Subplot

The Subplot project

2020-09-26 16:50

Contents

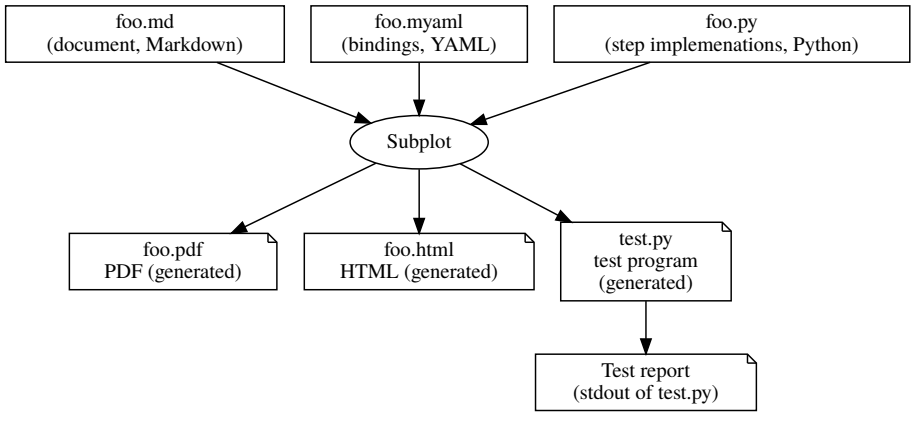
1 Introduction

Subplot is software to help capture and communicate acceptance criteria for software and systems, and how they are verified, in a way that's understood by all project stakeholders. The current document contains the acceptance criteria for Subplot itself, and its architecture.

The acceptance criteria are expressed as *scenarios*, which roughly correspond to use cases. The scenario as accompanied by explanatory text to explain things to the reader. Scenarios use a given/when/then sequence of steps, where each step is implemented by code provided by the developers of the system under test. This is very similar to the Cucumber[1] tool, but with more emphasis of producing a standalone document.

1.1 Subplot architecture

Subplot reads an input document, in Markdown, and generates a typeset output document, as PDF or HTML, for all stakeholders to understand. Subplot also generates a test program, in Python, that verifies the acceptance criteria are met, for developers and testers and auditors to verify the sustem under test meets its acceptance criteria. The generated program uses code written by the Subplot user to implement the verification steps. The graph below illustrates this and shows how data flows through the system.

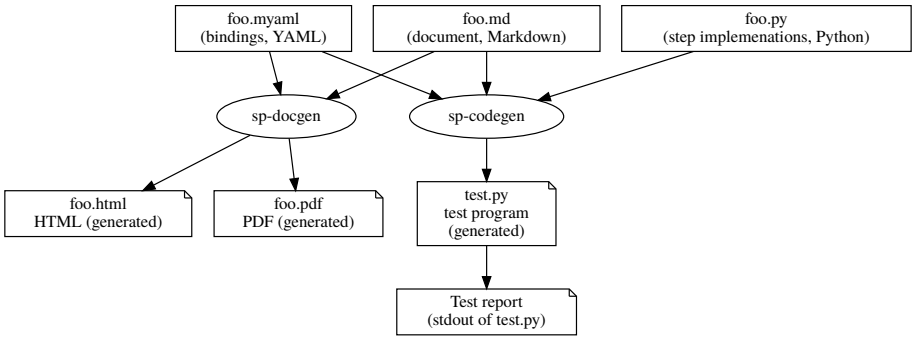


[1] `https://en.wikipedia.org/wiki/Cucumber_(software)`

Subplot uses the Pandoc[2] software for generating PDF and HTML output documents. In fact, any output format supported by Pandoc can be requested by the user. Depending on the output format, Pandoc may use, for example, LaTeX. Subplot interprets parts of the Markdown input file itself.

Subplot actually consists mainly of two separate programs: **sp-docgen** for generating output documents, and **sp-codegen** for generating the test program. There are a couple of additional tools (**sp-meta** for reporting meta data about a Subplot document, and **sp-filter** for doing the document generation as a Pandoc filter).

Thus a more detailed architecture view is shown below.



1.2 A fairy tale of acceptance testing

The king was upset. This naturally meant the whole court was in a tizzy and chattering excitedly at each other, while trying to avoid the royal wrath.

"Who will rid me of this troublesome chore?" shouted the king, and quaffed a flagon of wine. "And no killing of priests, this time!"

The grand hall's doors were thrown open. The grand wizard stood in the doorway, robe, hat, and staff everything, but quite still. After the court became silent, the wizard strode confidently to stand before the king.

"What ails you, my lord?"

The king looked upon the wizard, and took a deep breath. It does not do to shout at wizards, for they control dragons, and even kings are tasty morsels to the great beasts.

"I am tired of choosing what to wear every day. Can't you do something?"

The wizard stoke his long, grey beard. He turned around, looked at the magnificent outfits worn by members of the court. He turned back, and looked at the king.

[2] `https://pandoc.org/`

"I believe I can fix this. Just to be clear, your beef is with having to choose clothing, yes?"

"Yes", said the king, "that's what I said. When will you be done?"

The wizard raised his staff and brought it back down again, with a loud bang.

"Done" said the wizard, smugly.

