

Document .Net

(Multi-platform .Net library)

[SautinSoft](http://SautinSoft.com)

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	10

1. Preparing environment

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

We need to install .NET 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:

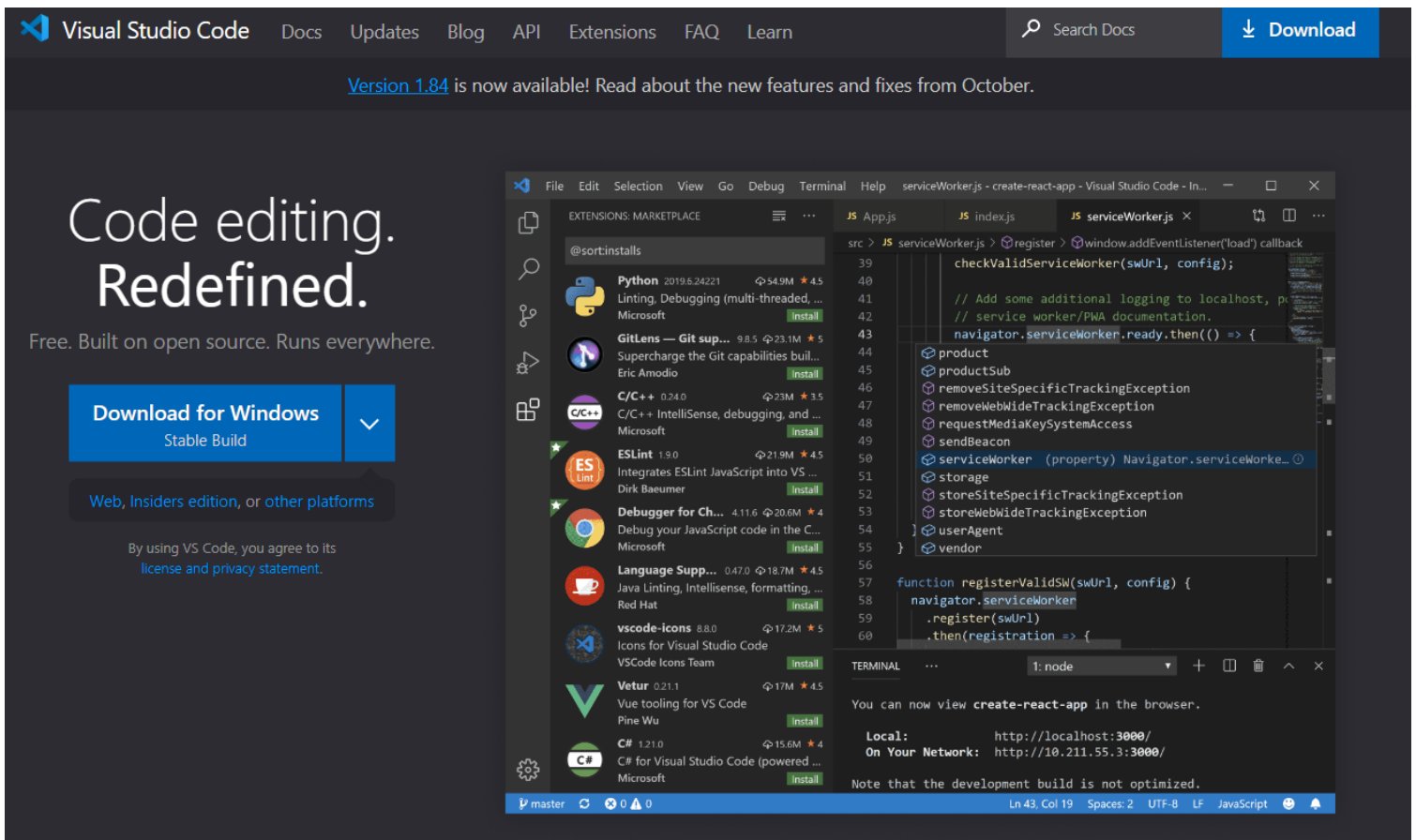
[Install .NET SDK for Linux.](#)



