancel | OK

Figure 4: Setting the conversion flags

Figure 5: The converted result is part of your project

❺ CONCLUSION

At the time of writing this article, the **DefinitelyTyped** contained well over 36.000 declaration files. Theoretically, these can now all be used in **PAS2JS** You may wonder why the converted units are not made part of the **PAS2JS** repository. The answer to this question is twofold:

◼ The archive evolves continously: the **PAS2JS** units would be outdated almost daily.
◼ The conversion is not always perfect:
sometimes some manual work is needed to fix the generated unit.

Javascript and **Typescript** have a lot of idioms which do not always translate well to **Pascal.** What is more, the declaration files are sometimes 'messy' – despite being more strict than **Javascript, TypeScript** still leaves a lot of room for interpretation and the translator sometimes simply cannot translate correctly what is being defined in **TypeScript:**

Different people may have used different methods to describe the same **Javascript** interface and some descriptions may translate better to pascal than others.
Some declarations are simply outdated:
TypeScript has evolved, but the declaration files have not been updated accordingly.
The **Javascript/Typescript** parser included in **Free Pascal** is not perfect either:

IT TRANSLATES WELL OVER 99% OF THE FILES IN DEFINITELYTYPED, BUT NOT 100%.

Still, using the tool does most of the work for you. Even if some manual work is involved, the amount of work that you must still do will be negligable compared to writing the import units manually.
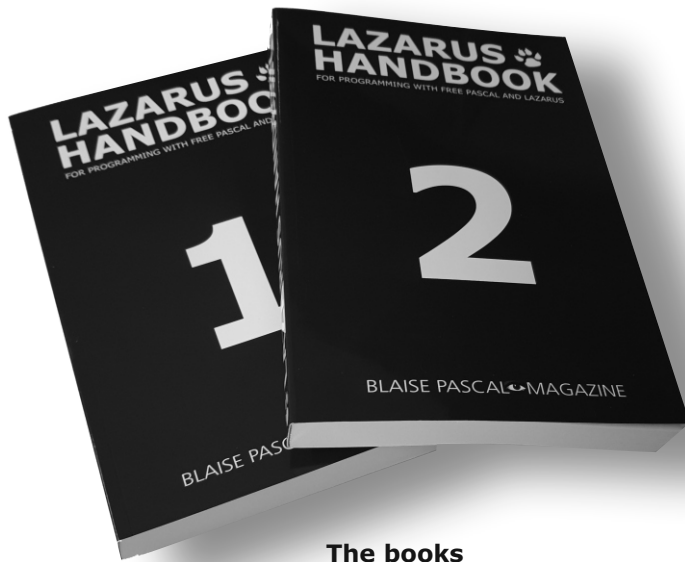
The books

Sewn POCKET ,
almost thousand pages
written by the makers of FPC and Lazarus

934 Pages in two books **50 €** (euro)

+ +
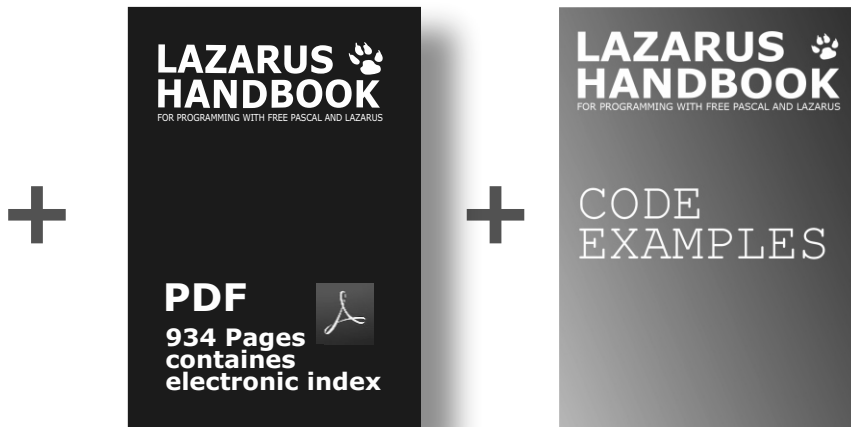
Including the PDF, INDEX and Code Examples

LIBRARY SUPPORT IN PAS2✓JS
BY MICHAËL VAN CANNEYT

D11

starter ————→ expert

PAS2✓JS & LIBRARIES

ABSTRACT

This article is meant to show a new feature: With version 2.2, **PAS2JS** introduces library support in the compiler. Libraries in **PAS2JS** translate to **Javascript** modules: independent blocks of **Javascript** code which must be explicitly imported in another block. In this article we show to use them.

❶ INTRODUCTION

For the experienced pascal programmer, using libraries is not uncommon. Till recently, using libraries in **PAS2JS** was not possible.
With release 2.2 (released on 22-02-2022) of **PAS2JS** libraries can now also be used in **PAS2JS**

For the pascal programmer, libraries - (**DLL**s in **Windows**) - are independent programs which export certain functions and variables.
In **Javascript,** a similar concept exists: **Modules.**

Modules can import symbols from other modules, and can export symbols to other modules.
It is therefore natural to transpile a **Pascal** library to a **Javascript** module, and this is now what can be done with **PAS2JS**
◘ Import symbols from a **module.**
◘ Create a **module** that exports functions and variables.

In contrast to **Delphi,** no special precautions are needed for using strings or classes in a **PAS2JS** library: in particular, there is no need to enable a module to use shared memory.

2 JAVASCRIPT MODULES

Javascript modules are nothing but **Javascript** files which export a number of symbols, but which otherwise do not share any code or namespaces. Especially the latter is important. By default, if you link 2 **Javascript** scripts to a **HTML** page:

```
<script src="script1.js"></script>
<script src="script2.js"></script>
```

then the code script1 has access to all symbols (variables, functions etc.) of `script2`, and vice versa. This means they can modify or even annihilate each others' working.
With Javascript modules, this is not the case. Take the following HTML snippet:

```
<script type="module"
src="script1.js"></script>
<script type="module"
src="script2.js"></script>
<script src="script3.js"></script>
```

Here script1, `script2` and `script3` have distinct namespaces. They do not interfere with each other: both script1 and `script2` can have a variable `MyVar`, but each has a local copy of this variable. If `script2` writes to `MyVar`, it will only modify its own copy.
What is more, `script3` has no access to the symbols defined in `script1` and `script2`.
Only modules can import symbols of other modules. Imagine `script1.js` has the following content

```
export const MyText1 = "Hello,";
export const MyText2 = " World!";
```

As you can see, it exports 2 constants, **MyText1** and **MyText2**.
This means `script2.js` can use these constants as follows:

```
import { MyText1, MyText2 } from "./script1.js";
document.title = MyText1+MyText2;
```

When loading `script2,` the browser will also automatically load `script1.js,` there is no need to include it explicitly in the **HTML** file. The file script1 must of course exist in the specified location.
In contrast with `script2, script3` can never access the symbols, because it is not a module itself. Only modules can import and export symbols.
It is of course possible to share some symbols between modules and non-modules by attaching them to a global symbol such as the window.

❸ IMPORTING LIBRARIES

To import symbols from a module (written in Pascal or not) 2 things are needed:
- ◘ a `linklib` directive:

    `{$linklib ./my-file.js myfile}`
    this will be transformed to the following Javascript
    `import * as myfile from "./my-file.js";`

    Javascript supports some more fine-grained import statements, but these are not yet supported in Pas2JS. The myfile name is optional, in which case the filename without path or extension will be used.

- ◘ an external declaration for each function or variable exported by the module (the declaration has been split over 2 lines for readability):

    ```
    Function MyFunction (S: String) : Integer;
       external name 'myfile.myFunction';
    var
       MyVar : String; external name 'myfile.myVar';
    ```

    Note that these external names contain the prefix 'myfile.' as part of their name: this is because all symbols of the module are available as `myfile.NNN`, due to the way the import statement is constructed from the `{$Linklib }` directive.

Note that the use of the `{$Linklib }` directive also requires the use of the module compiler target. More about this later.

❹ WRITING LIBRARIES

To write a library using **Pas2JS,** you can write a library just as you would in **Delphi** or **Free Pascal,** using the library keyword, instead of the default program keyword:

```
library htmlutils;
  {$mode objfpc}
  // Your exports Here
  // exports a, b, c;
begin
  // Your library initialization code
here
```

However, this is not enough. A new transpiler target was introduced: module.
The reason for introducing a new target is the following:

Depending on the target, the transpiler will include a **Pas2JS** `rtl.run();` statement at the
end of the generated Javascript (or not). The output for the **nodejs** target includes such a
statement, but the browser target does not - because, as a rule, the `rtl.run()` statement is
included in the `.html` file: this will ensure that **HTML** tags and their ids have been processed by
the browser before the program is run.

Since a library *(or module)* can be used both in **node.js** and in the browser, a new target has
been created: module. This target will always include the `rtl.run()` statement.
The `{$Linklib}` directive also requires the use of the module target.
No import statement will be generated, unless the target is set to module.

❺ CREATING JAVASCRIPT MODULES USING PASCAL

So, how to use libraries and `{$Linklib}` directives to create Javascript modules?
We will demonstrate this with an example.
We create a library that allows to clear the **HTML** page below a certain tag (*identified by
it's id attribute*), and which allows to set the page title. This is quite simple:

```
library htmlutils;

uses web;

Var   DefaultClearID : String;

Procedure SetPageTitle(aTitle : String);
begin
    Document.Title:=aTitle;
end;

Procedure ClearPage(aBelowID : String);
Var
    EL : TJSElement;
begin
    if (aBelowID='') then aBelowID:=DefaultClearID;
    if (aBelowID='') then el:=Document.body
    else
        el:=Document.getElementById(aBelowID);
    if Assigned(El) then El.innerHTML:='';
end;

exports
    DefaultClearID, SetPageTitle, ClearPage;

end.
```
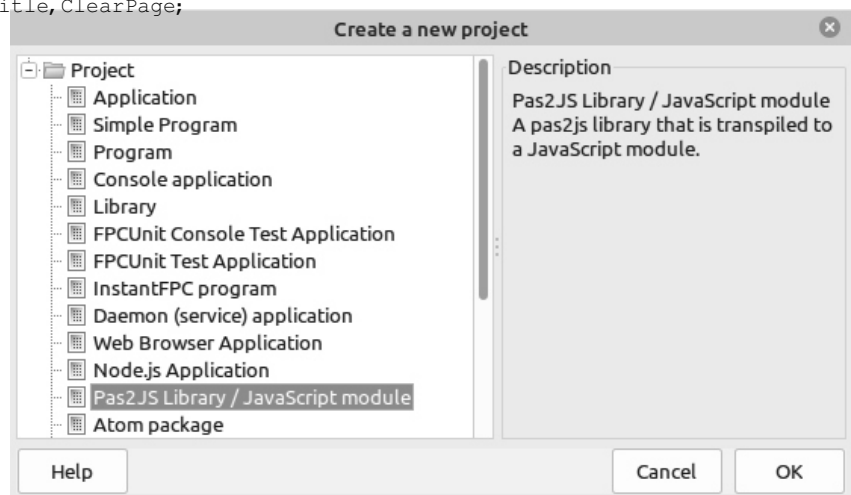

Figure 1: Pas2JS module support in the Lazarus new project menu

To demonstrate the export of variables, we also export a variable `DefaultClearID`.
The value of this variable is used to determine which **HTML** tag to clear. If it is not set,
and no tag ID was specified in the call to `ClearPage`, the whole **HTML** body element is cleared.