The king was amazed and started smiling, until he noticed that everyone, including himself, was wearing identical burlap sacks and nothing on their feet. His voice was high, whiny, like that of a little child.

"Oh no, that's not at all what I wanted! Change it back! Change it back now!"

The morale of this story is to be clear and precise in your acceptance criteria, or you might get something other than what you really, really wanted.

1.3 Motivation for Subplot

Keeping track of requirements and acceptance criteria is necessary for all but the simplest of software projects. Having all stakeholders in a project agree to them is crucial, as is that all agree how it is verified that the software meets the acceptance criteria. Subplot provides a way for documenting the shared understanding of what the acceptance criteria are and how they can be checked automatically.

Stakeholders in a project may include:

- those who pay for the work to be done; this may be the employer of the developers for in-house projects ("*customer*")
- those who use the resulting systems, whether they pay for it or not ("*user*")
- those who install and configure the systems and keep them functional ("*sysadmin*")
- those who support the users ("*support*")
- those who test the project for acceptability ("*tester*")
- those who develop the system in the first place ("*developer*")

The above list is incomplete and simplistic, but suffices as an example.

All stakeholders need to understand the acceptance criteria, and how the system is evaluated against the criteria. In the simplest case, the customer and the developer need to both understand and agree so that the developer knows when the job is done, and the customer knows when they need to pay their bill.

However, even when the various stakeholder roles all fall upon the same person, or only on people who act as developers, the Subplot tooling can be useful. A developer would understand acceptance criteria expressed only in code, but doing so may take time and energy that are not always available. The Subplot approach aims to encourage hiding unnecessary detail and documenting things in a way that is easy to understand with little effort.

Unfortunately, this does mean that for a Subplot output document to be good and helpful, writing it will require effort and skill. No tool can replace that.

2 Requirements

This chapter lists requirements for Subplot. These requirements are not meant to be automatically verifiable. For specific, automatically testable acceptance criteria, see the later chapter with acceptance tests for Subplot[3].

Each requirement here is given a unique mnemnoic id for easier reference in discussions.

UnderstandableTests Acceptance tests should be possible to express in a way that's easily understood by all stakeholders, includcing those who are not software developers.

Done but requires the Subplot document to be written with care.

EasyToWriteDocs The markup language for writing documentation should be easy to write.

Done by using Markdown.

AidsComprehension The formatted human-readable documentation should use good layout and typography to enhance comprension.

In progress — typesetting via Pandoc works, but may need review and improvement.

CodeSeparately The code to implement the acceptance criteria should not be embedded in the documentation source, but be in separate files. This makes it easier to edit without specialised tooling.

Done by keeping scenario step implementations in a separate file.

AnyProgammingLanguage The developers implementing the acceptance tests should be free to use a language they're familiar and comfortable with. Subplot should not require them to use a specific language.

Not done — only Python supported at the moment.

FastTestExecution Executing the acceptance tests should be fast.

Not done &mash; the generated Python test program is simplistic and linear.

NoDeployment The acceptance test tooling should assume the system under test is already deployed and available. Deploying is too big of a problem space to bring into the scope of acceptance testing, and there are already good tools for deployment.

[3] `#acceptance`

Done by virtue of letting those who implement the scenario steps worry about it.

MachineParseableResults The tests should produce a machine parseable result that can be archived, post-processed, and analyzed in ways that are of interest to the project using Subplot. For example, to see trends in how long tests take, how often tests fail, to find regressions, and to find tests that don't provide value.

Not done — the generated test program is simplistic.

3 Subplot input language

Subplot reads three input files, each in a different format:

- The document file, which uses the Markdown dialects understood by Pandoc.
- The bindings file, in YAML.
- The functions file, in Bash or Python.

Subplot interprets marked parts of the input document specially. It does this via the Pandoc abstract syntax tree, rather than text manipulation, and thus anything that Pandoc understands is understood by Subplot. We will not specify Pandoc's dialect of Markdown here, only the parts Subplot pays attention to.

3.1 Document metadata

Pandoc supports, and Subplot makes use of, a YAML metadata block[4] in a Markdown document. This can and should be used to set the document title, authors, date (version), and can be used to control some of the typesetting. Crucially for Subplot, the bindings and functions files are named in the metadata block, rather than Subplot deriving them from the input file name.

As an example, the metadata block for the Subplot document might look as follows. The `---` before and `...` after the block are mandatory: they are how Pandoc recongizes the block.

```
---
title: "Subplot"
author: The Subplot project
date: work in progress
bindings: subplot.yaml
functions: subplot.py
...
```

There can be more than one bindings or functions file: use a YAML list.

[4] `https://pandoc.org/MANUAL.html#extension-yaml_metadata_block`

3.2 Document markup

Subplot uses Pandoc[5], the universal document converter, to parse the Markdown file, and thus understands the variants of Markdown that Pandoc supports. This includes traditional Markdown, CommonMark, and GitHub-flavored Markdown.

Subplot extends Markdown by treating certain certain tags for fenced code blocks[6] specially. A scenario, for example, would look like this:

```
1 ```scenario
2 given a standard setup
3 when peace happens
4 then everything is OK
5 ```
```

The `scenario` tag on the code block is recognized by Subplot, which will typeset the scenario (in output documents) or generate code (for the test program) accordingly. Scenario blocks do not need to be complete scenario. Subplot will collect all the snippets into one block for the test program. Snippets under the same heading belong together; the next heading of the same or a higher level ends the scenario.