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

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

```
linuxconfig@linuxconfig-org: ~  
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"](#).

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

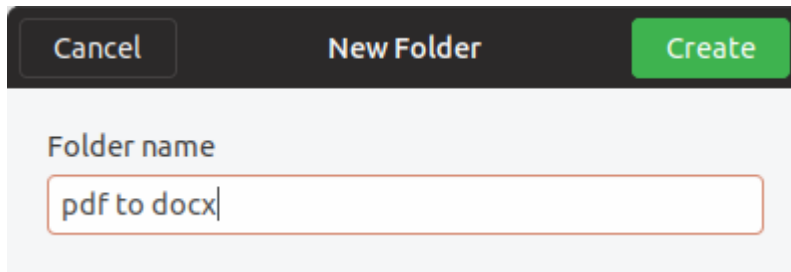
[Get Started with C# and Visual Studio Code](#)

Not only is possible to create .NET 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 2022.

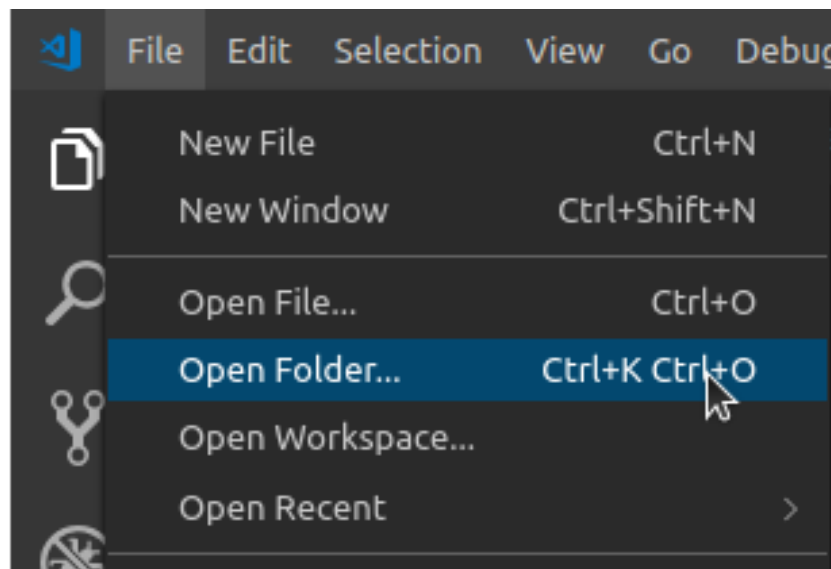
2. Creating “Convert PDF to DOCX” application

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

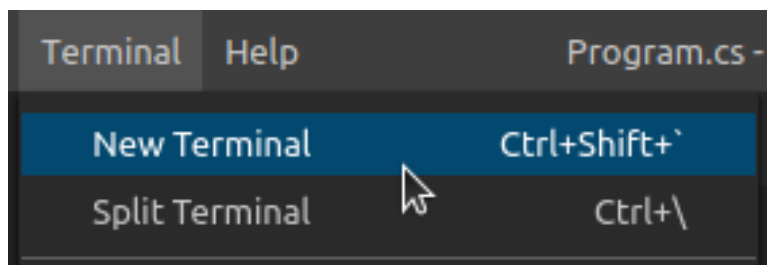
For example, let’s create the folder “**pdf to docx**” on Desktop (Right click-> New Folder):



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

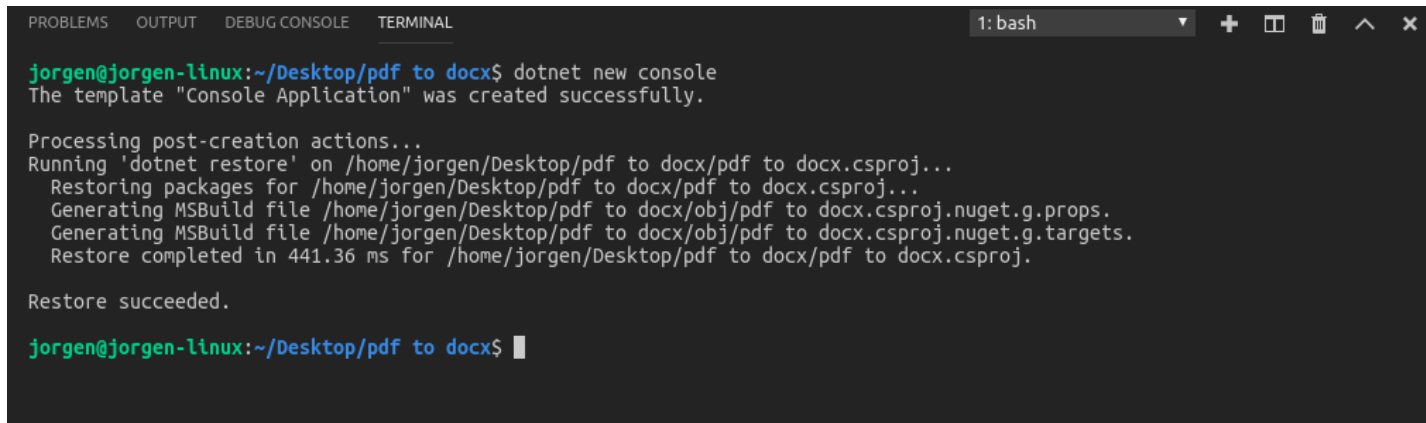


Now, open the integrated console – the Terminal: follow to the menu **Terminal -> New Terminal** (or press Ctrl+Shift+’):



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

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

A screenshot of a terminal window with a dark background. The title bar shows '1: bash'. The prompt is 'jorgen@jorgen-linux:~/Desktop/pdf to docx\$'. The command 'dotnet new console' has been entered, and the output shows that the 'Console Application' template was created successfully. It then lists post-creation actions: running 'dotnet restore', restoring packages, generating MSBuild files, and restoring the project in 441.36 ms. The final output is 'Restore succeeded.' followed by a new prompt.

