# Narrative

Rob Rohan

2022-04-08

**Abstract**

"Narrative is a technique to comment code to produce both the code itself, and documentation / a manual / a tutorial about the code. It is a similar idea to literate programming, but focused more on the code than on producing an academic paper."

## Contents

## 1 Introduction

Narrative is a technique to comment code to produce both the code itself, and at the same time create documentation (a manual, a tutorial, etc) about the code.

It is a similar idea to, and was inspired by, literate programming. Narrative is more focused on guiding someone through the code than on producing an academic paper.

View example output from this codebase (work in progress)

# 2 Narrative

Narrative is a program to help with a minimalist literate programming pipeline. It takes code and “inverts it” - turns the code into examples and the code comments into a human readable document. (Web of Stories - Life Stories of Remarkable People 2016)

## 2.1 Includes

Here we are importing several packages. Since this application mostly just combines files and concatenates strings, almost all of these imports are to support those activities.

The *ardanlabs/conf* include is a library to help make using command line parameters a bit easier. More can be seen on their website.

```
package main

import (
   "fmt"
   "github.com/ardanlabs/conf"
   "github.com/robrohan/narrative/internal"
   "log"
   "os"
)
```

This *build* variable will be overwritten by our build script. This value will be the git hash of the current, build time commit.

```
var build = "develop"
```

## 2.2 Run Wrapper

```
func run(log *log.Logger) error {
   cfg := narrative.Config{}
   if err := conf.Parse(os.Args[1:], "NT", &cfg); err != nil {
       if err == conf.ErrHelpWanted {
           usage, err := conf.Usage("NT", &cfg)
           if err != nil {
               return err
           }
           fmt.Println(usage)
           return nil
       }
       return err
   }

   narrative.ParseNarrative(cfg, log)
   return nil
}
```

### 2.3 Program Main Entry

You've got to start somewhere.

Here we just setup the logger and kick off the main *run* method.

```
func main() {
   log := log.New(os.Stdout, "NT: ", log.LstdFlags|log.Lmicroseconds|log.Lshortfile)

   if err := run(log); err != nil {
       log.Println("error: ", err)
       os.Exit(1)
   }
}
```

## 3 Narrative Package

The narrative package holds most of the code.

### 3.1 Includes

```
package narrative

import (
   "bufio"
   "errors"
   "fmt"
   "gopkg.in/yaml.v2"
   "io"
   "io/ioutil"
   "log"
   "os"
   "path/filepath"
   "strings"
)
```

### 3.2 Config Struct

This structure is used to hold the command line parameters passed when the application was started. Note that only *input* is required, and *input* needs to be a line separated file that has a list of files to concatenate together.

```
type Config struct {
   Input   string `conf:"short:i,default:NARRATIVE"`
   Output  string `conf:"short:o,default:final.md"`
   Markers string `conf:"short:m,default:./narrative.yaml"`
}
```

## 3.3 Comment Markers

In order to handle comments in different file types, we allow for different *Comment Markers*. A comment marker is a way to define an area we will use to look for markdown text. Meaning, it becomes the human readable part of the application.

```
type CommentMarkers struct {
   Markers []Marker `yaml:"Marker"`
}
```

A *Marker* is a single file type's markdown area definition.

```
type Marker struct {
   Ext   []string `yaml:"Ext"`
   Start string   `yaml:"Start"`
   End   string   `yaml:"End"`
}
```

## 3.4 Parse the NARRATIVE File

The Narrative file is used to describe the parse order of the files - and also which files to include or exclude.

The format of this file is:

- A singe file per line.
- A '#' on the start of a line to denote a single line comment.
- The files will be processed in order they appear in the file..