For embedding test data files in the Markdown document, Subplot understands the `file` tag:

```
~~~{#filename .file}
This data is accessible to the test program as 'filename'.
~~~
```

The `.file` attribute is necessary, as is the identifier, here `#filename`. The generated test program can access the data using the identifier (without the #). The mechanism used is generic to Pandoc, and can be used to affect the typesetting by adding more attributes. For example, Pandoc can typeset the data in the code block using syntax highlighting, if the language is specified: `.markdown`, `.yaml`, or `.python`, for example.

Subplot also understands the `dot` and `roadmap` tags, and can use the Graphviz dot program, or the roadmap[7] Rust crate, to produce graphs. These can useful for describing things visually.

3.3 Bindings file

The bindings file binds scenario steps to code functions that implement the steps. The YAML file is a list of objects (also known as dicts or hashmaps or key/value pairs), specifying a step kind (given, when, then), a pattern matching the text of the step and optionally capturing interesting parts of the text, and the name of a function that implements the step.

[5] `https://pandoc.org/`
[6] `https://pandoc.org/MANUAL.html#fenced-code-blocks`
[7] `https://crates.io/search?q=roadmap`

Patterns can be simple or full-blown Perl-compatible regular expresssions (PCRE[8]).

```
- given: "a standard setup"
  function: create_standard_setup
- when: "{thing} happens"
  function: make_thing_happen
- when: "I say (?P<sentence>.+) with a smile"
  regex: true
  function: speak
- then: "everything is OK"
  function: check_everything_is_ok
```

In the example above, there are four bindings:

- A binding for a "given a standard setup" step. The binding captures no part of the text, and causes the `create_standard_setup` function to be called.
- A binding for a "when" step consisting of one word followed by "happens". For example, "peace", as in "then peace happens". The word is captured as "thing", and given to the `make_thing_happen` function as an argument when it is called.
- A binding for a "when" followed by "I say", an arbitrary sentence, and then "with a smile", as in "when I say good morning to you with a smile". The function `speak` is then called with capture named "sentence" as "good morning to you".
- A binding for a "then everything is OK" step, which captures nothing, and calls the `check_everything_is_ok` function.

3.4 Simple patterns

The simple patterns are of the form `{name}` and match a single word consisting of printable characters. This can be varied by adding a suffix, such as `{name:text}` which matches any text. The following kinds of simple patterns are supported:

- `{name}` or `{name:word}` – a single word
- `{name:text}` – any text
- `{name:int}` – any whole number, including negative
- `{name:uint}` – any unsigned whole number
- `{name:number}` – any number

A pattern uses simple patterns by default, or if the `regex` field is set to false. To use regular expressions, `regex` must be set to true. Subplot complains if typical regular expression characters are used, when simple patterns are expected, unless `regex` is explicitly set to false.

[8] `https://en.wikipedia.org/wiki/Perl_Compatible_Regular_Expressions`

3.5 Regular expression patterns

Regular expression patterns are used only if the binding `regex` field is set to `true`.

The regular expressions use PCRE[9] syntax as implemented by the Rust regex[10] crate. The `(?P<name>pattern)` syntax is used to capture parts of the step. The captured parts are given to the bound function as arguments, when it's called.

3.6 Functions file

Functions implementing steps are supported in Bash and Python. The language is chosen by setting the `template` field in the document YAML metadata to `bash` or `python`.

The functions files are not parsed by Subplot at all. Subplot merely copies them to the output. All parsing and validation of the file is done by the programming language being used.

The conventions for calling step functions vary by language. All languages support a "dict" abstraction of some sort. This is most importantly used to implement a "context" to store state in a controlled manner between calls to step functions. A step function can set a key to a value in the context, or retrieve the value for a key.

Typically, a "when" step does something, and records the results into the context, and a "then" step checks the results by inspecting the context. This decouples functions from each other, and avoids having them use global variables for state.

3.6.1 Bash

The step functions are called without any arguments.

The context is managed using shell functions provided by the Bash template:

- `ctx_set key value`
- `ctx_get key`

Captured values from scenario steps are passed in via another dict and accessed using another function:

- `cap_get key`

Similarly, there's a dict for embedded data files:

- `files_get filename`

The template provides assertion functions: `assert_eq`, `assert_contains`.

Example:

[9] `https://en.wikipedia.org/wiki/Perl_Compatible_Regular_Expressions`
[10] `https://crates.io/crates/regex`

```
_run()
{
    if "$@" < /dev/null > stdout 2> stderr
    then
        ctx_set exit 0
    else
        ctx_set exit "$?"
    fi
    ctx_set stdout "$(cat stdout)"
    ctx_set stderr "$(cat stderr)"
}

run_echo_without_args()
{
    _run echo
}

run_echo_with_args()
{
    args="$(cap_get args)"
    _run echo "$args"
}

exit_code_is()
{
    actual_exit="$(ctx_get exit)"
    wanted_exit="$(cap_get exit_code)"
    assert_eq "$actual_exit" "$wanted_exit"
}

stdout_is_a_newline()
{
    stdout="$(ctx_get stdout)"
    assert_eq "$stdout" "$(printf '\n')"
}

stdout_is_text()
{
    stdout="$(ctx_get stdout)"
    text="$(cap_get text)"
    assert_contains "$stdout" "$text"
}

stderr_is_empty()
{
    stderr="$(ctx_get stderr)"
```

```
    assert_eq "$stderr" ""
}
```

3.6.2 Python

The context is implemented by a dict-like class.

