Cottle Documentation

Release 2.0.0

Cottle

Jan 11, 2021

Contents

CHAPTER 1

Table of contents

1.1 Overview

1.1.1 What does it looks like?

Cottle (short for **C**ompact **O**bject to **T**ext **T**ransform **L**anguag**e**) is a lightweight template engine for .NET (.NET Framework >= 4.7.2 & .NET Standard >= 2.0) allowing you to transform structured data into any text-based format output such as plain text, HTML or XML. It uses a simple yet easily extensible template language, thus enabling clean separation of document contents and layout.

A simple Cottle template printing an HTML document showing how many messages are in your mailbox could look like this:

```
{wrap html:
    <h1>Hello, {name}!</h1>
    <p>
        {if len(messages) > 0:
            You have {len(messages)} new message{if len(messages) > 1:s} in your␣
↪mailbox!
        |else:
            You have no new message.
        }
    </p>
}
```

As you can guess by looking at this code, a Cottle template contains both plain text printed as-is as well as commands used to output dynamic contents. Cottle supports most common template engine features, such as:

- Text substitution through variables,
- Mathematical and boolean expressions,
- Built-in and used-defined functions,
- Variables & functions declaration and assignments,

- Text escaping control (wrap, unwrap),
- Conditional statements (if),
- Loops (for, while).

Source code is open for reviews and contributions!

1.1.2 Download the library

Cottle is available as an installable package on NuGet official website. Just open your extension manager in Visual Studio, search for "Cottle" and install from there.

You can also read, download or contribute to the source code on GitHub.

1.1.3 Getting started

To start using Cottle, first reference the package in your solution (using NuGet or manual install as detailed above). You'll then need two things:

- An input template written with Cottle's template language, used to define how your data will be rendered. This template can be contained in a `string` or streamed from any source compatible with IO.TextReader class (text file, memory buffer, network socket...) as shown in the example below.
- An executable code that reads your input template, create a *IDocument* object from it then render it to an output string or IO.TextWriter instance.

Here is a basic sample rendering a template with a single injected variable. Copy the **C# source** snippet somewhere in your program and get it executed. You should see the content of **Rendering output** snippet printed to standard output:

Listing 1: C# source

```
void RenderAndPrintTemplate()
{
    var template = "Hello {who}, stay awhile and listen!";

    var documentResult = Document.CreateDefault(template); // Create from template␣
↪string
    var document = documentResult.DocumentOrThrow; // Throws ParseException on error

    var context = Context.CreateBuiltin(new Dictionary<Value, Value>
    {
        ["who"] = "my friend" // Declare new variable "who" with value "my friend"
    });

    // TODO: customize rendering if needed

    Console.Write(document.Render(context));
}
```

Listing 2: Rendering output

```
Hello my friend, stay awhile and listen!
```

For following code samples we'll introduce **Cottle template**, **C# source** and **Rendering output** snippets to hold corresponding fragments. You'll always need a C# wrapper similar to the one above in your code, so only new features will be specified in following examples ; they should replace the **TODO** comment highligted in above **Rendering outout** snippet.

1.2 Template language

1.2.1 Language syntax

Plain text and commands

A Cottle template can contain plain text printed as-is as well as commands that will be executed when document is rendered. These commands can either print dynamic content or have side-effects such as defining variables or controlling the rendering flow.

The most important command you'll need is the `echo` command that takes an argument and outputs its contents. Here is how it works:

Listing 3: Cottle template

```
Value of x is {echo x}.
```

Listing 4: C# source

```
var context = Context.CreateBuiltin(new Dictionary<Value, Value>
{
    ["x"] = 53
});
```

Listing 5: Rendering output

```
Value of x is 53.
```

In this example we're creating a *variable* named `x` with value 53 and pass it when rendering our template. Then we're using the `echo` command to output the value of this variable after some constant plain text.

Implicit echo command

Since `echo` is the most commonly used command it supports a shorter implicit form: you can omit the "echo" keyword as long as the name of the variable you're printing doesn't conflict with another command. This means the following example will produce the same output:

Listing 6: Cottle template

```
Value of x is {x}.
```

Implicit form of `echo` command can be used everywhere as long as you're not printing a variable having the same name than a Cottle command such as `for`. While technically possible, using Cottle command names as variables should be avoided for readability reasons anyway.

Command delimiters

All commands must be specified between **{** (*start of command*) and **}** (*end of command*) delimiters, which can be redefined in configuration if needed (read section *Delimiters customization* to learn how). Some commands having a plain text body also use **|** (*continue*) delimiter as we'll see later. Delimiters must be escaped if you want to use them in plain text, otherwise they would be misinterpreted as commands. This can be achieved by using **** (*escape*) delimiter as shown below:

Listing 7: Cottle template

```
Characters \{, \}, \| and \\ must be escaped when used in plain text.
```

Listing 8: Rendering output

```
Characters {, }, | and \ must be escaped when used in plain text.
```

As visible in this example, backslash character \ must also be used to escape itself when you want to output a backslash. Similar to other delimiters, the *escape* delimiter can be redefined through configuration.

1.2.2 Expressions

Passing variables

To send variables so they can be used when a document is rendered you must provide them through a *IContext* instance which is used as a render-time storage. This interface behaves quite like an immutable `Dictionary<Cottle.Value, Cottle.Value>` where *Value* is a data structure able to store any value Cottle can handle. Key and value pairs within this dictionary are used as variable names and their associated values.

Implicit constructors from some native .NET types to *Value* type are provided so you usually don't have to explicitly do the conversion yourself but you can also create values using `Value.FromSomething()` static construction methods (where "Something" is a known .NET type). See API documentation about *Value* type for details.

Once you assigned variables to a context, pass it to your document's rendering method so you can read them from your template (see section *Getting started* for a full example):

Listing 9: Cottle template

```
Hello {name}, you have no new message.
```

Listing 10: C# source

```
var context = Context.CreateBuiltin(new Dictionary<Value, Value>
{
    ["name"] = "John" // Implicit conversion from string to Value
});
```

Listing 11: Rendering output

```
Hello John, you have no new message.
```

Instances of *IContext* are passed at document render time so they can be changed from one render to another, while instances of *IDocument* can then be *rendered* as many time as you want. Compiling a template string into an *IDocument* is a costly process implying parsing the string, validating its contents, applying code optimizations and storing it as an internal data structure. You should organize your code to avoid re-creating documents from the same template multiple time, as compiling a document is significantly more costly than rendering it.

Value types

Cottle supports immutable values which can either be declared as constants in templates or set in contexts you pass when rendering a document. Values have a type which can be one of the following:

- Boolean (value is either true or false),

- Number (equivalent to .NET's double),
- String (sequence of character),
- Map (associative key/value container),
- Void (value is undefined ; any undeclared variable has void type).

Map values are associative tables that contain multiple children values stored as key/value pairs. Values within a map can be accessed directly by their key, using either dotted or subscript notation:

Listing 12: Cottle template

```
You can use either {mymap.f1} or {mymap["f2"]} notations for map values.
```

Listing 13: C# source

```
var context = Context.CreateBuiltin(new Dictionary<Value, Value>
{
    ["mymap"] = new Dictionary<Value, Value> // Implicit conversion to Value
    {
        ["f1"] = "dotted",
        ["f2"] = "subscript"
    }
});
```

Listing 14: Rendering output

```
You can use either dotted or subscript notations for map values.
```

Please note the quotes used in subscript notation. Trying to access value of `{mymap[f2]}` will result in a very different behavior, since it will search for the value whose key is the value of `f2` (which hasn't be defined), leading to an undefined result. It is valid to have a map in which two or more keys are equal, but you will only be able to access the last one when using direct access. Iterating over the map's elements will however show you its entire contents.

Implicit constructors on *`Value`* class allow you to convert most .NET standard types into a Cottle value instance. To get an undefined value your from C# code use the `Cottle.Value.Undefined` static field.

You can also declare constant values in your templates with following constructs:

Listing 15: Cottle template

```
{17.42}
{"Constant string"}
{'String with single quotes'}
{["key1": "value1", "key2": "value2"]}
{["map", "with", "numeric", "keys"]}
```

When declaring a constant map without keys, numeric increasing keys (starting at index 0) are implied. Also remember that both keys and values can be of any value type (numbers, strings, other nested maps...).

Note: There are no *false* nor *true* constants in Cottle. You can inject them as variables if needed, but numeric values 0 and 1 can be considered as equivalent in most scenarios.

Expression operators

Cottle supports common mathematical and logical operators. Here is the list of all operators sorted by decreasing precedence order:

- `+`, `-` and `!`: unary plus, minus and logical "not" operator ;
- `*`, `/` and `%`: binary multiplication, division and modulo operators ;
- `+` and `-`: binary addition and subtraction operators ;
- `<`, `<=`, `=`, `!=`, `>=` and `>`: binary logical comparison operators ;
- `&&` and `||`: binary "and" and "or" logical operators.

You can also use `(` and `)` to group sub-expressions and change natural precedence order. Here are some example of valid expressions:

Listing 16: Cottle template

```
{1 + 2 * 3}
{(1 + 2) * 3}
{!(x < 1 || x > 9)}
{value / 2 >= -10}
{"aaa" < "aab"}
```

Note: Mathematical operators (`+`, `-`, `*`, `/` and `%`) only accept numeric operands and will try to cast other types to numbers (see *`Value`* type for details about conversion to number).

Note: Logical operators can compare any type of operand and uses the same comparison algorithm than built-in function *cmp(x, y)*.