```
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.

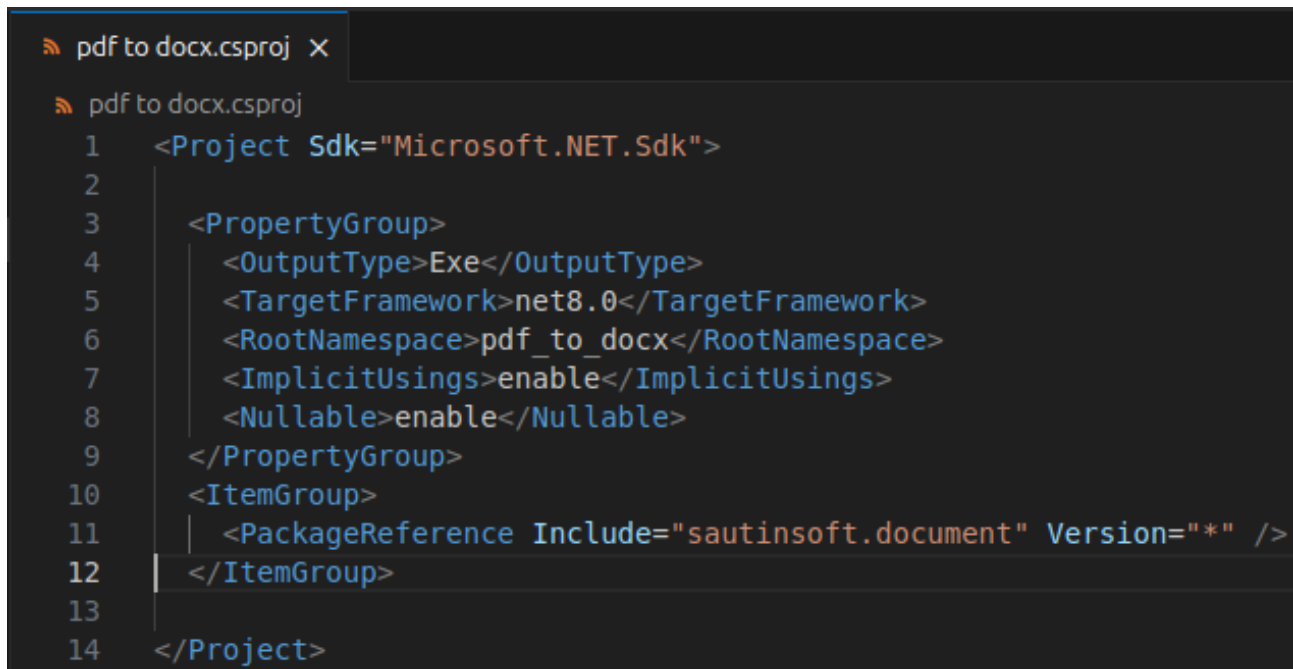
jorgen@jorgen-linux:~/Desktop/pdf to docx$
```

Now we are going to modify this simple application 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 or the file SautinSoft.Document.dll with additional references.

In order to do it, follow to the **Explorer** and open project file "**pdf to docx.csproj**":

In the first case (NuGet):

A screenshot of a Visual Studio Code editor window. The file 'pdf to docx.csproj' is open. The XML content shows a project configuration for Microsoft.NET.Sdk. It includes a PropertyGroup with OutputType set to Exe, TargetFramework set to net8.0, RootNamespace set to pdf_to_docx, ImplicitUsings set to enable, and Nullable set to enable. An ItemGroup contains a PackageReference to 'sautinsoft.document' with Version set to '*'. The project is closed with the closing tag for Project.