The step functions are called with a `ctx` argument that has the current state of the context, and each capture from a step as a keyword argument. The keyword argument name is the same as the capture name in the pattern in the bindings file.

Embedded files are accessed using a function:

- `get_file(filename)`

Example:

```
import json

def exit_code_is(ctx, wanted=None):
    assert_eq(ctx.get("exit"), wanted)

def json_output_matches_file(ctx, filename=None):
    actual = json.loads(ctx["stdout"])
    expected = json.load(open(filename))
    assert_dict_eq(actual, expected)

def file_ends_in_zero_newlines(ctx, filename=None):
    content = open(filename, "r").read()
    assert_ne(content[-1], "\n")
```

4 Acceptance criteria for Subplot

Add the acceptance criteria test scenarios for Subplot here.

4.1 Test data shared between scenarios

The scenarios below test Subplot by running it against specific input files. This section specifies the bindings and functions files. They're separate from the scenarios so that the scenarios are shorter and clearer, but also so that the input files do no need to be duplicated for each scenario.

File: **simple.md**

```
---
title: Test scenario
bindings: b.yaml
functions: f.py
```

```
...

# Simple
This is the simplest possible test scenario

```scenario
given precondition foo
when I do bar
then bar was done
```
```

File: **b.yaml**

```
- given: precondition foo
  function: precond_foo
- when: I do bar
  function: do_bar
- when: I do foobar
  function: do_foobar
- then: bar was done
  function: bar_was_done
- then: foobar was done
  function: foobar_was_done
```

File: **f.py**

```
def precond_foo(ctx):
    ctx['bar_done'] = False
    ctx['foobar_done'] = False
def do_bar(ctx):
    ctx['bar_done'] = True
def bar_was_done(ctx):
    assert_eq(ctx['bar_done'], True)
def do_foobar(ctx):
    ctx['foobar_done'] = True
def foobar_was_done(ctx):
    assert_eq(ctx['foobar_done'], True)
```

4.2 Smoke test

This tests that Subplot can build a PDF and an HTML document, and execute a simple scenario successfully. The test is based on generating the test program from an input file, running the test program, and examining the output.

given file **simple.md**
and file **b.yaml**
and file **f.py**
when I run sp-docgen **simple.md** -o **simple.pdf**

then file **simple.pdf** exists
when I run sp-docgen **simple.md** -o **simple.html**
then file **simple.html** exists
when I run sp-codegen --run **simple.md** -o **test.py**
then scenario "**Simple**" was run
and step "**given precondition foo**" was run
and step "**when I do bar**" was run
and step "**then bar was done**" was run
and program finished successfully

4.3 Keywords

Subplot supports the keywords **given**, **when**, and **then**, and the aliases **and** and **but**. The aliases stand for the same (effective) keyword as the previous step in the scenario. This chapter has scenarios to check the keywords and aliases in various combinations.

4.3.1 All the keywords

given file **allkeywords.md**
and file **b.yaml**
and file **f.py**
when I run sp-docgen **allkeywords.md** -o **foo.pdf**
then file **foo.pdf** exists
when I run sp-codegen --run **allkeywords.md** -o **test.py**
then scenario "**All keywords**" was run
and step "**given precondition foo**" was run
and step "**when I do bar**" was run
and step "**then bar was done**" was run
and program finished successfully

File: **allkeywords.md**

```
---
title: All the keywords scenario
bindings: b.yaml
functions: f.py
...

# All keywords

This uses all the keywords.

```scenario
given precondition foo
when I do bar
and I do foobar
```
```

```
then bar was done
but foobar was done
```

4.3.2 Keyword aliases

given file **aliases.md**
and file **b.yaml**
and file **f.py**
when I run sp-docgen **aliases.md** -o **aliases.html**
then file **aliases.html** matches regex /**given<[^>]*> precondition foo**/
and file **aliases.html** matches regex /**when<[^>]*> I do bar**/
and file **aliases.html** matches regex /**and<[^>]*> I do foobar**/
and file **aliases.html** matches regex /**then<[^>]*> bar was done**/
and file **aliases.html** matches regex /**but<[^>]*> foobar was done**/
and program finished successfully

File: **aliases.md**

```
---
title: Keyword aliasesG
bindings: b.yaml
functions: f.py
...

# Aliases

```scenario
given precondition foo
when I do bar
and I do foobar
then bar was done
but foobar was done
```
```

4.3.3 Misuse of continuation keywords

When continuation keywords (`and` and `but`) are used, they have to not be the first keyword in a scenario. Any such scenario will fail to parse because subplot will be unable to determine what kind of keyword they are meant to be continuing.

given file **continuationmisuse.md**
and file **b.yaml**
and file **f.py**
when I run sp-docgen **continuationmisuse.md** -o **foo.pdf**
then file **foo.pdf** exists
when I try to run sp-codegen --run **continuationmisuse.md** -o **test.py**
then exit code is non-zero

File: **continuationmisuse.md**

```
---
title: Continuation keyword misuse
bindings: b.yaml
functions: f.py
...

# Continuation keyword misuse

This scenario should fail to parse because we misuse a
continuation keyword at the start.