Calling functions

Functions in Cottle are special values that can be invoked with arguments. They must be set in a context as any other value type, and a helper method is available so you can start with a predefined set of built-in functions when rendering your documents. Create a context using *`Context.CreateBuiltin`* method to have all built-in functions available in your document:

Listing 17: Cottle template

```
You have {len(messages)} new message{when(len(messages) > 1, 's')} in your inbox.
```

Listing 18: C# source

```
var context = Context.CreateBuiltin(new Dictionary<Value, Value>
{
    ["messages"] = new Value[]
    {
        "message #0",
        "message #1",
        "message #2"
    }
});
```

Listing 19: Rendering output

```
You have 3 new messages in your inbox.
```

The list of all built-in functions as well as their behavior is available in section *Built-in functions*. For all following samples in this document we'll assume that built-in functions are available when rendering a template.

Note: If you don't want any built-in function to be available in your template, you can create an blank context by calling `Context.CreateCustom` method.

1.2.3 Commands

Text escaping: wrap, unwrap

You'll most probably want to escape unsafe values (e.g. user input) before printing their contents from your templates, like making sure characters "<" and ">" are replaced by "<" and ">" when printing variables to an HTML document. While this can be done by injecting an escaping function and using it to wrap all the expressions you want to print with `echo` command, a nice alternative is using `wrap` command with a function such as Web.HttpUtility.HtmlEncode to ensure nothing is left unescaped before printing:

Listing 20: Cottle template

```
{wrap html:
    <p data-description="{op_description}">
        {op_name}
    </p>
}
```

Listing 21: C# source

```
var htmlEncode = Function.CreatePure1((s, v) => HttpUtility.HtmlEncode(v));
var context = Context.CreateBuiltin(new Dictionary<Value, Value>
{
    ["html"] = Value.FromFunction(htmlEncode),
    ["op_description"] = "Three-way comparison or \"spaceship operator\"",
    ["op_name"] = "<=>"
});
```

Listing 22: Rendering output

```
<p data-description="Three-way comparison or &quot;spaceship operator&quot;">
    &lt;=&gt;
</p>
```

The `wrap` command expects a function parameter, then a **:** (*body declaration*) separator character and the body you want the escaping to be applied to. Use **}** (*end of command*) delimiter to close the `wrap` command and stop applying its effect. Command body is a Cottle template, meaning it can contain plain text and commands as well. Inside `wrap` command's body, the function you passed as a parameter will be invoked on every value printed by inner `echo` commands, and the result of this function will be printed instead of original value. This means our previous example will produce an output equivalent to this template:

Listing 23: Cottle template

```
<p data-description="{html(op_description)}">
    {html(op_name)}
</p>
```

You may occasionally want to cancel wrapping for printing a safe HTML snippet without wrapping it. This can be achieved with the `unwrap` command that cancels its parent `wrap` command:

Listing 24: Cottle template

```
{wrap html:
    <p>This {variable} will be HTML-escaped.</p>
    {unwrap:
        <p>This {raw} one won't so make sure it doesn't contain unvalidated user␣
↪input!</p>
    }
    <p>We're back in {safe} context here with HTML escaping enabled.</p>
}
```

Multiple `wrap` commands can be nested, resulting in their functions being called from the innermost to outermost `wrap` command.

Conditionals: if

You can write conditional statements by using the `if` command which uses an expression as a predicate to check whether its body should be printed or not. Predicate is verified if value, once converted to a boolean type, is true (see *`Value`* type for details about conversion to boolean).

Listing 25: Cottle template

```
{if 1:
    A condition on a numeric value is true if the value is non-zero.
}

{if "aaa":
    {if 1 + 1 = 2:
        Commands can be nested.
    }
}
```

Listing 26: Rendering output

```
A condition on a numeric value is true if the value is non-zero.

Commands can be nested.
```

The `if` command uses a syntax similar to `wrap` command and expects a predicate expression ended by a **:** (*body declaration*) separator character and followed by body of the `if` command, then a **}** (*end of command*) delimiter. This command also supports optional `elif` (else if) and `else` blocks that behave like in any other programming language. These can be specified using the **|** (*continue*) delimiter followed by either `elif` and a predicate or `else`, then a **:** (*body declaration*) separator character, and a body similar to the `if` command. Last block must be ended by a **}** (*end of command*) delimiter:

Listing 27: Cottle template

```
{if test:
    Variable "test" is true!
|else:
    Variable "test" is false!
}

{if len(items) > 2:
    There are more than two items in map ({len(items)}, actually).
}

{if x < 0:
    X is negative.
|elif x > 0:
    X is positive.
|else:
    X is zero.
}
```

Listing 28: C# source

```
var context = Context.CreateBuiltin(new Dictionary<Value, Value>
{
    ["items"] = new Value[]
    {
        "item #0",
        "item #1",
        "item #2"
    },
    ["test"] = 42,
    ["x"] = -3
});
```

Listing 29: Rendering output

```
Variable "test" is true!

There are more than two items in map (3, actually).

X is negative.
```

Enumerations: for

Keys and values within a map can be enumerated using the `for` command, which repeatedly evaluates its body for each key/value pair contained within the map. The `for` command also supports an optional `empty` block evaluated when the map you enumerated doesn't contain any key/value pair.

Syntax of the `for` keyword and its optional `empty` block is similar to the `else` block of the `if` command (see section *Conditionals: if*):

Listing 30: Cottle template

```
{for index, text in messages:
    Message #{index + 1}: {text}
```

(continues on next page)

(continued from previous page)

```
|empty:
    No messages to display.
}

Tags: {for tag in tags:{tag} }
```

Listing 31: C# source

```
var context = Context.CreateBuiltin(new Dictionary<Value, Value>
{
    ["messages"] = new Value[]
    {
        "Hi, this is a sample message!",
        "Hi, me again!",
        "Hi, guess what?"
    },
    ["tags"] = new Value[]
    {
        "action",
        "horror",
        "fantastic"
    }
});
```

Listing 32: Rendering output

```
Message #1: Hi, this is a sample message!
Message #2: Hi, me again!
Message #3: Hi, guess what?

Tags: action horror fantastic
```

Note: Use syntax `for value in map` instead of `for key, value in map` if you don't need to use map keys.

Assignments: set

You can assign variables during rendering with the `set` command. Variable assignment helps you improving performance by storing intermediate results (such as function calls) when using them multiple times.

Listing 33: Cottle template

```
{set nb_msgs to len(messages)}

{if nb_msgs > 0:
    You have {nb_msgs} new message{if nb_msgs > 1:s} in your mailbox!
|else:
    You have no new message.
}

{set nb_long to 0}
```

(continues on next page)

(continued from previous page)

```
{for message in messages:
    {if len(message) > 20:
        {set nb_long to nb_long + 1}
    }
}

{nb_long} message{if nb_long > 1:s is|else: are} more than 20 characters long.
```

Listing 34: C# source

```
var context = Context.CreateBuiltin(new Dictionary<Value, Value>
{
    ["messages"] = new Value[]
    {
        "Hi, this is a sample message!"
        "Hi, me again!",
        "Hi, guess what?"
    }
});
```

Listing 35: Rendering output

```
You have 3 new messages in your mailbox!

1 message is more than 20 characters long.
```

Note: Cottle variables have visibility scopes, which are described in section *Variable scope*.

Loops: while

The `while` command evaluates a predicate expression and continues executing its body until predicate becomes false. Be sure to check for a condition that will become false after a finite number of iterations, otherwise rendering of your template may never complete.

Listing 36: Cottle template

```
{set min_length to 64}
{set result to ""}
{set words to ["foo", "bar", "baz"]}

{while len(result) < min_length:
    {set result to cat(result, words[rand(len(words))])}
}

{result}
```

Listing 37: Rendering output

```
barbazfoobarbazbazbazbarbarbarbarfoofoofoobarfoobazfoofoofoofoobaz
```

Warning: Prefer the use of the `for` command over `while` command whenever possible.

Debug: dump

When your template doesn't render as you would expect, the `dump` command can help you identify issues by showing value as an explicit human readable string. For example undefined values won't print anything when passed through the `echo` command, but the `dump` command will show them as `<void>`.

Listing 38: Cottle template

```
{dump "string"}
{dump 42}
{dump unknown(3)}
{dump [856, "hello", "x": 17]}
```

Listing 39: Rendering output

```
"string"
42
<void>
[856, "hello", "x": 17]
```

Note: Command `dump` is a debugging command. If you want to get type of a value in production code, see *type(value)* method.

Comments: _

You can use the _ (underscore) command to add comments to your template. This command can be followed by an arbitrary plain text and will be stripped away when template is rendered.

Listing 40: Cottle template

```
{_ This is a comment that will be ignored when rendering the template}

Hello, World!
```

Listing 41: Rendering output

```
Hello, World!
```

1.3 Built-in functions

1.3.1 Logical

and(x, y, ...)

Perform logical "and" between given boolean values, i.e. return `true` if all arguments are equivalent to `true` (see *Value* type for details about conversion to boolean).

Listing 42: Cottle template

```
{and(2 < 3, 5 > 1)}
```

Listing 43: Rendering output

```
true
```

Note: This function is equivalent to operator `&&`.

cmp(x, y)

Compare `x` against `y`, and return -1 if `x` is lower than `y`, 0 if they're equal, or 1 otherwise. When used on numeric values, the `cmp` function uses numerical order. When used on strings, it uses alphabetical order. When used on maps, it first performs numerical comparison on their length then compares keys and values two by two. Two values of different types are always different, but the order between them is undefined.

