

# Table of content

On this section, you will find articles related to the use of the ConsoleAppVisuals library and general articles for C# projects.

Regarding the library,

- [Create your own visual element](#)
- [Create and use font](#)
- [Debugging using the library](#)

Regarding C# projects,

- [Create your project documentation](#)
- [Publish a library](#)
- [C# for Visual Studio Code](#)

---

Have a question, give a feedback or found a bug? Feel free to [open an issue](#) or [start a discussion](#) on the GitHub repository.

# Create your own elements

## Introduction

This article will guide you through the process of creating custom visual elements using the library. This will enable you to create passive elements as well as interactive elements that can be used in your applications.

## Prerequisites

- .NET framework 6 or later
- ConsoleAppVisuals library: 3.0.0 or later
- Having looked at the project from the [Introduction section](#)

## Setup workspace

We will take the example project of the [Introduction section](#).

As a reminder, here is the file structure of the project:

```
Example_project <-- root
├── MyApp
│   ├── bin
│   ├── MyApp.csproj
│   └── Program.cs
```

## Passive elements

Passive elements are visual elements that do not have any interactive behavior. They are used to display information to the user. They can be updated and change display properties.

## Setup of a passive element

Start by creating a new file in your project and name it `PassiveExample.cs`. Then, add the following code to the file (see real example in the [example project](#)):

```
using ConsoleAppVisuals;

namespace MyApp
{
    public class PassiveExample : PassiveElement
    {
        #region Fields
```

```

// Add your custom fields here.
#endregion

#region Properties
// Add overridden properties here.
// You may also add your custom properties here.
#endregion

#region Constructor
/// <summary>
/// The natural constructor of the PassiveExample element.
/// </summary>
public PassiveExample(){}
#endregion

#region Methods
// Add your custom methods here.
#endregion

#region Rendering
/// <summary>
/// Renders the PassiveExample element.
/// </summary>
protected override void RenderElementActions()
{
    // This method is mandatory to render correctly your element. If not, an
    error will be thrown.
    // Add what the display code here.
}
#endregion
}
}

```

## Customize your new passive element

Now let's look at the `Element` class. This class is the base class for all visual elements. It contains all the properties and methods that are necessary for the rendering of the elements. You can override some of these properties and methods to customize the behavior of your element (the `PassiveElement` class inherits from all the `Element` class attributes, so you can take the `Element` class as a model to create `PassiveElements`).

The method that you can override are highlighted in yellow here:

```

/*
Copyright (c) 2024 Yann M. Vidamment (MorganKryze)

```

Licensed under GNU GPL v2.0. See full license at:  
<https://github.com/MorganKryze/ConsoleAppVisuals/blob/main/LICENSE.md>  
\*/

```
namespace ConsoleAppVisuals.Models;
```

```
/// <summary>  
/// The <see cref="Element"/> class is an abstract class that represents an element  
/// that can be rendered on the console.
```

```
/// </summary>  
/// <remarks>  
/// For more information, consider visiting the documentation available <a  
href="https://morgankryze.github.io/ConsoleAppVisuals/">here</a>.
```

```
/// </remarks>
```

```
public abstract class Element  
{
```

```
    #region Constants
```

```
    /// <summary>  
    /// The default visibility of the elements when they are added to the window.
```

```
    /// </summary>
```

```
    /// <remarks>
```

```
    /// This value should not be changed.
```

```
    /// Each time the user adds an element to the window, it will try to toggle the  
visibility of the element.
```

```
    /// </remarks>
```

```
    private const bool DEFAULT_VISIBILITY = false;
```

```
    private const int DEFAULT_HEIGHT = 0;
```

```
    private const int DEFAULT_WIDTH = 0;
```

```
    private const int DEFAULT_MAX_NUMBER_OF_THIS_ELEMENT = int.MaxValue;
```

```
    #endregion
```

```
    #region Sealed Properties
```

```
    /// <summary>
```

```
    /// Gets the id number of the element.
```

```
    /// </summary>
```

```
    /// <remarks>This property is sealed. The ID of an element is automatically  
generated and managed by the <see cref="Window"/> class.</remarks>
```

```
    public int Id { get; set; }
```

```
    /// <summary>
```

```
    /// Gets the visibility of the element.
```

```
    /// </summary>
```

```
    /// <remarks>This property is sealed. The visibility of an element is managed by  
the <see cref="ToggleVisibility"/> method.</remarks>
```

```

public bool Visibility { get; private set; } = DEFAULT_VISIBILITY;
#endregion

#region Properties
/// <summary>
/// Gets the type of the element.
/// </summary>
[Visual]
public virtual ElementType Type { get; }

/// <summary>
/// Gets the height of the element, the vertical number of lines taken in
the console.
/// </summary>
/// <remarks>This property is marked as virtual. It is recommended to override
this property in derived classes to make it more specific.</remarks>
public virtual int Height { get; } = DEFAULT_HEIGHT;

/// <summary>
/// Gets the width of the element, the horizontal number of lines taken in
the console.
/// </summary>
/// <remarks>This property is marked as virtual. It is recommended to override
this property in derived classes to make it more specific.</remarks>
public virtual int Width { get; } = DEFAULT_WIDTH;

/// <summary>
/// Gets the placement of the element in the console. See the <see
cref="Placement"/> enum to know the possible values.
/// </summary>
/// <remarks>This property is marked as virtual. It is recommended to override
this property in derived classes to make it more specific.</remarks>
public virtual Placement Placement { get; set; }

/// <summary>
/// Gets the text alignment of the text of the element. See the <see
cref="TextAlignment"/> enum to know the possible values.
/// </summary>
/// <remarks>This property is marked as virtual. It is recommended to override
this property in derived classes to make it more specific.</remarks>
public virtual TextAlignment TextAlignment { get; set; }

/// <summary>
/// Gets the maximum number of this element that can be drawn on the console.
/// </summary>
/// <remarks>This property is marked as virtual. It is recommended to override

```

```

this property in derived classes to make it more specific.</remarks>
    public virtual int MaxNumberOfThisElement { get; } =
DEFAULT_MAX_NUMBER_OF_THIS_ELEMENT;

    /// <summary>
    /// Gets a line to place the element in the console.
    /// </summary>
    /// <exception cref="ArgumentOutOfRangeException">Thrown when the placement of
the element is invalid.</exception>
    /// <remarks>ATTENTION: This property is not marked as virtual. Override this
property only to give it a constant value.</remarks>
    public virtual int Line
    {
        get
        {
            var elements = Window.Range(0, Id);
            return Placement switch
            {
                Placement.TopCenterFullWidth
                    => elements
                        .Where(e => e.Placement == Placement.TopCenterFullWidth
&& e.Visibility)
                        .Sum(e => e.Height)
                        + elements
                        .Where(e => e.Placement == Placement.TopCenter
&& e.Visibility)
                        .Sum(e => e.Height)
                        + elements
                        .Where(e => e.Placement == Placement.TopLeft
&& e.Visibility)
                        .Sum(e => e.Height)
                        + elements
                        .Where(e => e.Placement == Placement.TopRight
&& e.Visibility)
                        .Sum(e => e.Height),
                Placement.TopCenter
                    => elements
                        .Where(e => e.Placement == Placement.TopCenterFullWidth
&& e.Visibility)
                        .Sum(e => e.Height)
                        + elements
                        .Where(e => e.Placement == Placement.TopCenter
&& e.Visibility)
                        .Sum(e => e.Height),
                Placement.TopLeft
                    => elements

```

```

        .Where(e => e.Placement == Placement.TopCenterFullWidth
&& e.Visibility)
        .Sum(e => e.Height)
    + elements
        .Where(e => e.Placement == Placement.TopLeft
&& e.Visibility)
        .Sum(e => e.Height),

    Placement.TopRight
    => elements
        .Where(e => e.Placement == Placement.TopCenterFullWidth
&& e.Visibility)
        .Sum(e => e.Height)
    + elements
        .Where(e => e.Placement == Placement.TopRight
&& e.Visibility)
        .Sum(e => e.Height),
    Placement.BottomCenterFullWidth
    => (Console.WindowHeight == 0 ? 0 : Console.WindowHeight - 1)
        - (Height - 1)
        - elements
        .Where(e =>
            e.Placement == Placement.BottomCenterFullWidth
&& e.Visibility
        )
        .Sum(e => e.Height),
    _ => throw new ArgumentOutOfRangeException(nameof(Placement),
"Invalid placement.")
    };
}
#endregion

#region Methods
/// <summary>
/// Toggles the visibility of the element. If the maximum number of this element
is reached, an exception is thrown.
/// </summary>
/// <exception cref="InvalidOperationException">Thrown when the maximum number
of this element is reached.</exception>
/// <remarks>This method is effectively sealed. The only way to change the
visibility of an element is to use this method.</remarks>
public void ToggleVisibility()
{
    if (Visibility)
    {