```scenario
and precondition foo
when I do bar
then bar was done
```
```

4.4 Empty lines in scenarios

This scenario verifies that empty lines in scenarios are ignored.

given file **emptylines.md**
and file **b.yaml**
and file **f.py**
when I run sp-docgen **emptylines.md** -o **emptylines.pdf**
then file **emptylines.pdf** exists
when I run sp-docgen **emptylines.md** -o **emptylines.html**
then file **emptylines.html** exists
when I run sp-codegen --run **emptylines.md** -o **test.py**
then scenario "**Simple**" was run
and step "**given precondition foo**" was run
and step "**when I do bar**" was run
and step "**then bar was done**" was run
and program finished successfully

File: **emptylines.md**

```
---
title: Test scenario
bindings: b.yaml
functions: f.py
...

# Simple
This is the simplest possible test scenario

```

```
```scenario
given precondition foo

when I do bar

then bar was done

```
```

4.5 Automatic cleanup in scenarios

A binding can define a cleanup function, which gets called at the end of the scenario in reverse order for the successful steps. If a step fails, all the cleanups for the successful steps are still called. We test this for every language templat we support.

File: **cleanup.yaml**

```yaml
- given: foo
  function: foo
  cleanup: foo_cleanup
- given: bar
  function: bar
  cleanup: bar_cleanup
- given: failure
  function: failure
  cleanup: failure_cleanup
```

File: **cleanup.py**

```python
def foo(ctx):
   pass
def foo_cleanup(ctx):
   pass
def bar(ctx):
   pass
def bar_cleanup(ctx):
   pass
def failure(ctx):
   assert 0
def failure_cleanup(ctx):
   pass
```

File: **cleanup.sh**

```
foo() {
    true
}
foo_cleanup() {
```

```
    true
}
bar() {
    true
}
bar_cleanup() {
    true
}
failure() {
   return 1
}
failure_cleanup() {
   true
}
```

4.5.1 Cleanup functions gets called on success (Python)

given file **cleanup-success-python.md**
and file **cleanup.yaml**
and file **cleanup.py**
when I run sp-codegen --run **cleanup-success-python.md** -o **test.py**
then scenario "**Cleanup**" was run
and step "**given foo**" was run, and then step "**given bar**"
and cleanup for "**given bar**" was run, and then for "**given foo**"
and program finished successfully

File: **cleanup-success-python.md**

```
---
title: Cleanup
bindings: cleanup.yaml
functions: cleanup.py
template: python
...

# Cleanup

~~~scenario
given foo
given bar
~~~
```

4.5.2 Cleanup functions get called on failure (Python)

given file **cleanup-fail-python.md**
and file **cleanup.yaml**
and file **cleanup.py**

when I try to run sp-codegen --run **cleanup-fail-python.md** -o **test.py**
then scenario "**Cleanup**" was run
and step "**given foo**" was run, and then step "**given bar**"
and cleanup for "**given bar**" was run, and then for "**given foo**"
and cleanup for "**given failure**" was not run
and exit code is non-zero

File: **cleanup-fail-python.md**

```
---
title: Cleanup
bindings: cleanup.yaml
functions: cleanup.py
template: python
...

# Cleanup

~~~scenario
given foo
given bar
given failure
~~~
```

4.5.3 Cleanup functions gets called on success (Bash)

given file **cleanup-success-bash.md**
and file **cleanup.yaml**
and file **cleanup.sh**
when I run sp-codegen --run **cleanup-success-bash.md** -o **test.sh**
then scenario "**Cleanup**" was run
and step "**given foo**" was run, and then step "**given bar**"
and cleanup for "**given bar**" was run, and then for "**given foo**"
and program finished successfully

File: **cleanup-success-bash.md**

```
---
title: Cleanup
bindings: cleanup.yaml
functions: cleanup.sh
template: bash
...

# Cleanup

~~~scenario
given foo
```

```
given bar
~~~
```

4.5.4 Cleanup functions get called on failure (Bash)

If a step fails, all the cleanups for the preceding steps are still called, in reverse order.

given file **cleanup-fail-bash.md**
and file **cleanup.yaml**
and file **cleanup.sh**
when I try to run sp-codegen --run **cleanup-fail-bash.md** -o **test.sh**
then scenario "**Cleanup**" was run
and step "**given foo**" was run, and then step "**given bar**"
and cleanup for "**given bar**" was run, and then for "**given foo**"
and cleanup for "**given failure**" was not run
and exit code is non-zero

File: **cleanup-fail-bash.md**

```
---
title: Cleanup
bindings: cleanup.yaml
functions: cleanup.sh
template: bash
...

# Cleanup

~~~scenario
given foo
given bar
given failure
~~~
```

4.6 Capturing parts of steps for functions

A scenario step binding can capture parts of a scenario step, to be passed to the function implementing the step as an argument. Captures can be done using regular expressions.