Listing 44: Cottle template

```
{cmp("abc", "bcd")}
{cmp(9, 6)}
{cmp([2, 4], [2, 4])}
```

Listing 45: Rendering output

```
-1
1
0
```

default(primary, fallback)

Return `primary` if `primary` is equivalent to `true` (see *Value* type for details about conversion to boolean) or `fallback` otherwise.

Listing 46: Cottle template

```
{set x to 3}
{default(x, "invisible")}
{default(y, "visible")}
```

Listing 47: Rendering output

```
3
visible
```

defined(x)

Check whether value `x` is defined by checking it has a non-void type.

This is different than checking whether a value is equivalent to `true` (see *`Value`* type for details about conversion to boolean), for example integer `0` is equivalent to `false` when used as a boolean expression but `defined(0)` is `true`. This function is mostly useful for testing whether a variable has been assigned a value or not.

Listing 48: Cottle template

```
{dump defined(undefined)}
{set a to 0}
{dump defined(a)}
```

Listing 49: Rendering output

```
<false>
<true>
```

eq(x, y, . . .)

Return `true` if all arguments are equal or `false` otherwise. It uses the same comparison algorithm than function *cmp(x, y)*.

Listing 50: Cottle template

```
{eq(7, 7)}
{eq(1, 4)}
{eq("test", "test")}
{eq(1 = 1, 2 = 2, 3 = 3)}
```

Listing 51: Rendering output

```
true
false
true
true
```

Note: This function is equivalent to operator = when used with 2 arguments.

ge(x, y)

Return `true` if `x` has a value greater than or equal to `y` or `false` otherwise. It uses the same comparison algorithm than function *cmp(x, y)*.

Listing 52: Cottle template

```
{ge(7, 3)}
{ge(2, 2)}
{ge("abc", "abx")}
```

Listing 53: Rendering output

```
true
true
false
```

Note: This function is equivalent to operator >=.

gt(x, y)

Return `true` if `x` has a value greater than `y` or `false` otherwise. It uses the same comparison algorithm than function *cmp(x, y)*.

Listing 54: Cottle template

```
{gt(7, 3)}
{gt(2, 2)}
{gt("abc", "abx")}
```

Listing 55: Rendering output

```
true
false
false
```

Note: This function is equivalent to operator >.

has(map, key)

Return `true` if given map has a value associated to given key or `false` otherwise.

Listing 56: Cottle template

```
{has(["name": "Paul", "age": 37, "sex": "M"], "age")}
```

Listing 57: Rendering output

```
true
```

Note: Result of this function is close to but not strictly equivalent to `defined(map[key])` as the former will return `true` if `map` contains a key `key` associated to an undefined value while the later will return `false`.

le(x, y)

Return `true` if `x` has a value lower than or equal to `y` or `false` otherwise. It uses the same comparison algorithm than function *cmp(x, y)*.

Listing 58: Cottle template

```
{le(3, 7)}
{le(2, 2)}
{le("abc", "abx")}
```

Listing 59: Rendering output

```
true
true
true
```

Note: This function is equivalent to operator <=.

lt(x, y)

Return `true` if `x` has a value lower than `y` or `false` otherwise. It uses the same comparison algorithm than function *cmp(x, y)*.

Listing 60: Cottle template

```
{lt(3, 7)}
{lt(2, 2)}
{lt("abc", "abx")}
```

Listing 61: Rendering output

```
true
false
true
```

Note: This function is equivalent to operator <.

ne(x, y)

Return `true` if `x` equals `y` or `false` otherwise. It uses the same comparison algorithm than function *cmp(x, y)*.

Listing 62: Cottle template

```
{ne(7, 7)}
{ne(1, 4)}
{ne("test", "test")}
```

Listing 63: Rendering output

```
false true false
```

Note: This function is equivalent to operator `!=` when used with 2 arguments.

not(x)

Perform logical "not" on given boolean value, i.e return `false` if value was equivalent to `true` (see *Value* type for details about conversion to boolean) or `false` otherwise.

Listing 64: Cottle template

```
{not(1 = 2)}
```

Listing 65: Rendering output

```
true
```

Note: This function is equivalent to operator `!`.

or(x, y, ...)

Perform logical "or" between given boolean values, i.e. return `true` if at least one argument is equivalent to `true` (see *Value* type for details about conversion to boolean).

Listing 66: Cottle template

```
{or(2 = 3, 5 > 1)}
```

Listing 67: Rendering output

```
true
```

Note: This function is equivalent to operator `||`.

xor(x, y, ...)

Perform logical "xor" between given boolean values, i.e. return `true` if exactly one argument is `true` and all the others are `false`.

Listing 68: Cottle template

```
{xor(2 < 3, 1 = 2)}
```

Listing 69: Rendering output

```
true
```

when(condition[, truthy[, falsy]])

Return `truthy` if `condition` is equivalent to `true` (see *Value* type for details about conversion to boolean) or `falsy` otherwise (or an undefined value if `falsy` is missing). This function is intended to act as the ternary operator you can find in some programming languages.

Listing 70: Cottle template

```
{set x to 3}
{set y to 0}
{when(x, "x is true", "x is false")}
{when(y, "y is true", "y is false")}
```

Listing 71: Rendering output

```
x is true
y is false
```

1.3.2 Mathematical

abs(x)

Return the absolute value of given numeric value `x`.

Listing 72: Cottle template

```
{abs(-3)}
{abs(5)}
```

Listing 73: Rendering output

```
3
5
```

add(x, y)

Return the sum of two numeric values.

Listing 74: Cottle template

```
{add(3, 7)}
```

Listing 75: Rendering output

```
10
```

Note: This function is equivalent to operator +.

ceil(x)

Returns the smallest integer greater than or equal to number value x.

Listing 76: Cottle template

```
{ceil(2.7)}
```

Listing 77: Rendering output

```
3
```

cos(x)

Get the cosine of angle x in radians.

Listing 78: Cottle template

```
{cos(-1.57)}
```

Listing 79: Rendering output

```
0.000796326710733263
```

div(x, y)

Return the numeric value of x divided by the numeric value of y, or an undefined value if y was equal to zero.

Listing 80: Cottle template

```
{div(5, 2)}
```

Listing 81: Rendering output

```
2.5
```

Note: This function is equivalent to operator /.

floor(x)

Returns the largest integer less than or equal to number value `x`.

Listing 82: Cottle template

```
{floor(2.7)}
```

Listing 83: Rendering output

```
2
```

max(x[, y[, z, ...]])

Return the highest numeric value among given ones.

Listing 84: Cottle template

```
{max(7, 5)}
{max(6, 8, 5, 7, 1, 2)}
```

Listing 85: Rendering output

```
7
8
```

Note: Combine with function *call(func, map)* if you want to get the highest numeric value from an array.

min(x[, y[, z, ...]])

Return the lowest numeric value among given ones.

Listing 86: Cottle template

```
{min(9, 3)}
{min(6, 8, 5, 7, 1, 2)}
```

Listing 87: Rendering output

```
3
1
```

Note: Combine with function *call(func, map)* if you want to get the lowest numeric value from an array.

mod(x, y)

Return the value of `x` modulo `y`, or an undefined value if `y` was equal to zero.

Listing 88: Cottle template

```
{mod(7, 3)}
```

Listing 89: Rendering output

```
1
```

Note: This function is equivalent to operator `%`.

mul(x, y)

Return the numeric value of `x` times `y`.

Listing 90: Cottle template

```
{mul(3, 4)}
```

Listing 91: Rendering output

```
12
```

Note: This function is equivalent to operator `*`.

pow(x, y)

Get specified number `x` raised to the power `y`.

Listing 92: Cottle template

```
{pow(2, 10)}
```

Listing 93: Rendering output

```
1024
```

rand([a[, b]])

Get a pseudo-random numeric value between 0 and 2.147.483.647 inclusive. If numeric `a` value is specified, return a pseudo-random numeric value between 0 and `a` exclusive. If both numeric values `a` and `b` are specified, return a pseudo-random numeric value between `a` inclusive and `b` exclusive.

Listing 94: Cottle template

```
{rand()}
{rand(1, 7)}
```

Listing 95: Rendering output

```
542180393
5
```

round(x[, digits])

Rounds number value `x` to a specified number of fractional digits `digits`, or to the nearest integral value if `digits` is not specified.

Listing 96: Cottle template

```
{round(1.57)}
{round(1.57, 1)}
```

Listing 97: Rendering output

```
2
1.6
```

sin(x)

Get the sine of angle `x` in radians.

Listing 98: Cottle template

```
{sin(1.57)}
```

Listing 99: Rendering output

```
0.999999682931835
```

sub(x, y)

Return the numeric value of `x` minus `y`.

Listing 100: Cottle template

```
{sub(3, 5)}
```

Listing 101: Rendering output

```
-2
```

Note: This function is equivalent to operator −.

1.3.3 Collection

cat(a, b, . . .)

Concatenate all input maps or strings into a single one. Keys are **not** preserved when this function used on map values.

Listing 102: Cottle template

```
{dump cat("Hello, ", "World!")}
{dump cat([1, 2], [3])}
```

Listing 103: Rendering output

```
"Hello, World!"
[1, 2, 3]
```

Warning: All arguments must share the same type than first one, either map or string.

cross(map1, map2, . . .)

Return a map containing all pairs from `map1` having a key that also exists in `map2` and all following maps. Output pair values will always be taken from `map1`.