This library can be compiled with the `-Tmodule` target:

```
/home/michael/bin/Pas2JS -Tmodule -Jirtl.js -Jc htmlutils.pas
```

The **Lazarus IDE** has support for creating a **Pas2JS** library in the `Project-New project`
menu, which will set all necessary options, as can be seen in *figure 1 on page 3.*
As indicated earlier, to use a library (or module), we must use again a **Javascript** module:
only **javascript** modules can use other modules. To create this module, we have 2 options:
◘   Create another library.
◘   Create a program.
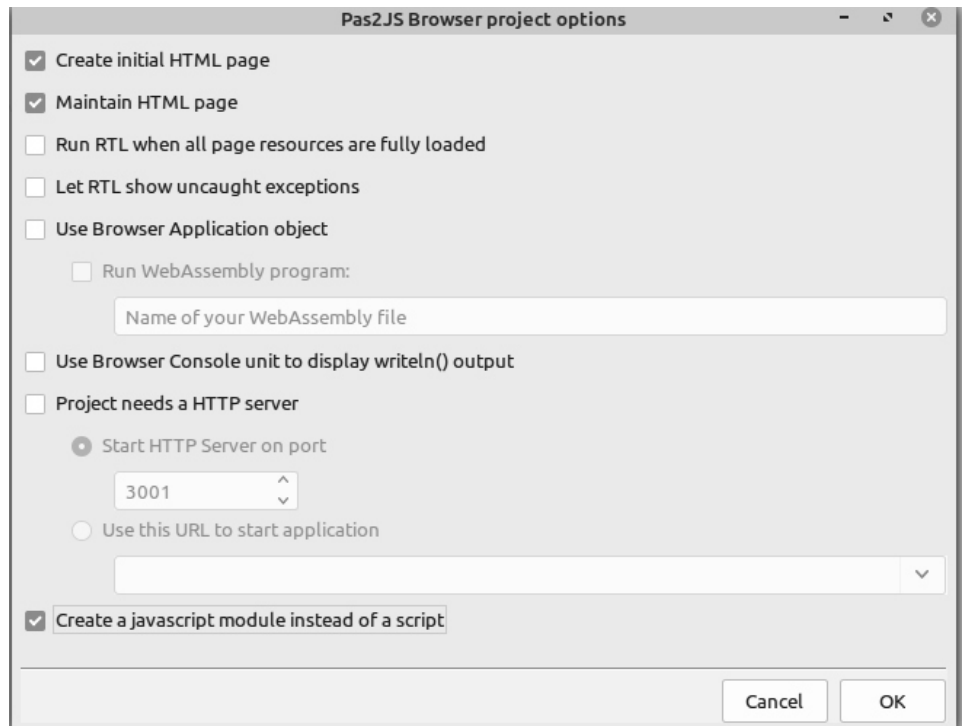
Figure 2: Pas2JS program as module support in the Lazarus new project menu

It is clear why a library will work: the **Javascript**
script will need to have the `module` type and must be compiled with
the module target. However, a program will also work.

From the **Javascript** point of view, there is no difference between a library and a program.
From a **Pascal** point of view, the only factor of importantce is whether you want to export
symbols from your module. If you do, then you must create a library.

For demonstration purposes, we'll create a program, because the **Lazarus IDE** wizard then
also creates a **HTML** page which we will need to show the functionality of our library.
In older versions of **Lazarus** the **TargetOS** of our program must manually be set to module in the
compiler options. In the latest (*trunk, hence not yet released*) version, the
Project - New project dialog already offers an option which does this for you,
see figure 2 on page 4.

We start by creating all code that is needed to import the library:

```
program htmlutilsdemo;

{$mode objfpc}
{$linklib ./htmlutils.js utils}

uses
  Web;
Procedure SetPageTitle(aTitle : String);
  external name 'utils.SetPageTitle';
Procedure ClearPage(aBelowID : String);
  external name 'utils.ClearPage';
var
  DefaultClearID : string;
      external name 'utils.vars.DefaultClearID';
```

Note the `utils.vars.DefaultClearID`: the prefix vars is needed for all variables
exported by a **Pas2JS**-created library.
To use these routines, we create a HTML page with 2 edits (IDs `edtTitle`, `edtBelowID`)
and a checkbox (ID `cbUserDefaultClearID`) and 2 buttons (IDs `btnSetTitle`
and `btnClear`) . These edits can be used to specify a page title and an element ID,
the `onclick` event handlers of the buttons will call our imported functions.
The element definitions are bound to the **HTML** tags in the `BindElements` function:

```
Var
   BtnSetTitle,BtnClear : TJSHTMLButtonElement;
   edtTitle,edtBelowID,cbUseDefaultClearID : TJSHTMLInputElement;
Procedure BindElements;

begin
   TJSElement(BtnSetTitle):=Document.getElementById('btnSetTitle');
   BtnSetTitle.OnClick:=@DoSetTitle;
   TJSElement(BtnClear):=Document.getElementById('btnClear');
   BtnClear.onclick:=@DoClear;
   TJSElement(edtTitle):=Document.getElementById('edtTitle');
   TJSElement(edtBelowID):=Document.getElementById('edtBelowID');
   TJSElement(cbUseDefaultClearID):= Document.getElementById('cbUseDefaultClearID');
end;
```

The `BindElements` function is called in the program startup code.
The `DoSetTitle` and `DoClear` methods are callbacks that will call our imported function:

```
function DoSetTitle(aEvent: TJSMouseEvent): boolean;
begin
  Result:=False;
  SetPageTitle(edtTitle.Value);
end;
```

The `DoClear` function is a little longer, since it must take into account the value of the `cbUseDefaultClearID` element:

```
function DoClear(aEvent: TJSMouseEvent): boolean;
begin
  Result:=False;
  if cbUseDefaultClearID.Checked then
    begin
      DefaultClearID:=edtBelowID.value;
      ClearPage('');
    end
  else
    begin
      DefaultClearID:='';
      ClearPage(edtBelowID.value);
    end;
end;
```
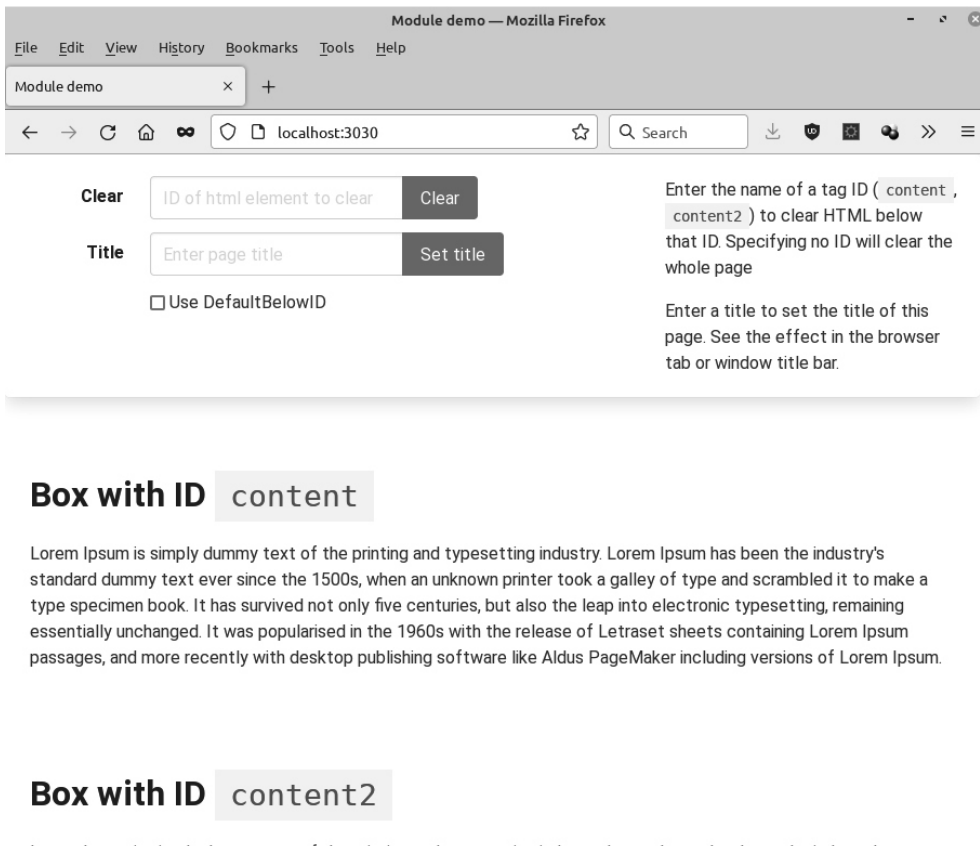

Figure 3: Our page in action

The HTML will not be presented here, except to show that the script tag must be modified, the type of the script must be set to module:
```
<script type="module" src="htmlutilsdemo.js"></script>
```
(again, in the latest development version of Lazarus, this is already done for you).
The resulting HTML page can be seen in see *figure 3 on page 6*, it is available online at
```
https://www.freepascal.org/~michael/pas2js-demos/modules/htmlutils/
```

6 EXPORTING CLASSES

In the **exports** statement only variables and functions can be specified. Despite this
restriction, it is possible to use classes which are exported from libraries.
The simplest way to do so is to create a function that creates an instance of a class.
Alternatively, for global instances, you can declare a variable of type of the desired class.
To demonstrate this, we'll rewrite our example to use a class called **THTMLUtils**:

```pascal
library htmlutils;

uses web;

Type
  THTMLUtils = class(TObject)
Public
  DefaultClearID : String;
  Procedure SetPageTitle(aTitle : String);
  Procedure ClearPage(aBelowID : String);
end;

Procedure THTMLUtils.SetPageTitle(aTitle : String);
begin
  Document.Title:=aTitle;
end;

Procedure THTMLUtils.ClearPage(aBelowID : String);

Var EL : TJSElement;

begin
  if (aBelowID='') then aBelowID:=DefaultClearID;
  if (aBelowID='') then el:=Document.body
  else
    el:=Document.getElementById(aBelowID);
  if Assigned(El) then El.innerHTML:='';
end;
```

Since we cannot export a class directly from our module, in order for users of the library to
be able to use the class, we must export a function that creates an instance of the class:

```pascal
Function CreateUtils : THTMLUtils;
begin
Result:=THTMLUtils.Create;
end;
exports
CreateUtils;
end.
```

Obviously, if you need to specify options to your class' constructor you'll need to define
these options in your function.

> **Note:** Due to a bug in the released **Pas2JS** compiler it is necessary to disable optimizations
> when compiling this library:
> in the `Custom options` part of the compiler options dialog, the -O- option must be added.
> This bug has meanwhile been fixed.

To use this class, we must also rewrite our program.

We start by defining the `THTMLUtils` class as an external class:

```pascal
program htmlutilsdemo;

{$mode objfpc}
{$linklib ./htmlutils.js utils}
{$modeswitch externalclass}

uses JS, Web;

type
  THTMLUtils = class external name 'Object' (TJSObject)
  Public
    DefaultClearID : String;
    Procedure SetPageTitle(aTitle : String);
    Procedure ClearPage(aBelowID : String);
  end;

Function CreateUtils : THTMLUtils; external name 'utils.CreateUtils';
```

> **Note** the use of the `{$moduleswitch externalclass}`, needed to be able to define
> external classes.

Now, to use this class, we must also rewrite our program a little. We define a variable of
the class, which we use in our callbacks:

```pascal
Var
  BtnSetTitle,BtnClear : TJSHTMLButtonElement;
  edtTitle,edtBelowID,cbUseDefaultClearID : TJSHTMLInputElement;
  UtilsObj : THTMLUtils;

function DoSetTitle(aEvent: TJSMouseEvent): boolean;
begin
  Result:=False;
  UtilsObj.SetPageTitle(edtTitle.Value);
end;

function DoClear(aEvent: TJSMouseEvent): boolean;
begin
  Result:=False;
  if cbUseDefaultClearID.Checked then
    begin
      UtilsObj.DefaultClearID:=edtBelowID.value;
      UtilsObj.ClearPage('');
    end
  else
    begin
      UtilsObj.DefaultClearID:='';
      UtilsObj.ClearPage(edtBelowID.value);
    end;
end;
```

We initialize the variable with the CreateUtils call exported from our library:

```
begin
   UtilsObj:=CreateUtils;
   BindElements;
end.
```

The resulting page works in exactly the same way as the original example, only
now it uses a class. You can test this at:

```
https://www.freepascal.org/~michael/pas2js-demos/modules/classes/
```

For this simple example, exporting a variable of the correct type is also sufficient.
It requires only a few changes. In the library, the `CreateUtils` function can be
replaced with an exported variable declaration:

```
var Utils:THTMLUtils;

exports Utils;

initialization
   Utils:=THTMLUtils.Create;
end.
```

The variable is initializd in the initialization section of the library.
To use this variable, only a small change is needed in our program.
We remove the 'CreateUtils' function, and change the declaration of the UtilsObj variable:

```
 var
   UtilsObj:THTMLUtils;external name'utils.vars.Utils';
```

And of course the statement to assign the variable must be removed.
After these changes, again the example will function as the original example.
You can convince yourself at the live demo:
```
https://www.freepascal.org/~michael/pas2js-demos/modules/classusingvar/
```

❼ CONCLUSION

In this article we've shown one of the latest features of the **Pas2JS** transpiler: libraries and
how to use them. We've also shown that libraries in **Pas2JS** are more powerful than libraries
in native code: there is no need for special memory managers, and classes can be used as-is.
There are some small glitches in the library support for classes: the optimization switch,
and using overloads is possible but requires some tweaking of the external names.
Despite this, the support for modules is sufficiently mature to be used in production.

barnsten

Promotions

Delphi & C++Builder are the best development tools on the market to design and develop modern, cross-platform native apps and services. Also for Windows 11! It's easier than ever to create stunning, high performing apps for Windows, macOS, iOS, Android and Linux Server (Linux Server is supported in Delphi Enterprise or higher), using the same native code base. Share visually designed UIs across multiple platforms that make use of native controls and platform behaviors, and leverage powerful and modern languages with enhancements that help you code faster.