4.6.1 Capture using simple patterns

given file **simplepattern.md**
and file **simplepattern.yaml**
and file **capture.py**
when I run sp-codegen --run **simplepattern.md** -o **test.py**
then scenario "**Simple pattern**" was run

and step "**given I am Tomjon**" was run
and output matches /**function got argument name as Tomjon**/
and program finished successfully

File: **simplepattern.md**

```
---
title: Simple pattern capture
bindings: simplepattern.yaml
functions: capture.py
...

# Simple pattern

~~~scenario
given I am Tomjon
~~~
```

File: **simplepattern.yaml**

```
- given: I am {name}
  function: func
```

File: **capture.py**

```
def func(ctx, name=None):
    print('function got argument name as', name)
```

4.6.2 Simple patterns with regex metacharacters: forbidden case

Help use to avoid accidental regular expression versus simple pattern confusion. The rule is that a simple pattern mustn't contain regular expression meta characters unless the rule is explicitly marked as not being a regular expression pattern.

given file **confusedpattern.md**
and file **confusedpattern.yaml**
and file **capture.py**
when I try to run sp-codegen --run **confusedpattern.md** -o **test.py**
then exit code is non-zero
and stderr matches /**simple pattern contains regex**/

File: **confusedpattern.md**

```
---
title: Simple pattern capture
bindings: confusedpattern.yaml
functions: capture.py
...

```

```
# Simple pattern

~~~scenario
given I* am Tomjon
~~~
```

File: **confusedpattern.yaml**

```
- given: I* am {name}
  function: func
```

4.6.3 Simple patterns with regex metacharacters: allowed case

given file **confusedbutok.md**
and file **confusedbutok.yaml**
and file **capture.py**
when I run sp-codegen --run **confusedbutok.md** -o **test.py**
then program finished successfully

File: **confusedbutok.md**

```
---
title: Simple pattern capture
bindings: confusedbutok.yaml
functions: capture.py
...

# Simple pattern

~~~scenario
given I* am Tomjon
~~~
```

File: **confusedbutok.yaml**

```
- given: I* am {name}
  function: func
  regex: false
```

4.6.4 Capture using regular expressions

given file **regex.md**
and file **regex.yaml**
and file **capture.py**
when I run sp-codegen --run **regex.md** -o **test.py**
then scenario "**Regex**" was run
and step "**given I am Tomjon**" was run
and output matches /**function got argument name as Tomjon**/
and program finished successfully

File: **regex.md**

```
---
title: Regex capture
bindings: regex.yaml
functions: capture.py
...

# Regex

~~~scenario
given I am Tomjon
~~~
```

File: **regex.yaml**

```
- given: I am (?P<name>\S+)
  function: func
  regex: true
```

4.7 Avoid changing typesetting output file needlessly

4.7.1 Avoid typesetting if output is newer than source files

This scenario make sure that if docgen generates the bitwise identical output to the existing output file, it doesn't actually write it to the output file, including its timestamp. This avoids triggering programs that monitor the output file for changes.

given file **simple.md**
and file **b.yaml**
and file **f.py**
when I run sp-docgen **simple.md** -o **simple.pdf**
then file **simple.pdf** exists
when I remember metadata for file **simple.pdf**
and I run sp-docgen **simple.md** -o **simple.pdf**
then file **simple.pdf** has same metadata as before
and only files **simple.md, b.yaml, f.py, simple.pdf** exist

4.7.2 Do typeset if output is older than markdown

given file **simple.md**
and file **b.yaml**
and file **f.py**
when I run sp-docgen **simple.md** -o **simple.pdf**
then file **simple.pdf** exists
when I remember metadata for file **simple.pdf**
and I touch file **simple.md**

and I run sp-docgen **simple.md** -o **simple.pdf**
then file **simple.pdf** has changed from before

4.7.3 Do typeset if output is older than functions

given file **simple.md**
and file **b.yaml**
and file **f.py**
when I run sp-docgen **simple.md** -o **simple.pdf**
then file **simple.pdf** exists
when I remember metadata for file **simple.pdf**
and I touch file **f.py**
and I run sp-docgen **simple.md** -o **simple.pdf**
then file **simple.pdf** has changed from before

4.7.4 Do typeset if output is older than bindings

given file **simple.md**
and file **b.yaml**
and file **f.py**
when I run sp-docgen **simple.md** -o **simple.pdf**
then file **simple.pdf** exists
when I remember metadata for file **simple.pdf**
and I touch file **b.yaml**
and I run sp-docgen **simple.md** -o **simple.pdf**
then file **simple.pdf** has changed from before

4.8 Document structure

Subplot uses chapters and sections to keep together scenario snippets that form a complete scenario. The lowest level heading before a snippet starts a scenario and is the name of the scenario. If there's subheadings, they divide the description of the scenario into parts, but don't start a new scenario. The next heading at the same or a higher level starts a new scenario.