```

```

        Visibility = false;
    }
    else if (Window.IsElementActivatable(Id))
    {
        Visibility = true;
    }
    else
    {
        throw new InvalidOperationException(
            $"Operation not allowed, too many elements of {GetType()} already
            toggled from the maximum of {MaxNumberOfThisElement}. Consider turning off one
            element of this type."
        );
    }
}
#endregion

#region Rendering
/// <summary>
/// Renders the element on the console.
/// </summary>
/// <remarks>
/// For more information, consider visiting the documentation available <a
href="https://morgankryze.github.io/ConsoleAppVisuals/">here</a>.
/// </remarks>
[Visual]
public void RenderElement()
{
    if (Visibility)
    {
        RenderOptionsBeforeHand();
        RenderElementActions();
        RenderOptionsAfterHand();
    }
}

/// <summary>
/// Defines the actions to perform when the element is called to be rendered on
the console.
/// </summary>
/// <remarks>This method is marked as virtual. It is recommended to override
this method in derived classes to make it more specific.</remarks>
[Visual]
protected virtual void RenderElementActions()
{
    throw new NotImplementedException("Consider overriding this method in the

```



```

derived class.");
    }

    /// <summary>
    /// Defines actions to perform before rendering the element on the console.
    /// </summary>
    [Visual]
    protected virtual void RenderOptionsBeforeHand() { }

    /// <summary>
    /// Defines actions to perform after rendering the element on the console.
    /// </summary>
    [Visual]
    protected virtual void RenderOptionsAfterHand() { }

    /// <summary>
    /// Renders the space taken by the element on the console.
    /// </summary>
    /// <param name="ignoreVisibility">Whether to ignore the visibility of the
element or not.</param>
    /// <remarks>
    /// For more information, consider visiting the documentation available <a
href="https://morgankryze.github.io/ConsoleAppVisuals/">here</a>.
    /// </remarks>
    [Visual]
    public void RenderElementSpace(bool ignoreVisibility = false)
    {
        if (Visibility || ignoreVisibility)
        {
            Core.SaveColorPanel();
            Core.SetForegroundColor(Core.GetRandomColor());
            Core.WriteMultiplePositionedLines(
                false,
                TextAlignment.Center,
                Placement,
                true,
                Line,
                GetRenderSpace()
            );
            Core.LoadSavedColorPanel();
        }
    }

    /// <summary>
    /// Gets the space taken by the element on the console.
    /// </summary>

```

```

    /// <returns>The space taken by the element.</returns>
    /// <remarks>This method is marked as virtual. It is recommended to override
this method in derived classes to make it more specific.</remarks>
    [Visual]
    protected virtual string[] GetRenderSpace()
    {
        var space = new string[Height];
        for (int i = 0; i < space.Length; i++)
        {
            space[i] = new string(' ', Width);
        }
        return space;
    }

    /// <summary>
    /// Clears the space taken by the element on the console.
    /// </summary>
    /// <remarks>
    /// For more information, consider visiting the documentation available <a
href="https://morgankryze.github.io/ConsoleAppVisuals/">here</a>.
    /// </remarks>
    [Visual]
    public void Clear()
    {
        Core.ClearMultiplePositionedLines(Placement, Line, GetRenderSpace());
    }
    #endregion
}

```

### TIP

Depending on the element you want to create, you may not need to override all of these methods. You can override only the ones that are necessary for your element. However I highly recommend to override these:

- **MaxNumberOfThisElement**: Define the maximum number of this element that can be displayed on the screen simultaneously.
- **RenderElementActions()**: Describe how the element should be displayed.
- **Height** and **Width**: Depending on the element, you may want to override these properties to define the size of your element.

Once your customization is done, you may use your element in your application just like a default element.

# Interactive elements

Interactive elements are visual elements that have interactive behavior. They can be used to create buttons, prompts, menus, and other interactive elements. They can be updated and change display properties. But they also always give a response that the user can catch. The type of the response depends on the element.

## Setup of an interactive element

Similar to the passive elements, you can create interactive elements but this time they inherit from the `InteractiveElement` class. This class contains all the properties and methods that are necessary for the rendering of the elements. You can override some of these properties and methods to customize the behavior of your element.

Start by creating a new file in your project and name it `InteractiveExample.cs`. Then, create your new element following this template (see real example in the [example project](#)):

```
using ConsoleAppVisuals;

namespace MyApp
{
    public class InteractiveExample : InteractiveElement<T>
    {
        #region Fields
        // Add your custom fields here.
        #endregion

        #region Properties
        // Add overridden properties here.
        // You may also add your custom properties here.
        #endregion

        #region Constructor
        /// <summary>
        /// The natural constructor of the InteractiveExample element.
        /// </summary>
        public InteractiveExample(){}
        #endregion

        #region Methods
        // Add your custom methods here.
        #endregion

        #region Rendering
        /// <summary>
```

```

    /// Renders the InteractiveExample element.
    /// </summary>
    protected override void RenderElementActions()
    {
        // This method is mandatory to render correctly your element. If not, an
        error will be thrown.
        // Add what the display code here.
    }
    #endregion
}
}

```

## Customize your new interactive element

Now let's look at the `InteractiveElement` class. This class inherits from the `Element` class and contains all the properties and methods that are necessary for the rendering of the elements. You can override some of these properties and methods to customize the behavior of your element.

### ⊗ IMPORTANT

To define a new interactive element, you must define the type of the response that the element will give. This type can be pretty much everything, but a classic type like `int`, `string`, ... is to prefer. In the example above, the type `T` is used. You can replace it with the type you want to use.

The method and properties that you can override are the same as the `PassiveElement` class at some exceptions:

- `MaxNumberOfThisElement`: is set to one.
- `RenderOptionsBeforeHand` & `RenderOptionsBeforeHand`: cannot be modified.

Two new methods are available and cannot be modified:

- `SendResponse()`: This method is called when the user interacts with the element. It is used to send a response to the window (highly recommended to see the [example project](#) to understand its implementation).
- `GetResponse()`: This method is called when the user has interacted with the element. It is used to get the response from the user (you also have `GetResponseHistory()` to get the history of the responses).

To understand how is defined the interaction response, I highlighted the two attributes that are used to define the response:

```
/*
    Copyright (c) 2024 Yann M. Vidamment (MorganKryze)
    Licensed under GNU GPL v2.0. See full license at:
    https://github.com/MorganKryze/ConsoleAppVisuals/blob/main/LICENSE.md
*/
namespace ConsoleAppVisuals.Models;

/// <summary>
/// The <c>InteractionEventArgs</c> class is a generic class that represents the
/// event arguments for the interactive elements.
/// </summary>
/// <remarks>
/// For more information, consider visiting the documentation available <a
/// href="https://morgankryze.github.io/ConsoleAppVisuals/">here</a>.
/// </remarks>
public class InteractionEventArgs<T> : EventArgs
{
    #region Fields
    /// <summary>
    /// Gets the status after exiting the interactive element. See the <see
    cref="Status"/> enumeration to know the possible values.
    /// </summary>
    /// <value>Status.Escaped : pressed escape, Status.Deleted : pressed backspace,
    Status.Selected : pressed enter</value>
    public Status Status { get; set; }

    /// <summary>
    /// Gets the <typeparamref name="T"/> value of the response after exiting the
    interactive element.
    /// </summary>
    public T Value { get; set; }
    #endregion

    #region Constructor
    /// <summary>
    /// The <c>InteractionEventArgs</c> class is a generic class that represents the
    event arguments for the interactive elements.
    /// </summary>
    /// <param name="status">The status of the exit from the menu.</param>
    /// <param name="value">The value of the response after exiting the interactive
    element.</param>
    /// <remarks>
```

```

    /// For more information, consider visiting the documentation available <a
href="https://morgankryze.github.io/ConsoleAppVisuals/">here</a>.
    /// </remarks>
    public InteractionEventArgs(Status status, T value)
    {
        Status = status;
        Value = value;
    }
    #endregion
}

```

Where **Status** depends on the values of the [Status enum](#) and **Value** depends on the **T** type of the **InteractiveElement** you created.

Once your customization is done, you may use your element in your application just like a default element.

