

Document .Net

(Multi-platform .Net library)

[SautinSoft](#)

Linux development manual

Table of Contents

1. Preparing environment	2
1.1. Check the installed Fonts availability	3
2. Creating "Convert PDF to DOCX" application	5
3. Creating new DOCX document from scratch	13

1. Preparing environment

In order to build multi-platform applications using .NET Core on Linux, the first steps are for installing in our Linux machine the required tools.

We need to install .NET Core SDK from Microsoft and to allow us to develop easier, we will install an advance editor with a lot of features, Visual Studio Code from Microsoft.

Both installations are very easy and the detailed description can be found by these two links:

[Install .NET Core SDK for Linux.](#)

Windows

Linux

macOS

.NET
Core

.NET Core 2.2

.NET Core is a cross-platform version of .NET for building websites, services, and console apps.

Build Apps ⓘ

Install .NET Core SDK

Run Apps ⓘ

Install .NET Core Runtime

[Install VS Code for Linux.](#)

Once installed VS Code, you need to install a C# extension to facilitate us to code and debugging:

Install [C# extension](#).

1.1. Check the installed Fonts availability

Check that the directory with fonts `"/usr/share/fonts/truetype"` is exist.

Also check that it contains `*.ttf` files.

If you don't see this folder, make these steps:

1. Download the archive with `*.ttf` fonts: <https://sautinsoft.com/components/fonts.tar>
2. Uncompress the downloaded font's archive to a directory and add it to the font path, a list of directories containing fonts:

```
# tar xvzf
```

3. Create a directory for new fonts

```
# mkdir /usr/share/fonts/truetype
```

4. Move the uncompressed font files to the new font directory

```
# mv *.ttf /usr/share/fonts/truetype
```

5. Navigate to the font directory

```
# cd /usr/share/fonts/truetype
```

6. Create `fonts.scale` and `fonts.dir`

```
# mkfontscale && mkfontdir
```

```
# fc-cache
```

7. Add the new font directory to the X11 font path

```
# chkfontpath --add /usr/share/fonts/truetype
```

8. Restart X font server

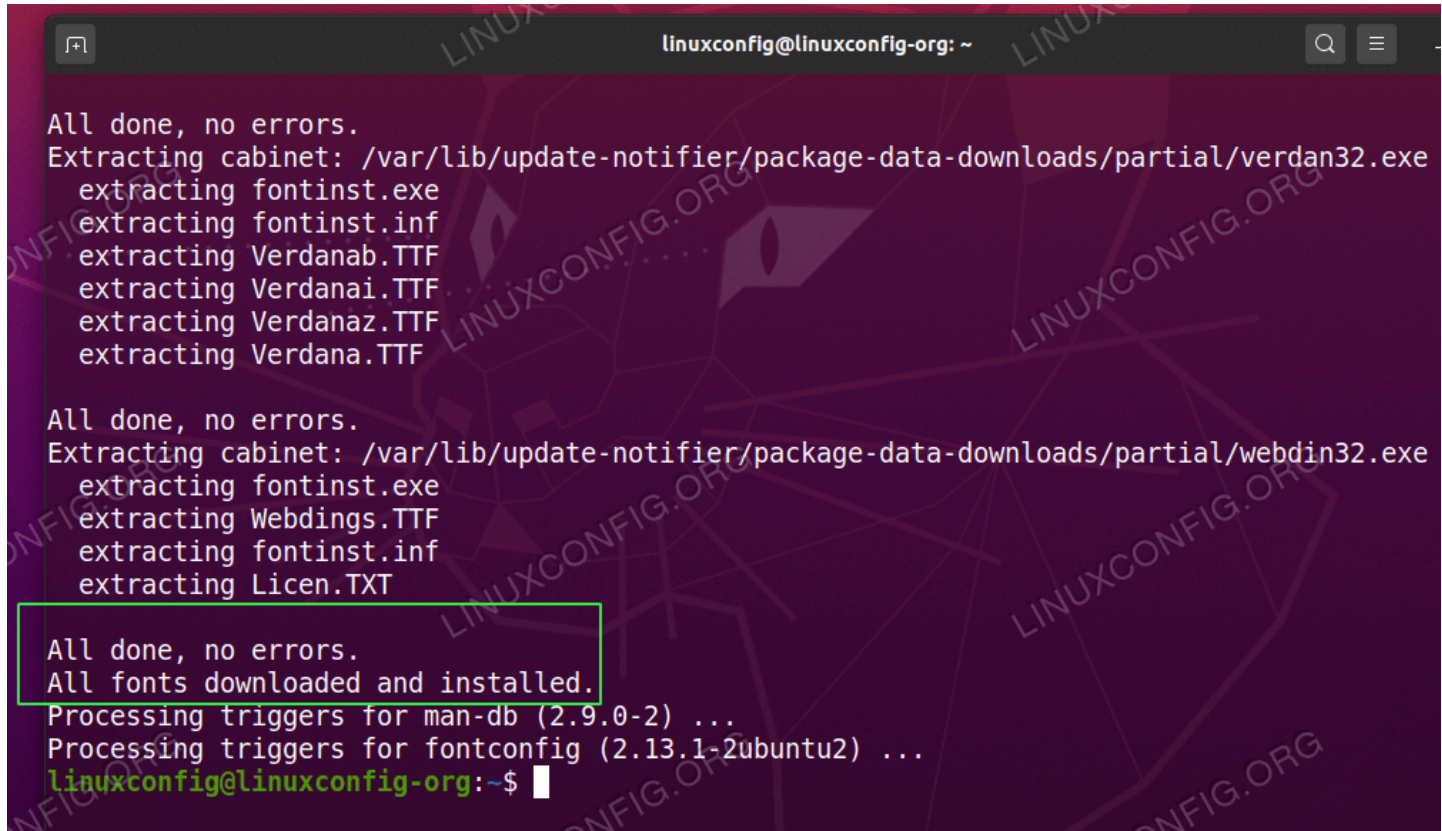
```
# /etc/rc.d/init.d/xfs restart
```

You can verify the successful addition of the new path by running `chkfontpath` command or by listing X font server's `/etc/X11/XF86Config` file.

If you do not have root access, copy the `*.ttf` to `~/.fonts` directory instead.

Or you may install "Microsoft TrueType core fonts" using terminal and command:

```
$ sudo apt install ttf-mscorefonts-installer
```

A terminal window screenshot with a dark background and light-colored text. The window title is 'linuxconfig@linuxconfig-org: ~'. The output shows the successful installation of 'ttf-mscorefonts-installer'. It lists the extraction of various font files including Verdana and Webdings. A green box highlights the final status: 'All done, no errors. All fonts downloaded and installed.' followed by processing triggers for 'man-db' and 'fontconfig'. The prompt 'linuxconfig@linuxconfig-org:~\$' is visible at the bottom.

```
All done, no errors.  
Extracting cabinet: /var/lib/update-notifier/package-data-downloads/partial/verdan32.exe  
  extracting fontinst.exe  
  extracting fontinst.inf  
  extracting Verdanab.TTF  
  extracting Verdanai.TTF  
  extracting Verdanz.TTF  
  extracting Verdana.TTF  
  
All done, no errors.  
Extracting cabinet: /var/lib/update-notifier/package-data-downloads/partial/webdin32.exe  
  extracting fontinst.exe  
  extracting Webdings.TTF  
  extracting fontinst.inf  
  extracting Licen.TXT  
  
All done, no errors.  
All fonts downloaded and installed.  
Processing triggers for man-db (2.9.0-2) ...  
Processing triggers for fontconfig (2.13.1-2ubuntu2) ...  
linuxconfig@linuxconfig-org:~$
```

Read more about [TrueType Fonts and "How to install Microsoft fonts, How to update fonts cache files, How to confirm new fonts installation"](#) .