Introduction offer on RAD Studio, Delphi and C++Builder 11.1 Alexandria – until March 31, 2022:

20% discount on Professional
30% discount on Enterprise
30% discount on Architect
Buy directly in the webshop or ask us for a quote.

This offer is not valid on Academic licenses, term licenses and/or existing contracts.
You can not combine this with other offers.

MOST SOLD!

SALE

**DELPHI 11
ALEXANDRIA**

**Delphi 11.1 Alexandria
Professional**

Delphi

~~€1.699,00~~ €1.359,00

ADD TO CART

SALE

**DELPHI 11
ALEXANDRIA**

**Delphi 11.1 Alexandria
Enterprise**

Delphi

~~€3.999,00~~ €2.799,00

ADD TO CART

SALE

**DELPHI 11
ALEXANDRIA**

**Delphi 11.1 Alexandria
Architect**

Delphi

~~€6.499,00~~ €4.549,00

ADD TO CART

https://www.barnsten.com/promotions/

BY MICHAËL VAN CANNEYT

D11   starter ------- expert

PAS2√JS 3

ABSTRACT

In this article we show how to reduce coding when creating forms in a **PAS2JS** web application. Additionally we show how routing can be used to show multiple forms in an **SPA (Single Page Application)** and keep the browser experience of the user intact.

❶ INTRODUCTION

The previous articles showed how to implement a **PAS2JS** dialog, and how to switch to another dialog when the user logged in. All the examples shared a common approach:
whether they used the **WebWidget** components or plain HTML classes, they always had one field per HTML tag element in the web page: the field was either a **TWebWidget** component or one of the HTML classes found in the Web unit. This is identical to how Delphi code deals with forms.
For example, the login page resulted in this declaration:

```
TMyApplication =
class(TBrowserApplication)
edtEmail    : TJSHTMLInputElement;
edtPassword : TJSHTMLInputElement;
btnLogin    : TJSHTMLButtonElement;
procedure doLoginClick(aEvent: TJSEvent);
```

This is of course similar to a form declaration in Delphi. In the previous articles, these "form declarations" were created manually. In the following lines, we show how to generate such a declaration directly from the HTML file.

In **Delphi,** it is very common to show a second form with code like this:

```
Procedure TMainform.mnShowUserClick(Sender : Tobject);
var
frm : TUserForm;
begin
frm:=TUserForm.Create(Self);
frm.Show;
end
```

It is possible to mimic this behaviour in a web application. But this is in fact not really how a user will expect a web application to function:
when the user form appears as shown in the code, the user expects to be able to use the browser's back button to return to the previous form, or to reload the page using the refresh button.

The solution for this problem is called routing. With each form of the application, a URL is associated.
The URL must contain enough information to reconstruct the form.
For example, the following 3 URLs could be used to respectively show the overview of users, create a new user and edit user with ID 123:

```
/users/
/users/new
/users/123
```

If the user is currently viewing URL /`users`, and navigates to the details of user 123 then the URL becomes /`users/123`.
When the user wants to go back to the overview of users, he'll press the back button.

The application should catch this event and present the user again with the overview of the users. We'll explain how this can be achieved in a **PAS2JS** application.

2 GENERATING FORM DECLARATIONS

To avoid having to manually create a form declaration for each HTML file in a web application, a tool called **html2form** has been created. Its sources are distributed with **Pas2JS,** in the directory `tools/html2form`. It is a command-line application. When executed with the `-h` command-line option, you get some help messages which explain the various options:

◘   **help**   show a help message
◘   **below-id=ID**   Only create fields for child elements of element ID in the **HTML** page.
◘   **formclass=NAME**   The name of the pascal "form" class to create.
◘   **form-file**   Generate also a form .frm file (see below?).

◘   **getelementfunction=NAME**
                              Name of `getelementByID` function: this is the function that is used in a
                              `BindElements method to look up an HTML element based on`
                              their ID attribute.

◘   **events**   When specified, the tool will emit code to bind event handlers to methods.
◘   **input=file**   With this option, you specify the html file to read.
◘   **map=file**   Read a mapping file, which is used to map HTML tags to **Pascal** classes,
                              based on tag and attributes. By default, the tool maps HTML tags to the
                              native Javascript `HTMLElement` child classes.

◘   **no-bind**   By default, the `BindElements` call which maps variables to actual instances
                              is called from the class constructor. When this option is specified,
                              the call to `BindElements` is omitted from the constructor

◘   **output=file**   The pascal file to write a unit to.
◘   **parentclass=NAME**   Name of pascal "form" parent class.
                              There is no fixed `TForm` class in **Pas2JS,** so the tool needs a class name.
                              By default, this class is `Tcomponent`.
◘   **exclude=List**   You can specify a comma-separated list of IDs to exclude:
                              for these Ids, no field will be created.
                              If the value for this option starts with **@,**
                              then the remainder of the option is assumed to be a filename,
                              and the list is loaded from the file.

These options give you an idea of the possibilities.

So, how to use this tool?
Let's take the `index.html` file from our previous examples
– it contains a login dialog – and run it through the tool using the following command-line:

```
html2form --input=index.html -o frmlogin.pas -f TLoginForm
```

The result is a file that looks like this (*some comments have been removed*):

```pascal
unit frmlogin;
{$MODE ObjFPC}
{$H+}

interface

uses js, web, Classes;

Type
  TLoginForm = class(TComponent)
  Published
    edtEmail : TJSHTMLInputElement;
    error : TJSHTMLElement;
    edtPassword : TJSHTMLInputElement;
    btnContinue : TJSHTMLButtonElement;
  Public
    Constructor create(aOwner : TComponent); override;
    Procedure BindElements; virtual;
end;

implementation

Constructor TLoginForm.create(aOwner : Tcomponent);

begin
  Inherited;
  BindElements;
end;

Procedure TLoginForm.BindElements;
begin
  edtEmail:=TJSHTMLInputElement(document.getelementByID('edtEmail'));
  error:=TJSHTMLElement(document.getelementByID('error'));
  edtPassword:=TJSHTMLInputElement(document.getelementByID('edtPassword'));
  btnContinue:=TJSHTMLButtonElement(document.getelementByID('btnContinue'));
end;

end.
```

This "form" declaration will compile as-is and can be added to the **Pas2JS** project.
Many controls on a page need some kind of event handler: a button without event handler
is of little use.
Luckily, the **html2form tool** can also generate event handlers for you. For this, a convention is used.
When looking at a tag, all attributes that begin and end with an underscore character (_) are considered
event names. The value of the attribute is the event handler method name.
To demonstrate this, we modify the index.html a little.
The login button becomes:

```html
<button id="btnContinue"
  class="button is-block is-info is-large is-fullwidth"
  _click_="DoLoginClick">
  Continue <i class="fa fa-sign-in aria-hidden="true"></i>
</button>
```

The idea is that the 'click' event for the `btnContinue` button is handled by a method called `DoLoginClick`.
We run again the **HTML2FORM TOOL** on this file, but we also pass the -event command-line option:

```
html2form --input=index.html -o frmloginbase.pas --event -f TBaseLoginForm
```

As you see, we also specify another name for the class file and the unit name. The reason for this will become apparent soon.
The resulting class has more methods:

```
TBaseLoginForm = class(TComponent)
Published
  edtEmail : TJSHTMLInputElement;
  error : TJSHTMLElement;
  edtPassword : TJSHTMLInputElement;
  btnContinue : TJSHTMLButtonElement;
  Procedure DoLoginClick(Event : TJSEvent); virtual; abstract;
Public
  Constructor create(aOwner : TComponent); override;
  Procedure BindElements; virtual;
  Procedure BindElementEvents; virtual;
end;
```

The `BindElementEvents` is where the events are bound to the callbacks:

```
Procedure TBaseLoginForm.BindElementEvents;
begin
  btnContinue.AddEventListener('click',@DoLoginClick);
end;
```

Note that the callbacks are marked `virtual; abstract;`.
This is configurable:
If you prefer, you can also simply generate virtual methods with an empty body.

But there is a reason for making these methods abstract:
The class above is not meant to be used directly:
If you generate a class from the HTML file, it can happen that the HTML changes, and you must change the class definition.
If you do this and regenerate the file, any changes you made to the file will be lost.
This is of course not very convenient.
Instead, the above file is generated with abstract methods.
To actually code the form's business logic, you create a new unit with a descendent of `TBaseLoginForm`:

```
unit frmlogin;
{$MODE ObjFPC}
{$H+}

interface
uses js, web, Classes, frmloginbase;

Type
  TLoginForm = class(TBaseLoginForm)
  Public
end;
implementation

end.
```
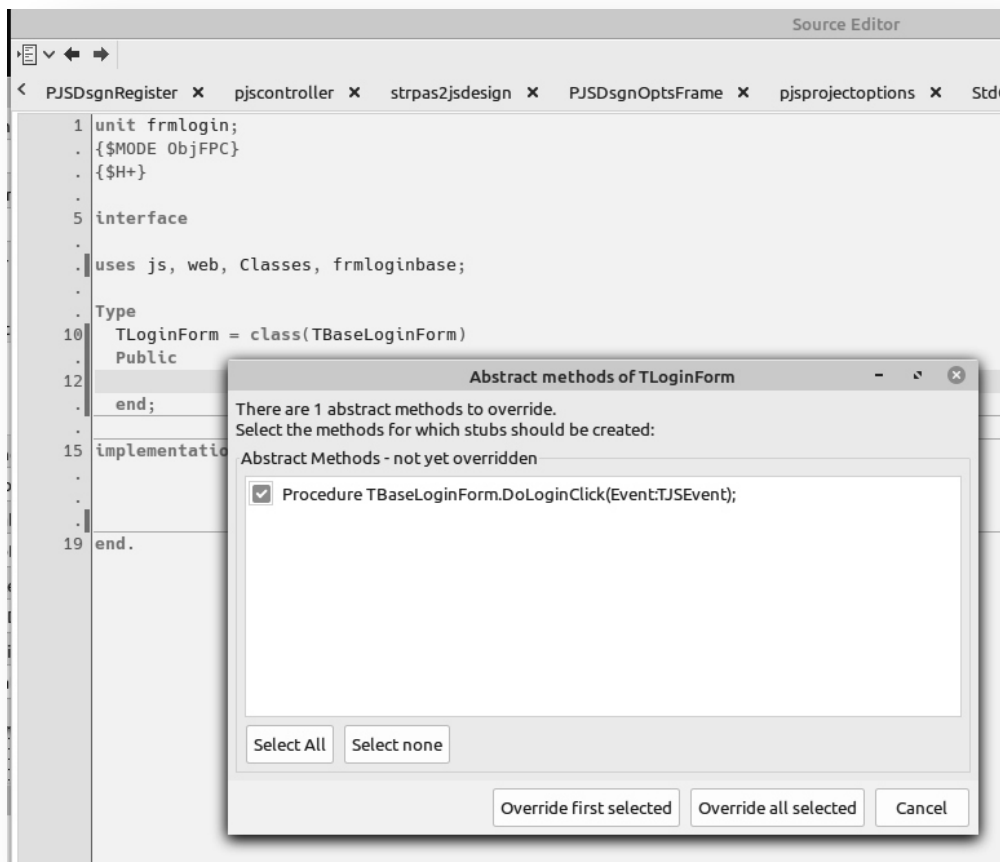
Figure 1: override abstract methods

In this 'form' class, we override the abstract methods, and implement the
GUI logic of the form. Now, when the **HTML** File changes, we can simply regenerate the
`frmloginbase` unit, and continue to work in the `frmlogin` unit.