4.8.1 Lowest level heading is name of scenario

given file **scenarioislowest.md**
and file **b.yaml**
and file **f.py**
when I run sp-codegen --run **scenarioislowest.md** -o **test.py**
then scenario "**heading 1.1.1**" was run
and program finished successfully

File: **scenarioislowest.md**

```
1
2 ---
```

```
title: Test scenario
bindings: b.yaml
functions: f.py
...

# heading 1
## heading 1.1
### heading 1.1.1

```scenario
given precondition foo
```
```

4.8.2 Subheadings don't start new scenario

given file **subisnotnewscenario.md**
and file **b.yaml**
and file **f.py**
when I run sp-codegen --run **subisnotnewscenario.md** -o **test.py**
then scenario "**heading 1.1a**" was run
and program finished successfully

File: **subisnotnewscenario.md**

```

---
title: Test scenario
bindings: b.yaml
functions: f.py
...

# heading 1
## heading 1.1a

```scenario
given precondition foo
```

heading 1.1.1
heading 1.1.2
```

4.8.3 Next heading at same level starts new scenario

given file **samelevelisnewscenario.md**
and file **b.yaml**
and file **f.py**
when I run sp-codegen --run **samelevelisnewscenario.md** -o **test.py**

then scenario "**heading 1.1.1**" was run
and scenario "**heading 1.1.2**" was run
and program finished successfully

File: **samelevelisnewscenario.md**

````
---
title: Test scenario
bindings: b.yaml
functions: f.py
...

# heading 1
## heading 1.1
### heading 1.1.1

```scenario
given precondition foo
```
### heading 1.1.2

```scenario
given precondition foo
```
````

4.8.4 Next heading at higher level starts new scenario

given file **higherisnewscenario.md**
and file **b.yaml**
and file **f.py**
when I run sp-codegen --run **higherisnewscenario.md** -o **test.py**
then scenario "**heading 1.1.1**" was run
and scenario "**heading 1.2**" was run
and program finished successfully

File: **higherisnewscenario.md**

````
---
title: Test scenario
bindings: b.yaml
functions: f.py
...

# heading 1
## heading 1.1
### heading 1.1.1
````

```
~~~scenario
given precondition foo
~~~
## heading 1.2

~~~scenario
given precondition foo
~~~
```

4.8.5 Document titles

The document and code generators require a document title, because it's a common user error to not have one, and Subplot should help make good documents. The Pandoc filter, however, mustn't require a document title, because it's used for things like formatting websites using ikiwiki, and ikiwiki has a different way of specifying page titles.

4.8.5.1 Document generator gives an error if input document lacks title

given file **notitle.md**
when I try to run sp-docgen **notitle.md** -o **foo.md**
then exit code is non-zero

File: **notitle.md**

```
---
bindings: b.yaml
functions: f.py
...


# Introduction

This is a very simple Markdown file without a YAML metadata block,
and thus also no document title.

~~~scenario
given precondition foo
when I do bar
then bar was done
```

4.8.5.2 Code generator gives an error if input document lacks title

given file **notitle.md**
when I try to run sp-codegen --run **notitle.md** -o **test.py**

then exit code is non-zero

4.9 Running only chosen scenarios

To make the edit-test loop more convenient for the test programs generated by Subplot, we allow the user to specify patterns for scenarios to run. Default is to run all scenarios.

4.9.1 Running only chosen scenarios with Python

This verifies that the generated Python test program can run only chosen scenarios.

given file **twoscenarios-python.md**
and file **b.yaml**
and file **f.py**
when I run sp-codegen **twoscenarios-python.md** -o **test.py**
and I run python3 **test.py on**
then scenario "**One**" was run
and scenario "**Two**" was not run
and program finished successfully

File: **twoscenarios-python.md**

```
---
title: Test scenario
bindings: b.yaml
functions: f.py
...

# One

```scenario
given precondition foo
when I do bar
then bar was done
```

Two

```scenario
given precondition foo
when I do bar
then bar was done
```
```

4.9.2 Running only chosen scenarios with Bash

This verifies that the generated Bash test program can run only chosen scenarios.

given file **twoscenarios-bash.md**
and file **b.yaml**
and file **f.sh**
when I run sp-codegen **twoscenarios-bash.md** -o **test.sh**
and I run bash **test.sh on**
then scenario "**One**" was run
and scenario "**Two**" was not run
and program finished successfully

File: **twoscenarios-bash.md**

```
---
title: Test scenario
bindings: b.yaml
functions: f.sh
template: bash
...

# One

```scenario
given precondition foo
when I do bar
then bar was done
```

Two

```scenario
given precondition foo
when I do bar
then bar was done
```
```

File: **f.sh**