Some projects choose to create this file dynamically to include all files within the project. For example, you could add something like the following before the build step:

```
find ./ -name "*.tf" >> NARRATIVE
```

You can then feed the *NARRATIVE* file into the narrative process.

```
func ParseNarrative(cfg Config, log *log.Logger) {
   // open the NARRATIVE input file
   narrativeFile, err := os.Open(cfg.Input)
   if err != nil {
       log.Fatal(err)
   }
```

```
    defer narrativeFile.Close()

    // open the output file
    fout, err := os.OpenFile(cfg.Output, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        log.Fatal(err)
    }
    defer fout.Close()

    log.Printf("Marker config: %s\n", cfg.Markers)
    markers, err := ParseMarkerConfig(cfg.Markers)
    if err != nil {
        log.Fatal(err)
    }

    dir := filepath.Dir(narrativeFile.Name())
    rd := bufio.NewReader(narrativeFile)
    for {
        line, err := rd.ReadString('\n')
        if err != nil {
            if err == io.EOF {
                break
            }
            log.Fatal(err)
            return
        }

        line = strings.Trim(line, "\n ")
        if line == "" || line[0] == '#' {
            continue
        }
        inputFile := fmt.Sprintf("%s%c%s", dir, filepath.Separator, line)
        log.Printf("Processing: %s\n", inputFile)

        // call the main "markdown finding code" for this file
        Parse(markers, log, inputFile, fout)
    }
}
```

## 3.5 Read Marker Config YAML

This is some boilerplate code to read in the YAML file that describes how comment blocks should start and stop.

```
func ParseMarkerConfig(markersFile string) (*CommentMarkers, error) {
    filename, _ := filepath.Abs(markersFile)
    yamlFile, err := ioutil.ReadFile(filename)
    if err != nil {
        return nil, err
    }

    var markers CommentMarkers

    err = yaml.Unmarshal(yamlFile, &markers)
    if err != nil {
        return nil, err
    }

    return &markers, nil
}
```

## 3.6 Find A Marker

This function finds the maker definition based on the passed file extension. While this function "doesn't scale", the expected amount of configuration data means it should not be a problem.

---

Note: while looking for an extension match, the *Ext* array is expected to be in alpha order. If the first letter of the defined extension is before the first letter of the sought after extension, the loop moves on to the next section.

---

The function double loops over the passed in yaml config file looking for the section that matches the extension. If this becomes a problem, the file could be indexed by extension instead.

```
func FindMarker(markers *CommentMarkers, extension string) (*Marker, error) {
    if len(extension) < 2 {
        return nil, errors.New("File extension must have at least a '.' and one characte
    }

    for i := range markers.Markers {
        testExt := markers.Markers[i].Ext
        for m := range testExt {
            if testExt[m][0] > extension[1] {
                break
            }
            if string("."+testExt[m]) == extension {
                return &markers.Markers[i], nil
            }
        }
    }
    return nil, errors.New("Marker definition not found. Edit narrative.yaml.")
}
```

## 3.7 Parse a Code file and Extract the Markdown

While processing the files from the NARRATIVE file, we then look within the code file and find areas marked as markdown. We also "invert" the rest of the file to be markdown code blocks.

```
func Parse(markers *CommentMarkers, log *log.Logger, filePath string, fout io.Writer) 
    file, err := os.Open(filePath)
    if err != nil {
        log.Fatal(err)
        return
    }
    defer file.Close()

    // try to get the file extension
    extension := strings.ToLower(filepath.Ext(filePath))

    // find the start and end markers for this file type
    marker, err := FindMarker(markers, extension)
    if err != nil {
        log.Fatal(err)
    }

    code_mode := false
    scanner := bufio.NewScanner(file)
    line := ""
    for scanner.Scan() {
        line = scanner.Text()

        // if the makers are blank, take the file as is
        if marker.Start == "" && marker.End == "" {
            _, err := fmt.Fprintf(fout, "%s\n", line)
            if err != nil {
                log.Fatal(err)
            }
        } else {
            // line = strings.Trim(line, "\n ")
            if strings.Trim(line, "\n ") == marker.Start {
                code_mode = true
                continue
            }
            if strings.Trim(line, "\n ") == marker.End {
                code_mode = false
                continue
            }

            if code_mode {
                _, err := fmt.Fprintf(fout, "%s\n", line)
                if err != nil {
                    log.Fatal(err)
                }
            } else {
                _, err := fmt.Fprintf(fout, "     %s\n", line)
                if err != nil {
                    log.Fatal(err)
                }
            }
        }
    }
    if err := scanner.Err(); err != nil {
        log.Fatal(err)
    }
}
```

Web of Stories - Life Stories of Remarkable People. 2016. ‘Donald Knuth - Literate Programming (66/97).’