With these steps, we will ready to start developing.

In next paragraphs we will explain in detail how to create simple console application. All of them are based on this VS Code guide:

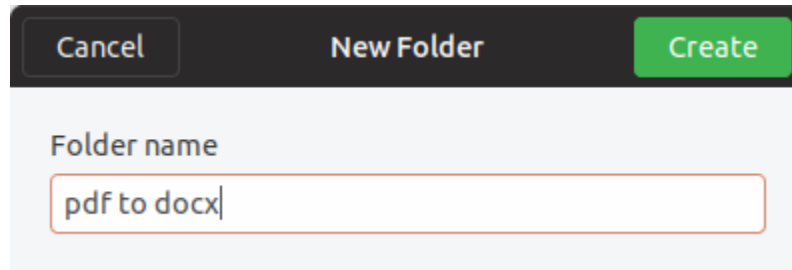
[Get Started with C# and Visual Studio Code](#)

Not only is possible to create .NET Core applications that will run on Linux using Linux as a developing platform. It is also possible to create it using a Windows machine and any modern Visual Studio version, as Microsoft Visual Studio Community 2017.

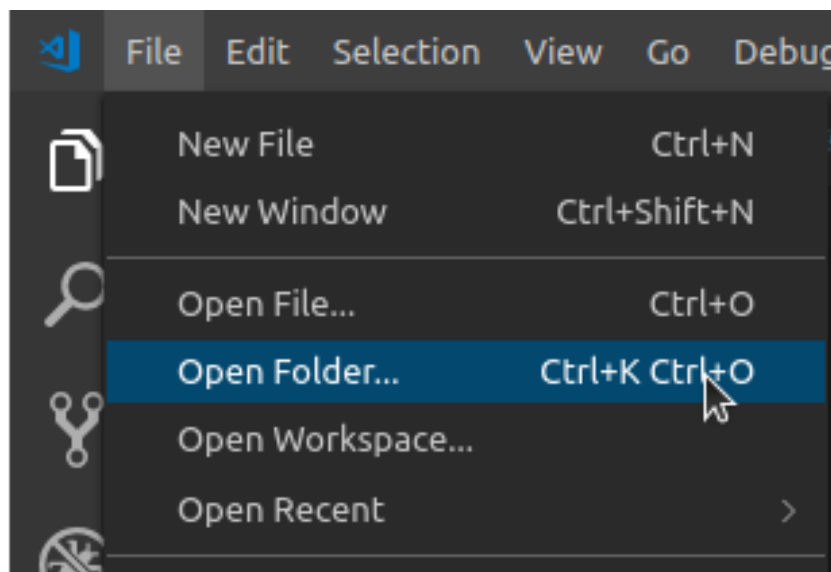
2. Creating “Convert PDF to DOCX” application

Create a new folder in your Linux machine with the name ***pdf to docx***.

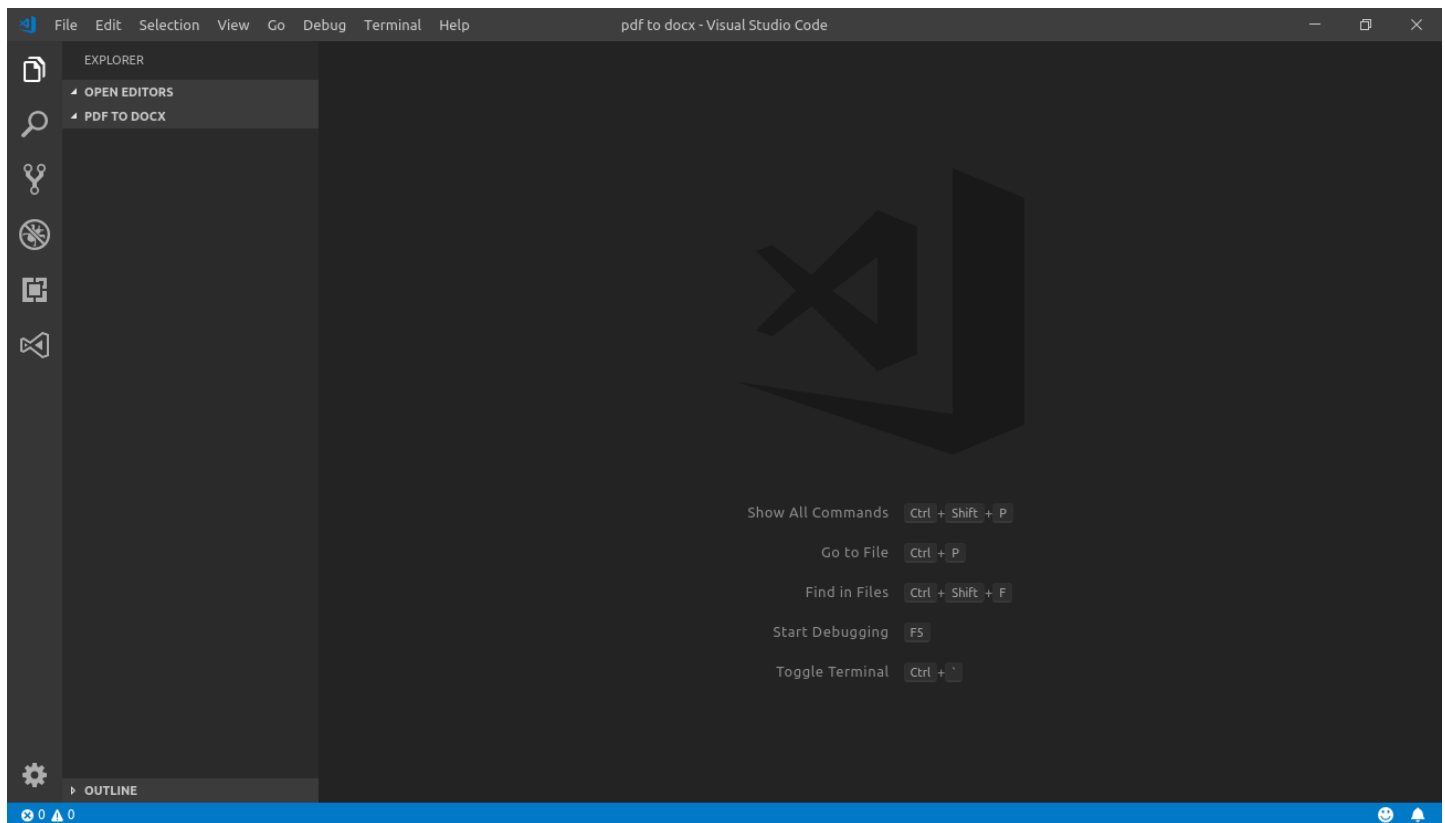
For example, let’s create the folder “***pdf to docx***” on Desktop (Right click-> New Folder):



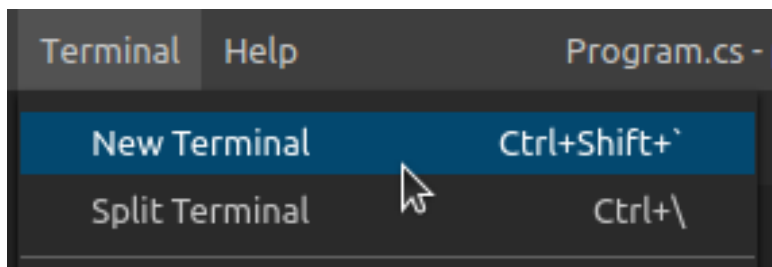
Open VS Code and click in the menu ***File->Open Folder***. From the dialog, open the folder you’ve created previously:



Next you will see the similar screen:

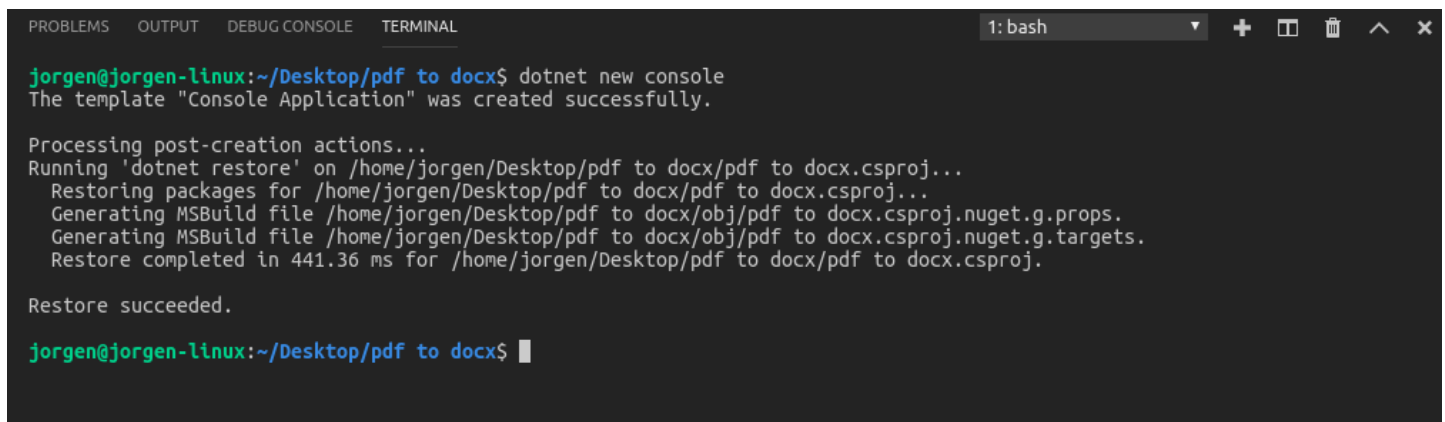


Now, open the integrated console – the Terminal: follow to the menu **Terminal** -> **New Terminal** (or press Ctrl+Shift+`):

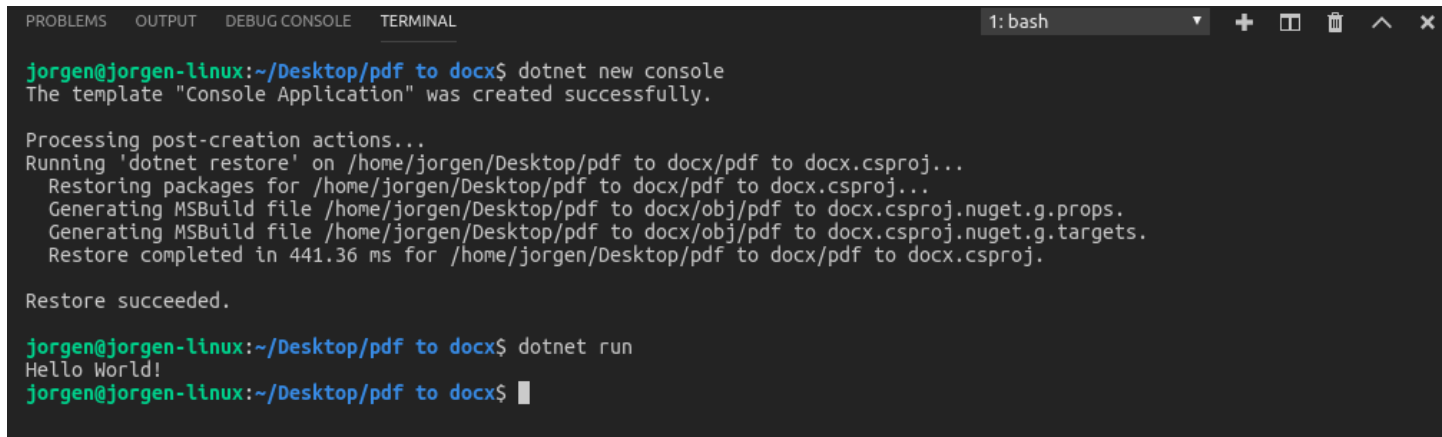


Create a new console application, using **dotnet** command.

Type this command in the Terminal console: **dotnet new console**



A new simple ***Hello world!*** console application has been created. To execute it, type this command: ***dotnet run***



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: bash
jorgen@jorgen-linux:~/Desktop/pdf to docx$ dotnet new console
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on /home/jorgen/Desktop/pdf to docx/pdf to docx.csproj...
  Restoring packages for /home/jorgen/Desktop/pdf to docx/pdf to docx.csproj...
  Generating MSBuild file /home/jorgen/Desktop/pdf to docx/obj/pdf to docx.csproj.nuget.g.props.
  Generating MSBuild file /home/jorgen/Desktop/pdf to docx/obj/pdf to docx.csproj.nuget.g.targets.
  Restore completed in 441.36 ms for /home/jorgen/Desktop/pdf to docx/pdf to docx.csproj.

Restore succeeded.