## Visualize all elements available

Now that you know how to create your own elements, you can check if they are available in the library. To do so, you can use built-in elements to display all the elements available in the library.

```

Window.Open();

ElementsList passiveList = new ElementsList(ElementType.Passive);
Window.AddElement(passiveList);

Window.Render(passiveList);
Window.Freeze();

Window.DeactivateElement(passiveList);
Window.RemoveElement(passiveList);

Window.Close();

```

| Passive element types available |                   |                   |
|---------------------------------|-------------------|-------------------|
| Id                              | Type              | Project           |
| 0                               | PassiveDemo       | example           |
| 1                               | Banner            | ConsoleAppVisuals |
| 2                               | Footer            | ConsoleAppVisuals |
| 3                               | Header            | ConsoleAppVisuals |
| 4                               | HeightSpacer      | ConsoleAppVisuals |
| 5                               | ElementsDashboard | ConsoleAppVisuals |
| 6                               | ElementsList      | ConsoleAppVisuals |
| 7                               | LoadingBar        | ConsoleAppVisuals |
| 8                               | Matrix`1          | ConsoleAppVisuals |
| 9                               | TableView         | ConsoleAppVisuals |
| 10                              | Title             | ConsoleAppVisuals |

Or target only interactive elements:

```
Window.Open();
```

```
ElementsList interactiveList = new ElementsList(ElementType.Interactive);  
Window.AddElement(interactiveList);
```

```
Window.Render(interactiveList);  
Window.Freeze();
```

```
Window.DeactivateElement(interactiveList);  
Window.RemoveElement(interactiveList);
```

```
Window.Close();
```

| Interactive element types available |                 |                   |
|-------------------------------------|-----------------|-------------------|
| Id                                  | Type            | Project           |
| 0                                   | InteractiveDemo | example           |
| 1                                   | EmbedText       | ConsoleAppVisuals |
| 2                                   | FakeLoadingBar  | ConsoleAppVisuals |
| 3                                   | FloatSelector   | ConsoleAppVisuals |
| 4                                   | IntSelector     | ConsoleAppVisuals |
| 5                                   | Prompt          | ConsoleAppVisuals |
| 6                                   | ScrollingMenu   | ConsoleAppVisuals |
| 7                                   | TableSelector   | ConsoleAppVisuals |

### NOTE

You may repeat the same process for the `ElementType.Default` and `ElementType.Animated` to see all the elements available in the library. Note also that creating an `AnimatedElement` is just like creating a `InteractiveElement` but without sending a response. You may add a way for the user to press a key to skip the animation or to stop it. [see the loading bars](#)

Have a question, give a feedback or found a bug? Feel free to [open an issue](#) or [start a discussion](#) on the GitHub repository.



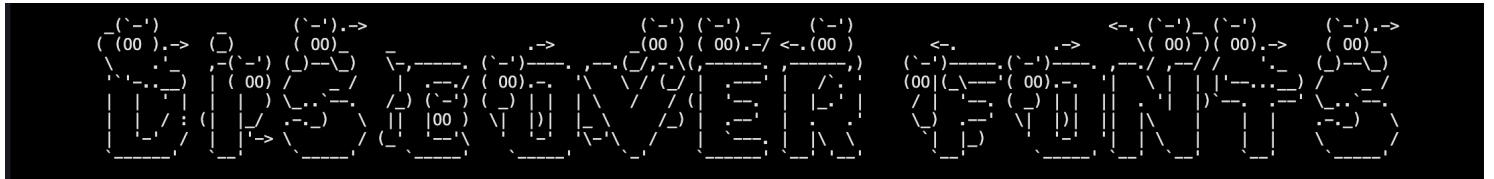
# Create and use fonts

## Introduction

This article will guide you through the process of creating your own font using the `Font enum`.



*ANSI\_Shadow*



*Lil\_Devil*



*Merlin*

## Prerequisites

- .NET framework 6 or later
- ConsoleAppVisuals library: 3.0.0 or later
- Having looked at the project from the [Introduction section](#) ✓

## Setup workspace

We will take the example project of the [Introduction section](#) ✓.

As a reminder, here is the file structure of the project:

```
Example_project <-- root
├── MyApp
│   ├── bin
│   ├── MyApp.csproj
│   └── Program.cs
```

## What are fonts

In ConsoleAppVisuals, a font is a collection of ASCII characters on multiple lines associated to keys (e.g. `abc123?!/`). Some fonts are already available in the `Font` enum like the `ANSI_Shadow`, `Bulbhead`, `Lil_Devil` accessible using: `Font.ANSI_Shadow`, `Font.Bulbhead`, `Font.Lil_Devil`.

Fonts are not available to all elements, for example, find it in the `Title` element:

```
Title title = new Title("Example project", 1, TextAlignment.Center,
    Font.ANSI_Shadow);
```

To use a custom font, update the font variable and add the font path:

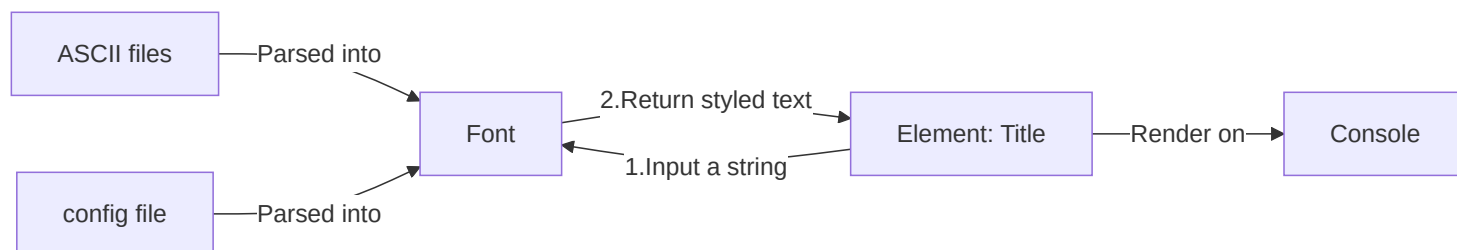
```
Title title = new Title("Example project", 1, TextAlignment.Center,
    Font.Custom, "path/to/font");
```

### ⊗ CAUTION

In elements that use fonts:

- If you use built-in fonts, you MUST NOT specify a font path.
- If you use a custom font, you MUST specify a font path.

Here is a recap of fonts work:



## Creating a font Structure

Here is briefly the structure of a font for the `ANSI_Shadow` font:

```
ANSI_Shadow
├── data
│   ├── alphabet.txt
│   └── numbers.txt
```

```
| └─ symbols.txt
└─ config.yml
```


## Config file

The config.yml is the config file of the font. It contains the name, the author, the height of the characters and all the supported ones. Find an example below for the ANSI\_Shadow font:

```
name: ANSI Shadow
author: Unknown
height: 6

chars:
  alphabet: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
  numbers: 0123456789
  symbols: '?!:.,;/_-()[ ]%$^*@ '
```

### NOTE

Pay attention to the alphabet as the fonts are case-sensitive. In the `alphabet` category, mind to include both the upper and lower case letters. If your font does not support both, just copy and paste the letters twice. `ANSI_Shadow` font is a good example. [Find it here](#) 

Here, `name`, `author`, `height` and `chars` are required. They must not be null or empty. If the author is unknown, you can put `Unknown` by convention.

In contrast, the `alphabet`, `numbers` and `symbols` are optional. If you don't have a specific category, you can let it empty and not include the ASCII file associated.

An example of empty `numbers`:

```
name: Bloody
author: Unknown
height: 10

chars:
  alphabet: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
  numbers:
  symbols: ' '
```

**NOTE**

It is highly recommended to have a ' ' (space) char in the `symbols` category for a readable font.

By convention, the À, é, ü... should be included in the **symbols** category.

## ASCII files

The ASCII files are the representation of the characters in the font. They are stored in the **data** folder. The name of the file must be the category name (e.g. **alphabet.txt**, **numbers.txt**, **symbols.txt**).

Here is an example of the `alphabet.txt` file for the `ANSI_Shadow` font:

|  |    |
|--|----|
|  | @  |
|  | @  |
|  | @  |
|  | @  |
|  | @  |
|  | @@ |
|  | @  |
|  | @  |
|  | @  |
|  | @  |
|  | @@ |
|  | @  |
|  | @  |
|  | @  |
|  | @  |
|  | @@ |

As you noticed:

1. All characters are 6 lines high as specified in the config file.
2. Each line ends with a @.
3. The final line of a character ends with a @@.
4. The width of the characters is not fixed. It can be different from one character to another.
5. The @characters are aligned for a given character.

6. One space is added between the characters and the @ to make the result styled text more readable.
7. DO NOT USE @ CHAR IN YOUR FONT ITSELF.

Points 1, 2, 3 are mandatory and will throw an exception if not respected. Points 4, 5, 6, 7 are optional but recommended for a better result.

### **i NOTE**

The characters are ordered following the order given by the `chars` categories in the config file.