Overriding the abstract methods can be done trivially in the **Lazarus** IDE:
The dialog under the **Source - Refactoring - Abstract** methods menu (*see figure 1 on page 5*)
allows you to do this with a couple of mouseclicks.
This dialog is also available from the source editor context menu popup,
or you can attach a shortcut key to it.
The resulting code looks like this:

```
TLoginForm = class(TBaseLoginForm)
procedure DoLoginClick(Event: TJSEvent); override;
Public
end;

implementation
procedure TLoginForm.DoLoginClick(Event: TJSEvent);
begin
end;
```

Figure 2: Create a class definition from an HTML File

All that is needed is to code the necessary **UI** or business logic.
If you forget to implement some abstract methods, the compiler will warn you about this
when you create an instance of a class which has abstract methods:

```
frmlogin.pas(29,14) Warning:
Constructing a class "TLoginForm" with abstract method "DoLoginClick"
```

If you have the **latest development version of Lazarus**,
this whole process has been automated in the IDE.
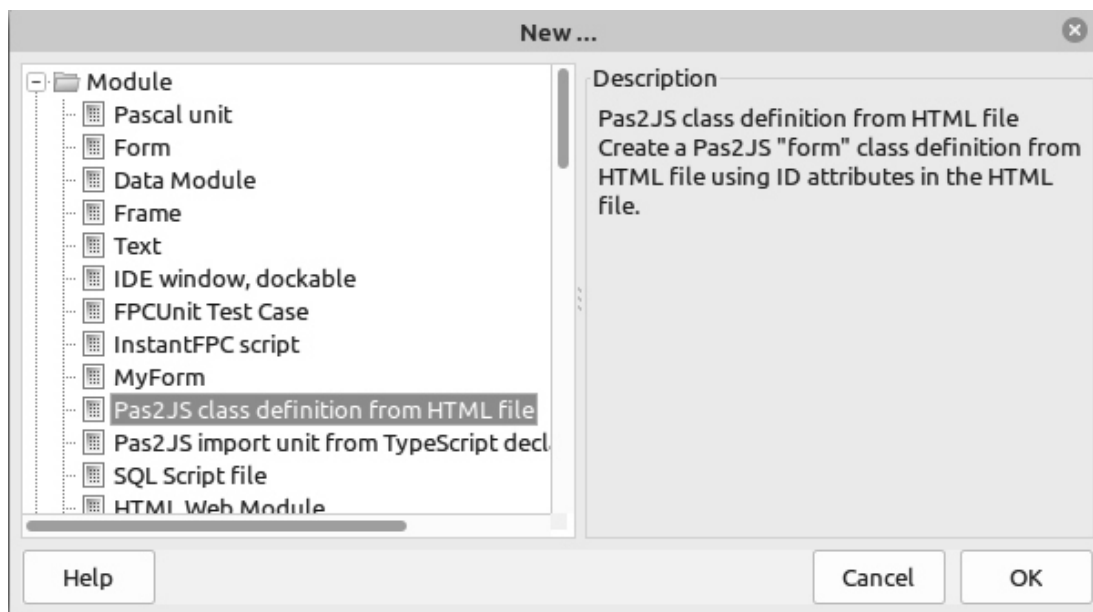In the **File-New** dialog, you can choose the **Pas2JS** Class definition from HTML file option (*see figure
2 on page 6*). When you choose this, you will be presented with a dialog that allows you to enter all
possible options for the generating of the class definition, *see figure 3 on page 7* and *figure 4 on page 7.*
In this dialog, you can also opt to add the HTML file to the **Lazarus** project.

Once all the options have been set, the **IDE** will create the unit with the class declaration, and adds the
new file to the project. In *figure 4 on page 7* you can see that more options are available in the dialog
than on the command-line.

In these screenshots, you see two toolbuttons: With these buttons you can load and save theoptions
set in this dialog: this allows you to quickly re-use the same options for all forms in your application,
and also allows you to use the saved options in an automated build procedure: the command-line
appplication can read this file as well.

To ensure that you can recreate the class definitions at any given moment, the IDE
automatically stores the options used to generate the unit in the Lazarus project file
(*the .lpi file*). In the project inspector, you can use the context menu to regenerate one
or more (*the selected units*) or all html form class files (*see figure 5 on page 8*).
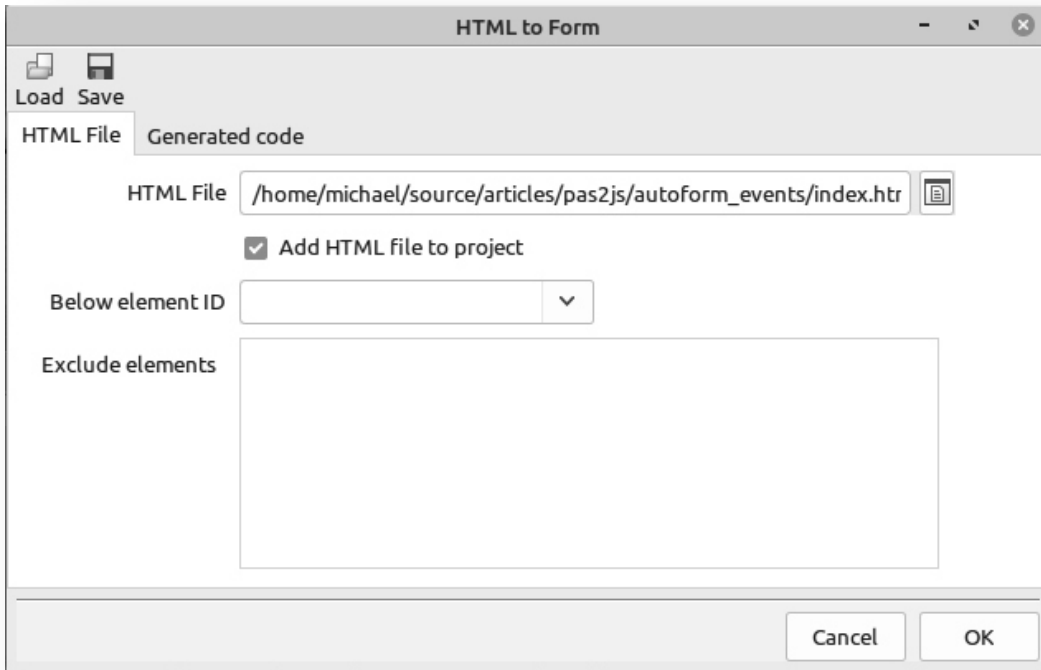
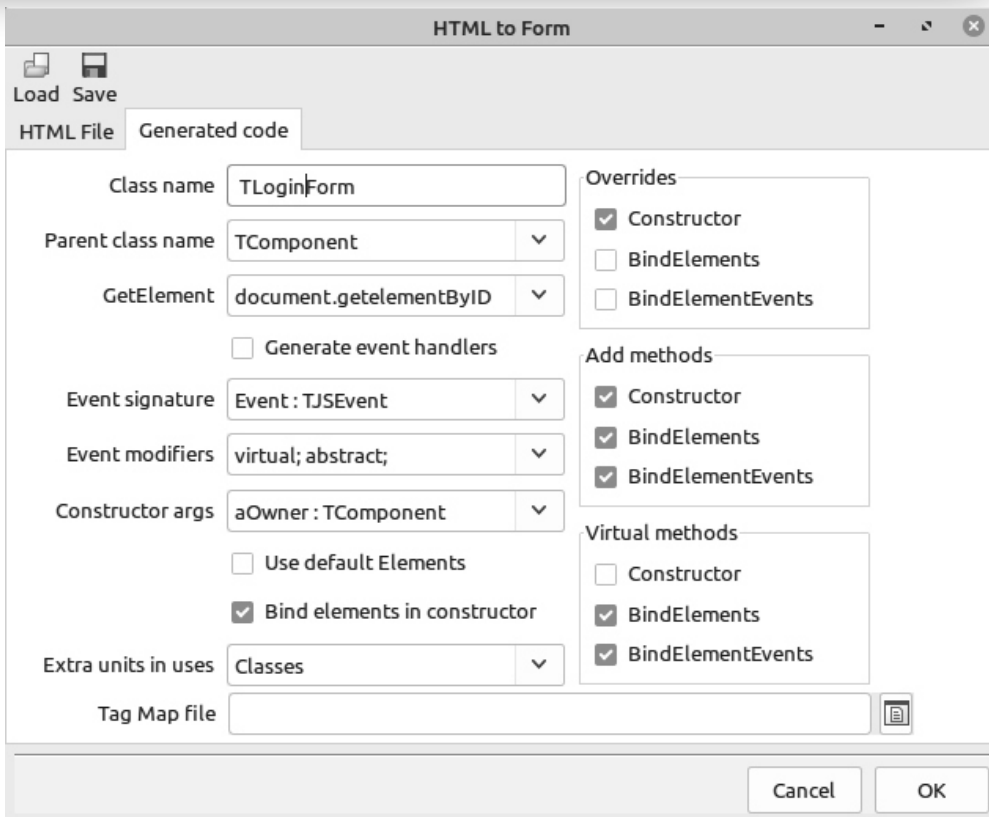Figure 3: Options for creating a class definition from an HTML File

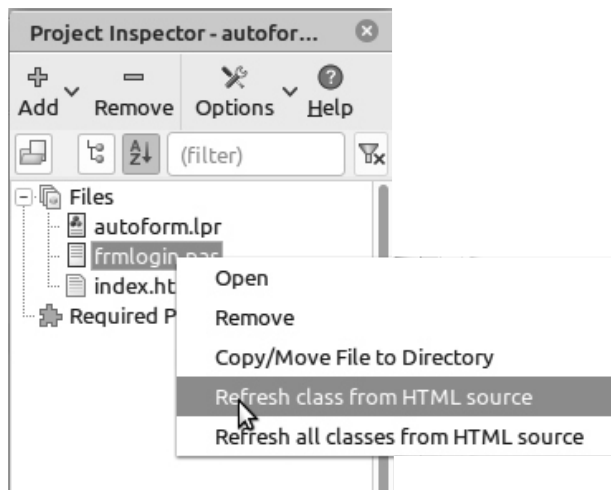Figure 4: More options for creating a class definition from an HTML File

Figure 5: Quickly regenerate the class definitions from their HTML files

3 NAVIGATING FROM ONE FORM TO THE NEXT

A web application usually shows one form at a time:
for instance, an overview of projects is shown, and when the user clicks a project,
the overview disappears, and the details for the selected project is shown.

In a **SPA** (**Single Page Application**) this usually happens by showing all 'forms' below a
designated HTML tag (*let's give it an id: form-parent*). This operation resembles docking a form in a
main form in **Delphi.**

There are several ways to do this: all forms can be made part of the html - you just insert their HTML
below the designated tag `form-parent,` give each form's top level HTML an ID. Then we can just
show or hide parts of the HTML by adding or removing the following style element to the top level tag
of the forms: `style="display: none;"`

You could make the routine that does this part of the form constructor, and just create the
form you need. This is easy and convenient if there are only a few forms in your application.
But in an application with many forms, the page's HTML will become unwieldy.
Far better and easier is to have the HTML for each form in a separate file. By loading the HTML file at
runtime, we can replace the HTML below the `form-parent` tag, and the browser will
happily refresh the screen with your new form.

A difficulty with this approach is that loading a file from the server is an asynchronous operation;
it takes some time. But this is not a big issue: we can start loading the forms as soon as the page is
loaded. A second issue is of course that we should not reload a form each time it is opened:
once it was loaded, we better keep the HTML somewhere in the browser, so we don't need to
download it again next time the form is shown.

To help with all this, **Pas2JS** comes with a unit called `Rtl.TemplateLoader`. This unit will load
a bunch of files (*called templates*) and keep them in some memory structure.
When it is time to load a form, the needed template is requested from the template loader,
and the form can be shown. If the template loader does not have it yet, you will need to tell
it to load it and wait till it is loaded: the component will notify you when it was loaded
so you can display the form.

The **TTemplateLoader** class is defined as follows:

```
TTemplateLoader = Class(TComponent)
   Procedure RemoveRemplate(aName : String);
   Function FetchTemplate(Const aName,aURL : String) : TJSPromise;
   Procedure LoadTemplate(Const aName,aURL : String;
                                 aOnSuccess : TTemplateNotifyEvent = Nil;
                                 AOnFail : TTemplateErrorNotifyEvent= Nil);
   Procedure LoadTemplates(Const Templates : Array of String;
                                 aOnSuccess : TTemplateNotifyEvent = Nil;
                                 AOnFail : TTemplateErrorNotifyEvent= nil);
   Property BaseURL : String;
   Property Templates[aName : String] : String;
   Property OnLoad : TTemplateNotifyEvent;
   Property OnLoadFail : TTemplateErrorNotifyEvent;
end;
```

The method names speak for themselves:

RemoveTemplate	clears the template with name **aName**.
FetchTemplate	Loads the template from URL **aURL** and stores the template with name **aName**. Returns a promise you can use to wait for the result.
LoadTemplate	Loads the template from URL **aURL** and stores the template with name **aName**. You can optionally specify 2 event handlers, which will be called when the template is loaded or when the load fails.
LoadTemplates	Passes a list of strings, strings at even indexes are the names of templates, strings at odd indexes are the URLS to load. You can optionally specify 2 event handlers, which will be called when a template is loaded.

The property names are equally clear:

BaseURL	All urls in **FetchTemplate, LoadTemplate(s)** are relative to this URL.
Templates	Here you can access a loaded template by name. If the template does not exist, an empty string is returned.
OnLoad	Allows you to set a global template load notification event. This is called in addition to the ones specified in the load call.
OnFail	Allows you to set a global template load failure notification event.