jorgen@jorgen-linux:~/Desktop/pdf to docx$ dotnet run
Hello World!
jorgen@jorgen-linux:~/Desktop/pdf to docx$
```

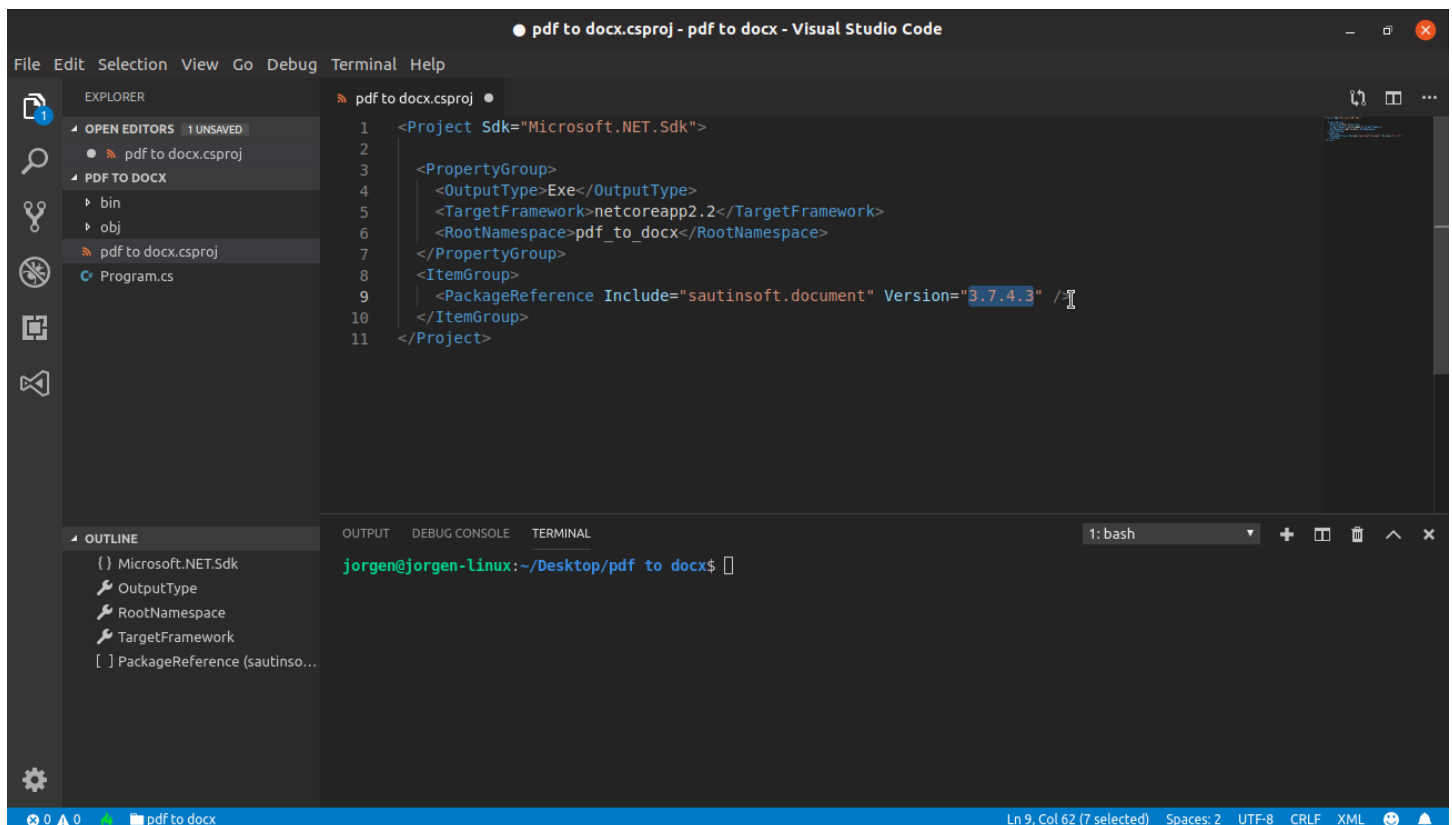
You can see the typical “Hello world!” message.

Now we are going to convert this simple application into something more interesting.

We’ll transform it into an application that will convert a pdf file to a docx file.

First of all, we need to add the package reference to the ***sautinsoft.document*** assembly using Nuget.

In order to do it, follow to the ***Explorer*** and open project file “***pdf to docx.csproj***” within VS Code to edit it:



```
pdf to docx.csproj - pdf to docx - Visual Studio Code
File Edit Selection View Go Debug Terminal Help
EXPLORER
  OPEN EDITORS 1 UNSAVED
  • pdf to docx.csproj
  PDF TO DOCX
    bin
    obj
    pdf to docx.csproj
    Program.cs
  OUTLINE
    ( ) Microsoft.NET.Sdk
    OutputType
    RootNamespace
    TargetFramework
    [ ] PackageReference (sautinso...)

pdf to docx.csproj
1 <Project Sdk="Microsoft.NET.Sdk">
2
3   <PropertyGroup>
4     <OutputType>Exe</OutputType>
5     <TargetFramework>netcoreapp2.2</TargetFramework>
6     <RootNamespace>pdf_to_docx</RootNamespace>
7   </PropertyGroup>
8   <ItemGroup>
9     <PackageReference Include="sautinsoft.document" Version="3.7.4.3" />
10  </ItemGroup>
11 </Project>
```

Add these lines into the file "**pdf to docx.csproj**":

```
<ItemGroup>  
    <PackageReference Include="sautinsoft.document" Version="3.7.4.3" />  
</ItemGroup>
```

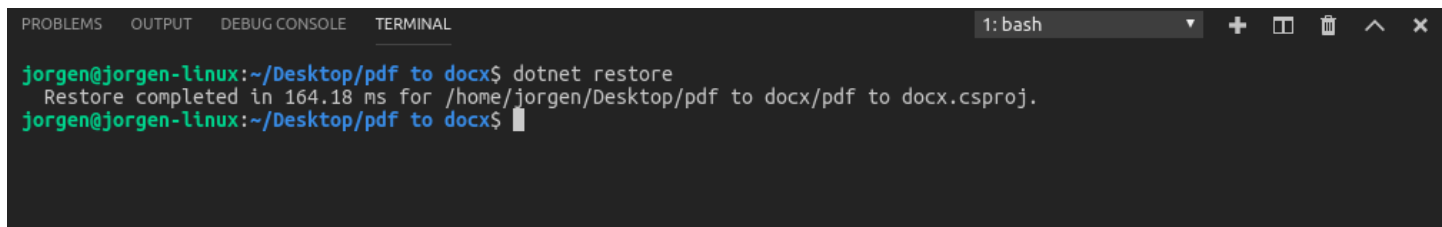
It's the reference to **sautinsoft.document** package from Nuget.

At the moment of writing this manual, the latest version of **sautinsoft.document** was 3.7.4.3. But you may specify the latest version, to know what is the latest, follow:

<https://www.nuget.org/packages/sautinsoft.document/>

At once as we've added the package reference, we have to save the "**pdf to docx.csproj**" and restore the added package.

Follow to the **Terminal** and type the command: **dotnet restore**

A screenshot of a terminal window with a dark background. The window has tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL', with 'TERMINAL' being the active tab. The terminal shows the command 'dotnet restore' being executed. The output indicates that the restore was completed in 164.18 ms for the project located at '/home/jorgen/Desktop/pdf to docx/pdf to docx.csproj'. The prompt shows the user is 'jorgen' on a 'jorgen-linux' machine, in the directory '~/Desktop/pdf to docx\$'.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: bash  
jorgen@jorgen-linux:~/Desktop/pdf to docx$ dotnet restore  
Restore completed in 164.18 ms for /home/jorgen/Desktop/pdf to docx/pdf to docx.csproj.  
jorgen@jorgen-linux:~/Desktop/pdf to docx$
```

Very important!

There are a lot of Linux varieties: Suse, Fedora, Debian, Ubuntu, etc.

In addition, there are cloud platforms: AWS Lambda, Google Cloud, Azure, Apex, etc.

Because our dll works with Graphics and using GDI+ API, you need to check, that your system has [System.Drawing.Common](#) is the graphics library which ships as part of .NET Core. On macOS and Linux, it uses [libgdiplus](#) as its back-end.

There are existing ways to install libgdiplus on macOS and Linux. On macOS, you can use the [mono-libgdiplus](#) Homebrew package; on Ubuntu Linux, you can install the [libgdiplus-dev](#) package.