Listing 104: Cottle template

```
{dump cross([1: "a", 2: "b", 3: "c"], [1: "x", 3: "y"])}
```

Listing 105: Rendering output

```
[1: "a", 3: "c"]
```

except(map1, map2, . . .)

Return a map containing all pairs from `map1` having a key that does not exist in `map2` and any of following maps. This function can also be used to remove a single pair from a map (if you are sure that it's key is not used by any other pair, otherwise all pairs using that key would be removed also).

Listing 106: Cottle template

```
{dump except([1: "a", 2: "b", 3: "c"], [2: "x", 4: "y"])}
```

Listing 107: Rendering output

```
[1: "a", 3: "c"]
```

find(subject, search[, start])

Find index of given `search` value in a map or sub-string in a string. Returns 0-based index of match if found or -1 otherwise. Search starts at index 0 unless `start` argument is specified.

Listing 108: Cottle template

```
{find([89, 3, 572, 35, 7], 35)}
{find("hello, world!", "o", 5)}
{find("abc", "d")}
```

Listing 109: Rendering output

```
3
8
-1
```

filter(map, predicate[, a, b, . . .])

Return a map containing all pairs having a value that satisfies given predicate. Function `predicate` is invoked for each value from `map` with this value as its first argument, and pair is added to output map if predicate result is equivalent to `true` (see *Value* type for details about conversion to boolean).

Optional arguments can be specified when calling `filter` and will be passed to each invocation of `predicate` as second, third, forth argument and so on.

Listing 110: Cottle template

```
{dump filter(["a", "", "b", "", "c"], len)}

{declare multiple_of(x, y) as:
    {return x % y = 0}
}

{dump filter([1, 6, 7, 4, 9, 5, 0], multiple_of, 3)}
```

Listing 111: Rendering output

```
["a", "b", "c"]
[6, 9, 0]
```

flip(map)

Return a map were pairs are created by swapping each key and value pair from input map. Using resulting map with the `for` command will still iterate through each pair even if there was duplicates, but only the last occurrence of each duplicate can be accessed by key.

Listing 112: Cottle template

```
{dump flip([1: "hello,", 2: "world!"])}
{dump flip(["a": 0, "b": 0])}
```

Listing 113: Rendering output

```
["hello,": 1, "world!": 2]
["a", 0: "b"]
```

join(map[, string])

Concatenate all values from given map pairs, using given string as a separator (or empty string if no separator is provided).

Listing 114: Cottle template

```
{join(["2011", "01", "01"], "/")}
```

Listing 115: Rendering output

```
2011/01/01
```

len(x)

Return number of elements in given value, which means the number of pairs for a map or the number of character for a string.

Listing 116: Cottle template

```
{len("Hello!")}
{len([17, 22, 391, 44])}
```

Listing 117: Rendering output

```
6
4
```

map(source, modifier[, a, b, . . .])

Return a map where values are built by applying given modifier to map values, while preserving keys. Function `modifier` is invoked for each value in `source` with this value as its first argument.

Optional arguments can be specified when calling `map` and will be passed to each invocation of `modifier` as second, third, forth argument and so on.

Listing 118: Cottle template

```
{declare square(x) as:
    {return x * x}
}

{dump map([1, 2, 3, 4], square)}
{dump map(["a": 1, "b": 7, "c": 4, "d": 5, "e": 3, "f": 2, "g": 6], lt, 4)}
```

Listing 119: Rendering output

```
[1, 4, 9, 16]
["a": 1, "b": 0, "c": 0, "d": 0, "e": 1, "f": 1, "g": 0]
```

range([start,]stop[, step])

Generate a map where value of the *i*-th pair is *start* + *step* * *i* and last value is lower (or higher if `step` is a negative integer) than `stop`. Default base index is 0 if the `start` argument is omitted, and default value for `step` is 1 if `start` < `stop` or -1 otherwise.

Listing 120: Cottle template

```
{for v in range(5): {v}}
{for v in range(2, 20, 3): {v}}
```

Listing 121: Rendering output

```
0 1 2 3 4
2 5 8 11 14 17
```

slice(subject, index[, count])

Extact sub-string from a string or elements from a map (keys are not preserved when used with maps). `count` items or characters are extracted from given 0-based numeric `index`. If no `count` argument is specified, all elements starting from given `index` are extracted.

Listing 122: Cottle template

```
{for v in slice([68, 657, 54, 3, 12, 9], 3, 2): {v}}
{slice("abchello", 4)}
```

Listing 123: Rendering output

```
3 12
hello
```

sort(map[, callback])

Return a sorted copy of given map. First argument is the input map, and will be sorted using natural order (numerical or alphabetical, depending on value types) by default. You can specify a second argument as comparison delegate, that should accept two arguments and return -1 if the first should be placed "before" the second, 0 if they are equal, or 1 otherwise.

Listing 124: Cottle template

```
{set shuffled to ["in", "order", "elements" "natural"]}
{for item in sort(shuffled):
    {item}
}
```

(continues on next page)

(continued from previous page)

```
{declare by_length(a, b) as:
    {return cmp(len(b), len(a))}
}
{set shuffled to ["by their", "are sorted", "length", "these strings"]}
{for item in sort(shuffled, by_length):
    {item}
}
```

Listing 125: Rendering output

```
elements in natural order
these strings are sorted by their length
```

union(map1, map2, . . .)

Return a map containing all pairs from input maps, but without duplicating any key. If a key exists more than once in all input maps, the last one will overwrite any previous pair using it.

Listing 126: Cottle template

```
{dump union([1: "a", 2: "b"], [2: "x", 3: "c"], [4: "d"])}
```

Listing 127: Rendering output

```
[1: "a", 2: "x", 3: "c", 4: "d"]
```

zip(k, v)

Combine given maps of same length to create a new one. The n-th pair in result map will use the n-th value from `k` as its key and the n-th value from `v` as its value.

Listing 128: Cottle template

```
{set k to ["key1", "key2", "key3"]}
{set v to ["value1", "value2", "value3"]}
{dump zip(k, v)}
```

Listing 129: Rendering output

```
["key1": "value1", "key2": "value2", "key3": "value3"]
```

1.3.4 Text

char(codepoint)

Get a 1-character string from its Unicode code point integer value. See more about Unicode and code points on Wikipedia.

Listing 130: Cottle template

```
{char(97)}
{char(916)}
```

Listing 131: Rendering output

```
a
Δ
```

format(value, format[, culture])

Convert any `value` to a string using given formatting from `format` string expression. Format should use syntax `str` or `t:str` where `t` indicates the type of the formatter to use and `str` is the associated .NET format string. Available formatter types are:

- `a`: automatic (default, used if `t` is omitted)
- `b`: System.Boolean
- `d` or `du`: System.DateTime (UTC)
- `dl`: System.DateTime (local)
- `i`: System.Int64
- `n`: System.Double
- `s`: System.String

Format string depends on the type of formatter selected, see help about Format String Component for more information about formats.

Listing 132: Cottle template

```
{format(1339936496, "d:yyyy-MM-dd HH:mm:ss")}
{format(0.165, "n:p2", "fr-FR")}
{format(1, "b:n2")}
```

Listing 133: Rendering output

```
2012-06-17 12:34:56
16,50 %
True
```

Formatters use current culture, unless a culture name is specified in the `culture` argument. See documentation of CultureInfo.GetCultureInfo method to read more about culture names.

lcase(string)

Return a lowercase conversion of given string value.

Listing 134: Cottle template

```
{lcase("Mixed Case String"}
```

Listing 135: Rendering output

```
mixed case string
```

match(subject, pattern)

Match `subject` against given regular expression pattern. If match is successful, a map containing full match followed by captured groups is returned, otherwise result is an undefined value. See .NET Framework Regular Expressions for more information.

Listing 136: Cottle template

```
{dump match("abc123", "^[a-z]+([0-9]+)$")}
{dump match("xyz", "^[a-z]+([0-9]+)$")}
```

Listing 137: Rendering output

```
["abc123", "123"]
<void>
```

ord(character)

Get the Unicode code point value of the first character of given string. See more about Unicode and code points on Wikipedia.

Listing 138: Cottle template

```
{ord("a")}
{ord("Δ")}
```

Listing 139: Rendering output

```
97
916
```

split(subject, separator)

Split `subject` string according to given string separator `separator`. Result is an map where pair values contain split sub-strings.

Listing 140: Cottle template

```
{dump split("2011/01/01", "/")}
```

Listing 141: Rendering output

```
["2011", "01", "01"]
```

token(subject, search, index[, replace])

Either return the n-th section of a string delimited by separator substring `search` if no `replace` argument is provided, or replace this section by `replace` else. This function can be used as a faster alternative to combined split/slice/join calls in some cases.

Listing 142: Cottle template

```
{token("First.Second.Third", ".", 1)}
{token("A//B//C//D", "//", 2)}
{token("XX-??-ZZ", "-", 1, "YY")}
{token("1;2;3", ";", 3, "4")}
```

Listing 143: Rendering output

```
Second
C
XX-YY-ZZ
1;2;3;4
```

ucase(string)

Return an uppercase conversion of given string value.