## Using your font

If you followed the previous steps rigorously, you should have a font ready to use. Consider the following project:

```
Example_project <-- root
├── MyApp
│   ├── ANSI_Shadow
│   │   ├── data
│   │   │   ├── alphabet.txt
│   │   │   ├── numbers.txt
│   │   │   └── symbols.txt
│   │   └── config.yml
│   ├── bin
│   ├── MyApp.csproj
│   └── Program.cs
```

### **! WARNING**

Do not forget to add at the beginning of your `Program.cs` file the following using statements:

```
using ConsoleAppVisuals;
using ConsoleAppVisuals.PassiveElements;
using ConsoleAppVisuals.InteractiveElements;
using ConsoleAppVisuals.Enums;
using ConsoleAppVisuals.Models;
```

To check that `ANSI_Shadow` is working, update the `Program.cs` file:

```
Title title = new Title("Example project", 1, TextAlignment.Center,
    Font.Custom, "../ANSI_Shadow/");
```

The path here leads to the font directory. The library will automatically target or the `config.yml` file and the `data` folder.

If no error was thrown, that means that the font is working. You can now use it. Here is how to manipulate the styler:

Creation:

```
TextStyler styler = new TextStyler(Font.Custom, "ANSI_Shadow/");
```

Usage (String -> Styled text):

```
string[] styledText = styler.Style("Hello, world!");
```

Display:

```
Core.WritePositionedStyledText(styledText);
```

For more information about the `TextStyler` class and `WritePositionedStyledText()` method, please refer to the [References](#) section.

## Contributing

If you want to contribute to the library by adding a font, you can do so by creating a pull request on the [GitHub repository](#).

Here are the steps to follow:



1. Fork the repository and create a new branch for your new font.
2. Add your font to the `src/ConsoleAppVisuals/fonts` directory.
3. Make sure to match all the requirements for the font defined above in the article.
4. Add your font name to the `Font` enum (`src/ConsoleAppVisuals/enums/Font.cs`) and precise the author and the height of the characters in the metadata comments.
5. Submit a pull request to the dev branch of the repository.

After these steps, your font will be reviewed and merged into the library to be available for everyone.

## Resources

- [Test fonts Figlet](#)
- [Fonts collection](#)

---

Have a question, give a feedback or found a bug? Feel free to [open an issue](#) or [start a discussion](#) on the GitHub repository.

# Debugging in ConsoleAppVisuals

## Introduction

Debugging is a crucial part of software development. It is the process of identifying and removing errors from a computer program. We put a lot of effort into making debugging easier and more efficient in ConsoleAppVisuals. This article will guide you through the debugging tools and resources available in ConsoleAppVisuals to help you overcome your challenges.

## Pre-requisites

- Basic knowledge of C# programming language
- A code editor (Visual Studio, Visual Studio Code, etc.)
- ConsoleAppVisuals library installed (v3.3.0 or later)

## In-code tools

### ElementsDashboard

The first tool that we will discuss is the `ElementsDashboard`. It is a debugging tool that allows you to visualize the elements added to the Window. they are presented like a list of elements with their properties. This tool is very useful when you want to see the updates of the elements properties in real-time after activating, deactivating or modifying them.

```
Window.Open();
```

```
ElementsDashboard dashboard = new ElementsDashboard(Placement.TopCenter, false);  
Window.AddElement(dashboard);
```

```
Window.Render(dashboard);  
Window.Freeze();
```

```
Window.Close();
```



| Window Elements Dashboard |                   |            |        |       |      |           |               |
|---------------------------|-------------------|------------|--------|-------|------|-----------|---------------|
| Id                        | Type              | Visibility | Height | Width | Line | Placement | IsInteractive |
| 0                         | ElementsDashboard | True       | 6      | 78    | 0    | TopCenter | False         |

### TIP

The arguments available for the `ElementsDashboard` are:

- `Placement` - The position of the dashboard on the window to place it at a convenient location. (see [Placement enum](#) for more details)
- `RoundedCorners` - A boolean value that indicates if the dashboard should have rounded corners. (purely aesthetic)

## ElementsList

The `ElementsList` is another debugging tool that allows you to visualize the elements inherited from the `Element`, `PassiveElement` or `InteractiveElement` classes. It lets you see their type and project location (library or your own). This tool is very useful when you want to add your custom element and see if they are recognized by the library.

```
Window.Open();
```

```
ElementsList list = new ElementsList(ElementType.Default, Placement.TopCenter,
false);
Window.AddElement(list);
```

```
Window.Render(list);
Window.Freeze();
```

```
Window.Close();
```

| Element types available |                   |                   |
|-------------------------|-------------------|-------------------|
| Id                      | Type              | Project           |
| 0                       | InteractiveDemo   | example           |
| 1                       | PassiveDemo       | example           |
| 2                       | Banner            | ConsoleAppVisuals |
| 3                       | Footer            | ConsoleAppVisuals |
| 4                       | Header            | ConsoleAppVisuals |
| 5                       | HeightSpacer      | ConsoleAppVisuals |
| 6                       | ElementsDashboard | ConsoleAppVisuals |
| 7                       | ElementsList      | ConsoleAppVisuals |
| 8                       | LoadingBar        | ConsoleAppVisuals |
| 9                       | Matrix`1          | ConsoleAppVisuals |
| 10                      | TableView         | ConsoleAppVisuals |
| 11                      | Title             | ConsoleAppVisuals |
| 12                      | EmbedText         | ConsoleAppVisuals |
| 13                      | FakeLoadingBar    | ConsoleAppVisuals |
| 14                      | FloatSelector     | ConsoleAppVisuals |
| 15                      | IntSelector       | ConsoleAppVisuals |
| 16                      | Prompt            | ConsoleAppVisuals |
| 17                      | ScrollingMenu     | ConsoleAppVisuals |
| 18                      | TableSelector     | ConsoleAppVisuals |

### TIP

The arguments available for the `ElementsList` are:

- `ElementType` - The type of elements to display in the list. (see [ElementType enum](#) for more details)
- `Placement` - The position of the list on the window to place it at a convenient location. (see [Placement enum](#) for more details)
- `RoundedCorners` - A boolean value that indicates if the list should have rounded corners. (purely aesthetic)

## Core.WriteDebugMessage()

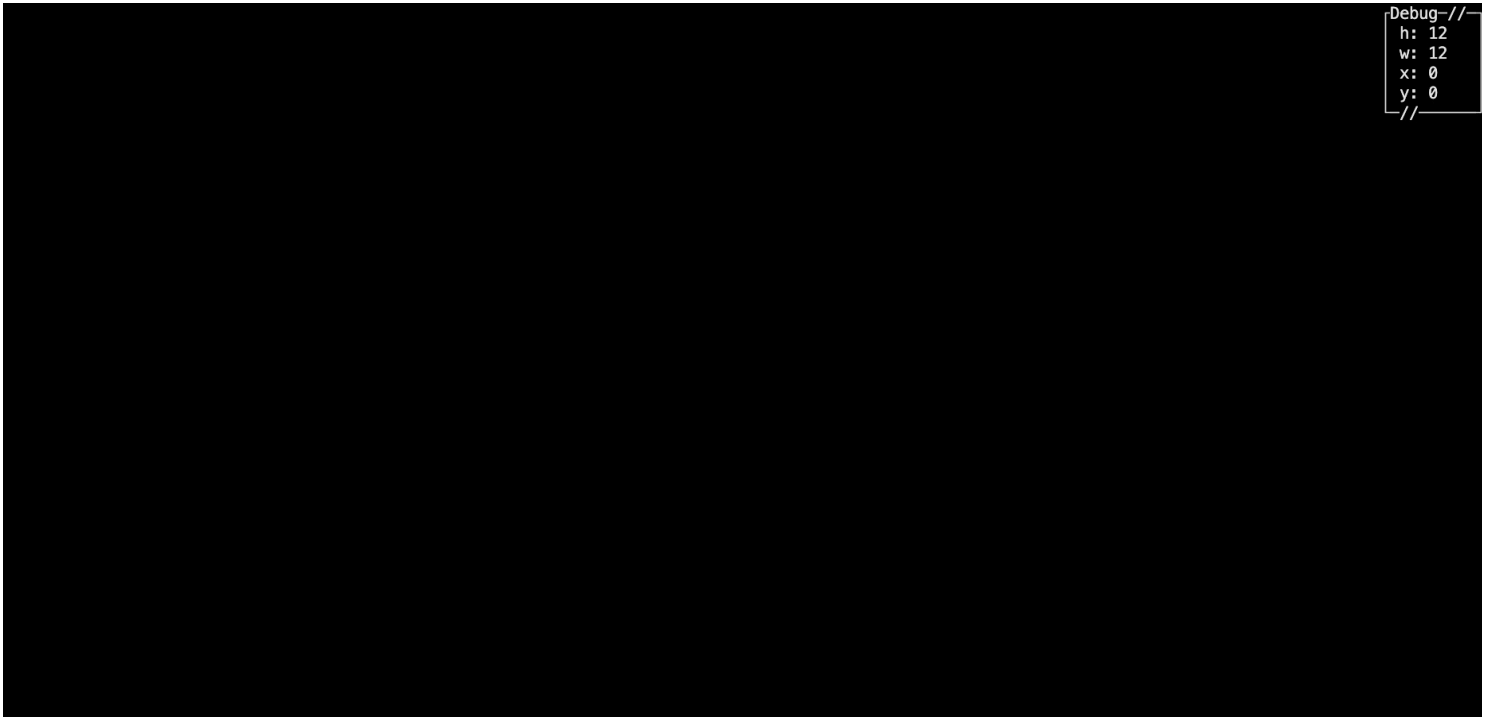
The `Core.WriteDebugMessage()` method is a simple debugging tool that allows you to print a debug message in the console. This message overrides whatever is currently in the console and is useful when you want to display information without creating an `EmbedText` for example (it is also a quick alternative to the `Console.WriteLine()` method).

```
Window.Open();
```

```
// Here the info are dummy but you may pass real elements or window variables
Core.WriteDebugMessage(Placement.TopRight, "h: 12", "w: 12", "x: 0", "y: 0");
```

```
Window.Freeze();
```

```
Window.Close();
```



### **TIP**

The arguments available for the `Core.WriteDebugMessage()` are:

- **Placement** - The position of the message on the window to place it at a convenient location. (values can only be `TopLeft`, `TopCenter` or `TopRight`)
- **params string[]** - The information to display in the message. (you can pass as many strings as you want) An empty string will be displayed as a blank space.

