AsciiDoc Template

Table of Contents

1. About

This `.adoc` file is an example of how to use `AsciiDoc` as an alternative to Word / Markdown or Latex to cut down on time needed for writing documentation. The official documentation of `AsciiDoc` can be found at AsciiDoc.org.

The provided template, uses custom theming and fonts and was made for use by students studying at `FH Hagenberg`, but can be used by anyone that wants to pick up AsciiDoc.

The Template inside the `template` folder is structured like the requirements for most exercise submissions, containing a src and doc folder. The template and its dependencies itself is inside `./template/doc`. I recommend just copying the doc folder into the same level as your source code folder.

Though, I created this template, I want to offer my thanks to **Florian W.** (flohero on github), for providing the `basic-theme.yaml` and the `auto-include.sh` script. The script can be used for creating automatic includes for all your code files inside your project. It works by providing the starting directory and the file extension to search for. This script is a `bash` script, i.e. to run it on Windows, use the `wsl` (Windows Subsystem for Linux). For Example: `auto-include.sh ./path/to/src cs` searches for `.cs` files in the provided directory and its subdirectories.

2. Intro

AsciiDoc offers many features, that make it very easy to write a clean documentation for your code. This document is a showcase of how to achieve different markups, include images, and code.

A very convenient aspect of AsciiDoc is, that many popular IDE's offer some sort of support for AsciiDoc, be

it natively or via plugins. This means, that you can manage both your Code and Documentation in the same Editor, not needing to switch between programs.

This document for example was written in IntelliJ IDEA, with the help of the `AsciiDoc` plugin, which is available for all JetBrain IDE's. VS Code offers a similar experience via the `AsciiDoc` Extension. These plugins, also offer the ability to export the asciidoc file to a pdf, though some IDE's might require an installation of `AsciiDoc` via `Roby`, not JetBrains and VS Code though.

Since our submissions often require the exercise specification to be included, I recommend, using a pdf editor, filing out the required fields, and setting the completed exercise specification as the cover-image, this vastly reduces the hassle of needing to manually merge the pdfs if you have already submitted your exercise, but found something that needs fixing.

To set a pdf as the cover-image / page, declare `:front-cover-image: path-to-the-specification.pdf` in the top section.

3. Working with AsciiDoc

3.1. Text formatting

This section covers only the basic formatting options. For more info visit AsciiDoc.org.

Text can be made **Bold** by placing it between two stars * *.

Italic is achieved, by placing it between two underscores _ _.

`Monospace` can be achieved, by placing it between two apostrophes ` `:

The option can also be `***combined as needed***`.

3.2. Images

To include images, start a line with the `imgae::` macro:

```
image::wator-parallel-aproach.png[]
```

The square brackets, are needed for the image to be displayed instead of the text. The square brackets can also be used to change the image width / height or other properties.

For an image to be displayed, it is also important, that there is an empty line before the `image` macro.