To demonstrate the use of this component, we'll make a web page with 3 "forms" – actually an HTML template file, and a button to show each form. The HTML template files will have an accompanying form declaration (we now know how to generate one quickly), which we will instantiate once the HTML has been loaded. For this, we need 3 html files:

❶ The global HTML file. We'll name it **index.html**, and it will contain the buttons to display the 2 forms. This file would normally contain a menu, nav bar etc: the things which are always the same in every form.
❷ The HTML file for the first form, a login page: we'll name it **login.html**.
❸ The HTML file for the second form, a projects list page: we'll name it **projects.html**.
❹ The HTML file for the third form, a users list page: we'll name it **users.html**.

Each HTML file will be accompagnied by a class form file, and we'll add some events to it, to demonstrate the capability of the html-to-form converter.

The `index.html` file is quite simple (*we show just the HTML body*):

```html
<div class="container">
  <div class="box">
    <button class="button is-primary" id="btnLogin"
                _click_="DoLoginClick">Login</button>
    <button class="button is-info" id="btnProjects"
                _click_="DoProjectsClick">Projects</button>
    <button class="button is-info" id="btnUsers"
                _click_="DoUsersClick">Users</button>
  </div>
  <div class="box form-container" >
    <div id="form-parent" >
      <div class="notification is-info is-light">
        Click one of the buttons above.
      </div>
    </div>
  </div>
</div>
```

As you can see, there are 3 buttons, plus some tags that use **Bulma CSS** to create a visually more pleasing HTML page.

From this we use the File-New wizard to create a `frmIndex.pp` unit with the following class:

```pascal
TIndexForm = class(TComponent)

Published
  btnLogin : TJSHTMLButtonElement;
  btnProjects : TJSHTMLButtonElement;
  btnUsers : TJSHTMLButtonElement;
  form_parent : TJSHTMLElement;
  procedure DoLoginClick(Event : TJSEvent);
  procedure DoProjectsClick(Event : TJSEvent);
  procedure DoUsersClick(Event : TJSEvent);
Public
  constructor create(aOwner : TComponent); override;
  procedure BindElements; virtual;
  procedure BindElementEvents; virtual;
end;
```

We do the same for the login, projects and users HTML files:

For these files, the IDE will generate a class definition that looks much like the above.

After doing this, we end up with 4 units in our project: f`rmIndex, frmLogin, frmProjects` and `frmUsers`.

For simplicity, we will deviate from the 'proper' way to do things and simply implement the needed functionality in the units themselves.

The `TIndexForm` class is the 'main' form of our application. In this form, we must implement the logic for navigation between the login, projects and users form. Here is the logic to show the login page:

```pascal
procedure TIndexForm.DoLoginClick(Event: TJSEvent);

Procedure ShowLogin;
begin
  form_parent.innerHTML:=GlobalTemplates.Templates['login'];
  FreeAndNil(FCurrentForm);
  FCurrentForm:=TLoginForm.Create(Self);
end;

procedure DoShowLogin(Sender: TObject; const aTemplate: String);
begin
  ShowLogin;
end;

begin
if GlobalTemplates.Templates['login']<>'' then ShowLogin
else
  GlobalTemplates.LoadTemplate('login','login.html',@DoShowLogin);
end;
```
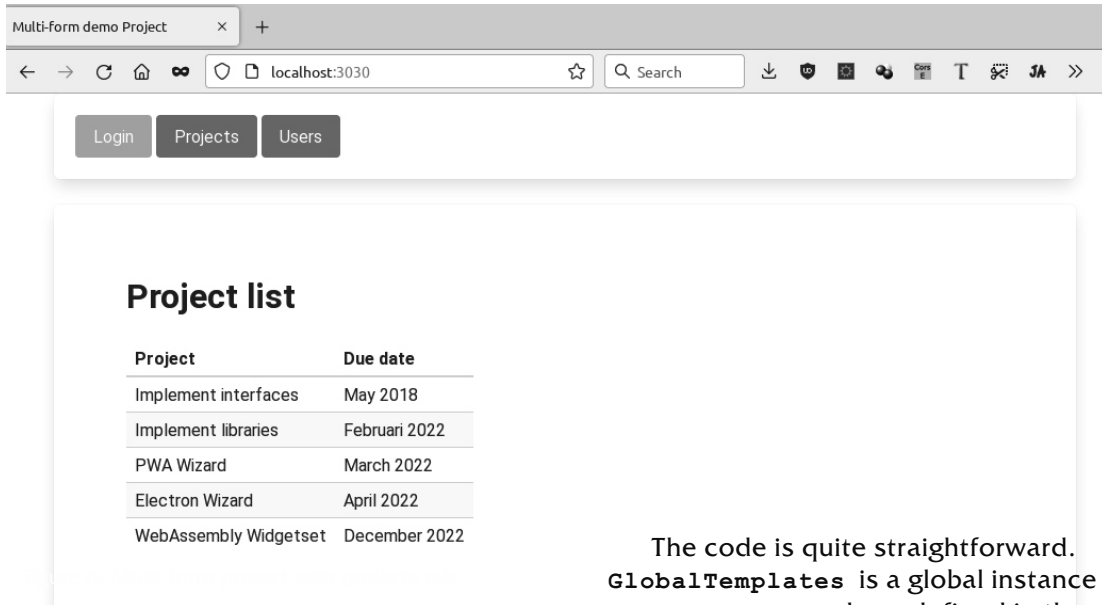
The code is quite straightforward. **GlobalTemplates** is a global instance of the **TTemplateLoader** class, defined in the **Rtl.TemplateLoader** unit. If the template is known, then the **ShowLogin** is called. If the template is not yet known, it is loaded, and in the success handler, **ShowLogin** is called. For simplicity, we didn't use a failure event handler.

The ShowLogin routine enters the template HTML below the HTML tag with id **form-parent**. It then destroys any previous form instance in **FCurrentForm** - a variable that keeps the current form. Finally it creates the new form class and saves it.

That's all there is to it. For the **Projects** and **Users** pages, a similar routine is made, only the names differ. The result after pressing the **Projects** button is shown in *figure 6on page 11.*

4 USING A FACTORY PATTERN

The routines to show the login, projects, an users pages are the same. All that differs is the class name, and the name of the template and html file. If there are a lot of forms, then repeating the above code is of course not very efficient.

So, an obvious improvement to reduce code is to create a routine (*or better, a class*) which does all this in one call. It would also be nice if we could just pass a form name which says which form must be shown, without having to specify a class or a HTML file name.

To achieve this, we create a TFormManager class in a frmBase unit, which looks like this:

WIKIPEDIA *In class-based programming, the* **factory method pattern** *is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method - either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes - rather than by calling a constructor.*

```
TFormManager = Class(TComponent)
Public
  Procedure RegisterForm(aClass : TBaseFormClass;
                         const aName : String = '';
                         aHTMLFile : String = '');
  Procedure UnregisterForm(aName : string);
  Procedure ShowForm(aName : string;
                     OnShow : TFormProcedure = nil);
  Property CurrentForm : TBaseForm;
  Property FormParent : TJSHTMLElement;
  Class property Instance : TFormManager;
end;
```

The `Instance` class property returns a global instance, which can be used to manage all forms.

With the `RegisterClass` routine, we can register a form class, using a name with which it can be shown, and a HTML file with which to load the HTML for the form. You can choose these last 2 parameters at will, but if you don't specify them, some defaults will be taken.

The `ShowForm` method can then be used to show a form using just the name used to register the form; A callback handler can be specified: it will be called when the form is shown.

The `ShowForm` routine looks much like the `OnClick` handler which we presented before, with as an addition a call to the `OnShow` handler that can be passed to the method:

```pascal
procedure TFormManager.ShowForm(aName: string; OnShow: TFormProcedure);
Var Idx : Integer; Reg : TFormRegistration;

  Procedure ShowForm;
  var html : string;
  begin
     If Assigned(FCurrentForm) then FreeAndNil(FCurrentForm);
     html:=GlobalTemplates.Templates['form:'+Reg.Name];
     FFormParent.innerHTML:=html;
     FCurrentForm:=Reg.FFormClass.Create(Self);
     If Assigned(OnShow) then OnShow(Self,FCurrentForm);
  end;

  procedure FormFailed(Sender: TObject;
                        const aTemplate, aError: String;
                        aErrorcode: Integer);
  begin
     Writeln('Error loading form template',aTemplate,':',
     aError,' (Code:',aErrorCode,')');
  end;

  procedure HaveForm(Sender: TObject; const aTemplate: String);
  begin
     ShowForm;
  end;

begin
   Idx:=FForms.IndexOf(aName);
   if Idx=-1 then
      Raise EForms.CreateFmt(SErrUnknownForm,[aName]);
   Reg:=TFormRegistration(FForms.Objects[Idx]);
   if GlobalTemplates.Templates['form:'+Reg.Name]='' then
      GlobalTemplates.LoadTemplate('form:'+Reg.Name,Reg.HTML,
                                    @HaveForm,@FormFailed)
   else
      ShowForm;
end;
```

The `OnClick` handlers of our menu buttons in the index form can now be reduced to the following:

```pascal
procedure TIndexForm.DoLoginClick(Event: TJSEvent);
begin
   FormManager.ShowForm('login');
end;

procedure TIndexForm.DoProjectsClick(Event: TJSEvent);
begin
   FormManager.ShowForm('projects');
end;

procedure TIndexForm.DoUsersClick(Event: TJSEvent);
begin
   FormManager.ShowForm('users');
end;
```

Obviously, before this can work, the login, projects and users forms need to be registered.
In the **RegisterForm** method of the **TFormManager** class, the **aClass** parameter is
of type **TBaseFormClass**. This class reference type is also defined in the **frmBase** unit:

```
 TBaseForm = class(TComponent)
Public
   Class Function FormName : String; virtual;
   Class Function FormHTMLFileName : String; virtual;
   Class Procedure Register;
end;
TBaseFormClass = class of TBaseForm;
```

The **Register** class method looks like this:

```
class procedure TBaseForm.Register;
begin
   With TFormManager.Instance do
      RegisterForm(Self,FormName,FormHTMLFileName);
end;
```

The FormName and FormHTMLFileName look like this:

```
 class function TBaseForm.FormName: String;
 Var P : integer;

 begin
    Result:=LowerCase(ClassName);
    if Result.StartsWith('tfrm') then
       Result:=Copy(Result,5,Length(Result)-4)
    else if Result.StartsWith('t') then
       Result:=Copy(Result,2,Length(Result)-1);
    if Result.EndsWith('form') then
    begin
       P:=Pos('form',Result);
       Result:=Copy(Result,1,P-1);
    end;
 end;

 class function TBaseForm.FormHTMLFileName: String;
 begin
    Result:=FormName+'.html';
  end;
```

The result of all this code is that the line

```
   TFrmLogin.Register;
```

will register the form class **TFrmLogin** with name **login** and html file **login.html**.
The mechanism presented here is of course just a convention which makes life easier;
you can perfectly invent other algorithms. The start of our program becomes therefore:

```
      TUsersForm.Register;
      TProjectsForm.Register;
      TLoginForm.Register;
      FIndex:=TIndexForm.Create(Self);
      FormManager.FormParent:=FIndex.form_parent;
```

Note that the **TIndexForm** is not registered:
It has no associated HTML which must be loaded:
the index.html file is already loaded.

5 ROUTING

We have now reduced the code it takes to show a form to a one-liner in an `onclick` handler.
However, this does not solve our principal problem: the use of the back and
forward buttons in the browser:
if the user first opens the projects list and then goes to the users list, he will naturally assume he
can go back to the projects list by hitting the back button.
With the application as it is coded now, if you press the back button while the users list is
shown, either
- ◘ Nothing will happen if the demo is the first page loaded in your browser.
- ◘ Or you will be taken back to the website you were looking at before you opened the demo.

The solution to this problem is called routing: with each form we associate an URL.
As the user navigates between the forms, the URL changes.
This is easy with a website where each form is an actual and separate HTML page.

But how to do this in a **Single Page Application (SPA)**?
Luckily, in **HTML 5**, this is possible: the browser offers access to the history mechanism
of your browser page. You can be notified if the URL changes, and you can also change
the URL. Since we are creating a **SPA (Single Page Application)** we must of course try to
avoid a page reload, and remain in the current page.
But how to stay on the same page when we require that the URL must change when navigating
from one form to another? This also is possible: the hash part of the URL can
be used. The following 3 URLs are the same page:

```
http://localhost:3000/index.html#/login
http://localhost:3000/index.html#/users
http://localhost:3000/index.html#/projects
```

These are 3 different URLs, but they all refer to the same HTML page. When you are on
the last URL in the list, and press the back button, the browser will see that the previous
URL is actually the same page, and will not reload the page from the webserver.
This mechanism can be further expanded, you can pass more information in the URL.
The following can refer to 1 page (*a fictitious project detail page*), which will – in turn –
show the details for project 1, a new project and project 2.

```
http://localhost:3000/index.html#/project/1
http://localhost:3000/index.html#/project/new
http://localhost:3000/index.html#/project/2
```

What is more, the user can copy the URL, send it to someone else, and the receiver can
open the application and be presented with the same page.

