# Creating PDF document in .NET Core Web API

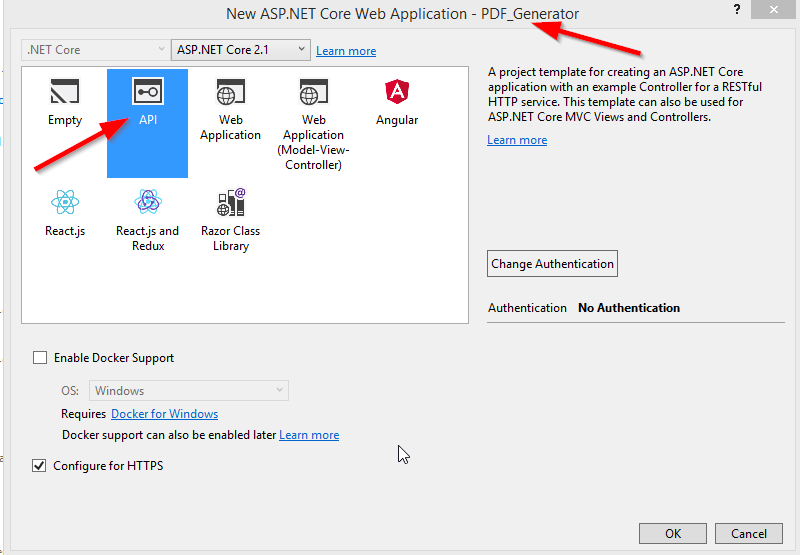
Let’s imagine that we have a .NET Core Web API project in which we need to generate a PDF report. Even though it shouldn’t suppose to be too hard to do something like that, we could end up in loosing to much time if we don’t know how to do it properly.

In this article, we are going to show you how to use DinkToPDF library to easily generate PDF documents while working on the .NET Core Web API project.

So, without further ado, let’s dive right into the fun part.

## Basic Project Preparations

Let’s start from the very beginning, by creating a new .NET Core Web Api project named PDF\_Generator:



After the project creation, let’s modify the launchSettings.json file to disable our browser to start automatically and to change the applicationUrl property to localhost:5000:

{

"$schema": "http://json.schemastore.org/launchsettings.json",

"iisSettings": {

"windowsAuthentication": false,

"anonymousAuthentication": true,

"iisExpress": {

"applicationUrl": "http://[::]:5000",

"sslPort": 44320

}

},

"profiles": {

"IIS Express": {

"commandName": "IISExpress",

"launchBrowser": false,

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

},

"PDF\_Generator": {

"commandName": "Project",

"launchBrowser": false,

"applicationUrl": "http://[::]:5000",

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

}

}

}

## DinkToPdf Library Configuration

DinkToPdf is a cross-platform oriented library which is the wrapper for the Webkit HTML to PDF library. It uses WebKit engine to convert HTML to PDF.

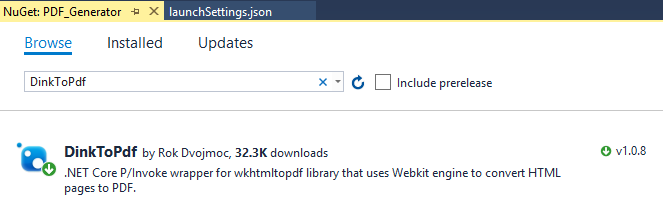
It will allow us to create a PDF document from our HTML string that we generate in the .NET Core project, or to create a PDF document form existing HTML page. Furthermore, we can download created PDF document or save it on a certain location or return a new HTML page with the PDF content.

We are going to cover all these features in this article.

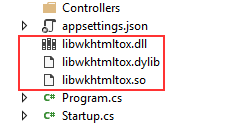
So, let’s first install the DinkToPdf library:

PM> Install-Package DinkToPdf

Or search for DinkToPdf inside the Nuget Package window:



After the installation completes, we need to import native library files to our root project. We can find those files in our source project in the NativeLibrary folder. Inside we will find two folders 32bit and 64bit, so we need to choose the appropriate library for our OS. We are going to choose the files from the 64bit folder:



Finally, we need to register this library with our IoC container in the StartUp class:

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

services.AddSingleton(typeof(IConverter), new SynchronizedConverter(new PdfTools()));

}

Excellent.

We have everything in place and we are ready for the coding part.

## Preparing Data for the PDF Document

In a real-world project, we can collect data from the database or receive it from other API. But for the sake of simplicity, we are going to collect data for our PDF document from the local storage. Then we are going to create an HTML template and store it in the PDF document.