🔗 Using libgdiplus on Ubuntu Linux

You can install libgdiplus on Ubuntu Linux using the Quamotion PPA. Follow these steps:

```
sudo add-apt-repository ppa:quamotion/ppa
sudo apt-get update
sudo apt-get install -y libgdiplus
```

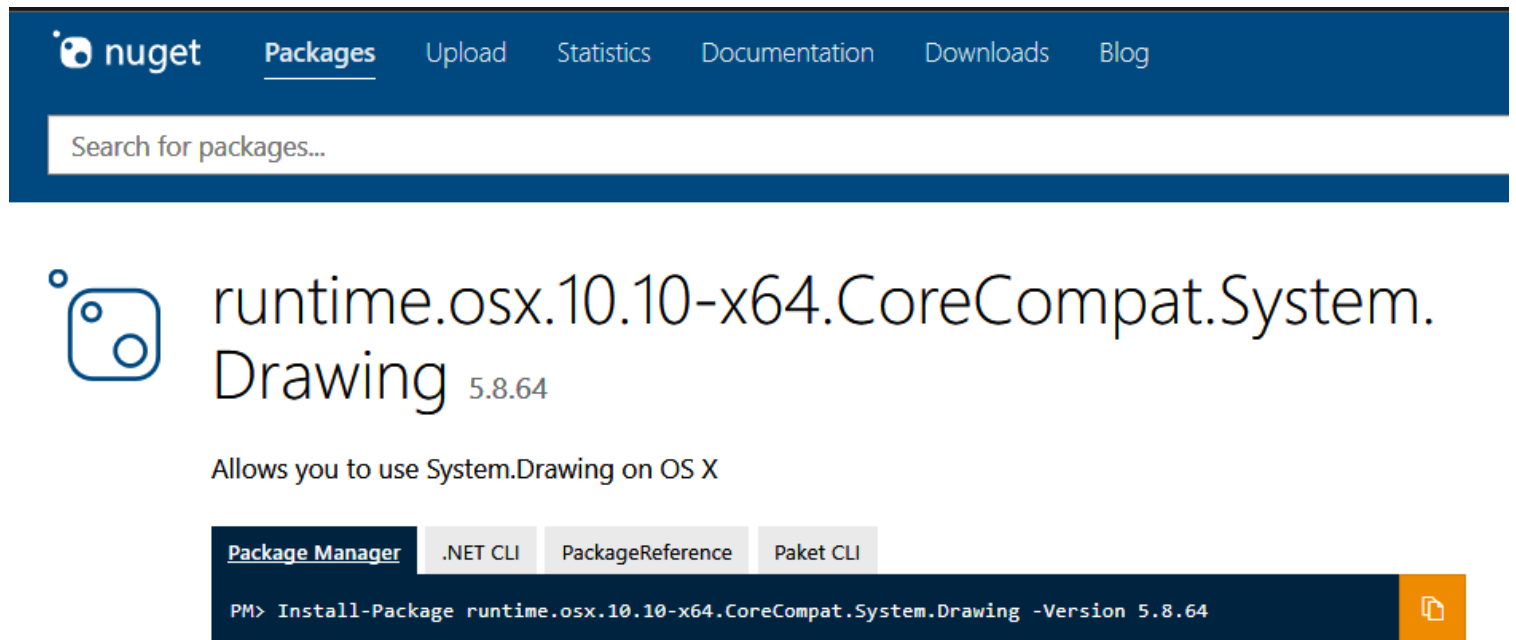
Using libgdiplus on macOS

On macOS, add a reference to the [runtime.osx.10.10-x64.CoreCompat.System.Drawing](#) package:

```
dotnet add package runtime.osx.10.10-x64.CoreCompat.System.Drawing
```

If you have installed LibGdiPlus' dll and it still doesn't work. Please add (update) this string in your project:

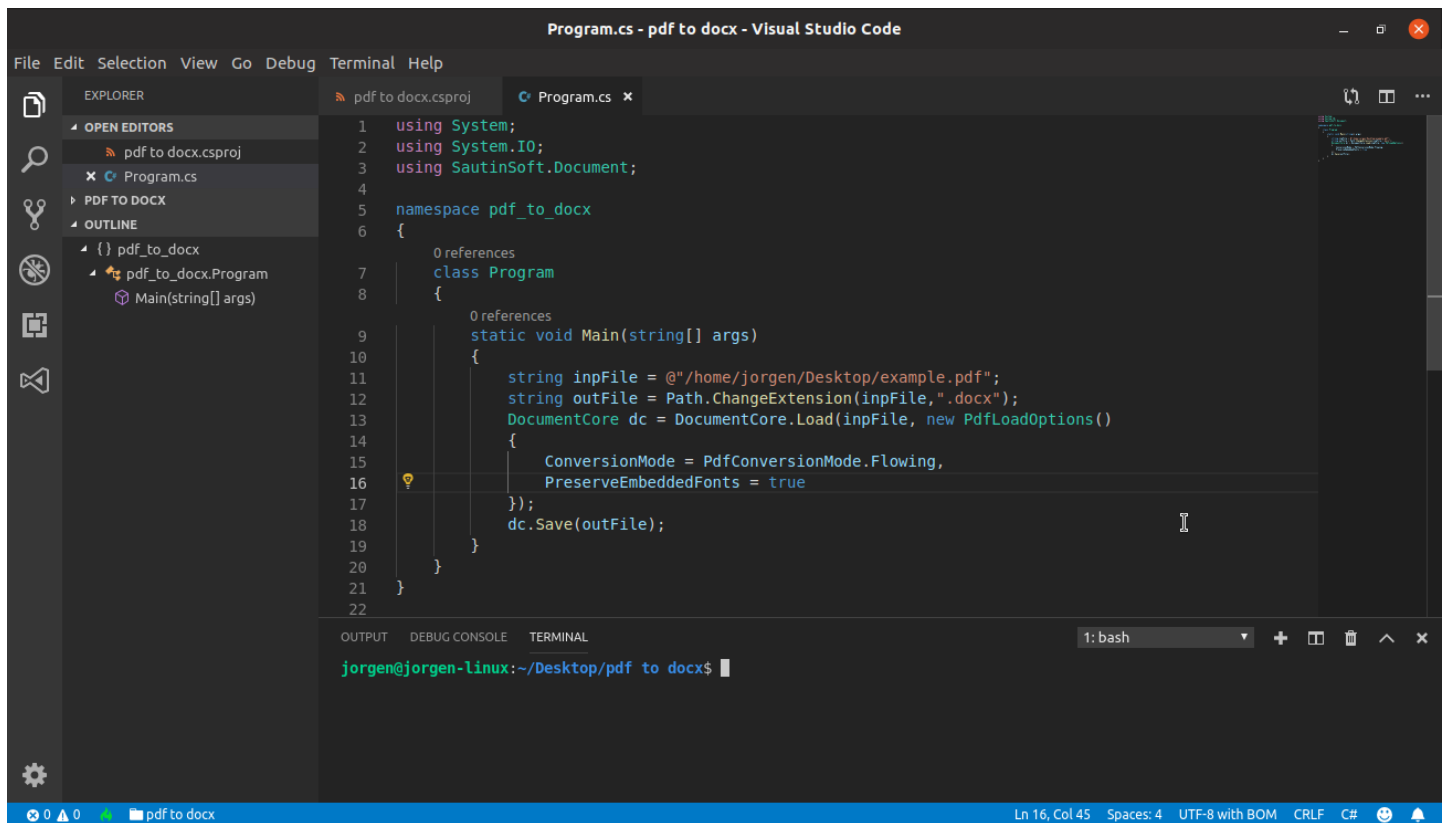
```
<PackageReference Include="runtime.osx.10.10-x64.CoreCompat.System.Drawing"
Version="5.23.62"/>
```



The screenshot shows the NuGet website interface. At the top is a dark blue navigation bar with the NuGet logo and links for Packages, Upload, Statistics, Documentation, Downloads, and Blog. Below this is a search bar with the placeholder text "Search for packages...". The main content area displays the package "runtime.osx.10.10-x64.CoreCompat.System.Drawing" with version "5.8.64". To the left of the package name is a small icon of a square with two circles inside. Below the package name, it says "Allows you to use System.Drawing on OS X". At the bottom, there is a row of tabs: "Package Manager" (selected), ".NET CLI", "PackageReference", and "Paket CLI". Below the tabs, a command prompt shows the command "PM> Install-Package runtime.osx.10.10-x64.CoreCompat.System.Drawing -Version 5.8.64".