So, how to achieve this?
The **Pas2JS RTL** comes with a webrouter unit, which implements
a `TRouter` class. This class allows you to associate a callback with a route. A route is
simply a URL fragment: when the URL changes, the router will catch the browse event
for it, and match the new URL with the list of known routes. If it finds a route definition
that matches the URL, it will call the registered callback for that route.

For example, these are possible routes for our application:

```
/login
/project
/project/new
/project/:ID
/user
/user/:ID/Tab/:TAB
/user/:ID/
/*
```

Notice the `:ID` and `:TAB` in these routes. They present parameters: any string that does not contain a / character. When the router matches the URL, it will replace ID with what was actually in the URL. This means that the following URL fragments:

```
/project/123
/project/789
```

will result in a match for the route `/project/:ID`, but with `ID` set to `123` and `789`, respectively.

You can also use the wildcard character ***** to match any **URL** fragment.
This can be used for example to register an error page if no matching URL was found,
or to handle all URLS that start with a certain fragment in a single route definition.
The following is the declaration of the **TRouter** class,
with only the most important methods:

```
TRouter = Class(TComponent)
  Procedure DeleteRoute(aIndex : Integer);
  Function RegisterRoute(Const aPattern : String;
                         aEvent: TRouteEvent;
                         IsDefault : Boolean = False) : TRoute;
  function FindHTTPRoute(const Path : String;
                         Params: TStrings) : TRoute;
  function GetRoute(const Path : String;
                    Params: TStrings) : TRoute;
  Function RouteRequest(Const aRouteURL : String;
                        DoPush : Boolean = False) : TRoute;
  Property Routes [AIndex : Integer] : TRoute ;
  Property RouteCount : Integer;
  Property BeforeRequest : TBeforeRouteEvent;
  Property AfterRequest : TAfterRouteEvent;
end;
```

The purpose of these methods should be clear:
- **DeleteRoute**   Delete given route by index.
- **RegisterRoute**   Register a callback for a route: the aPattern is a pattern to match with the URL. If the URL matches the route, then aEvent is called. If `isDefault` is `True` then this route is used if no matching route can be found for a given URL fragment.
- **FindHTTPRoute**   Find a route definition for Path, and return parameter values in Params. Returns the route definition. If no route is found, Nil is returned.
- **GetRoute**   calls FindHTTPRoute, and raises an exception if no route was found.
- **RouteRequest**   Perform the routing for a request with URL frament aRouteURL. If DoPush is true, the new route is pushed onto the browser's URL history.
- **Routes**   Array access to the registered routes.
- **RouteCount**   The number of known routes.
- **BeforeRequest**   An event that is fired before handling a routing request.
- **AfterRequest**   An event that is fired after handling a routing request.

How can we use this object to show our forms automatically in the application ?
A simple mechanism suggests itself: each form registers a route starting with the
form name used to create the form.
This means that our three forms must register 3 routes: `/login`
`/projects`
`/users`

Now we can pluck additional fruits of the factory pattern that we introduced earlier. We
can use the `RegisterForm` call to register a route for the form.
To allow a form to register multiple routes for itself, we create a `FormRoutes` method in
`TBaseForm`:

```
class function TBaseForm.FormRoutes: TStringDynArray;
begin
   Result:=[FormName];
end;
```

This method (*which can return multiple routes*) is then used to register the routes for the
form in the form manager's `RegisterForm` method. This method starts with some sanity
checks, before adding a form registration object to a list. The `FormRoutes` method is then
used to register the various routes for the form:

```
function TFormManager.RegisterForm(aClass: TBaseFormClass;
                                   const aName: String;
                                   aHTMLFile: String):
                                   TRouteDynArray;

Var
     aRoute,N,H : String;
     aRoutes : TStringDynArray;
     aReg : TFormRegistration;
     Idx : Integer;

begin
   // Some cleanup
   N:=aName;
   if N='' then N:=aClass.FormName;
   H:=aHTMLFile;
   if H='' then H:=aClass.FormHTMLFileName;
   // Create and save form registration.
   aReg:=TFormRegistration.Create(aClass,N,H);
   FForms.AddObject(N,aReg);
   // Register routes
   aRoutes:=aClass.FormRoutes;
   SetLength(Result,Length(aRoutes));
   Idx:=0;
   for aRoute in aRoutes do
   begin
      Result[Idx]:=Router.RegisterRoute(aRoute,@DoFormRoute,False);
      Inc(Idx);
   end;
   // Save routes in registration.
   aReg.FRoutes:=Result;
end;
```

As a last step, the created routes are saved in the form registration. This is needed in the
`DoFormRoute` method, which will be called when the route is matched.
In the `DoFormRoute` method, we start with looking up the form registration associated
with the route. The `HasRoute` helper function checks if the given route is in the array of
routes for that form registration.

```pascal
procedure TFormManager.DoFormRoute(URl: String;
             aRoute: TRoute;
             Params: TStrings);
Var
   Idx : Integer;
   Reg : TFormRegistration;
begin
   // Find the form registration for this route:
   Reg:=Nil;
   Idx:=FForms.Count-1;
While (Reg=Nil) and (Idx>=0) do
   begin
      Reg:=TFormRegistration(FForms.Objects[Idx]);
      if Not Reg.HasRoute(aRoute) then
         Reg:=Nil;
      Dec(Idx);
   end;
   // If we found a registration, show the form
   if Assigned(Reg) then
      ShowForm(Reg.Name,
      procedure (sender: TObject; aForm : TBaseForm)
      begin
         aForm.ShowRoute(URL,aRoute,Params);
      end);
end;
```

Finally, if a valid form registration is found, then we show the form using the existing
ShowForm method. In the `OnShow` callback we call a new method of our base form
class, `ShowRoute` :

```pascal
procedure TBaseForm.ShowRoute(const aURL: String; aRoute: Troute; aParams: TStrings);
begin
   Writeln('Showing route for URL ',aURL,' with pattern: ',
   aRoute.FullPath,' and params : ',aParams.CommaText);
end;
```

This virtual method can be overridden to let the form act on the particular route that
was used to show the form. For instance, to react on parameters in the route.
So, now that we have our routing in place, how to use it? This is simple, and we actually
end up with less code. The 3 buttons in the index.html page to show our 3 forms can now
be replaced with 3 anchor elements:

```html
<div class="box">
<a class="button is-primary" id="btnLogin" href="#/login">Login</a>
<a class="button is-info" id="btnProjects" href="#/projects">Projects</a>
<a class="button is-info" id="btnUsers"    href="#/users">Users</a>
</div>
```

As you can see, the `button` HTML tag has been replaced with an anchor HTML tag`(a)`. In the anchor tag's `href` attribute, we enter the route for the form that must be shown: `#/`, followed by the form name. The `click` handler has also been removed: it is no longer needed.

If we now regenerate the class file associated with our `index.html` file, we notice that the `click` handlers are gone. The navigation is now handled by the router.

The result can be seen in *figure 7 on page 18*. Notice how in in the address bar of the browser, the route is now displayed within the URL's hash. As you navigate between forms, the URL will change as you switch forms. Additionally, if you now use the back and forward buttons of the browser, you will actually switch forms !
With this mechanism, you are giving the user a real browser experience.
Incidentally, note that the hyperlink elements look exactly like button elements used before: this is one of the perks of using a **CSS framework**.
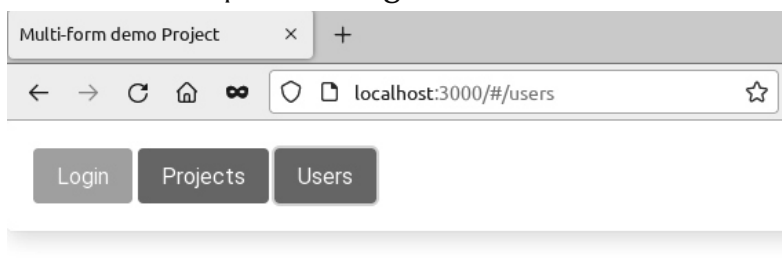
User list

Name	Country
Detlef overbeek	The netherlands
Mattias Gaertner	Germany
Sven Barth	Germany
Florian Klaempfl	Germany
Michael Van Canneyt	Belgium
Jonas Maebe	Belgium

Figure 7: Multi-form project using routing

To demonstrate the use of parameters in the URL, we change the projects overview page to show links to a 'project details' page for a project:

```html
<tr>
  <td>
  <a href="#/project/1">Implement interfaces </a>
  </td>
  <td>
  May 2018
  </td>
</tr>
```

The HTML of the project detail page (project.html) looks like this:

```html
<h1  id="pagetitle"
     class="title is-3">Project:
     <span id="hdrProjectName">?</span>
</h1>
<div  id="lblNotFound"
     class="notification is-danger is-light is-hidden">
     Project %d not found !</div>
<div  class="field">
   <label class="label">Project Name</label>
   <div class="control">
   <input   class="input"
           id="edtProjectName"
           type="text"
           placeholder="Project name">
   </div>
</div>

<div  class="field">
   <label class="label">date due</label>
   <div   class="control has-icons-left">
      <input   class="input is-success"
              type="text" id="edtDueDate"
              placeholder="project due date">
      <span class="icon is-small is-left">
         <i class="las la-calendar-check"></i>
      </span>
   </div>
</div>

<div class="field is-grouped">
   <div class="control">
      <button id="btnSave"
              class="button is-link">
      Save
      </button>
   </div>
   <div class="control">
      <button id="btnCancel"
              class="button is-link is-light">
      Cancel
      </button>
   </div>
</div>
```

When we generate the form for this HTML, we call the form class TProjectDetailForm, and we override the following methods:

```pascal
Procedure ShowRoute(Const aURL : String;
                          aRoute : TRoute;
                          aParams : TStrings); override;
Class function FormHTMLFileName: String; override;
Class function FormRoutes: TStringDynArray; override;
```

Since the form class name differs from the html file name (the convention that was presented earlier), we need to give the form factory the correct HTML file name:

```
class function TProjectDetailForm.FormHTMLFileName: String;
begin
Result:='project.html';
end;
```

Since we wish to obtain the value of the form ID as a parameter in the URL, we must register a fitting route for this:

```
class function TProjectDetailForm.FormRoutes: TStringDynArray;
begin
Result:=['/project/:ID']
end;
```

The result is that project ID will be passed to the ShowRoute in the ID parameter.

We can now use this parameter to load the correct project data. If a wrong ID or a false ID is loaded an error message is displayed:
The user can type an arbitrary or outdated URl in the browser address bar, and we must be prepared to deal with errors.
With a simple **Bulma CSS** class (`is-hidden`), a HTML element can be shown or hidden.

Showing a warning is thus simply a matter of removing the `is-hidden` CSS class from the HTML element that shows the warning.

The data is loaded from 2 arrays of values (`ProjectNames` and `ProjectDates`).

```
procedure TProjectDetailForm.ShowRoute(const aURL: String;
                                       aRoute: TRoute;
                                       aParams: TStrings);
Const
  NotFound = 'Project "%s" not found!';

Var
  aID : NativeInt;
  aError,aName,aDue : String;

begin
  aID:=StrToInt64Def(aParams.Values['ID'],-1);
  // Show an error if the ID is unknown.
  if (aID<1) or (aID>ProjectCount) then
  begin
     aError:=Format(NotFound,[aParams.Values['ID']]);
     lblNotFound.innerText:=aError;
     lblNotFound.classList.remove('is-hidden');
     Exit;
  end;
  // Show project data
  aName:=ProjectNames[aID];
  aDue:=ProjectDates[aID];
  hdrProjectName.InnerText:=aName;
  edtProjectName.value:=aName;
  edtDueDate.value:=aDue;
end;
```
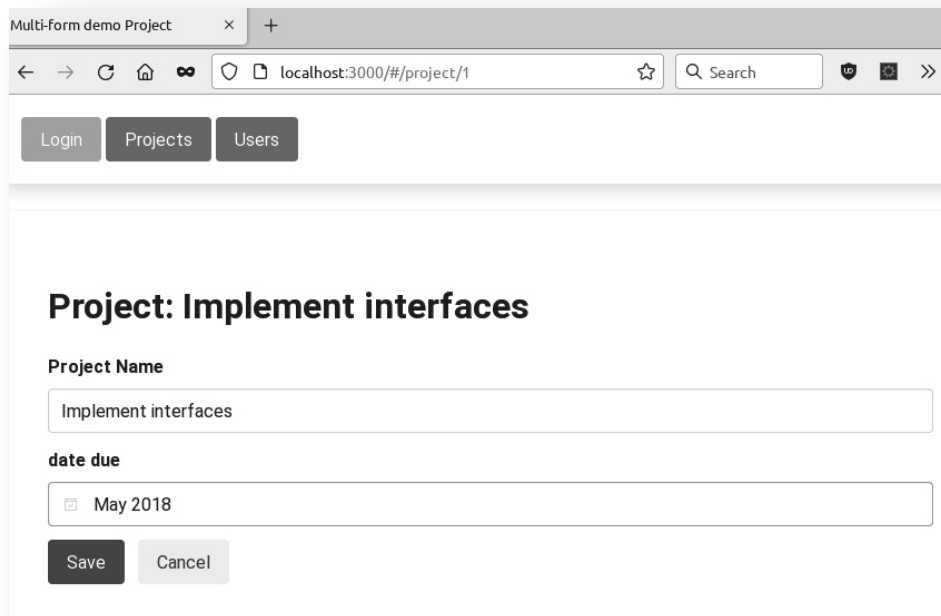
Figure 8: Routing parameters in action

The last lines are not very different from what you would do in a regular **VCL** Class: only the property names are different.