So let’s first create a new folder Models and inside it the Employee.cs file. We are going to modify it:

namespace PDF\_Generator.Models

{

public class Employee

{

public string Name { get; set; }

public string LastName { get; set; }

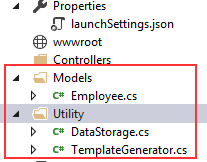
public int Age { get; set; }

public string Gender { get; set; }

}

}

To continue, we are going to create a new folder Utility and two class files inside it DataStoage.cs and TemplateGenerator.cs. Complete structure should look like this:



Now, let’s modify the DataStorage.cs file:

using PDF\_Generator.Models;

using System.Collections.Generic;

namespace PDF\_Generator.Utility

{

public static class DataStorage

{

public static List<Employee> GetAllEmployess()

{

return new List<Employee>

{

new Employee { Name="Mike", LastName="Turner", Age=35, Gender="Male"},

new Employee { Name="Sonja", LastName="Markus", Age=22, Gender="Female"},

new Employee { Name="Luck", LastName="Martins", Age=40, Gender="Male"},

new Employee { Name="Sofia", LastName="Packner", Age=30, Gender="Female"},

new Employee { Name="John", LastName="Doe", Age=45, Gender="Male"}

};

}

}

}

In the code avobe we just return a list of employees which will be displayed inside the HTML template.

### HTML Template Generation

We want to generate the HTML template, so we need to modify the TemplateGenerator.cs file:

using System.Text;

namespace PDF\_Generator.Utility

{

public static class TemplateGenerator

{

public static string GetHTMLString()

{

var employees = DataStorage.GetAllEmployess();

var sb = new StringBuilder();

sb.Append(@"

<html>

<head>

</head>

<body>

<div class='header'><h1>This is the generated PDF report!!!</h1></div>

<table align='center'>

<tr>

<th>Name</th>

<th>LastName</th>

<th>Age</th>

<th>Gender</th>

</tr>");

foreach (var emp in employees)

{

sb.AppendFormat(@"<tr>

<td>{0}</td>

<td>{1}</td>

<td>{2}</td>

<td>{3}</td>

</tr>", emp.Name, emp.LastName, emp.Age, emp.Gender);

}

sb.Append(@"

</table>

</body>

</html>");

return sb.ToString();

}

}

}

In the code above we are fetching data from our static DataStorage class and fill our template with it. The HTML template is nothing more than a pure HTML code.

But we want to style our table and h1 tag as well, so let’s create new folder assets and inside it a new styles.css file and modify it:

.header {

text-align: center;

color: green;

padding-bottom: 35px;

}

table {

width: 80%;

border-collapse: collapse;

}

td, th {

border: 1px solid gray;

padding: 15px;

font-size: 22px;

text-align: center;

}

table th {

background-color: green;

color: white;

}

This CSS file is going to be loaded later in the Controller class.

That is it, we have our HTML template to use in PDF creation and we can continue to the Controller logic.

## Saving the PDF Document on the Local Storage

In the Controllers folder, we are going to create a new empty API controller PdfCreatorController:

namespace PDF\_Generator.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class PdfCreatorController : ControllerBase

{

}

}

Let’s now modify the PdfCreatorController to support creating of PDF document and its saving to local drive:

using DinkToPdf;

using DinkToPdf.Contracts;

using Microsoft.AspNetCore.Mvc;

using PDF\_Generator.Utility;

using System.IO;

namespace PDF\_Generator.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class PdfCreatorController : ControllerBase

{

private IConverter \_converter;

public PdfCreatorController(IConverter converter)

{

\_converter = converter;

}

[HttpGet]

public IActionResult CreatePDF()

{

var globalSettings = new GlobalSettings

{

ColorMode = ColorMode.Color,

Orientation = Orientation.Portrait,

PaperSize = PaperKind.A4,

Margins = new MarginSettings { Top = 10 },

DocumentTitle = "PDF Report",

Out = @"D:\PDFCreator\Employee\_Report.pdf"

};

var objectSettings = new ObjectSettings

{

PagesCount = true,

HtmlContent = TemplateGenerator.GetHTMLString(),

WebSettings = { DefaultEncoding = "utf-8", UserStyleSheet = Path.Combine(Directory.GetCurrentDirectory(), "assets", "styles.css") },

HeaderSettings = { FontName = "Arial", FontSize = 9, Right = "Page [page] of [toPage]", Line = true },

FooterSettings = { FontName = "Arial", FontSize = 9, Line = true, Center = "Report Footer" }

};

var pdf = new HtmlToPdfDocument()

{

GlobalSettings = globalSettings,

Objects = { objectSettings }

};

\_converter.Convert(pdf);

return Ok("Successfully created PDF document.");

}

}

}

### Code Explanation

In the code above we first inject our registered Converter with the Dependency Injection inside our constructor by using IConverter interface. Then we create two objects globalSettings and objectSettings to use them as a configuration in the HtmlToPdfDcoument. Finally, we convert our pdf configuration into a real PDF Document on our local machine. Now let’s talk about the GlobalSettings and ObjectSettings classes.