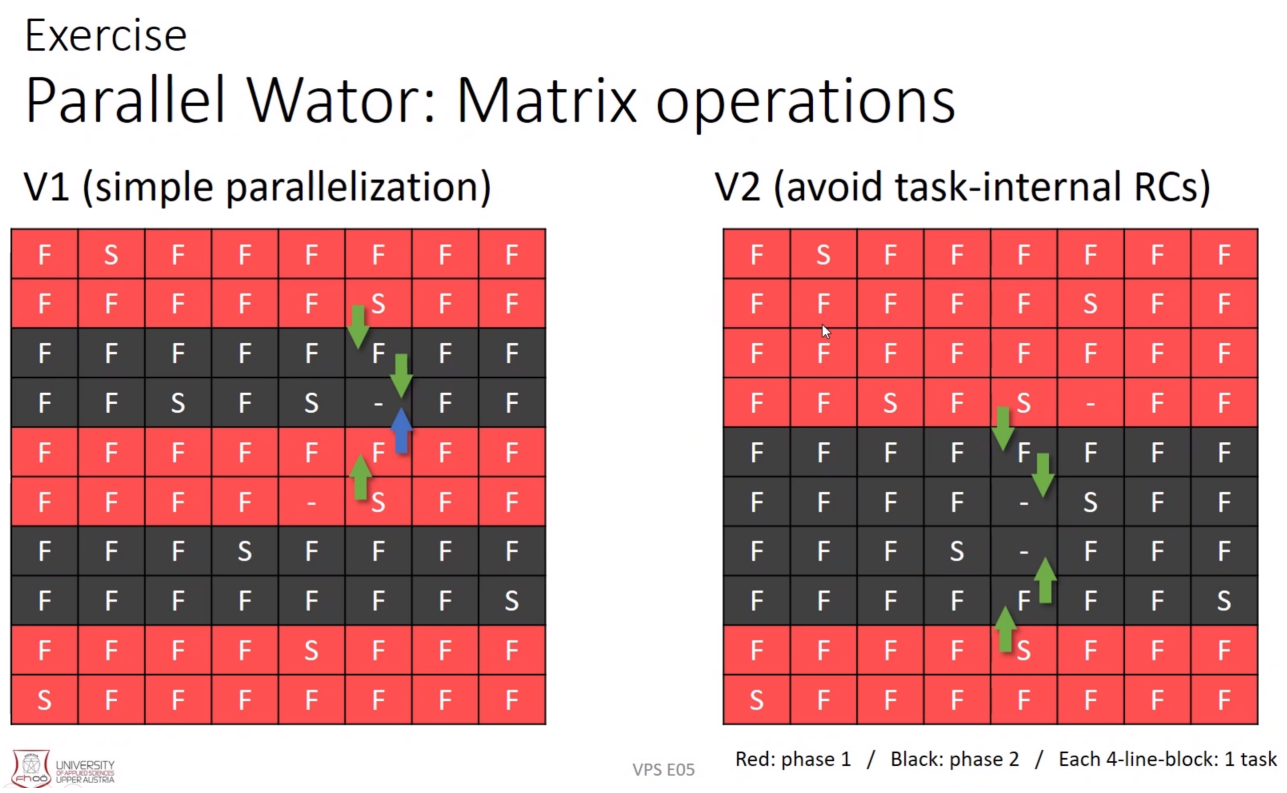

Figure 1. Caption for the example image

All paths to an image that are relative, start add the current directory or at the directory specified by `:imagesdir:`.

More on images can be found at Asciidoc.org.

3.3. Code

Code snippets can be included, by using the `include::` macro and wrapping it in a `Source Code Block`:

```
[source,cs]
----
 include::./example-code/csharp/ControlFlow.cs[lines=9..30]
----
```

The square brackets, declare it as a code block, and the second parameter defines the source language to get correct syntax highlighting.

The brackets in the include statement can be used to format the output. Lines for example, restrict the shown part to only the specified lines. `Lines=20..` would show everything after line 19 and could be used to exclude import / using statements.

Listing 1. Example C# code

```
    public static void PrintThread(string msg) =>
        Console.WriteLine(msg + ", thread: " + Thread.CurrentThread.ManagedThreadId);

    static async Task<int> ContentLengthAsync(Uri address) {
        using var client = new HttpClient();

        PrintThread("ContentLengthAsync start getStringTask");
        Task<string> getStringTask = client.GetStringAsync(address);
        PrintThread("getStringTask yielded control");

        DoSomethingUseful();

        PrintThread("awaiting getStringTask");
        string content = await getStringTask;
        PrintThread("getStringTask finished continuing");

        return content.Length;
    }

    static void DoSomethingUseful() {
        PrintThread("Doing something useful");
    }
```

If you do not want to always give the full relative path, you can declare your own shorthands like `:src: path_to_src_directory` or for this example `:csharp:` and `:kotlin:`

```
:kotlin: ./example-code/kotlin
:csharp: ./example-code/csharp
```

Listing 2. Example Kotlin code

```
fun main() {
    println("Synchronous World!  ${thread()}")

    runBlocking(Dispatchers.Default) {
        println("Concurrent world! ${thread()}")
        val job1 = launch { worker(1) }
        launch { startCalculations() }
        println("Job1 is completed? ${job1.isCompleted}")
        println("Coroutine completing ${thread()}")
    }

    println("Synchronous World!  ${thread()}")
}
/*  output:     Synchronous World!          main
                Concurrent world!           DefaultDispatcher-worker-2
                Job1 is completed? false
                Coroutine completing        DefaultDispatcher-worker-2
                Worker 1 started            DefaultDispatcher-worker-3
                not blocking...             DefaultDispatcher-worker-2
                Calculation 1 started       DefaultDispatcher-worker-2
                Worker 1 completed          DefaultDispatcher-worker-1
                Calculation completed: 69   DefaultDispatcher-worker-1
                Synchronous World!          main                            */
```

Listing 3. Example C# code which restricts the shown code to a few lines by specifying `lines=32..38`

```
    static async Task Main(string[] args) {
        PrintThread("Main");
        Task<int> getLengthTask = ContentLengthAsync(new Uri("https://www.fh-ooe.at/"));
        PrintThread("getLengthTask yielded control");
        int result = await getLengthTask;
        PrintThread("getLengthTask finished continuing");
    }
```

More about code blocks can be read on AsciiDoc.org.

3.4. Lists / Enumerations

AsciiDoc provides different ways to create lists and can be quite cumbersome at first.

Ordered list can be created, by starting a line with dots. Every dot represents a level of intendation.

1. The first entry
    a. The next level
        i. The third level
2. The second entry

Unorderd lists are started with stars (*).

- First entry
    - second level
        - third level
- Second entry

Ordered and unordered list can be nested like this:

```
* Nested List
+
Text that is intended by continuing with the `+`
+
. Ordered List
.. second level
* Second entry
```

- Mixed List

  Text that is intended by continuing with the +

  1. Ordered List
      a. second level
- Second entry

More on list can be found at AsciiDoc.org.