Listing 144: Cottle template

```
{ucase("Mixed Case String"}
```

Listing 145: Rendering output

```
MIXED CASE STRING
```

1.3.5 Type

cast(value, type)

Get value converted to requested scalar type. Type must be a string value specifying desired type:

- `"b"` or `"boolean"`: convert to boolean value
- `"n"` or `"number"`: convert to numeric value
- `"s"` or `"string"`: convert to string value

Listing 146: Cottle template

```
{dump cast("2", "n") = 2}
{dump ["value for key 0"][cast("0", "n")]}
{dump cast("some string", "b")}
```

Listing 147: Rendering output

```
<true>
"value for key 0"
<true>
```

type(value)

Retrieve type of given value as a string. Possible return values are `"boolean"`, `"function"`, `"map"`, `"number"`, `"string"` or `"void"`.

Listing 148: Cottle template

```
{type(15)}
{type("test")}
```

Listing 149: Rendering output

```
number
string
```

1.3.6 Dynamic

call(func, map)

Call function `func` with values from `map` as arguments (keys are ignored).

Listing 150: Cottle template

```
{call(cat, ["Hello", ", ", "World", "!"])}
{call(max, [3, 8, 2, 7])}
```

Listing 151: Rendering output

```
Hello, World!
8
```

1.4 Compiler configuration

1.4.1 Specifying configuration

You can specify configuration parameters by passing a `Cottle.DocumentConfiguration` instance when creating a new document. Here is how to specify configuration parameters:

Listing 152: C# source

```
void RenderAndPrintTemplate()
{
    var configuration = new DocumentConfiguration
    {
        NoOptimize = true
    };

    var template = "This is my input template file";

    var documentResult = Document.CreateDefault(template, configuration);

    // TODO: render document
}
```

Options can be set by assigning a value to optional fields of structure `Cottle.DocumentConfiguration`, as described below. Any undefined field will keep its default value.

1.4.2 Plain text trimming

Cottle's default behavior when rendering plain text is to output it without any modification. While this gives you a perfect character-level control of how a template is rendered, it may prevent you from writing clean indented code for target formats where whitespaces are not meaningful, such as HTML or JSON.

For this reason you can change the way plain text is transformed through the use of text *trimmers*. A text trimmer is a simple `Func<string, string>` function that takes a plain text value and returns it as it should be written to output. Some default trimmer functions are provided by Cottle, but you can inject any custom function you need as well.

TrimEnclosingWhitespaces

`DocumentConfiguration.TrimEnclosingWhitespaces` removes all leading and trailing blank characters from plain text blocks. You may need to use expression `{" "}` to force insertion of whitespaces between blocks:

Listing 153: Cottle template

```
{'white'}    {'spaces '} around plain    text    blocks will be coll    {'apsed'} .
```

Listing 154: C# source

```
var configuration = new DocumentConfiguration
{
    Trimmer = DocumentConfiguration.TrimEnclosingWhitespaces
};
```

Listing 155: Rendering output

```
whitespaces around plain    text    blocks will be collapsed.
```

TrimFirstAndLastBlankLines

`DocumentConfiguration.TrimFirstAndLastBlankLines` removes end of line followed by blank characters at beginning and end of plain text blocks. You may have to introduce two line breaks instead of one when interleaving plain texts and commands so one of them is preserved:

Listing 156: Cottle template

```
You have {len(messages)}
{if len(messages) > 1:
    s
}

in your inbox
```

Listing 157: C# source

```
var configuration = new DocumentConfiguration
{
    Trimmer = DocumentConfiguration.TrimFirstAndLastBlankLines
};
```

Listing 158: Rendering output

```
You have 4 messages
in your inbox
```

Note: This trimmer is used by default when no configuration is specified.

TrimNothing

`DocumentConfiguration.TrimNothing` doesn't changing anything on plain text blocks:

Listing 159: Cottle template

```
{'no'} change {'will'} be applied
{'on'} plain {'text'} blocks.
```

Listing 160: C# source

```
var configuration = new DocumentConfiguration
{
    Trimmer = DocumentConfiguration.TrimNothing
};
```

Listing 161: Rendering output

```
no change will be applied
on plain text blocks.
```

TrimRepeatedWhitespaces

`DocumentConfiguration.TrimRepeatedWhitespaces` replaces all sequences of white characters (spaces, line breaks, etc.) by a single space, similar to what HTML or XML languages do:

Listing 162: Cottle template

```
<ul>    {for s in ["First", "Second", "Third"]:    <li>    {s} </li>    } </ul>
```

Listing 163: C# source

```
var configuration = new DocumentConfiguration
{
    Trimmer = DocumentConfiguration.TrimRepeatedWhitespaces
};
```

Listing 164: Rendering output

```
<ul>  <li> First </li>  <li> Second </li>  <li> Third </li>  </ul>
```

1.4.3 Delimiters customization

Default Cottle configuration uses **{** character as *start of command* delimiter, **|** as *continue* delimiter and **}** as *end of command* delimiter. These characters may not be a good choice if you want to write a template that would often use them in plain text context, for example if you're writing a JavaScript template, because you would have to escape every **{**, **}** and **|** to avoid Cottle seeing them as delimiters.

A good solution to this problem is changing default delimiters to replace them by more convenient sequences for your needs. Any string can be used as a delimiter as long as it doesn't conflict with a valid Cottle expression (e.g. **[**, **+** or **<**). Make sure at least the first character of your custom delimiters won't cause any ambiguity when choosing them, as the compilation error messages you may have would be confusing.

Default escape delimiter **** can be replaced in a similar way, however it must be a single-character value.

Listing 165: Cottle template

```
Delimiters are {{block_begin}}, {{block_continue}} and {{block_end}}.
Backslash \ is not an escape character.
```

Listing 166: C# source

```
    var configuration = new DocumentConfiguration
    {
        BlockBegin = "{{",
        BlockContinue = "{|}",
        BlockEnd = "}}",
        Escape = '\0'
    };
```

(continues on next page)

(continued from previous page)

```
    var context = Context.CreateBuiltin(new Dictionary<Value, Value>
    {
        ["block_begin"] = "double left brace (" + configuration.BlockBegin + ")"
        ["block_continue"] = "brace pipe brace (" + configuration.BlockContinue + ")",
        ["block_end"] = "double right brace (" + configuration.BlockEnd + ")"
    });
```

Listing 167: Rendering output

```
Delimiters are double left brace ({{), brace pipe brace ({|}) and double right brace␣
↪(}}).
Backslash \ is not an escape character.
```

1.4.4 Optimizer deactivation

Cottle performs various code optimizations on documents after parsing them from a template to achieve better rendering performance. These optimizations have an additional cost at compilation, which you may not want to pay if you're frequently re-building document instances (which is something you should avoid if possible):

Listing 168: C# source

```
var configuration = new DocumentConfiguration
{
    NoOptimize = true
};
```

Warning: Disabling optimizations is not recommended for production usage.

1.5 Advanced features

1.5.1 Understanding value types

Every value has a type in Cottle, even if you usually don't have to worry about it (see *Value* type for details). Functions that expect arguments of specific types will try to cast them silently and fallback to undefined values when they can't. However in some rare cases you may have to force a cast yourself to get desired result, for example when accessing values from a map:

Listing 169: Cottle template

```
{set map to ["first", "second", "third"]}
{set key to "1"}
{dump map[key]}
```

Listing 170: Rendering output

```
<void>
```

You could have expected this template to display "second", but Cottle actually searches for the map value associated to key `"1"` (as a string), not key `1` (as a number). These are two different values and storing two different values for keys `"1"` and `1` in a map is valid, hence no automatic cast can be performed for you.

In this example, you can explicitly change the type of `key` to a number by using built-in function *cast(value, type)*. Also remember you can use the *Debug: dump* command to troubleshoot variable types in your templates.

1.5.2 Variables immutability

All variables in Cottle are immutable, meaning it's not possible to replace a section within a string or change the value associated to some key in a map. If you want to append, replace or erase a value in a map you'll have to rebuild a new one where you inject, filter out or replace desired value. There are a few built-in functions you may find handly to achieve such tasks:

- *cat(a, b, ...)* and *union(map1, map2, ...)* can merge strings (`cat` only) or maps ;
- *slice(subject, index[, count])* can extract part of a string or a map ;
- *except(map1, map2, ...)* can extract the intersection between two maps.

Here are a few examples about how to use them:

Listing 171: Cottle template

```
{set my_string to "Modify me if you can"}
{set my_string to cat("I", slice(my_string, 16), ".")}
{dump my_string}

{set my_array to [4, 8, 50, 90, 23, 42]}
{set my_array to cat(slice(my_array, 0, 2), slice(my_array, 4))}
{set my_array to cat(slice(my_array, 0, 2), [15, 16], slice(my_array, 2))}
{dump my_array}

{set my_hash to ["delete_me": "TODO: delete this value", "let_me": "I shouldn't be␣
↪touched"]}
{set my_hash to union(my_hash, ["append_me": "I'm here!"])}
{set my_hash to except(my_hash, ["delete_me": 0])}
{dump my_hash}
```

Listing 172: Rendering output

```
"I can."
[4, 8, 15, 16, 23, 42]
["let_me": "I shouldn't be touched", "append_me": "I'm here!"]
```

1.5.3 Function declaration

Cottle allows you to declare functions directly in your template code so you can reuse code as you would do with any other programming language. To declare a function and assign it to a variable, use the same `set` command you used for regular values assignments (see section *Assignments: set*) with a slightly different syntax. Function arguments must

be specified between parenthesis right after the variable name that should receive the function, and the `to` keyword must be followed by a ":" (semicolon) character, then function body declaration as a Cottle template.