The result of this code can be seen in *figure 8 on page 23*.
NOTE the **URL** which contains the project **ID.**
As you navigate between the various projects, you can always go back to a previously visited project with the browser's back button.

7 CONCLUSION

In this article, we've shown how to present the user with an actual browser experience:
back and forward buttons for navigation now work. In doing so, the work needed to show forms was significantly reduced:
Using a router and changing buttons to anchor elements in the html reduces code.
There are still small glitches: when reloading the page, you will return to the initial page, even though the URL contains the route for the last visited page. It would also be nice if data for the projects could be loaded from an actual database.
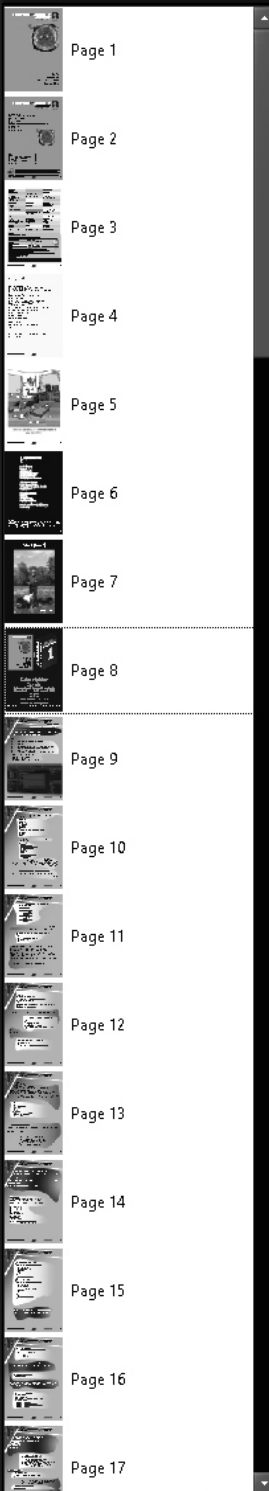We will deal with these issues in a next contribution.

ABSTRACT

Electron is a platform to enable you to create desktop applications of any size that can run on **Linux MacOS** and **Windows.** Because it is possible through **Pas2JS** to create Website applications it is worth exploring other options for creating applications for the desktop. **Electron** is one other possibility. In this article I 'll explain what **Electron** is and what one can achieve with it.

INTRODUCTION

Electron is basically a platform that enables you to easily built a **Graphical User Interface (GUI)**, It combines the **Node.JS (*1)** with **Chromium(*2)** (the open source foundation of Google Chrome).

Electron enables you to have easy access at the parts of your computer that the browser's sandbox can not access.
As an example: Web apps cant get through to the filesystem. It does not have access or hook into the operating system API which a desktop app needs.

Most web applications aren't available when there isn't a reliable internet connection. Electron is a runtime environment that allows you to create desktop applications with **HTML5, CSS,** and **JavaScript.**

It's an open source project started by **Cheng Zhao,** an engineer at **GitHub.** Previously called **Atom Shell, Electron** is the foundation for **Atom,** a cross-platform text editor by **GitHub** built with web tech.
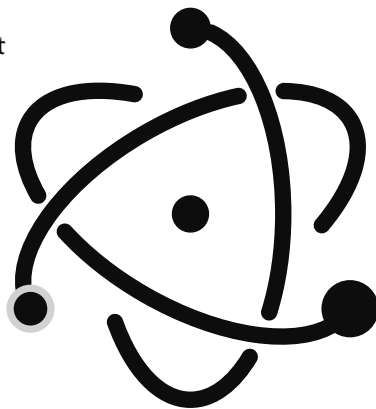
ELECTRON

(formerly known as **ATOM SHELL**) is a free and open-source (started by "**Cheng Zhao"**)software framework developed and maintained by **GitHub.** It allows for the development of desktop **GUI** applications using web technologies: it combines the **Chromium** rendering engine and the **Node.JS** runtime.
It was originally built for **Atom**. **Electron** is the main **GUI** framework behind several open-source projects including **Atom, GitHub Desktop, Light Table, Visual Studio Code, EverNote, WordPress Desktop** and **Eclipse Theia.**

ATOM is a free and open-source text and source code editor for macOS, Linux, and Windows with support for plug-ins written in JavaScript, and embedded Git Control. Developed by GitHub, Atom is a desktop application built using web technologies. Most of the extending packages have free software licenses and are community-built and maintained. Atom is based on Electron (formerly known as Atom Shell), a framework that enables cross-platform desktop applications using Chromium and Node.js.
Atom was initially written in CoffeeScript and Less, but much of it has been converted to JavaScript. Atom was released from beta, as version 1.0, on 25 June 2015. Its developers call it a "hackable text editor for the 21st Century", as it is fully customizable in HTML, CSS, and JavaScript.

Electron combines the CHROMIUM CONTENT MODULE and **Node.js** runtimes. **Chromium** and **Node** are both wildly popular application platforms in their own right, and both have been used independently to create ambitious applications.
Electron brings the two platforms together to allow you to use JavaScript to build an entirely new class of application.
Anything you can do in the browser, you can do with **Electron.**
Electron apps comprise multiple processes. There is the "main" process and several "renderer" processes. (*See schema on page 6 of this article).* The main process runs the application logic, and can then launch multiple renderer processes, rendering the windows that appear on a user's screen rendering **HTML** and **CSS.**
Both the main and renderer processes can run with **Node.JS** integration if enabled.

SIMPLE EXPLANATION:

suppose you create a form within the **Chromium** Browser and use that as your runtime environment for your Desktop application.

So that is actually a Web browser environment but one that can run on your desktop or even in your Web Bowser (as long as they are Chromium based: **Edge / Chrome /FireFox / Safari / Opera / Dolphin etc.**)

So remember it must be installed on your desktop and has the advantage that it will look on all OS's the same.

That is its great trump card.

WIKIPEDIA

Node.js *is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the* *V8* *engine and executes JavaScript code outside a web browser.*

Node.js lets developers use JavaScript to write command line tools and for server-side script-ing—running scripts server-side to produce dynamic web page content before the page is sent to the user's web browser.

Consequently, Node.js represents a "JavaScript everywhere" paradigm, unifying web-application development around a single programming lan-guage, rather than different languages for server-side and client-side scripts.

Though .**js** *is the standard filename extension for JavaScript code, the name "Node.js" does not refer to a particular file in this context and is merely the name of the product.*

Node.js has an event-driven architecture capable of ***asynchronous I/O.***

(In computer science, asynchronous I/O (also non-sequential I/O) is a form of input/output processing that permits other processing to continue before the transmission has finished.*)*

These design choices aim to optimize throughput and scalability in web applications with many input/output operations, as well as for real-time Web applications (e.g., real-time communication programs and browser games).

The Node.js distributed development project was previously governed by the Node.js Foundation, and has now merged with the JS Foundation to form the OpenJS Foundation, which is facilitated by the Linux Foundation's Collaborative Projects program.

V8 *is an open-source JavaScript engine developed by the Chromium Project for Google Chrome and Chromium web browsers.*
The project's creator is Lars Bak.
The first version of the V8 engine was released at the same time as the first version of Chrome: 2 September 2008.

Chromium *is a free and open-source web browser project, principally developed and maintained by Google.*
This codebase provides the vast majority of code for the Google Chrome browser, which is propri-etary software and has some additional features.

The Chromium codebase is widely used. Microsoft Edge, Samsung Internet, Opera, and many other browsers are based on the code. Moreover, significant portions of the code are used by several app frameworks.

Google does not provide an official or stable version of the Chromium browser. All versions released with the Chromium name and logo are built by other parties.

https://en.wikipedia.org/wiki/Chromium_(web_browser)

Figure1: Droste effect

Understanding Electron,
how to build Desktop apps using web technolgies

EXAMPLE:
You want to do something necessary and a must have: You need to view/search and edit a folder where ever your documents are. Browser applications are not capable of accessing the file system without user interaction.

With **Node.JS,** you can implement all the features necessary, but you can't create a **Graphical User Interface**, as a result your application would be worthless.
By combining the browser environment with **Node.JS,** you can use **Electron** to create an application where you can open and edit docs as well as provide a User Interface for doing so.
So you need **Node.JS** together with Chromium.
See figure 2 right top

CHROMIUM CONTENT MODULE
Chromium is the open source version of **Google's Chrome** web browser. It has most of the feature and same code with small differences and the licensing.
The **Google-authored** portion is shared under the 3-clause **BSD** license.
Other parts are subject to a variety of licenses, including **MIT, LGPL, Ms-PL,**
and an **MPL / GPL / LGPL** tri-license, while **Node.js** uses a permissive **MIT** license for the main library.
The **MIT** license applies to all parts of the **Node.**
The Content Module is the core code that allows **Chromium** to render web pages in independent processes and use **GPU** acceleration.
The Content Module includes only the core technologies required to render **HTML, CSS,** and **JavaScript.**

Electron is a simple runtime. Like the way you use Node from the command line, you can run Electron applications using the Electron command-line tool.
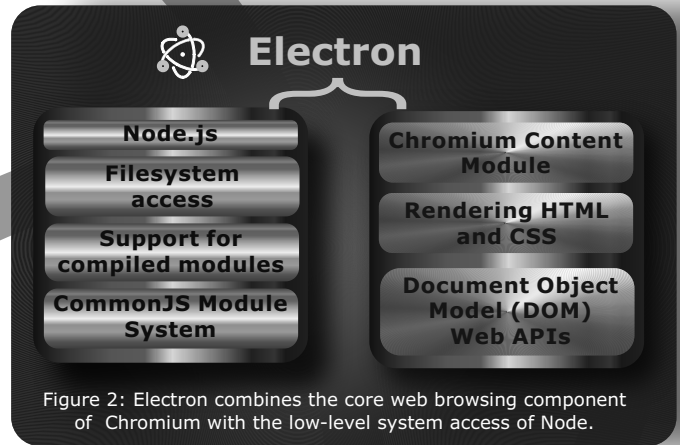

Figure 2: Electron combines the core web browsing component of Chromium with the low-level system access of Node.

You can find the Chromium licensing here:
https://www.chromium.org/chromium-os/licensing/

The part of the license that applies to Node.js here:
https://github.com/joyent/node/blob/

it is commonly known as the MIT license, which you can compare to here:
http://www.opensource.org/licenses/mit-license.php

This license, which is officially known as the Expat License, is here:
http://www.gnu.org/licenses/license-ist.html#Expat

a complete explanation you can find here:
https://en.wikipedia.org/wiki/MIT_License

WHAT IS NODE.JS?

For the first 15 years of its existence, **JavaScript** wasn't much used because it just applied within the web browser.

There wasn't a good way of support for running **JavaScript** on the server. There were projects, but hardly ever used.

The **Node.JS** project was initially released in 2009, as an open source, cross-platform runtime for developing server-side applications using JavaScript.

It used **Google's** open source **V8 engine** (*See page 3 of this article*) to interpret **JavaScript** and added API's for accessing the filesystem, creating servers, and loading code from external modules.

Over the last few years, **Node** has become very popular and is used for a wide range of purposes, from writing web servers, to control (*example*) **IOT (Internet Of Things)** or building desktop applications.