### About the GlobalSettings Class

The GlobalSettings class consists of the configuration properties for the PDF document overall. We use just a couple of those properties to set up the color mode, orientation, paper size, document title etc… but if we go to the implementation of the GlobalSettings class we are going to find more of those properties.

The Out property is very important if we want to save our file on a local machine. So we need to mention it and to give it a path where we want our document to reside. If we mention the Out property then we can use \_converter.Convert(pdf); to convert our document. We are going to find out how this will differ once we try to show our PDF document inside a browser.

One more important note is that we need to have all the folders in our path previously created or the creation won’t work. So in our example where we create a PDF document in the D: drive in PDFCreator folder, we had to create the PDFCreator folder prior to PDF document creation.

### About the ObjectSettings Class

The ObjectSettings class consists of the properties related to the content of the PDF document. So, we can setup do we want a page count to show in our document, how to format headers and footers, what is the body content of our document (HtmlContent property) or what are the web settings for our document.

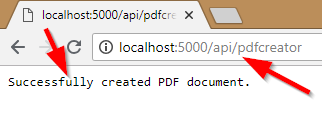
Of course, those are not all of the configuration properties but they are quite enough for our article.

In the ObjectSettings class, the HtmlContent property is very important one. It contains our generated HTML template and shows the main body of a PDF document.

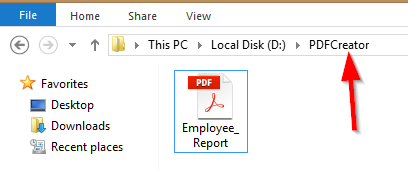
WebSettings is pretty important as well, especially if we have external CSS file for the styling as we do. In this property, we set up the encoding of our document and provide the path to our CSS file. If we inspect this property, we are going to find out more settings that we can configure like the background or should we load images or what is the minimum font size etc…

### Inspecting Results

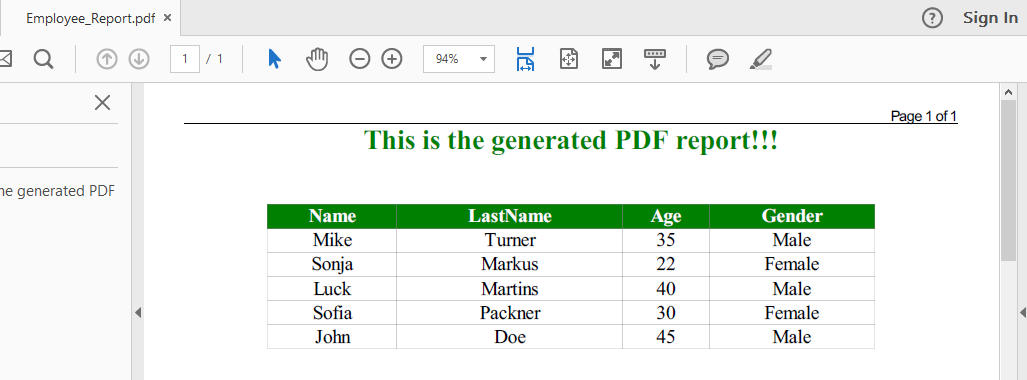
Let’s start our app, open our browser and send a simple request towards our PDF creator endpoint:



As a result, we have our document created in the PDFCreator folder:



And let’s inspect the content of the document:



That is awesome.

We can now continue on.

## Showing a PDF Document in a Browser

If we don’t want to save our document on our local machine but instead to show the document in a browser, we can make that quite easy.

First, we need to remove the Out property from the globalSettings object.

Then instead of this type of conversion:

\_converter.Convert(pdf);

We are going to use this type:

var file = \_converter.Convert(pdf);

Why is that?

Well as we said if we use the Out property than the file is sent to stdout and saved to our local machine. But without the Out property, our output will be stored in a buffer. While converting we need to create a byte array and to store it inside the file variable.

Finally, we are using that file variable and return it to the requester with a content type.

This is our CreatePDF() method after modification:

[HttpGet]

public IActionResult CreatePDF()

{

var globalSettings = new GlobalSettings

{

ColorMode = ColorMode.Color,

Orientation = Orientation.Portrait,

PaperSize = PaperKind.A4,

Margins = new MarginSettings { Top = 10 },

DocumentTitle = "PDF Report"

};

var objectSettings = new ObjectSettings

{

PagesCount = true,

HtmlContent = TemplateGenerator.GetHTMLString(),

WebSettings = { DefaultEncoding = "utf-8", UserStyleSheet = Path.Combine(Directory.GetCurrentDirectory(), "assets", "styles.css") },

HeaderSettings = { FontName = "Arial", FontSize = 9, Right = "Page [page] of [toPage]", Line = true },

FooterSettings = { FontName = "Arial", FontSize = 9, Line = true, Center = "Report Footer" }