Functions can return a value that can be used in any expression or stored in a variable. To make a function halt and return a value, use the `return` command within its body:

Listing 173: Cottle template

```
{set factorial(n) to:
    {if n > 1:
        {return n * factorial(n - 1)}
    |else:
        {return 1}
    }
}

Factorial 1 = {factorial(1)}
Factorial 3 = {factorial(3)}
Factorial 8 = {factorial(8)}

{set hanoi_recursive(n, from, by, to) to:
    {if n > 0:
        {hanoi_recursive(n - 1, from, to, by)}
        Move one disk from {from} to {to}
        {hanoi_recursive(n - 1, by, from, to)}
    }
}

{set hanoi(n) to:
    {hanoi_recursive(n, "A", "B", "C")}
}

{hanoi(3)}
```

Listing 174: Rendering output

```
Factorial 1 = 1
Factorial 3 = 6
Factorial 8 = 40320

Move one disk from A to C
Move one disk from A to B
Move one disk from C to B
Move one disk from A to C
Move one disk from B to A
Move one disk from B to C
Move one disk from A to C
```

You can see in this example that returning a value and printing text are two very different things. Plain text within function body is printed each time the function is called, or more precisely each time its enclosing block is executed (that means it won't print if contained in an `if` command that fails to pass, for example).

The value returned by the function won't be printed unless you explicitly require it by using the `echo` command (e.g. something like `{factorial(8)}`). If a function doesn't use any `return` command it returns an undefined value, that's why the call to `{hanoi(3)}` in the sample above does not print anything more than the plain text blocks it contains.

1.5.4 Variable scope

When writing complex templates using nested or recursive functions, you may have to take care of variable scopes to avoid potential issues. A scope is the local evaluation context of any function or command having a body. When assigning a value to a variable (see section *Assignments: set* for details) all variables belong to the same global scope. Consider this template:

Listing 175: Cottle template

```
{set depth(item) to:
    {set res to 0}

    {for child in item:
        {set res_child to depth(child) + 1}
        {set res to max(res, res_child)}
    }

    {return res}
}

{depth([["1.1", "1.2", ["1.3.1", "1.3.2"]], "2", "3", ["4.1", "4.2"]])}
```

The `depth` function is expected to return the level of the deepest element in a value that contains nested maps. Of course it could be written in a more efficient way without using non-necessary temporary variables, but it would hide the problem we want to illustrate. If you try to execute this code you'll notice it returns `2` where `3` would have been expected.

Here is the explanation: when using the `set` method to assign a value to variable `res` it always uses the same `res` instance. The `depth` function recursively calls itself but overwrite the unique `res` variable each time it tries to store a value in it, and therefore fails to store the actual deepest level as it should.

To solve this issue, the `res` variable needs to be local to function `depth` so that each invocation uses its own `res` instance. This can be achieved by using the `declare` command that creates a variable in current scope. Our previous example can then be fixed by declaring a new `res` variable inside body of function `depth`, so that every subsequent reference to `res` resolves to our local instance:

Listing 176: Cottle template

```
{set depth(item) to:
    {declare res}
    {set res to 0}

    {for child in item:
        {set res_child to depth(child) + 1}
        {set res to max(res, res_child)}
    }

    {return res}
}

{depth([["1.1", "1.2", ["1.3.1", "1.3.2"]], "2", "3", ["4.1", "4.2"]])}
```

You could even optimize the first `set` command away by assigning a value to `res` during declaration ; the `declare` command actually supports the exact same syntax than `set`, the only difference being than "to" should be replaced by "as":

Listing 177: Cottle template

```
{declare res as 0}
```

The same command can also be used to declare functions:

Listing 178: Cottle template

```
{declare square(n) as:
    {return n * n}
}
```

Note that the `set` command can also be used without argument, and assigns variable an undefined value (which is equivalent to reset it to an undefined state).

1.5.5 Native .NET functions

If you need new features or improved performance, you can assign your own .NET methods to template variables so they're available as Cottle functions. That's actually what Cottle does when you use *`Context.CreateBuiltin`* method: a set of Cottle methods is added to your context, and you can have a look at the source code to see how these methods work.

To pass a function in a context, use one of the methods from *`Function`* class, then pass it to *`Value.FromFunction`* method to wrap it into a value you can add to a context:

Listing 179: Cottle template

```
Testing custom "repeat" function:

{repeat("a", 15)}
{repeat("oh", 7)}
{repeat("!", 10)}
```

Listing 180: C# source

```
var context = Context.CreateBuiltin(new Dictionary<Value, Value>
{
    ["repeat"] = Value.CreateFunction(Function.CreatePure2((state, subject, count) =>
    {
        var builder = new StringBuilder();

        for (var i = 0; i < count; ++i)
            builder.Append(subject);

        return builder.ToString();
    }))
});
```

Listing 181: Rendering output

```
Testing custom "repeat" function:

aaaaaaaaaaaaaaa
ohohohohohohoh
!!!!!!!!!!
```

Static class *Function* supports multiple methods to create Cottle functions. Each method expects a .NET callback that contains the code to be executed when the method is invoked, and some of them also ask for the accepted number of parameters for the function being defined. Methods from *Function* are defined across a combination of 2 criteria:

- **Whether they're having side effects or not:**
  - Methods *Function.CreatePure*, *Function.CreatePure1* and *Function.CreatePure2* must be pure functions having no side effect and not relying on anything but their arguments. This assumption is used by Cottle to perform optimizations in your templates. For this reason their callbacks don't receive a `TextWriter` argument as pure methods are not allowed to write anything to output.
  - Methods *Function.Create*, *Function.Create1* and *Function.Create2* are allowed to perform side effects but will be excluded from most optimizations. Their callbacks receive a `TextWriter` argument so they can write any text contents to it.
- **How many arguments they accept:**
  - Methods *Function.Create* and *Function.CreatePure* with no integer argument accept any number of arguments, it is the responsibility of provided callback to validate this number.
  - Methods *Function.Create* and *Function.CreatePure* with a `count` integer accept exactly this number of arguments or return an undefined value otherwise.
  - Methods *Function.Create* and *Function.CreatePure* with two `min` and `max` integers accept a number of arguments contained between these two values or return an undefined value otherwise.
  - Methods `Function.CreateN` and `Function.CreatePureN` only accept exactly `N` arguments or return an undefined value otherwise.

The callback you'll pass to *Function* takes multiple arguments:

- First argument is always an internal state that must be forwarded to any nested function call ;
- Next arguments are either a list of values (for functions accepting variable number of arguments) or separate scalar values (for functions accepting a fixed number of arguments) received as arguments when invoking the function ;
- Last argument, for non-pure functions only, is a `TextWriter` instance open to current document output.

1.5.6 Lazy value evaluation

In some cases, you may want to inject to your template big and/or complex values that may or may not be needed at rendering, depending on other parameters. In such configurations, it may be better to avoid injecting the entire value in your context if there is chances it won't be used, and use lazy evaluation instead.

Lazy evaluation allows you to inject a value with a resolver callback which will be called only the first time value is accessed, or not called at all if value is not used for rendering. Lazy values can be created through implicit conversion from any `Func<Value>` instance or by using *Value.FromLazy* construction method:

Listing 182: Cottle template

```
{if is_admin:
    Administration log: {log}
}
```

Listing 183: C# source

```
var context = Context.CreateBuiltin(new Dictionary<Value, Value>
{
    ["is_admin"] = user.IsAdmin,
    ["log"] = () => log.BuildComplexLogValue() // Implicit conversion to lazy value
});

document.Render(context, Console.Out);
```

In this example, method `log.BuildComplexLogValue` won't be called unless `is_admin` value is true.

1.5.7 Reflection values

Instead of converting complex object hierarchies to Cottle values, you can have the library do it for you by using .NET reflection. Keep in mind that reflection is significantly slower than creating Cottle values manually, but as it's a lazy mechanism it may be a good choice if you have complex objects and don't know in advance which fields might be used in your templates.

To use reflection, invoke *`Value.FromReflection`* method on any .NET object instance and specify binding flags to indicate which members should be made visible to Cottle. Fields and properties resolved on the object will be accessible like if it were a Cottle map:

Listing 184: Cottle template

```
Your image has a size of {image.Width}x{image.Height} pixels.

{for key, value in image:
    {if value:
        {key} = {value}
    }
}
```

Listing 185: C# source

```
var context = Context.CreateBuiltin(new Dictionary<Value, Value>
{
    ["image"] = Value.FromReflection(new Bitmap(50, 50), BindingFlags.Instance |␣
↪BindingFlags.Public)
});
```

Listing 186: Rendering output

```
Your image has a size of 50x50 pixels.

Width = 50
Height = 50
HorizontalResolution = 96
VerticalResolution = 96
Flags = 2
```

Warning: Relying on reflection has a significant impact on execution performance. Use this feature only if performance is not important for your application, or you don't have other option like explicitly converting fields

and properties to a Cottle value.

1.5.8 Native documents

You can use "native" documents instead of default ones to achieve better rendering performance at a higher compilation cost. Native documents rely on IL code generation instead of runtime evaluation, and can provide a rendering performance boost from 10% to 20% depending on templates and environment (see benchmark). They're however two to three times most costly to build, so this feature should be used only when you need high rendering performances on long-lived documents.

To create native documents, simply invoke *Document.CreateNative* instead of default method:

Listing 187: C# source

```
var document = Document.CreateNative(template).DocumentOrThrow;
```

1.6 API reference

1.6.1 Public API definition

This page contains information about types that are part of Cottle's public API.