```
pdf to docx.csproj
1 <Project Sdk="Microsoft.NET.Sdk">
2
3   <PropertyGroup>
4     <OutputType>Exe</OutputType>
5     <TargetFramework>net8.0</TargetFramework>
6     <RootNamespace>pdf_to_docx</RootNamespace>
7     <ImplicitUsings>enable</ImplicitUsings>
8     <Nullable>enable</Nullable>
9   </PropertyGroup>
10  <ItemGroup>
11    <PackageReference Include="sautinsoft.document" Version="*" />
12  </ItemGroup>
13
14 </Project>
```

In the second case (SautinSoft.Document.dll):

```
pdf to docx.csproj X
pdf to docx.csproj
1  <Project Sdk="Microsoft.NET.Sdk">
2
3      <PropertyGroup>
4          <OutputType>Exe</OutputType>
5          <TargetFramework>net8.0</TargetFramework>
6          <RootNamespace>pdf_to_docx</RootNamespace>
7          <ImplicitUsings>enable</ImplicitUsings>
8          <Nullable>enable</Nullable>
9      </PropertyGroup>
10     <ItemGroup>
11         <PackageReference Include="Pkcs11Interop" Version="5.1.2" />
12         <PackageReference Include="Portable.BouncyCastle" Version="1.9.0" />
13         <PackageReference Include="SkiaSharp" Version="2.88.7" />
14         <PackageReference Include="SkiaSharp.HarfBuzz" Version="2.88.7" />
15         <PackageReference Include="SkiaSharp.NativeAssets.Linux" Version="2.88.7" />
16         <PackageReference Include="Svg.Skia" Version="1.0.0.18" />
17         <PackageReference Include="System.IO.Packaging" Version="4.5.0" />
18         <PackageReference Include="System.Resources.Extensions" Version="6.0.0" />
19         <PackageReference Include="System.Text.Encoding.CodePages" Version="4.5.0" />
20         <Reference Include="SautinSoft.Document">
21             <HintPath>/YourPathToDll/SautinSoft.Document.dll</HintPath>
22         </Reference>
23     </ItemGroup>
24
25 </Project>
```

At once as we’ve added the package references, we have to save the “*pdf to docx.csproj*” and restore the added packages.

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

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
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$
```

Good, now our application has all the references 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:

```
pdf to docx.csproj  Program.cs
Program.cs > ...
1  using SautinSoft.Document;
2  namespace pdf_to_docx
3  {
4      0 references
5      class Program
6      {
7          0 references
8          static void Main(string[] args)
9          {
10             string inpFile = "/example.pdf";
11             string outFile = Path.ChangeExtension(inpFile, ".docx");
12             DocumentCore dc = DocumentCore.Load(inpFile);
13             dc.Save(outFile);
14         }
15     }
16 }
```