## Issues

Before getting crazy over your code, try to find a related issue in the library that tackles the subject you are struggling with. If you can't find any, feel free to open an issue on the [GitHub repository](#) and we will be happy to help you.

### **NOTE**

As the v3 of the project is still in development, no issues have been reported yet. Be the first!

# References

If you struggle with the behavior of a specific class or method, you can refer to the [detailed documentation](#) to find the information you need. The documentation is updated regularly and contains all the information you need to use the library.

Or simply redo the [tutorials](#) to get a better understanding of the library.

# Examples

If you are looking for concrete examples to help you understand how to use the library in real use cases, you can refer to the [examples](#) section. The examples are updated regularly and cover a wide range of use cases.

# Source code

Finally, if none of the above solutions work, or if you are just curious, you can always refer to the [source code](#).

---

Have a question, give a feedback or found a bug? Feel free to [open an issue](#) or [start a discussion](#) on the GitHub repository.

# Create your project documentation

## Introduction

This article will guide you through the process of creating a documentation for your C# project using the tool [DocFX](#). Documentation is key to help user understand how to use the tools you create. It is also a good way to show the quality of your work.

## Prerequisites

- Having looked at the project from the [Introduction section](#)
- [.NET 6.0](#) or later

## Install DocFX

Ensure that you have dotnet (C#) installed by running:

```
dotnet --version
```

To install docfx, or update it, open any terminal and run the following command:

```
dotnet tool update -g docfx
```

## Setup workspace

As we are taking back the [Introduction project](#) to set the example, here is the file structure before generating the documentation:

```
Example_project <-- root
├── MyApp
│   ├── bin
│   ├── MyApp.csproj
│   └── Program.cs
```

Open a terminal to the root of the project(the end of your path should be [Example\\_project/](#)), and run the following command:

```
docfx init -y -o documentation
```

Now you should have a new folder called "documentation" in the root of your project. Your folder structure should look like this (files are specified with the dots):

```

Example_project <-- root
├── documentation
│   ├── docs
│   │   ├── getting-started.md
│   │   ├── introduction.md
│   │   └── toc.yml
│   ├── docfx.json
│   ├── index.md
│   └── toc.yml
└── MyApp
    ├── bin
    ├── MyApp.csproj
    └── Program.cs

```

Here should be the default content of `docfx.json`:

```

{
  "metadata": [
    {
      "src": [
        {
          "src": "../src",
          "files": ["**/*.csproj"]
        }
      ],
      "dest": "api"
    }
  ],
  "build": {
    "content": [
      {
        "files": ["**/*.md", "**/*.yml"],
        "exclude": ["_site/**"]
      }
    ],
    "resource": [
      {
        "files": ["img/**"]
      }
    ],
    "output": "_site",
    "template": ["default", "modern"],
    "globalMetadata": {
      "_appName": "",
      "_appTitle": "",

```

```

        "_enableSearch": true,
        "pdf": true
    }
}
}

```

For a more convenient display, features and to target to the project, I recommend you to update the file to the version below. For more information, see the [official documentation](#) of the references tags.

```

{
  "metadata": [
    {
      "src": [
        {
          "src": "../MyApp",
          "files": ["**/*.csproj"]
        }
      ],
      "dest": "api"
    }
  ],
  "build": {
    "content": [
      {
        "files": ["**/*.md,yml"],
        "exclude": ["_site/**"]
      }
    ],
    "output": "_site",
    "resource": ["assets/**"],
    "template": ["default", "modern"],
    "keepFileLink": false,
    "disableGitFeatures": false,
    "globalMetadata": {
      "_appName": "MyApp",
      "_appTitle": "MyApp",
      "_appFooter": "Copyright (C) 2024 Your Name",
      "_enableSearch": true,
      "_disableContribution": true,
      "pdf": true
    }
  }
}

```

## NOTE

You may want to select the channel of the documentation you want to generate. For example, if you want to generate the documentation for the Debug or Release version only. Feel free to update `files` to Debug or Release and `TargetFramework` to your dotnet version(available in the `MyApp.csproj`).

```
...
"metadata": [
  {
    "src": [
      {
        "src": "../MyApp",
        "files": ["**/bin/Debug/**/*.dll"]
      }
    ],
    "dest": "api",
    "properties": {
      "TargetFramework": "net8.0"
    }
  }
],
...
```

Do not forget to update your compiled files using the `dotnet build` command:

Debug

Release

```
dotnet build -c Debug
```

## Preview your doc

Now, back on your terminal from the root, run the following command:

```
docfx build documentation/docfx.json --serve
```

The output should end like this:



```
...
Serving ".../MyApp/documentation/_site" on http://localhost:8080. Press Ctrl+C to
shut down.
```

Your documentation is now available on <http://localhost:8080> if you want to see the preview on localhost.

## Customize your doc

### Add sections

By default, the only sections available are `Docs` and `Api Documentation`. You may want to add more sections to your documentation. To do so, you will have to do fe steps:

1. Add a new folder in the `documentation` folder. For example, `articles`.
2. Inside `articles`, add a `index.md` file and a `toc.yml` file.

Here is an example of the `index.md` file:

```
# Articles
```

```
This is the articles section. You can add articles to explain how to use
your library.
```

Here is an example of the `toc.yml` file:

```
items:
  - name: Articles
    href: index.md
```

#### NOTE

We added the `items` tag to the `toc.yml` file. This is the root of the table of contents and will remove the error `Incorrect Type. Expected "TOC"`.

3. Now, we need to update the `toc.yml` file in the `documentation` folder to add the new section. I recommend adding a homepage mention (will be the landing page when the section is clicked). Here is an example of the `toc.yml` file:

```
items:
  - name: Docs
    href: docs/
  - name: API
    href: api/
  - name: Articles
    href: articles/
    homepage: articles/index.md
```

## Add pages

Now that you know how to create new sections, to add pages you may just add markdown files to the sections folder, and add them to the `toc.yml` file. Here is an example of the `toc.yml` file:

```
items:
  - name: Getting Started
    href: index.md
  - name: How to use the library
    href: how_to_use.md
  - name: How to publish your work
    href: how_to_publish.md
```

However you may also be able to create collapsible menu in the `toc.yml` file. Here is an example of the `toc.yml` file:

```
items:
  - name: Getting Started
    href: index.md
  - name: Advanced
    items:
      - name: How to use the library
        href: how_to_use.md
      - name: How to publish your work
        href: how_to_publish.md
```

Or use another style and display the category name, and the pages without being collapsible:

```
items:
  - name: Getting Started
    href: index.md
```

- name: Other pages
  - href: how\_to\_use.md
  - href: how\_to\_publish.md

## Markdown features support

DocFX supports a lot of markdown features. All of them are listed in the [official documentation](#)🔗.

## Logo & favicon

To add a logo or favicon to your documentation, start by adding them into the assets folder (if you have not, create it in the `documentation` folder). Then, update the `docfx.json` file to add the `logo` and `favicon` tags. Here is an example:

```
...
"build": {
  ...
  "resource": ["assets/**"],
  "globalMetadata": {
    ...
    "_appLogoPath": "assets/logo.jpg",
    "_appFaviconPath": "assets/favicon.ico",
    ...
  }
  ...
}
```

For both I recommend you using svg files so that the logo and favicon are scalable and will not lose quality.

## Code documentation

Coding in C#, you may be aware of the use of the `///` comments to document your code. This is a good practice to help other developers understand your code. DocFX will take these comments into account to generate accurate documentation. Please refer to the [official documentation](#)🔗 for more information.