Warning: Types not listed in this page should not be considered as part of the public API, and are not taken into consideration when changing version number (see section *Versioning convention*).

Warning: You should avoid relying on method behaviors not documented in this page as they could change from one version to another.

1.6.2 Compiled documents

class IDocument

A document in Cottle is a compiled template, which means a template converted to an optimized in-memory representation.

Value **Render** (*IContext context*, TextWriter *writer*)

Render document and write output to given `TextWriter` instance. Return value is the value passed to top-level `return` command if any, or an undefined value otherwise.

String **Render** (*IContext context*)

Render document and return outout as a `string` instance.

class Document

Methods from this static class must be used to create instances of *DocumentResult*.

DocumentResult **CreateDefault** (TextReader *template*, *DocumentConfiguration configuration = default*)

Create a new default *IDocument* instance suitable for most use cases. Template is read from any non-seekable `TextReader` instance.

DocumentResult **CreateDefault** (String *template*, *DocumentConfiguration* *configuration* = *default*)
Create a new default `IDocument` instance similar to previous method. Template is read from given `string` instance.

DocumentResult **CreateNative** (TextReader *template*, *DocumentConfiguration* *configuration* = *default*)
Create a new native `IDocument` instance for better rendering performance but higher compilation cost. Template is read from any non-seekable `TextReader` instance. See section *Native documents* for details about native documents.

DocumentResult **CreateNative** (String *template*, *DocumentConfiguration* *configuration* = *default*)
Create a new native `IDocument` instance similar to previous method. Template is read from given `string` instance.

class DynamicDocument

: *Cottle.IDocument*

Deprecated class, use `Cottle.Document.CreateNative` to create native documents.

class SimpleDocument

: *Cottle.IDocument*

Deprecated class, use `Cottle.Document.CreateDefault` to create documents.

class DocumentConfiguration
Document configuration options, can be passed as an optional argument when creating a new document.

String **BlockBegin** { **get; set;** }
Delimiter for *start of command*, see section *Delimiters customization* for details.

String **BlockContinue** { **get; set;** }
Delimiter for *continue command*, see section *Delimiters customization* for details.

String **BlockEnd** { **get; set;** }
Delimiter for *end of command*, see section *Delimiters customization* for details.

Nullable<char> **Escape** { **get; set;** }
Delimiter for *escape*, see section *Delimiters customization* for details. Default escape character is \ when this property is null.

Boolean **NoOptimize** { **get; set;** }
Disable code optimizations after compiling a document, see *Optimizer deactivation* for details.

Func<String, String> **Trimmer** { **get; set;** }
Function used to trim unwanted character out of plain text when parsing a document, see section *Plain text trimming* for details.

class DocumentResult
This structure holds result of a template compilation, which can either be successful and provide compiled `IDocument` instance or failed and provide compilation error details as a list of `DocumentReport` elements:

IDocument **Document** { **get;** }
Instance of compiled document, only if compilation was successful (see `DocumentResult.Success`).

IReadOnlyList<*DocumentReport*> **Reports** { **get;** }
List of anomalies detected during compilation, as a read-only list of `DocumentReport` items.

Boolean **Success** { **get;** }
Indicate whether compilation was successful or not.

IDocument **DocumentOrThrow** { **get;** }
Helper to return compiled document when compilation was successful or throw a *Exceptions.ParseException* exception with details about first compilation error otherwise.

class DocumentReport
Anomaly report on compiled template, with references to related code location.

Int32 **Length** { **get;** }
Length of the last lexem recognized when encountering an anomaly.

String **Message** { **get;** }
Human-readable description of the anomaly. This value is meant for being displayed in a user interface but not processed, as its contents is not predictable.

Int32 **Offset** { **get;** }
Offset of the last lexem recognized when encountering an anomaly.

DocumentSeverity **Severity** { **get;** }
Report severity level.

enum DocumentSeverity
Report severity level.

Error
Template issue that prevents document from being constructed.

1.6.3 Rendering contexts

class IContext
This interface is used to pass variables to a document when rendering it.

Value **this**[, *Value symbol*] { **get;** }
Get variable by its symbol (usually its name), or an undefined value if no value was defined with this name.

class Context
Methods from this static class must be used to create instances of *IContext*.

IContext **CreateBuiltin** (*IContext custom*)
Create a rendering context by combining a given existing context with all Cottle built-in functions (see section *Built-in functions*). Variables from the input context always have priority over built-in functions in case of collision.

IContext **CreateBuiltin** (IReadOnlyDictionary<*Value*, *Value*> *symbols*)
Create a rendering context by combining variables from given dictionary with all Cottle built-in functions. This method is similar to previous one and only exists as a convenience helper.

IContext **CreateCascade** (*IContext primary*, *IContext fallback*)
Create a rendering context by combining two existing contexts that will be searched in order when querying a variable. Primary context is searched first, then fallback context is searched second if the result from first one was an undefined value.

IContext **CreateCustom** (Func<*Value*, *Value*> *callback*)
Create a rendering context using given callback for resolving variables. Callback must always expected to return a non-null result, possibly an undefined value.

IContext **CreateCustom** (IReadOnlyDictionary<*Value*, *Value*> *symbols*)
Create a rendering context from given variables dictionary.

(IContext,ISymbolUsage) **CreateMonitor** (*IContext context*)
Wrap given context inside a monitoring context to get statistics on which variables are read from it. Output *IContext* is the monitored one that should be passed to document, while the ISymbolUsage usage is

the structure you can query after rendering the document to get information about which variables were read.

1.6.4 Function declaration

class IFunction

Cottle function interface.

Boolean **IsPure** { **get;** }

Indicates whether function is pure or not. Pure functions have no side effects nor rely on any, and are eligible to various rendering optimizations.

Value **Invoke** (object *state*, IReadOnlyList<*Value*> *arguments*, TextWriter *output*)

Invoke function with given arguments. Variable `state` is a opaque payload that needs to be passed to nested function calls if any, `arguments` contains the ordered list of values passed to function, and `output` is a text writer to document output result.

class Function

Methods from this static class must be used to create instances of *IFunction*.

IFunction **Create** (Func<object, IReadOnlyList<*Value*>, TextWriter, *Value*> *callback*, Int32 *min*, Int32 *max*)

Create a non-pure function accepting between `min` and `max` arguments (included).

IFunction **Create** (Func<object, IReadOnlyList<*Value*>, TextWriter, *Value*> *callback*, Int32 *count*)

Create a non-pure function accepting exactly `count` arguments.

IFunction **Create** (Func<object, IReadOnlyList<*Value*>, TextWriter, *Value*> *callback*)

Create a non-pure function accepting any number of arguments.

IFunction **Create0** (Func<object, TextWriter, *Value*> *callback*)

Create a non-pure function accepting zero argument.

IFunction **Create1** (Func<object, *Value*, TextWriter, *Value*> *callback*)

Create a non-pure function accepting one argument.

IFunction **Create2** (Func<object, *Value*, *Value*, TextWriter, *Value*> *callback*)

Create a non-pure function accepting two arguments.

IFunction **Create3** (Func<object, *Value*, *Value*, *Value*, TextWriter, *Value*> *callback*)

Create a non-pure function accepting three arguments.

IFunction **CreatePure** (Func<object, IReadOnlyList<*Value*>, *Value*> *callback*, Int32 *min*, Int32 *max*)

Create a pure function accepting between `min` and `max` arguments (included).

IFunction **CreatePure** (Func<object, IReadOnlyList<*Value*>, *Value*> *callback*, Int32 *count*)

Create a pure function accepting exactly `count` arguments.

IFunction **CreatePure** (Func<object, IReadOnlyList<*Value*>, *Value*> *callback*)

Create a pure function accepting any number of arguments.

IFunction **CreatePure0** (Func<object, *Value*> *callback*)

Create a pure function accepting zero argument.

IFunction **CreatePure1** (Func<object, *Value*, *Value*> *callback*)

Create a pure function accepting one argument.

IFunction **CreatePure2** (Func<object, *Value*, *Value*, *Value*> *callback*)

Create a pure function accepting two arguments.

IFunction **CreatePure3** (Func<object, *Value*, *Value*, *Value*, *Value*> *callback*)

Create a pure function accepting three arguments.

1.6.5 Value declaration

class Value

Cottle values can hold instances of any of the supported types (see section *Value types*).

Value **EmptyMap** { **get;** }
Static and read-only empty map value, equal to `Value.FromEnumerable(new Value[0])`.

Value **EmptyString** { **get;** }
Static and read-only empty string value, equal to `Value.FromString(string.Empty)`.

Value **False** { **get;** }
Static and read-only boolean "false" value, equal to `Value.FromBoolean(false)`.

Value **True** { **get;** }
Static and read-only boolean "true" value, equal to `Value.FromBoolean(true)`.

Value **Undefined** { **get;** }
Static and read-only undefined value, equal to `new Value()` or `default(Value)`.

Value **Zero** { **get;** }
Static and read-only number "0" value, equal to `Value.FromNumber(0)`.

Boolean **AsBoolean** { **get;** }
Read value as a boolean after converting it if needed. Following conversion is applied depending on base type:

- From numbers, return `true` for non-zero values and `false` otherwise.
- From strings, return `true` for non-zero length values and `false` for empty strings.
- From undefined values, always return `false`.

IFunction **AsFunction** { **get;** }
Read value as a function, only if base type was already a function. No conversion is applied on this property, and return value is undefined if value was not a function.