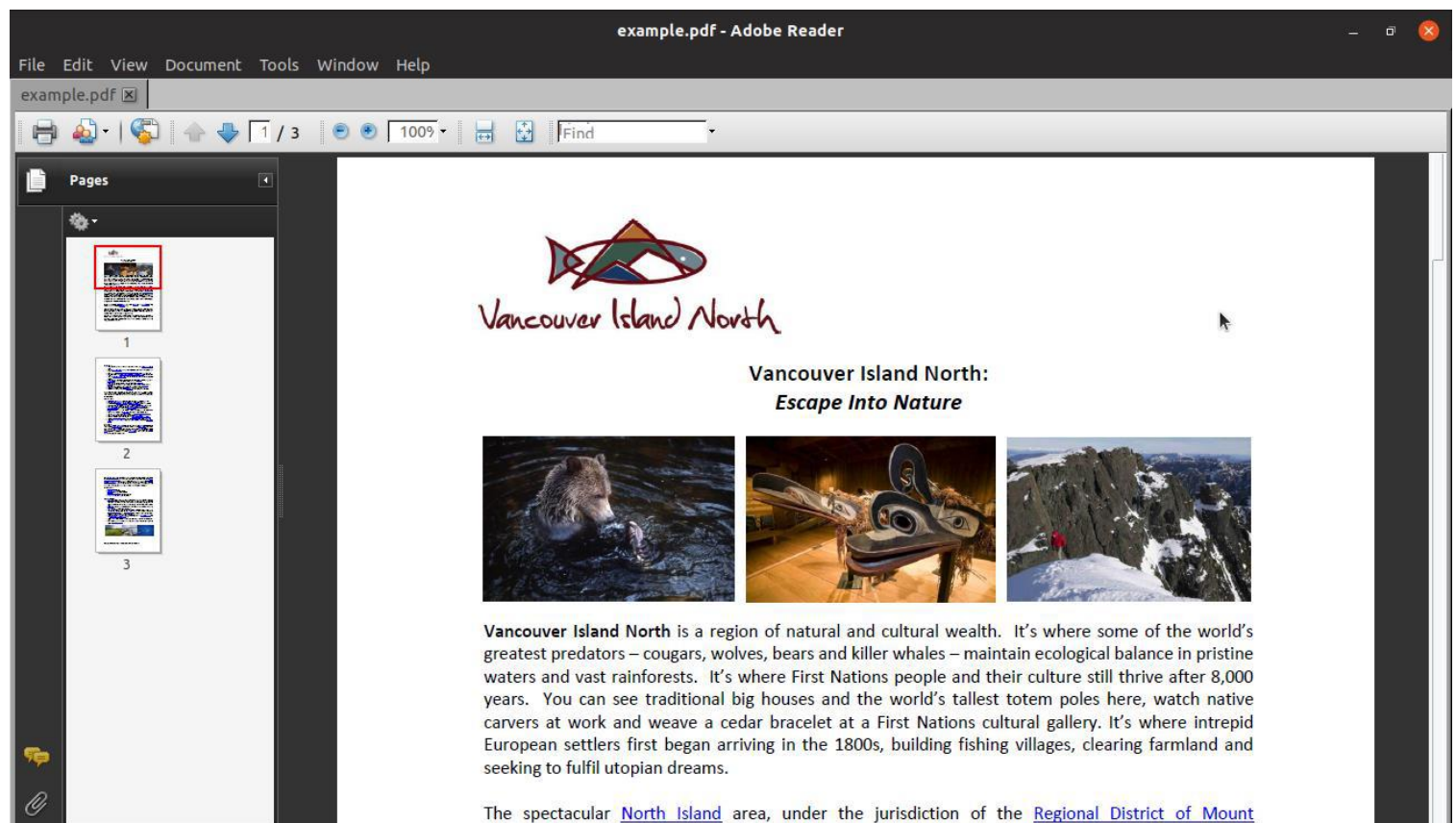

The code:

```
using SautinSoft.Document;
namespace pdf_to_docx
{
    class Program
    {
        static void Main(string[] args)
        {
            string inpFile = "/example.pdf";
            string outFile = Path.ChangeExtension(inpFile, ".docx");
            DocumentCore dc = DocumentCore.Load(inpFile);