Node comes bundled with a package manager called **NPM (Node Package Manager)**, which makes the more than 250,000 libraries available in its registry.
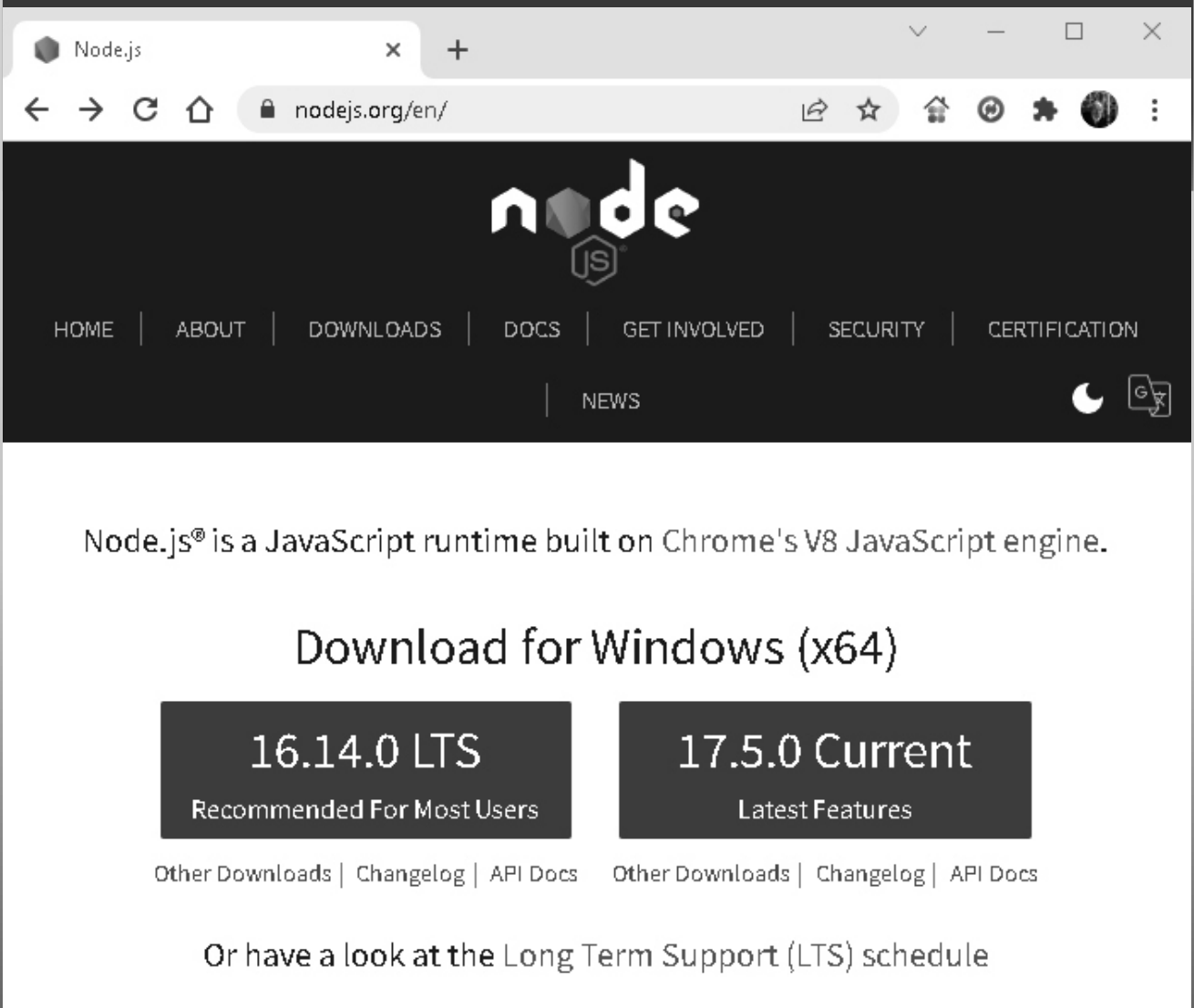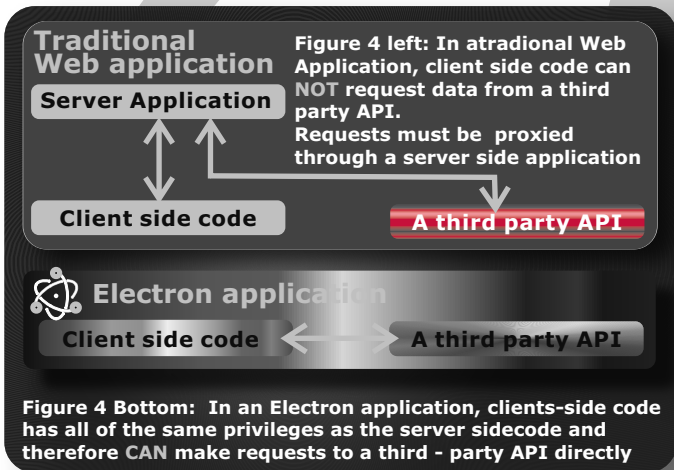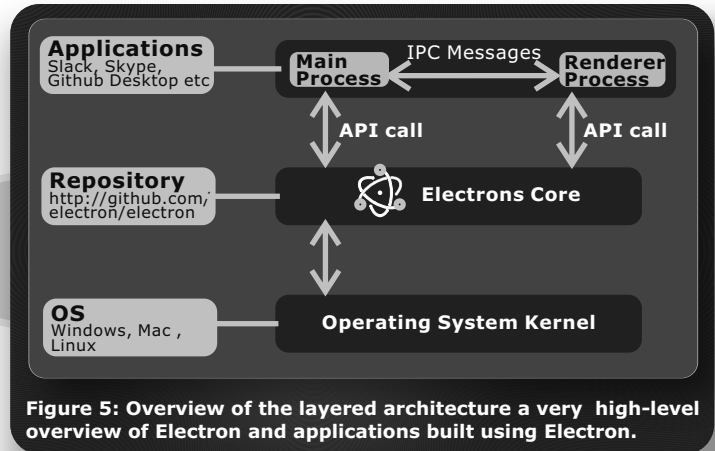
Figure 3: Node.Js Download

REASONS TO USE ELECTRON

When you create applications for a web browser, you have to be cautious in what technologies you choose to use and how you write your code: You're writing code that will be executed on a computer not owned by you.

Your users could be using the latest version of a modern browser such as Chrome or Firefox, or they could very well be using an outdated version of Internet Explorer.

❶ You have little to no say in where your code is being rendered and executed.
❷ You have to be ready for anything.
❸ You must write code for the lowest common denominator of features that have the widest support across all versions of all browsers in use today.

When you build your applications with **Electron,** you're packaging a particular version of **Chromium** and **Node.js,** so you can rely on whatever features are available in those versions. You don't have to concern yourself with what features other browsers and their versions support.

Figure 5: Overview of the layered architecture a very high-level overview of Electron and applications built using Electron.

**Figure 4 left: In atradional Web Application, client side code can NOT request data from a third party API.
Requests must be proxied through a server side application**

Figure 4 Bottom: In an Electron application, clients-side code has all of the same privileges as the server sidecode and therefore CAN make requests to a third - party API directly

MAIN PROCESS

Electron has 2 parts to it:
the main process and the rendering process.

❶ The main process has very important responsibilities.
It responds to application lifecycle events: such as starting up, quitting, preparing to quit, going to the background, coming to the foreground, and more.

❷ The main process is also responsible for communicating to native operating system APIs.
If you want to display a dialog box to open or save a file, you do it from the main process.

RENDERING

The main process can create and destroy renderer processes using **Electron's Browser-Window** module. Renderer processes can load web pages to display a **GUI**. Each process takes advantage of **Chromium's multiprocess architecture** and runs on its own thread. These pages can then load in additional **JavaScript** files and execute code in this process.

CRITICISM

Electron applications have been criticized for incurring significant overhead when compared with native applications with similar functionality. Applications built with **Electron** can take up more storage and **RAM,** and may run at less speed than a similar app built with technologies native to the operating system.

VERSIONS

In September of 2021, **Electron** moved to an 8 week release cycle between major versions to match the release cycle of **Chromium Extended Stable** and to comply with a new requirement from the **Microsoft Store** that requires browser-based apps to be within two major versions of the latest release of the browser engine.

Electron actively supports the latest three stable major versions. From September 2021 to May 2022, four major versions were temporarily supported due to the change in release cycles.

Unlike normal web pages, you have access to almost all the Node.js APIs in your renderer code. Renderer processes are isolated from each other and unable to access operating system integration **APIs. Electron** includes the ability to facilitate communication between processes to allow renderer processes to communicate with the main process in the event that they need to trigger an Open or Save File dialogue box or access any other OS-level integration.

Electron supports only **Windows 7** and later. For multimedia-focused applications, **Electron** is typically a better choice because **Electron** supports more codecs out of the box.
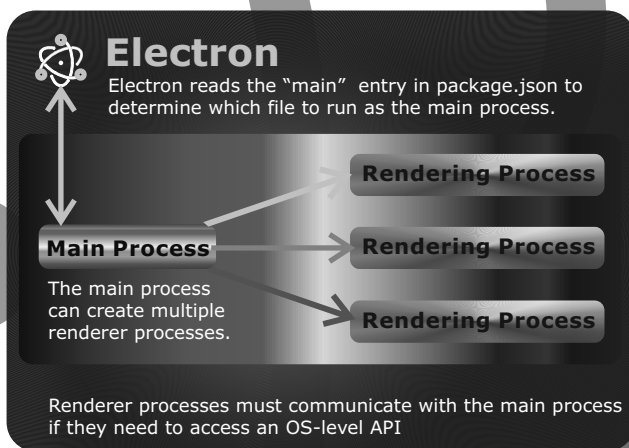

Figure X: Electron's multiprocess architecture

DONATE FOR UKRAINE AND GET A FREE LICENSE AT:
https://components4developers.blog/2022/02/26/donate-to-ukraine-humanitarian-aid/
(Just click)

.:: COMPONENTS
        DEVELOPERS *4*

If you are from Ukrainian origin you can get a free Subscription for Blaise Pascal Magazine, we will also give you a free pdf version of the Lazarus Handbook.

You need to send us your Ukrainian Name and Ukrainian email address (that still works for you), so that it proofs you are real Ukrainian.

please send it to editor@blaisepascal.eu **and you will receive your book and subscription**

BLAISE PASCAL◉MAGAZINE

Blaise Pascal

DONATE FOR UKRAINE AND GET A FREE LICENSE AT:

`https://components4developers.blog/2022/02/26/donate-to-ukraine-humanitarian-aid/`

(Just click)

KBMMW PROFESSIONAL AND ENTERPRISE EDITION V. 5.18.00 RELEASED!

- **RAD Studio XE5 to 11 Alexandria supported**
- Win32, Win64, Linux64, Android, IOS 32, IOS 64 and OSX client and server support
- Native high performance 100% developer defined application server
- Full support for centralized and distributed load balancing and failover
- Advanced ORM/OPF support including support of existing databases
- Advanced logging support
- Advanced configuration framework
- Advanced scheduling support for easy access to multithread programming
- Advanced smart service and clients for very easy publication of functionality
- High quality random functions.
- High quality pronouncable password generators.
- High performance LZ4 and Jpeg compression
- Complete object notation framework including full support for YAML, BSON, Messagepack, JSON and XML
- Advanced object and value marshalling to and from YAML, BSON, Messagepack, JSON and XML
- High performance native TCP transport support
- High performance HTTPSys transport for Windows.
- CORS support in REST/HTML services.
- Native PHP, Java, OCX, ANSI C, C#, Apache Flex client support!

kbmMemTable is the fastest and most feature rich in memory table for Embarcadero products.
- **Easily supports large datasets with millions of records**
- **Easy data streaming support**
- **Optional to use native SQL engine**
- **Supports nested transactions and undo**
- **Native and fast build in M/D, aggregation/grouping, range selection features**
- **Advanced indexing features for extreme performance**

- New I18N context sensitive internationalization framework to make your applications multilingual.
- New ORM LINQ support for Delete and Update.
- Comments support in YAML.
- New StreamSec TLS v4 support (by StreamSec)
- Many other feature improvements and fixes.

Please visit
`http://www.components4developers.com`
for more information about kbmMW

- High speed, unified database access (35+ supported database APIs) with connection pooling, metadata and data caching on all tiers
- Multi head access to the application server, via REST/AJAX, native binary, Publish/Subscribe, SOAP, XML, RTMP from web browsers, embedded devices, linked application servers, PCs, mobile devices, Java systems and many more clients
- Complete support for hosting FastCGI based applications (PHP/Ruby/Perl/Python typically)
- Native complete AMQP 0.91 support (Advanced Message Queuing Protocol)
- Complete end 2 end secure brandable Remote Desktop with near realtime HD video, 8 monitor support, texture detection, compression and clipboard sharing.
- Bundling kbmMemTable Professional which is the fastest and most feature rich in memory table for Embarcadero products.

.:: COMPONENTS 4 DEVELOPERS

EESB, SOA,MoM, EAI TOOLS FOR INTELLIGENT SOLUTIONS. kbmMW IS THE PREMIERE N-TIER PRODUCT FOR DELPHI / C++BUILDER