For docfx to support these metadata, ensure that a documentation file is generated correctly. Add this line to your `**.csproj` file, inside the `"PropertyGroup"` tag:

```
<GenerateDocumentationFile>true</GenerateDocumentationFile>
```

Here is a little troubleshooting if you have an error while building the documentation:

- Check the version of your dotnet.
- Update docfx.
- Check the `docfx.json` path to your project (e.g. `../MyApp`).
- Check if you have well put a `namespace` in your file.
- Your `program.cs` will not be used in the documentation, so you will need to have at least one more class. Here is a quick example to copy/paste in a new file:

```
namespace MyApp;

/// <summary>
/// Class <c>Point</c> models a point in a two-dimensional plane.
/// </summary>
public class Point
{
    private int x;
    private int y;

    /// <summary>
    /// Initializes a new instance of the <c>Point</c> class.
    /// </summary>
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    /// <summary>
    /// Gets the x-coordinate of the point.
    /// </summary>
    public int X
    {
        get { return x; }
    }

    /// <summary>
    /// Gets the y-coordinate of the point.
    /// </summary>
    public int Y
    {
        get { return y; }
    }

    /// <summary>
```

```

    /// Returns a string that represents the current object.
    /// </summary>
    public override string ToString()
    {
        return $"({x}, {y})";
    }
}

```

Now your documentation is ready to be generated in the section **API** in the generated site (you may change all sections names in your `toc.yml` file at the root of your documentation folder).

## Deploy the doc GitHub Pages

GitHub provides a service called GitHub Pages that allows you to host static websites directly from your repository. We will need to setup few things before deploying the documentation.

First of all, go to your repository settings, then to the "Pages" section. Select "Deploy from branch", then select the branch "gh-pages" branch and the root folder. Then click on "Save". If you do not have a "gh-pages" branch, you will have to create one (it is better if it is empty at the beginning but it is not mandatory).

## Deployment

Then, you will have to create a new folder called `.github` at the root of your project. Inside this folder, create a new folder called `workflows`. Inside this folder, create a new file called `deploy_docs.yml`. This file will contain the workflow to generate and deploy the documentation on GitHub Pages.

Here is an example of the `deploy_docs.yml` file:

```

name: Deploy docs
on:
  push:
    branches:
      - main
jobs:
  publish-docs:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v3

```

```
- name: Dotnet Setup
  uses: actions/setup-dotnet@v3
  with:
    dotnet-version: 8.x

- run: dotnet tool update -g docfx
- run: docfx documentation/docfx.json

- name: Deploy
  uses: peaceiris/actions-gh-pages@v3
  with:
    github_token: ${ secrets.GITHUB_TOKEN }
    publish_dir: docs/_site
```





Push the changes and go to the "Actions" section of your repository. You should see a new workflow called "Deploy docs". Click on it to see the logs. If everything went well, you should see a "Deployed" message at the end of the logs.

Now, on every push on the main branch, the documentation will be generated and deployed on GitHub Pages.



#### **NOTE**

In your github repository description, click on "Edit" then for the url select the "GitHub Pages" url option. So that your documentation is directly accessible from your repository.

## Sources

- [DocFX documentation](#)
- [Useful but not official documentation](#)
- [C# documentation comments](#)
- [DocFX markdown support](#)

---

Have a question, give a feedback or found a bug? Feel free to [open an issue](#) or [start a discussion](#) on the GitHub repository.

# Publish your library

## Introduction

This article will guide you through the process of publishing a package on NuGet.org and GitHub packages using GitHub actions. This will enable you to share your library with the world.

NuGet is a package manager for .NET that allows you to share your code with the world. It is a great way to share your library with the community and to make it easy for others to use your code.

## Prerequisites

- Having looked at the project from the [Introduction section](#)
- [.NET 6.0](#) or later
- [NuGet account](#), preferably for you on an outlook email address
- [Github account](#)
- Put your project on GitHub to be able to use GitHub actions

## Setup workspace

We will take the example project of the [Introduction section](#) and we will publish it on NuGet.org and GitHub packages as an example.

As a reminder, here is the file structure of the project:

```
Example_project <-- root
├── MyApp
│   ├── bin
│   ├── MyApp.csproj
│   └── Program.cs
```

## README.md & LICENSE

This part is not mandatory but highly recommended.

Readme files are a great way to introduce your project to the world. It is the first thing people will see when they visit your repository. It is a good practice to include a README file in your project. [Learn more](#)

The license file is also important. It is a way to tell people what they can and cannot do with your project. The default license is the MIT license that let the user a lot a freedom with your code. [Learn more](#)

Here is an example of a README file:

```
# MyApp

> A simple console app for demonstration purposes

## Installation

Describe how to install your project

## Usage

Describe how to use your project

## Contributing

Describe how to contribute to your project

## License

MIT
```

And here is an example of a LICENSE file:


```
MIT License

Copyright (c) 2024 YourName

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
```

## MyApp.csproj

C# project files (`.csproj`) are the files that contain all the information about your project. It is where you define the target framework, the dependencies, the version of your project, and much more. [Learn more](#) 



Here is a template for a `.csproj` file made for publishing a package:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <!-- Project Info-->
    <TargetFrameworks>net8.0</TargetFrameworks>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <PropertyGroup>
    <Title>MyApp</Title>
    <!-- Change this by the name of your package, it must be unique -->
    <PackageId>MyFirstApp1234</PackageId>
    <!-- Change this by the name of the publisher on nuget.org -->
    <Authors>YourNugetAccountName</Authors>
    <!-- Change this by the description of your package -->
    <Description>Descriptive description to describe the package use</Description>
    <PackageLicenseExpression>MIT</PackageLicenseExpression>
    <!-- Change this by the tags of your package -->
    <PackageTags>Test, Discovery</PackageTags>
    <!-- Change this by the url of your repository on GitHub -->
    <RepositoryUrl>https://github.com/MorganKryze/ConsoleAppVisuals</RepositoryUrl>
    <RepositoryType>git</RepositoryType>
    <PackageReadmeFile>README.md</PackageReadmeFile>
    <PackageLicenseFile>LICENSE</PackageLicenseFile>
  </PropertyGroup>

  <PropertyGroup>
    <!-- NuGet Package Explorer health standards -->
    <EmbedUntrackedSources>true</EmbedUntrackedSources>
    <PublishRepositoryUrl>true</PublishRepositoryUrl>
    <IncludeSymbols>true</IncludeSymbols>
    <SymbolPackageFormat>snupkg</SymbolPackageFormat>
  </PropertyGroup>

  <!-- This condition let you build locally, and on a test github action without
  issue. Only the action CD.yml (see later) will enable this condition. this is part
  of the package health standards for deterministic build. -->
  <PropertyGroup>
    <Condition="'$(GITHUB_ACTIONS)' == 'true' AND '$(GITHUB_ACTION)' == 'publish'">
      <ContinuousIntegrationBuild>true</ContinuousIntegrationBuild>
    </PropertyGroup>

  <PropertyGroup>
    <!-- Publishing Settings -->
```

```

    <GenerateDocumentationFile>true</GenerateDocumentationFile>
    <PublishRelease>true</PublishRelease>
    <PackRelease>true</PackRelease>
</PropertyGroup>

<ItemGroup>
  <!-- Assets load-->
  <!-- MANDATORY: give the filepath to the files declared -->
  <!-- OPTIONAL: give a custom path to store them inside your package -->
  <None Include="..\README.md" Pack="true" PackagePath="" />
  <None Include="..\LICENSE" Pack="true" PackagePath="" />
</ItemGroup>

<ItemGroup>
  <!-- Dependencies if you have-->
  <PackageReference Include="yamldotnet" Version="15.1.2" />
</ItemGroup>
</Project>

```

Consider checking that the filepaths are accurate. Here is the file structure of the project updated:

```

Example_project <-- root
├── MyApp
│   ├── bin
│   ├── MyApp.csproj
│   └── Program.cs
├── LICENSE
└── README.md

```

[Learn more](#) [about making your project deterministic.](#)

## Build the project

Now we will be able to build your project including the metadata for the package.

```
cd MyApp
```

```
dotnet build -c Release
```

Optional: You can also run the tests to make sure everything is working as expected by creating a local package:

```
dotnet pack -c Release
```

You will then find the NuGet package in the `bin/Release` folder of your project.

## Publish your package

### API keys

API keys are a way to authenticate yourself to a service. Using those keys will enable you to create automation to deploy and publish packages for example. You will need to create an API key for NuGet and GitHub.

Go to [Nuget.org](https://www.nuget.org) and sign in. Then jump to the [API keys page](#) and create a new API key. Copy it to your clipboard (I recommend you to store it somewhere safe afterward like in a password manager).

#### ⊗ IMPORTANT

Set your API key as a secret of your repository on GitHub and name it `NUGET_API_KEY`. Paste your API key in the value field.

To create a GitHub personal API key, go to [this page](#) and create a classic token. You will need to check the "write:packages" scope. Copy it to your clipboard (I recommend you to store it somewhere safe afterward like in a password manager).

## Automation

Now we will set up a github action to automate the process of publishing your package.

Create two folders in the root of your project: `.github` then `workflows` inside.

Create a new file in the `.github/workflows` folder and name it `CD.yml`.