            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**

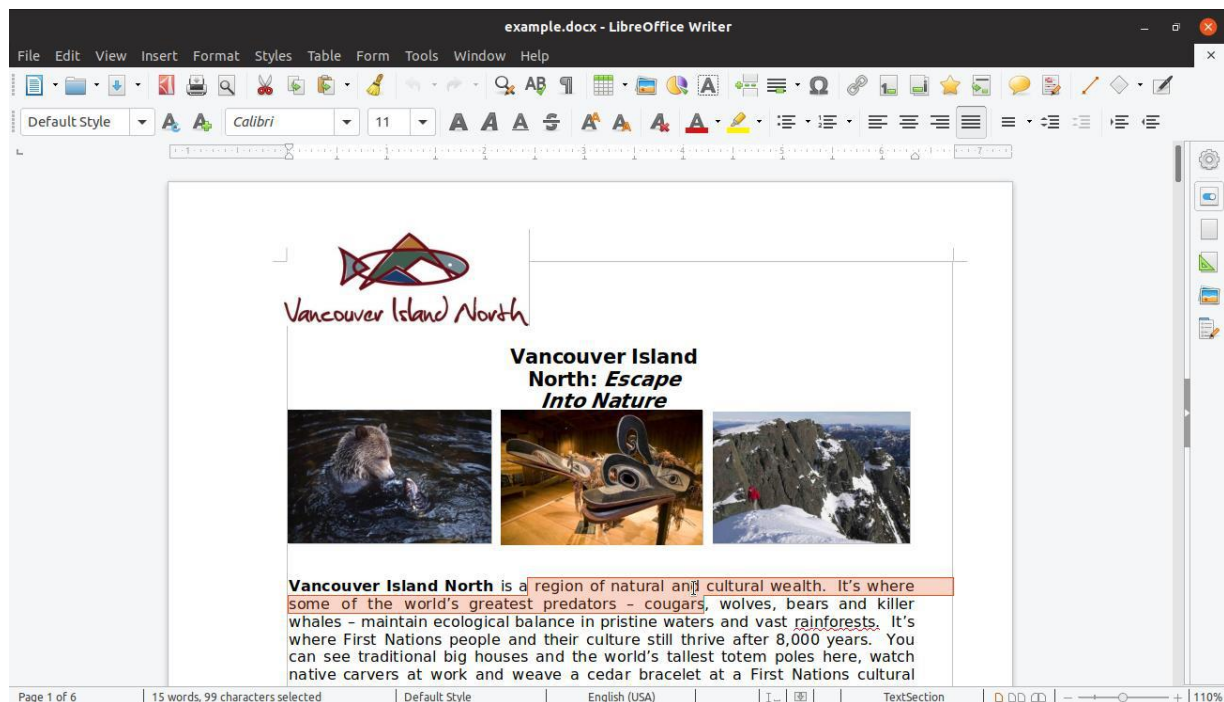
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: bash
jorgen@jorgen-linux:~/Desktop/pdf to docx$ dotnet run
Converting successfully!
jorgen@jorgen-linux:~/Desktop/pdf to docx$
```

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:



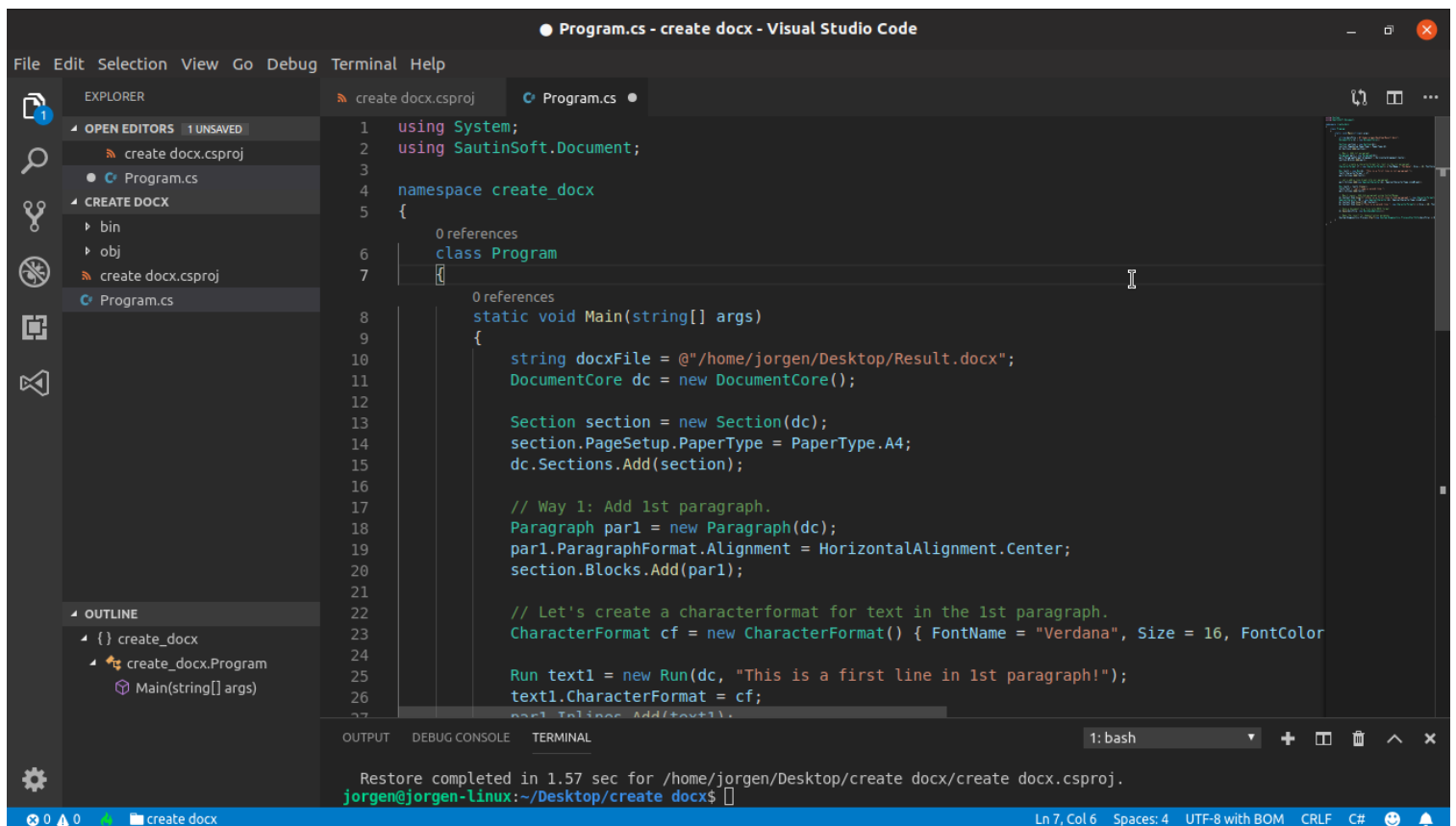
Well done! You have created the "PDF to DOCX" application under Linux!

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):

A screenshot of the Visual Studio Code editor interface. The title bar reads "Program.cs - create docx - Visual Studio Code". The Explorer sidebar on the left shows a project structure with folders "bin" and "obj", and files "create.docx.csproj" and "Program.cs". The Outline sidebar shows a class "create_docx" with a method "Main(string[] args)". The main editor area displays the following C# code:

```
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.Inlines.Add(text1);
30        }
31    }
32 }
```

The bottom of the window shows a terminal with the command "1: bash" and a status bar indicating "Ln 7, Col 6", "Spaces: 4", "UTF-8 with BOM", "CRLF", and "C#".