Good, now our application has the reference to **sautinsoft.document** package and we can write the code to convert pdf to docx and other formats.

Follow to the **Explorer**, open the **Program.cs**, remove all the code and type the new:



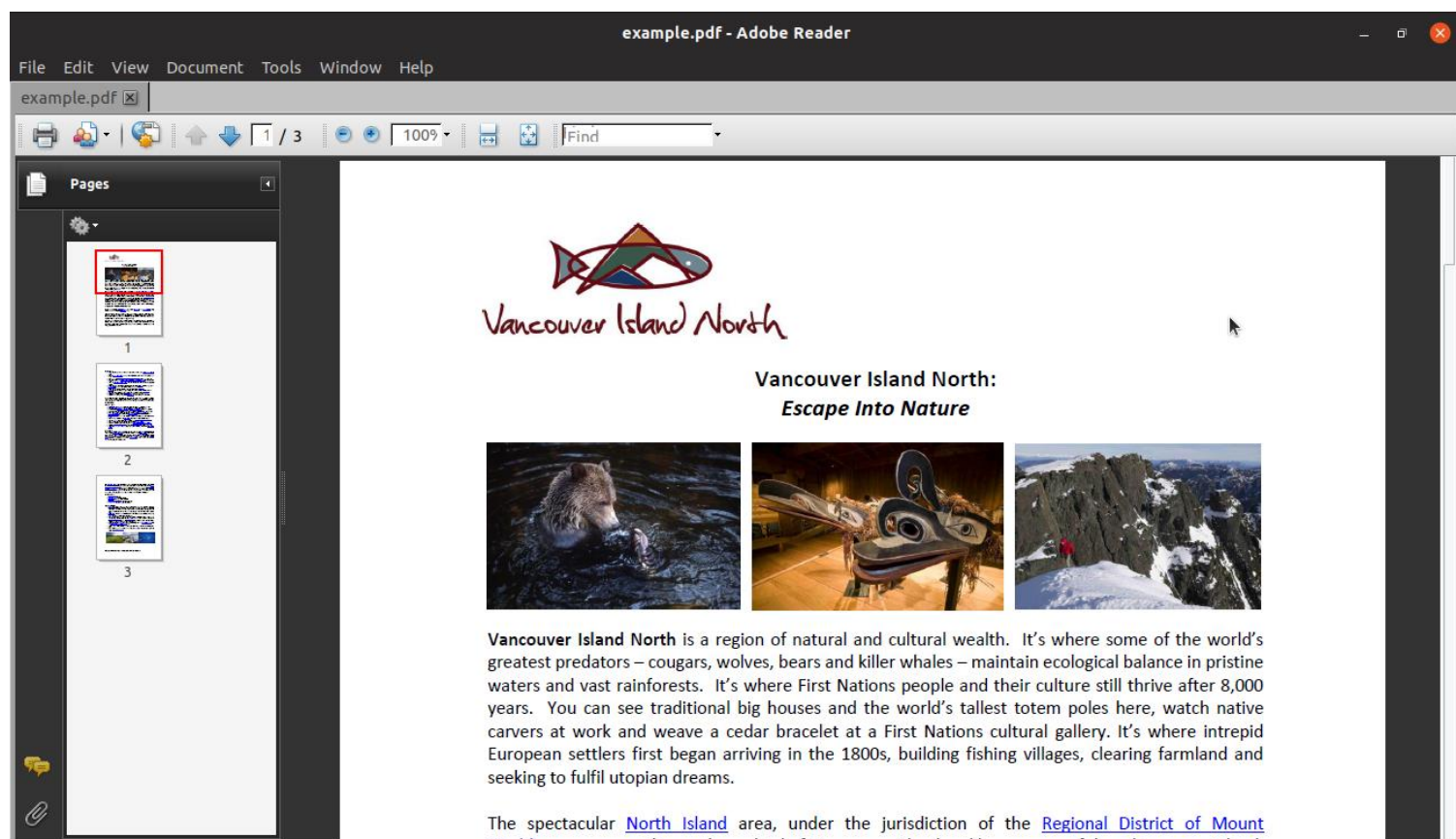
The code:

```
using System;
using System.IO;
using SautinSoft.Document;
namespace pdf_to_docx
{
    class Program
    {
        static void Main(string[] args)
        {
            string inpFile = @"/home/jorgen/Desktop/example.pdf";
            string outFile = Path.ChangeExtension(inpFile, ".docx");
            DocumentCore dc = DocumentCore.Load(inpFile, new PdfLoadOptions()
            {
                ConversionMode = PdfConversionMode.Flowing,
                PreserveEmbeddedFonts = true
            });
            dc.Save(outFile);
        }
    }
}
```

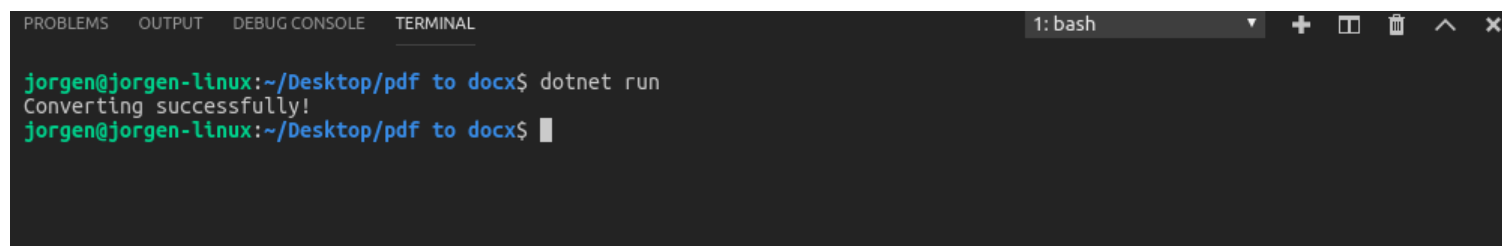
To make tests, we need an input PDF document. For our tests, let's place a PDF file with the name "example.pdf" at the Desktop.



If we open this file in the default PDF Viewer, we'll its contents:



Launch our application and convert the "example.pdf" into "example.docx", type the command: **dotnet run**