```
name: Publish package

on:
  push:
    tags:
      - 'v[0-9]+.[0-9]+.[0-9]+'

jobs:
  build:
    runs-on: ubuntu-latest
```

```

timeout-minutes: 15
steps:
  - name: Checkout
    uses: actions/checkout@v3
    with:
      fetch-depth: 0
  - name: Verify commit exists in origin/main
    run: git branch --remote --contains | grep origin/main
  - name: Extract release notes
    run: |
      git log --pretty=format:'%d %s' ${GITHUB_REF} | perl -pe 's| \(.?tag:
v(\d+.\d+.\d+(-preview\d{3})?)(, .?)*\)|\n## \1\n|g' > RELEASE-NOTES
  - name: Set VERSION variable from tag
    run: echo "VERSION=${GITHUB_REF/refs\*/tags\*/v/}" >> $GITHUB_ENV
  - name: Pack library
    run: dotnet pack <your_path_from_your_project_file.csproj>
/p:Version=${VERSION} /p:ContinuousIntegrationBuild=true --output .
  env:
    GITHUB_ACTIONS: true
    GITHUB_ACTION: 'publish'
  - name: Push to GitHub Packages
    run: dotnet nuget push <name_of_your_app>.${VERSION}.nupkg --source
https://nuget.pkg.github.com/<nuget_username>/index.json --api-key ${GITHUB_TOKEN}
  env:
    GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
  - name: Push to NuGet.org
    run: dotnet nuget push <name_of_your_app>.${VERSION}.nupkg --source
https://api.nuget.org/v3/index.json --api-key ${ secrets.NUGET_API_KEY }


```

### ⊗ IMPORTANT

Check that you updated:

- <your\_path\_from\_your\_project\_file.csproj> by the path to your .csproj file
- <name\_of\_your\_app> by the name of your app
- <nuget\_username> by your NuGet username (on the same line as the dotnet nuget push command)

Then, commit your changes and push them to your repository.

Finally, create a new release and add a tag to it as follow "vX.X.X" where X is a number representing the version of your package. [Learn more](#) 

### TIP

Wait a few minutes and you will find your package on NuGet.org and GitHub packages, you will be notified by email.

## Clean up

If that project was indeed for you for demo purposes, you cannot delete it from NuGet.org, but you can hide it by unlisting it: Go to Manage Packages > select the package > click on the Edit button > Listing category > unchecked the "List in search results" checkbox > Save.

## Bonus: Prefix ID

To protect the uniqueness of your package name, you can reserve a prefix for your package. This will prevent someone else from using the same name as your package. That way, I reserved "ConsoleAppVisuals" and "ConsoleAppVisuals.\*" (meaning that "ConsoleAppVisuals" and "ConsoleAppVisuals.MyApp" will be reserved for example).

To do so, you only need to send an email to [account@nuget.org](mailto:account@nuget.org) with the subject "Package ID prefix reservation" and give your NuGet username (or organization, or other name of collaborators) and the prefixes you want to reserve. The criteria are given in this [page](#).

## Resources

- [Official NuGet documentation](#)
- [Main Source](#)
- [Recap](#)
- [Deterministic Builds](#)
- [Package ID](#)

---

Have a question, give a feedback or found a bug? Feel free to [open an issue](#) or [start a discussion](#) on the GitHub repository.

# C# for Visual Studio Code

## Introduction

This guide will help you set up your development environment for C# using Visual Studio Code. It will guide you through the installation of the .NET SDK, Visual Studio Code, and the C# extension for a complete setup.

## Install .NET SDK

The .NET SDK is a free, open-source development platform for building many different types of applications. It includes the C# compiler, the .NET runtime, and the ASP.NET Core runtime.

1. Download the .NET SDK installer from the [official website](#).
2. Run the installer and follow the instructions.
3. Once installed, open a new terminal and run the following command to verify the installation:

```
dotnet --version
```

## Install Visual Studio Code

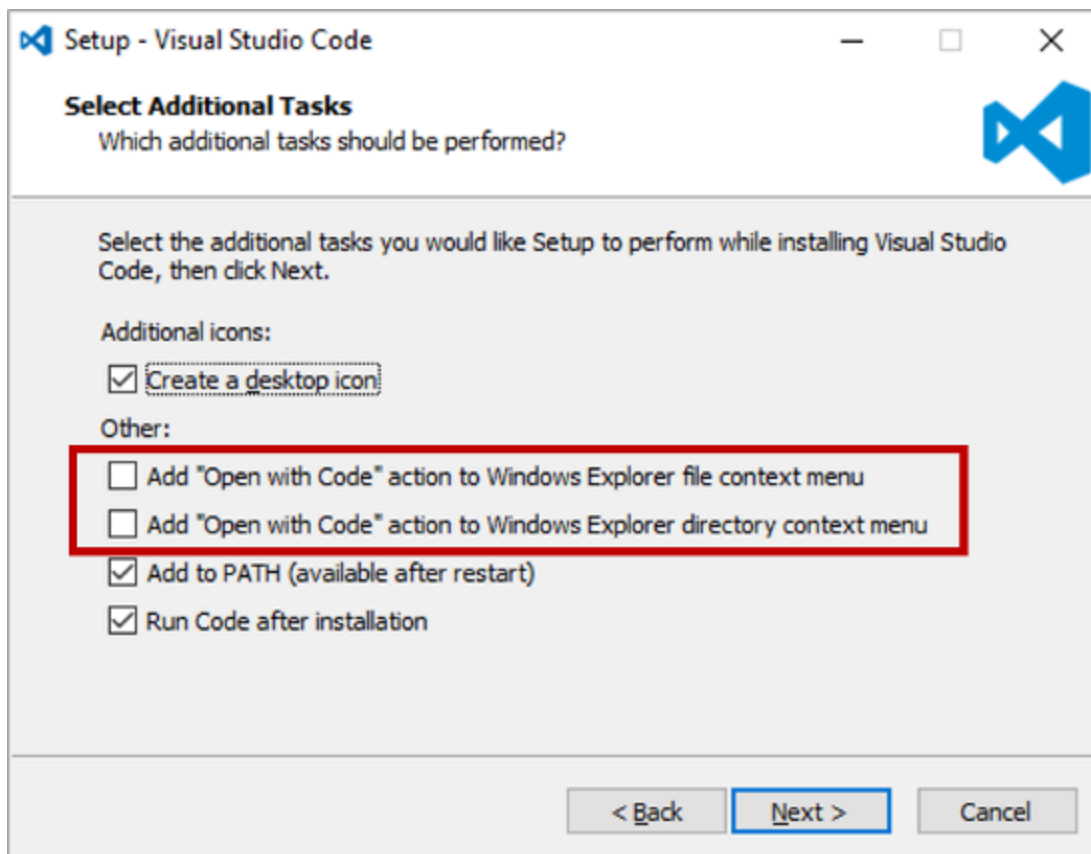
Visual Studio Code is a free source code editor developed by Microsoft for Windows, Linux and macOS. It includes support for debugging, embedded Git control, syntax highlighting, intelligent code completion, snippets, and code refactoring.

Windows

MacOS

Linux

1. Download the Visual Studio Code installer from the [official website](#).
2. Run the installer and follow the instructions (Consider adding **Open with Code** action to Windows Explorer context menu).



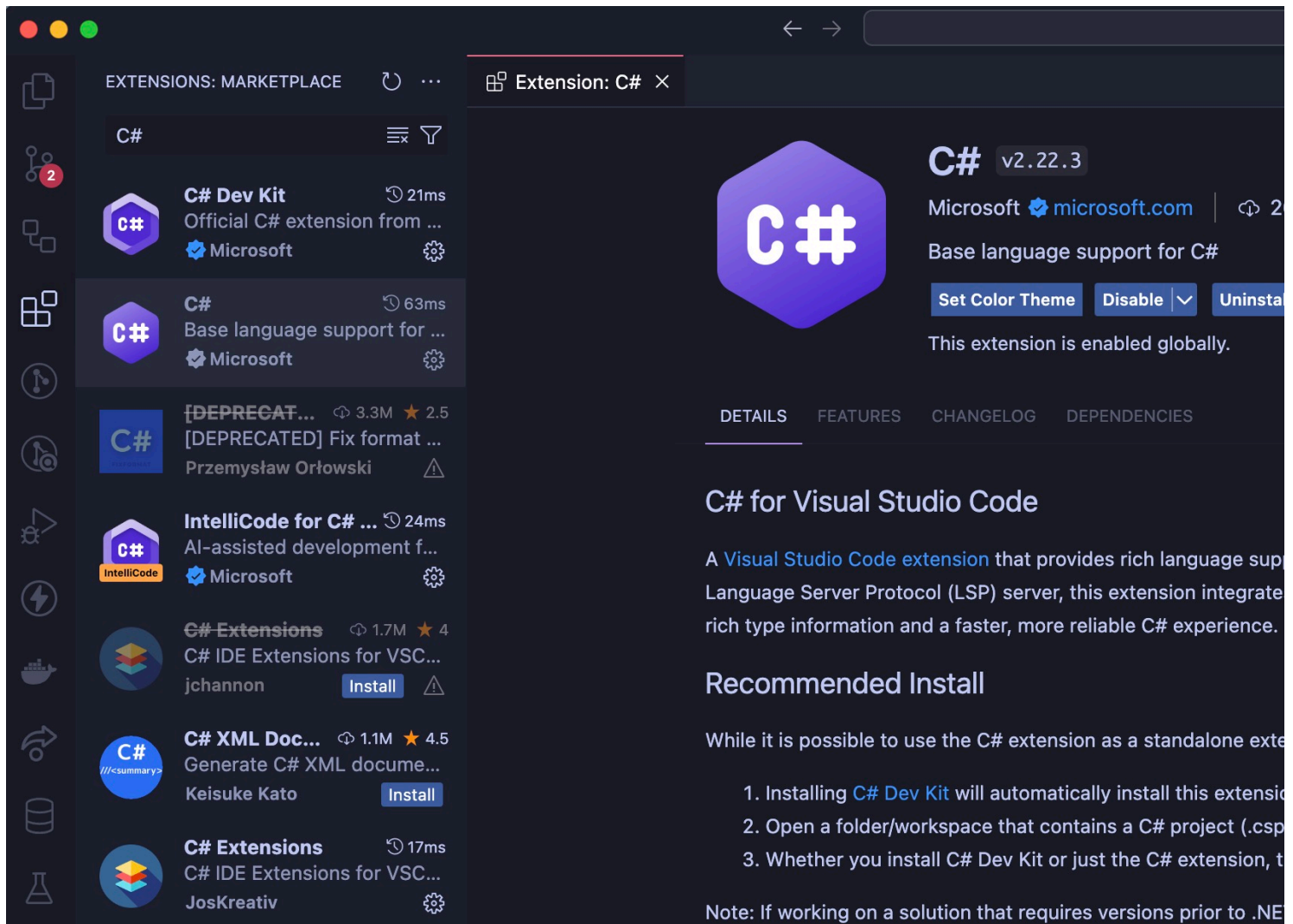
3. Once installed, open Visual Studio Code.

4. Click on **File** > **Auto Save** to enable the auto-save feature.

## Install the C# extension

The C# extension for Visual Studio Code adds support for C# to Visual Studio Code, including features such as syntax highlighting, IntelliSense (code completion), and debugging.

Find them here:



The extensions to install are:

- [C#](#)
- [C# Extensions](#)
- [C# Dev Kit](#)

## Bonus: Developer Tools

The following extensions are not required but can be useful:

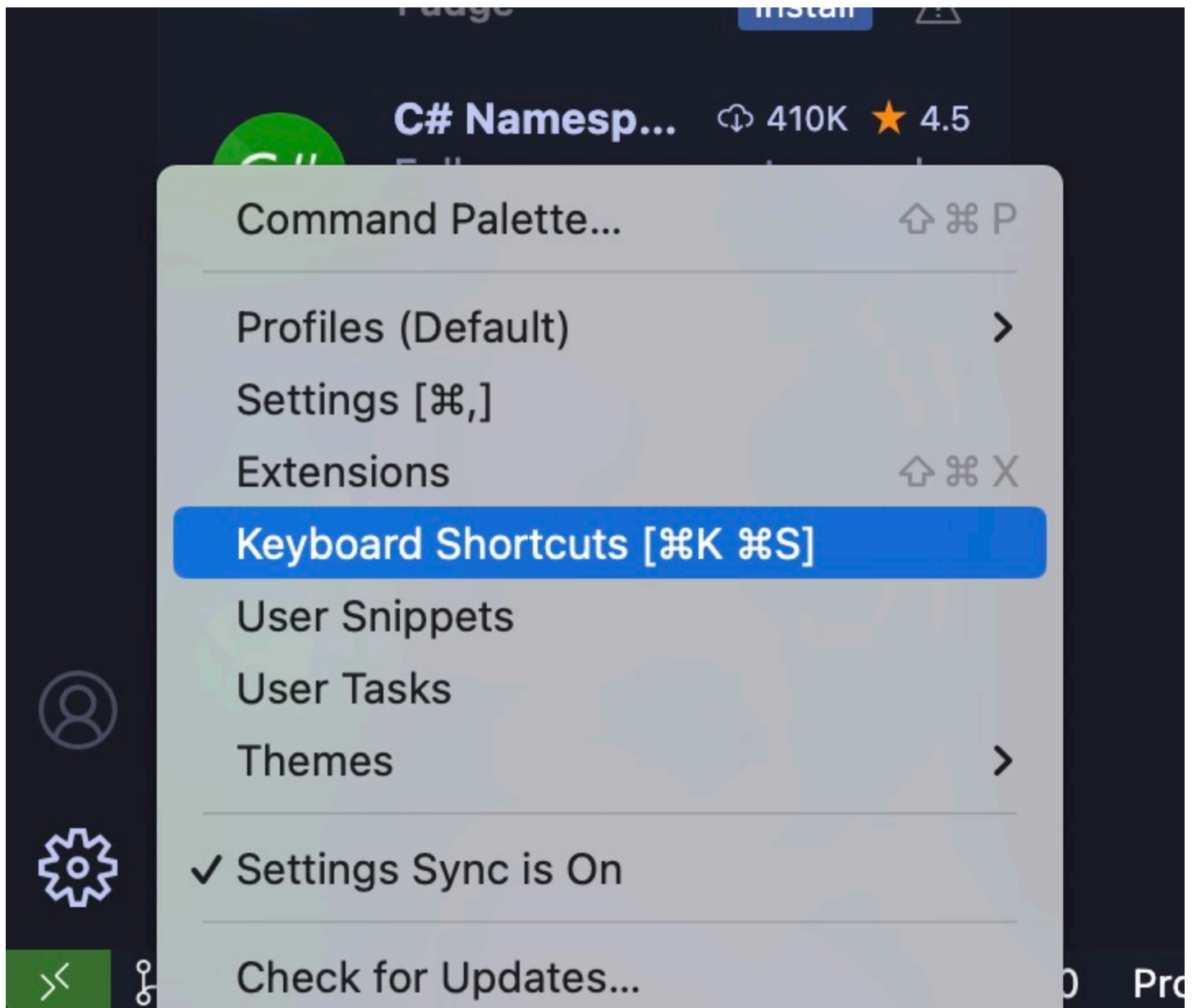
- [CSharpier](#)
- [tokyo-night](#)
- [Reload](#)

Open a terminal and run the following command to install CSharpier (code formatter):

```
dotnet tool install --global csharpier
```

Then, open KeyboardShortcuts:






Search for `Format Document` and set the keybinding to `Ctrl+S` (or `Cmd+S` on MacOS)

Keyboard Shortcuts X

Format Document

| Command   | Keybinding | When  | Source                    |
|---|------------|---|---------------------------|
| Format Cell<br>editor.action.formatDocument   | ⇧ ⌘ F      | editorHasDocumentFormattingProvider && editorTextFocus && inComp... | System                    |
| <b>Format Document</b><br>editor.action.formatDocument  | ⌘ S        | editorHasDocumentFormattingProvider && editorTextFocus && !edito... | User                      |
|  <b>Format Document</b><br>editor.action.formatDocument.none | ⇧ ⌘ F      | editorTextFocus && !editorHasDocumentFormattingProvider && !edit... | System                    |
| Format Selection<br>Format Document (editor.action.formatDocument.none)   | ⌘ K ⌘ F    | editorHasDocumentSelectionFormattingProvider && editorTextFocus ... | System                    |
| <b>Format Document (Forced)</b><br>prettier.forceFormatDocument   |            | -   | Prettier - Code formatter |
| <b>Format Document With...</b><br>editor.action.formatDocument.multiple   |            | -   | System                    |
| Ruff: <b>Format document</b>  |            | -   | Ruff                      |

Finally, open a C# file and press **Ctrl+S** (Cmd+S on MacOS) to format the document.

## Resources

- [To go further](#)

Have a question, give a feedback or found a bug? Feel free to [open an issue](#) or [start a discussion](#) on the GitHub repository.