The code:

```
using System;
using SautinSoft.Document;

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

```

// Change to your output path
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 });

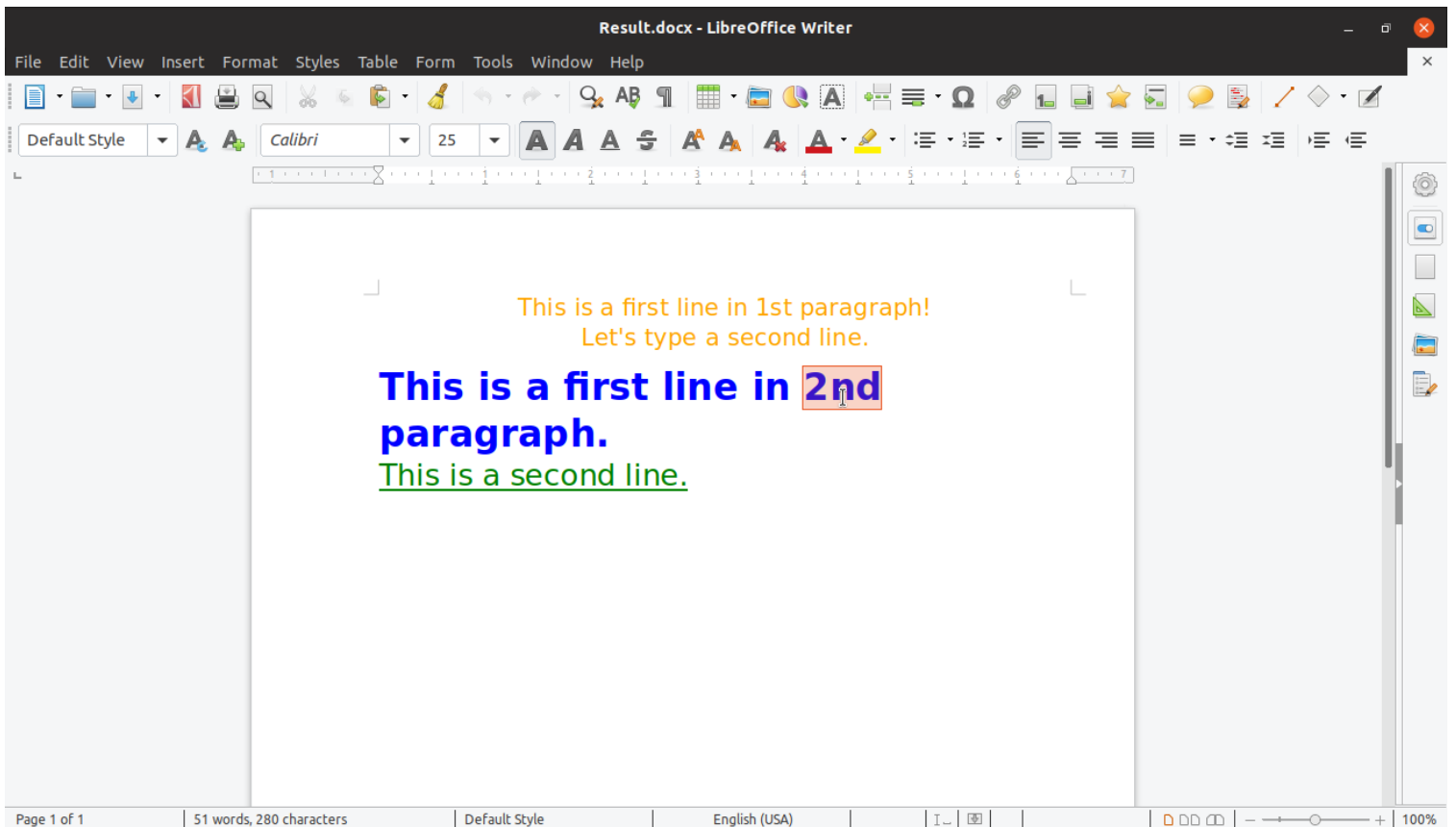
// Save a document to a file into DOCX format.
dc.Save(docxFile, new DocxSaveOptions());

// 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!