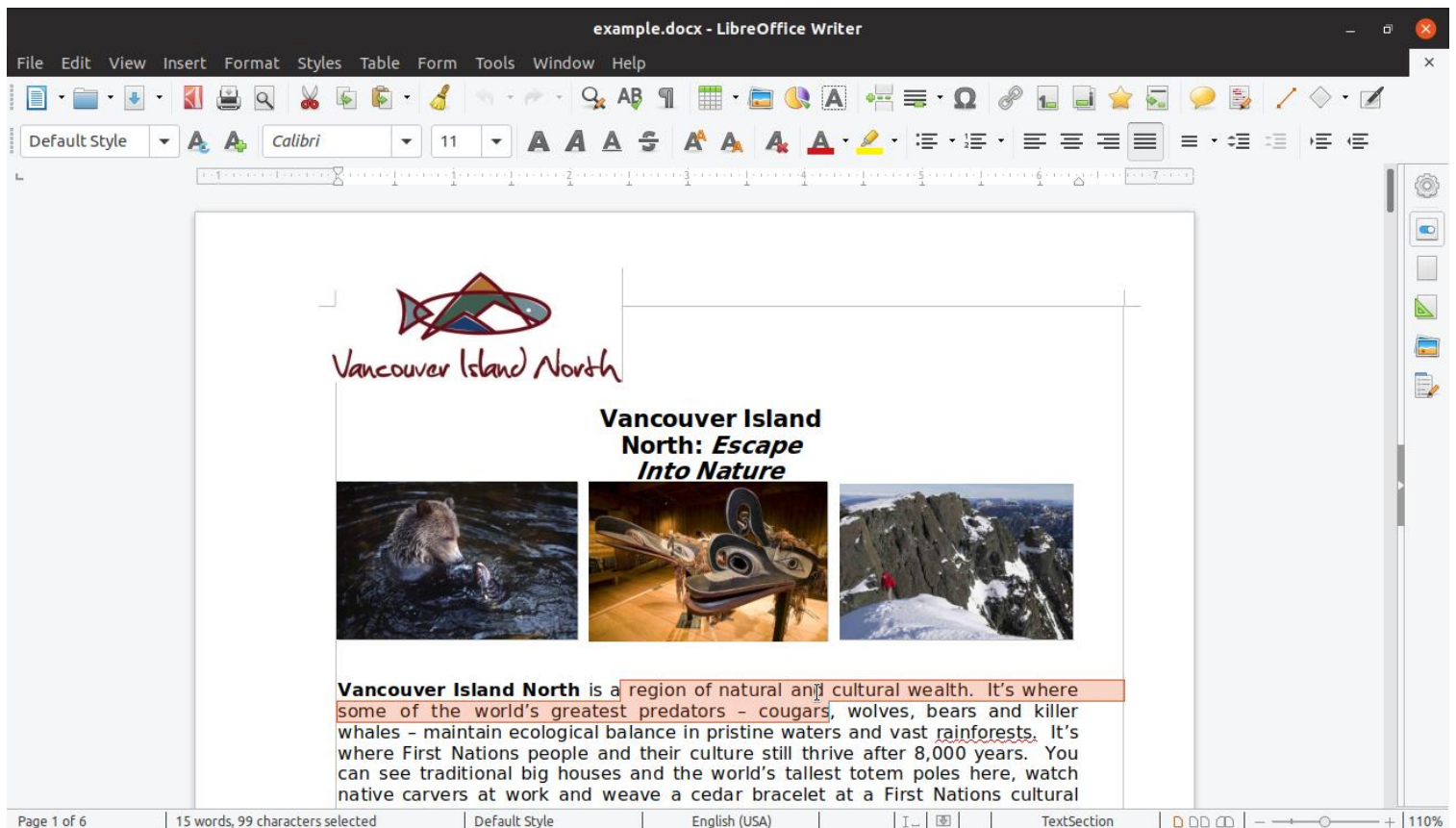


If you don't see any exceptions, everything is fine and we can check the result produced by the Document .Net library.

The new file "example.docx" has to appear on the Desktop:



Open the result in LibreOffice:



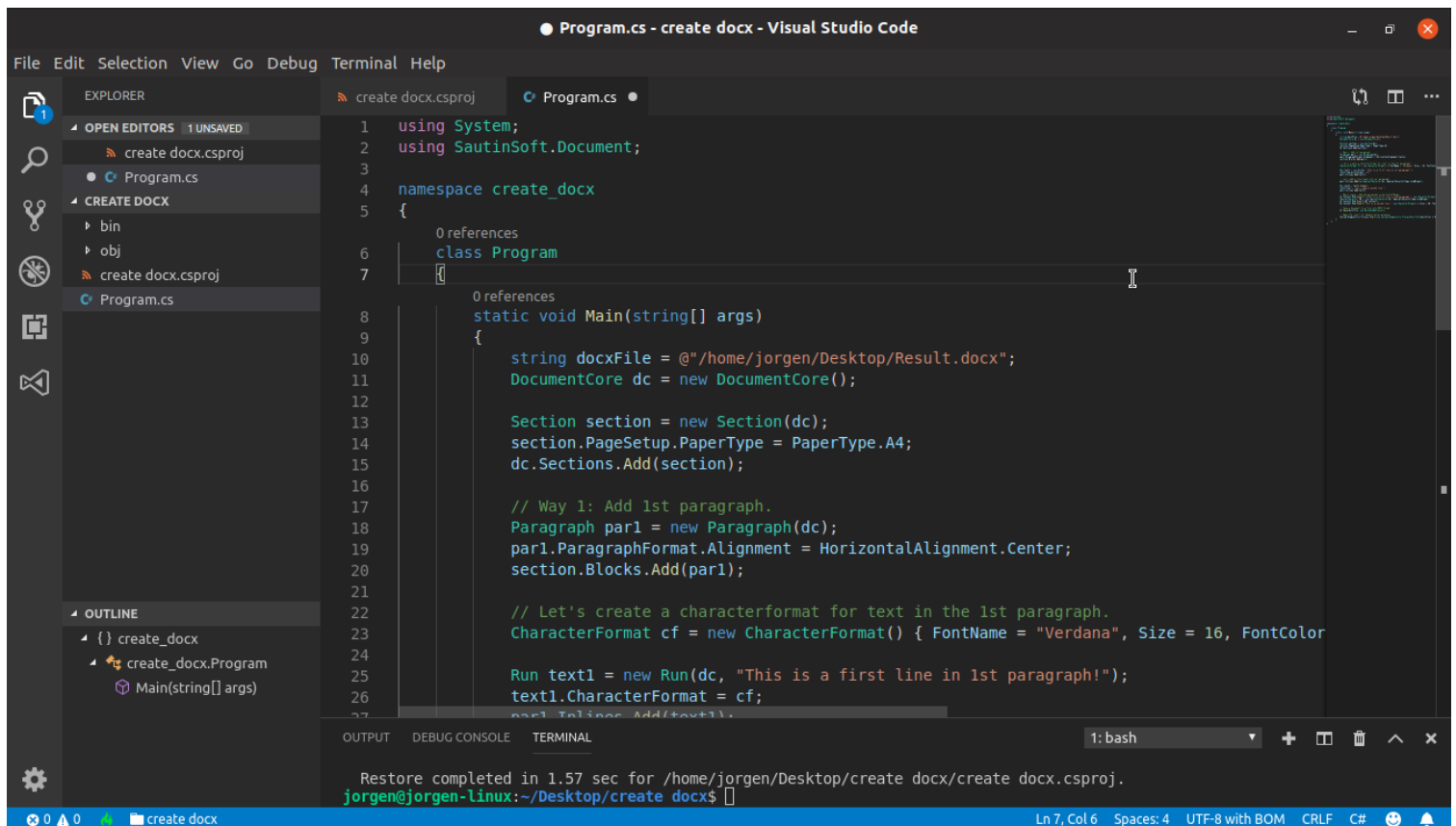
Well done! You have created the "PDF to DOCX" application under Linux!

3. Creating new DOCX document from scratch

Now we're going to develop a new application that will be able to create a new docx document and to add some content in it.

As we did before, create a new folder and name it "**create docx**". Open this folder within VS Code and repeat the same steps as done before, creating a new console project, adding dependencies and so on.