Double **AsNumber** { **get;** }
Read value as a double precision floating point number after converting it if needed. Following conversion is applied depending on base type:

- From booleans, return `0` for `false` or `1` for `true`.
- From strings, parse double number if value matches regular expression `[0-9]*(\\.[0-9]+)?`, or `0` otherwise.
- From undefined values, always return `0`.

String **AsString** { **get;** }
Read value as a string after converting it if needed. Following conversion is applied depending on base type:

- From booleans, return string `"true"` for `true` and empty string otherwise.
- From numbers, return result of call to `double.ToString()` method with invariant culture.
- From undefined values, always return an empty string.

IMap **Fields** { **get;** }
Get child field of current value if any, or an empty map otherwise.

ValueContent **Type** { **get;** }
Get base type of current value instance.

FromBoolean (Boolean *value*)
Create value from given boolean instance.

FromDictionary (IReadOnlyDictionary<*Value*, *Value*> *dictionary*)
Create a map value from given keys and associated value in given `dictionary`, without preserving any ordering. This override assumes input dictionary is immutable and simply keeps a reference on it without duplicating the data structure.

FromDictionary (IDictionary<*Value*, *Value*> *dictionary*)
Create a map value from given keys and associated value in given `dictionary`, without preserving any ordering. This override assumes input dictionary is mutable and duplicates the data structure to avoid further modifications.

FromEnumerable (IEnumerable<KeyValuePair<*Value*, *Value*>> *pairs*)
Create a map value from given `elements`, preserving element ordering but also allowing O(1) access to values by key.

FromEnumerable (IEnumerable<*Value*> *elements*)
Create a map value from given `elements`. Numeric keys are generated for each element starting at index `0`.

FromFunction (*IFunction function*)
Create a function value by wrapping an executable *`IFunction`* instance. See sections *Function declaration* and *Native .NET functions* for details about functions in Cottle.

FromGenerator (Func<Int32, *Value*> *generator*, Int32 *count*)
Create map value from given generator. Generator function `generator` is used to create elements based on their index, and the map will contain `count` values associated to keys `0` to `count - 1`. Values are created only when retrieved, so creating a generator value with 10000000 elements won't have any cost until you actually access these elements from your template.

FromLazy (Func<*Value*> *resolver*)
Create a lazy value from given value resolver. See section *Lazy value evaluation* for details about lazy value resolution.

FromMap (*IMap value*)
Create value from given *`IMap`* instance.

FromNumber (Double *value*)
Create value from given double instance.

FromReflection<TSource> (TSource *source*, BindingFlags *bindingFlags*)
Create a reflection-based value to read members from object `source`. Source object fields and properties are resolved using Type.GetFields and Type.GetProperties methods and provided binding flags for resolution. See section *Reflection values* for details about reflection-based inspection.

FromString (String *value*)
Create value from given string instance.

class FunctionValue

: *Cottle.Value*

Deprecated class, use *`Value.FromFunction`* to create function values.

FunctionValue (*IFunction function*)

Class constructor.

class LazyValue

: *Cottle.Value*

Deprecated class, use `Value.FromLazy` to create lazy values.

LazyValue (Func<*Value*> *resolver*)

Class constructor.

class ReflectionValue

: *Cottle.Value*

Deprecated class, use `Value.FromReflection` to create reflection values.

ReflectionValue (object *source*, BindingFlags *binding*)

Class constructor with explicit binding flags.

ReflectionValue (object *source*)

Class constructor with default binding flags for resolution (public + private + instance).

class IMap

Value fields container.

Value **this**[, *Value key*] { **get;** }
: Get field by its key (usually its name), or an undefined value if no field was defined with this name.

Int32 **Count** { **get;** }
: Get number of fields contained within this value.

Boolean **Contains** (*Value key*)
: Check whether current map contains a field with given key or not. Returns `true` if map contains requested field or `false` otherwise.

Boolean **TryGet** (*Value key*, *out Value value*)
: Try to read field by key. Returns `true` and set output `Value` instance if found, or return `false` otherwise.

enum ValueContent

Base value type enumeration.

Boolean
: Boolean value, either `true` or `false`.

Function
: Invokable function value.

Map
: Enumerable key/value collection.

Number
: Numeric value, either integer or floating point.

String
: Characters string value.

Void
: Undefined value.

1.6.6 Exceptions

class ParseException

: Exception

Exception class raised when trying to convert an invalid template string into a *IDocument* instance.

String **Lexem** { **get;** }
Lexem (text fragment) that was unexpectedly encountered in template.

Int32 **LocationLength** { **get;** }
Length of the last lexem recognized when encountering a parsing error.

Int32 **LocationStart** { **get;** }
Offset of the last lexem recognized when encountering parsing error.

1.7 Versioning

1.7.1 Versioning convention

Cottle versioning does **NOT** (exactly) follow SemVer convention but uses closely-related version numbers with form `MAJOR.MINOR.PATCH` where:

- `MAJOR` increases when breaking changes are applied and break source compatibility, meaning client code must be changed before it can compile.
- `MINOR` increases when binary compatibility is broken but source compatibility is maintained, meaning client code can be rebuilt with no source change.
- `PATCH` increases when binary compatibility is maintained from previous version, meaning new library version can be used as a drop-in replacement and doesn't require recompiling code.

The main difference between this approach and SemVer is the distinction made between binary compatibility and source compatibility. For example replacing a public field by a property, or doing the opposite, would break strict binary compatibility but wouldn't require any change when recompiling client code unless it's using reflection.

1.7.2 Migration guide

From 1.6.* to 2.0.*

- Cottle now uses Double type for number values instead of Decimal ; use builtin function *format(value, format[, culture])* if you need to control decimal precision when printing decimal numbers.
- Type *Value* is now a value type to reduce runtime allocations ; API was upgraded to be source-compatible with previous Cottle versions.
- Specialized value classes (e.g. *Values.FunctionValue*) are deprecated, use `Value.From*` static construction methods instead (e.g. *Value.FromFunction*).

Listing 188: Example of migration from 1.6.* code to equivalent 2.0.* version

```
// Version 1.6.*
var context = Context.CreateBuiltin(new Dictionary<Value, Value>
{
    ["f"] = new FunctionValue(myFunction),
    ["n"] = new NumberValue(myNumber)
};
```

(continues on next page)

(continued from previous page)

```
// Version 2.0.*
var context = Context.CreateBuiltin(new Dictionary<Value, Value>
{
    ["f"] = Value.FromFunction(myFunction),
    ["n"] = Value.FromNumber(myNumber) // Or just `myNumber` to use implicit␣
↪conversion
};
```

From 1.5.* to 1.6.*

- All documents should be constructed using methods from *Document* static class.
- All contexts should be constructed using methods from *Context* static class.
- All functions should be constructed using methods from *Function* static class.

Listing 189: Example of migration from 1.5.* code to equivalent 1.6.* version

```
// Version 1.5.*
IDocument document;

try
{
    document = new SimpleDocument(template, new CustomSetting
    {
        Trimmer = BuiltinTrimmers.FirstAndLastBlankLines
    });
}
catch (ParseException exception)
{
    MyErrorHandler(exception.Message);

    return string.Empty;
}

return document.Render(new BuiltinStore
{
    ["f"] = new NativeFunction((args, store, output) => MyFunction(args[0].AsNumber,␣
↪output), 1)
});

// Version 1.6.*
var result = Document.CreateDefault(template, new DocumentConfiguration
{
    Trimmer = DocumentConfiguration.TrimIndentCharacters
});

if (!result.Success)
{
    MyErrorHandler(result.Reports);

    return string.Empty;
}

// Can be replaced by result.DocumentOrThrow to factorize test on "Success" field and␣
↪use
```

(continues on next page)

(continued from previous page)

```
// the exception-based API which is closer to what was available in version 1.5.*
var document = result.Document;

return document.Render(Context.CreateBuiltin(new Dictionary<Value, Value>
{
    ["f"] = new FunctionValue(Function.Create1((state, arg, output) => MyFunction(arg.
↪AsNumber, output)))
});
```

From 1.4.* to 1.5.*

- `IStore` replaced by immutable *`IContext`* interface for rendering documents. Since the former extends the later, migration should only imply recompiling without any code change.
- Cottle function delegates now receive a `IReadOnlyList<Value>` instead of their mutable equivalent.
- Method `Save` from `DynamicDocument` can only be used in the .NET Framework version, not the .NET Standard one.

From 1.3.* to 1.4.*

- Change of version number convention, breaking source compatibility must now increase major version number.
- Cottle now requires .NET 4.0 or above.

From 1.2.* to 1.3.*

- Removed deprecated code (flagged as "obsolete" in previous versions).

From 1.1.* to 1.2.*

- `IScope` replaced by similar `IStore` interface (they mostly differ by the return type of their "Set" method which made this impossible to change without breaking the API).
- Callback argument of constructors for `NativeFunction` are not compatible with `IScope` to avoid ambiguous statements.

From 1.0.* to 1.1.*

- `LexerConfig` must be replaced by `CustomSetting` object to change configuration.
- `FieldMap` has been replaced by multiple implementations of the new `IMap` interface.
- Two values with different types are always different, even if casts could have made them equal (i.e. removed automatic casts when comparing values).
- Common functions `cross` and `except` now preserve duplicated keys.

1.8 Credits

1.8.1 Greetings

- Huge thanks to Zerosquare for the lovely project icon!

1.8.2 Contact

Contact me by e-mail: github+cottle [at] mirari [dot] fr

Remi Caput, 2019

CHAPTER 2

Indices and tables

- genindex
- search