};

var pdf = new HtmlToPdfDocument()

{

GlobalSettings = globalSettings,

Objects = { objectSettings }

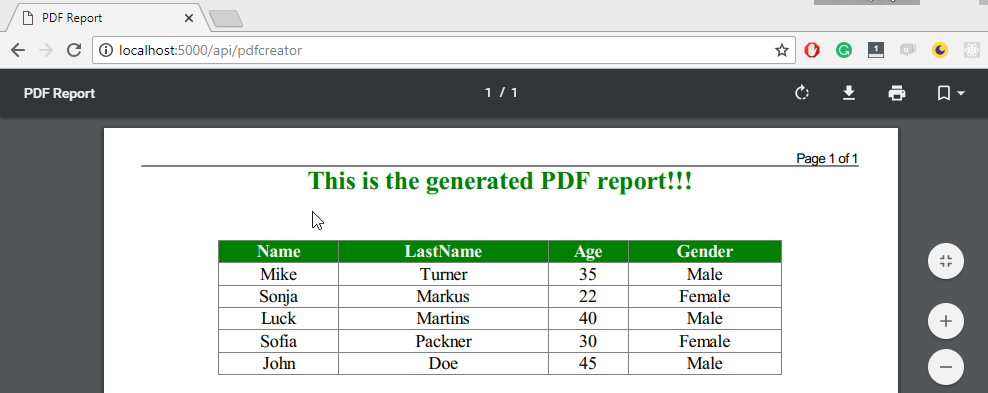
};

var file = \_converter.Convert(pdf);

return File(file, "application/pdf");

}

And this is the result:



## Using Existing HTML Page to Generate PDF Content

We don’t have to use our custom HTML template to generate a PDF content, we can use an existing HTML page. The effort for this is minimal. All we have to do is to remove the HtmlContent property and add the Page property of the ObjectSettings class.

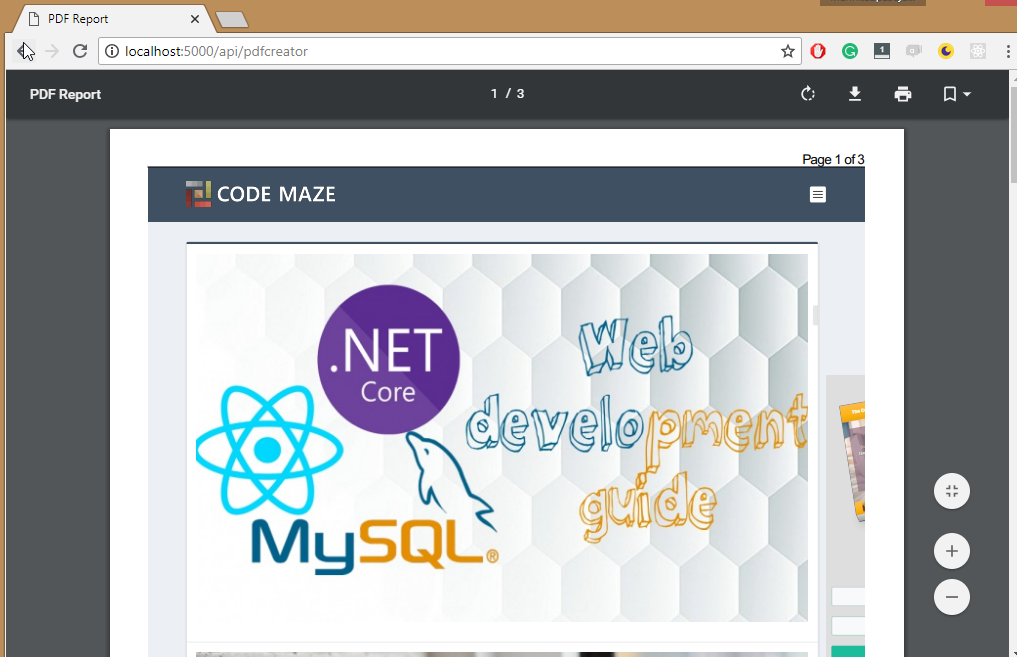
So instead of this code:

HtmlContent = TemplateGenerator.GetHTMLString()

let’s write this code:

Page = <https://code-maze.com/>

And let’s inspect the result:



## Enabling Download Mode

If we want to enable PDF document download all we have to do is to modify our return statement in our action method. All we have to do is to simply add the name of the file with its extension to the return statement:

return File(file, "application/pdf", "EmployeeReport.pdf");

As a result, we are going to have our file downloaded:



And there it is.

Everything is working as it supposed to do.

## Conclusion

In this article, we have shown how to use the DinkToPdf library to create PDF documents while working with the .NET Core Web API project. We have created our PDF’s in different ways to show many different features of DinkToPdf library.

[subscribe]