Once you have done and are ready to code your new program, type this within **Program.cs** as shown in the picture below (the complete code is after the picture):

The screenshot shows the Visual Studio Code interface with a project named 'create docx'. The Explorer sidebar on the left shows the file structure with 'create.docx.csproj' and 'Program.cs'. The Outline sidebar shows the 'create_docx' namespace containing a 'Program' class with a 'Main' method. The main editor displays the code for 'Program.cs', which includes using statements for 'System' and 'SautinSoft.Document', a namespace declaration 'create_docx', and a 'Program' class with a 'Main' method. The 'Main' method creates a 'DocumentCore' object, adds a 'Section', and then adds a paragraph with a specific format. The terminal at the bottom shows the command 'dotnet restore' being executed successfully.

```
1 using System;
2 using SautinSoft.Document;
3
4 namespace create_docx
5 {
6     0 references
7     class Program
8     {
9         0 references
10        static void Main(string[] args)
11        {
12            string docxFile = @"/home/jorgen/Desktop/Result.docx";
13            DocumentCore dc = new DocumentCore();
14
15            Section section = new Section(dc);
16            section.PageSetup.PaperType = PaperType.A4;
17            dc.Sections.Add(section);
18
19            // Way 1: Add 1st paragraph.
20            Paragraph par1 = new Paragraph(dc);
21            par1.ParagraphFormat.Alignment = HorizontalAlignment.Center;
22            section.Blocks.Add(par1);
23
24            // Let's create a characterformat for text in the 1st paragraph.
25            CharacterFormat cf = new CharacterFormat() { FontName = "Verdana", Size = 16, FontColor
26
27            Run text1 = new Run(dc, "This is a first line in 1st paragraph!");
28            text1.CharacterFormat = cf;
29            par1.Blocks.Add(text1);
30        }
31    }
32 }
```

The code:

```
using System;
using SautinSoft.Document;
```

```
namespace create_docx
{
    class Program
    {
        static void Main(string[] args)
```

```

{
    string docxFile = @"/home/jorgen/Desktop/Result.docx";
    DocumentCore dc = new DocumentCore();

    Section section = new Section(dc);
    section.PageSetup.PaperType = PaperType.A4;
    dc.Sections.Add(section);

    // Way 1: Add 1st paragraph.
    Paragraph par1 = new Paragraph(dc);
    par1.ParagraphFormat.Alignment = HorizontalAlignment.Center;
    section.Blocks.Add(par1);

    // Let's create a characterformat for text in the 1st paragraph.
    CharacterFormat cf = new CharacterFormat() { FontName = "Verdana", Size = 16,
    FontColor = Color.Orange };

    Run text1 = new Run(dc, "This is a first line in 1st paragraph!");
    text1.CharacterFormat = cf;
    par1.Inlines.Add(text1);

    // Let's add a line break into our paragraph.
    par1.Inlines.Add(new SpecialCharacter(dc, SpecialCharacterType.LineBreak));

    Run text2 = text1.Clone();
    text2.Text = "Let's type a second line.";
    par1.Inlines.Add(text2);

    // Way 2 (easy): Add 2nd paragraph using ContentRange.
    dc.Content.End.Insert("\nThis is a first line in 2nd paragraph.", new
    CharacterFormat() { Size = 25, FontColor = Color.Blue, Bold = true });
    SpecialCharacter lBr = new SpecialCharacter(dc, SpecialCharacterType.LineBreak);
    dc.Content.End.Insert(lBr.Content);
    dc.Content.End.Insert("This is a second line.", new CharacterFormat() { Size = 20,
    FontColor = Color.DarkGreen, UnderlineStyle = UnderlineType.Single });

    // Save a document to a file into DOCX format.
    dc.Save(docxFile, new DocxSaveOptions());

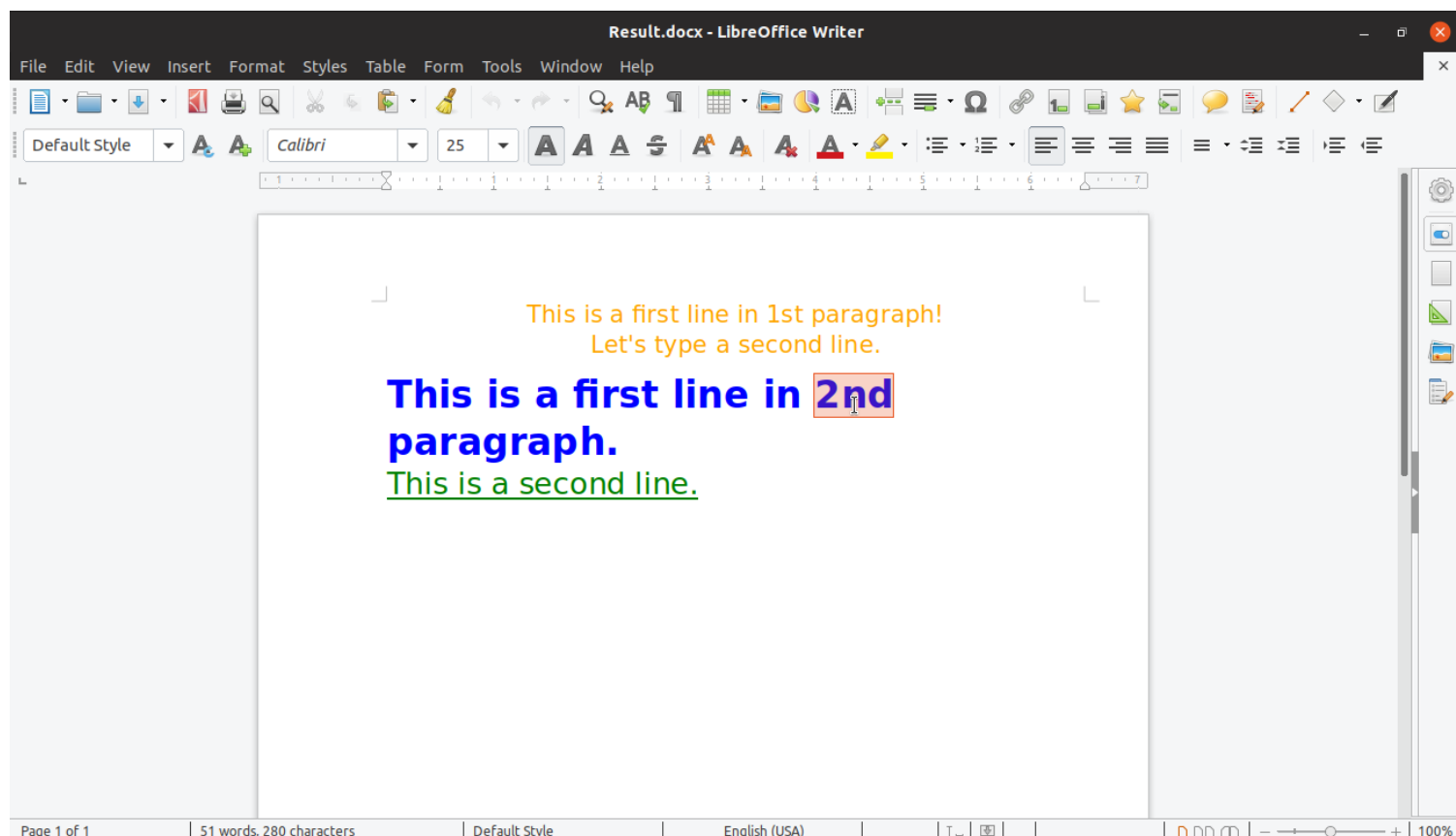
    // Open the result for demonstration purposes.
    System.Diagnostics.Process.Start(new
    System.Diagnostics.ProcessStartInfo(docxFile) { UseShellExecute = true });
}

```

```
}  
}
```

Launch our application to create a new DOCX document, type the command: ***dotnet run***

If you don't see any exceptions, the produced DOCX file will be opened automatically in the default DOCX viewer (in our case it's LibreOffice):



Well done! You have created the "Create DOCX" application under Linux!

If you have any troubles or need extra code, or help, don't hesitate to ask our SautinSoft Team at support@sautinsoft.com!