```
precond_foo() {
    ctx_set bar_none 0
    ctx_set foobar_none 0
}

do_bar() {
    ctx_set bar_done 1
}
```

```
9
10  do_foobar() {
11      ctx_set foobar_done 1
12  }
13
14  bar_was_done() {
15      actual="$(ctx_get bar_done)"
16      assert_eq "$actual" 1
17  }
18
19  foobar_was_done() {
20      actual="$(ctx_get foobar_done)"
21      assert_eq "$actual" 1
22  }
```

4.10 Document metadata

Some document metadata should end up in the typeset document, especially the title, authors. The document date is more complicated, to cater to different use cases:

- a work-in-progress document needs a new date for each revision
  - maintaining the `date` metadata field manually is quite tedious, so Subplot provides it automatically using the document source file modification time
  - some people would prefer a `git describe` or similar method for indicating the document revision, so Subplot allows the date to be specified via the command line
- a finished, reviewed, officiall stamped document needs a fixed date
  - Subplot allows this to be written as the `date` metadata field

The rules for what Subplot uses as the date or document revision information are, then:

- if there is `date` metadata field, that is used
- otherwise, if the user gives the `--date` command line option, that is used
- otherwise, the markdown file's modification time is used

4.10.1 Date given in metadata

This scenario tests that the `date` field in metadata is used if specified.

given file **metadate.md**
when I run sp-docgen **metadate.md** -o **metadate.html**
then file **metadate.html** exists
and file **metadate.html** contains "**<title>The Fabulous Title</title>**"
and file **metadate.html** contains "**Alfred Pennyworth**"

and file **metadate.html** contains "**Geoffrey Butler**"
and file **metadate.html** contains "**WIP**"

File: **metadate.md**

```
---
title: The Fabulous Title
author:
- Alfred Pennyworth
- Geoffrey Butler
date: WIP
...
# Introduction
This is a test document. That's all.
```

4.10.2 Date given on command line

This scenario tests that the `--date` command line option is used.

given file **dateless.md**
when I run sp-docgen **dateless.md** -o **dateoption.html** --date=**FANCYDATE**
then file **dateoption.html** exists
and file **dateoption.html** contains "**<title>The Fabulous Title</title>**"
and file **dateoption.html** contains "**Alfred Pennyworth**"
and file **dateoption.html** contains "**Geoffrey Butler**"
and file **dateoption.html** contains "**FANCYDATE**"

File: **dateless.md**

```
---
title: The Fabulous Title
author:
- Alfred Pennyworth
- Geoffrey Butler
...
# Introduction
This is a test document. It has no date metadata.
```

4.10.3 No date anywhere

This scenario tests the case of no metadata `date` and no command line option, either. The date in the typeset document shall come from the modification time of the input file, and shall have the date in ISO 8601 format, with time to the minute.

given file **dateless.md**
and file **dateless.md** has modification time **2020-02-26 07:53:17**
when I run sp-docgen **dateless.md** -o **mtime.html**
then file **mtime.html** exists

and file **mtime.html** contains "**<title>The Fabulous Title</title>**"
and file **mtime.html** contains "**Alfred Pennyworth**"
and file **mtime.html** contains "**Geoffrey Butler**"
and file **mtime.html** contains "**2020-02-26 07:53**"

4.10.4 Missing bindings file

If a bindings file is missing, the error message should name the missing file.

given file **missing-binding.md**
when I try to run sp-docgen **missing-binding.md** -o **foo.htmlh**
then exit code is non-zero
and stderr matches /**: missing-binding.yaml:**/

File: **missing-binding.md**

```
---
title: Missing binding
bindings: missing-binding.yaml
...
```

4.10.5 Missing functions file

If a functions file is missing, the error message should name the missing file.

given file **missing-functions.md**
and file **b.yaml**
when I try to run sp-codegen --run **missing-functions.md** -o **foo.py**
then exit code is non-zero
and stderr matches /**: missing-functions.py:**/

File: **missing-functions.md**

```
---
title: Missing functions
bindings: b.yaml
functions: missing-functions.py
...
```

4.10.6 Extracting metadata from a document

The **sp-meta** program extracts metadata from a document. It is useful to see the scenarios, for example. For example, given a document like this:

sp-meta would extract this information from the **simple.md** example:

```
title: Test scenario
bindings: b.yaml
functions: f.py
scenario Simple
```

This scenario check sp-meta works. Note that it requires the bindings or functions files.

given file **images.md**
and file **b.yaml**
and file **other.yaml**
and file **f.py**
and file **other.py**
and file **foo.bib**
and file **bar.bib**
and file **expected.json**
when I run sp-meta **images.md**
then output matches /**source: images.md**/
and output matches /**source: b.yaml**/
and output matches /**source: other.yaml**/
and output matches /**source: f.py**/
and output matches /**source: other.py**/
and output matches /**source: foo.bib**/
and output matches /**source: bar.bib**/
and output matches /**source: image.gif**/
and output matches /**bindings: b.yaml**/
and output matches /**bindings: other.yaml**/
and output matches /**functions: f.py**/
when I run sp-meta **images.md** -o json
then JSON output matches **expected.json**

File: **images.md**

```
---
title: Document refers to external images
bindings:
- b.yaml
- other.yaml
functions:
- f.py
- other.py
bibliography: [foo.bib, bar.bib]
...

![alt text](image.gif)
```

File: **other.yaml**

```
[]
```

File: **other.py**

```

```

File: **foo.bib**

```
@book{foo2020,
 author    = "James Random",
 title     = "The Foo book",
 publisher = "The Internet",
 year      =  2020,
 address   = "World Wide Web",
}
```

File: **bar.bib**

```
@book{foo2020,
 author    = "James Random",
 title     = "The Bar book",
 publisher = "The Internet",
 year      =  2020,
 address   = "World Wide Web",
}
```

File: **expected.json**

```
{
    "title": "Document refers to external images",
    "sources": [
      "images.md",
      "b.yaml",
      "other.yaml",
      "f.py",
      "other.py",
      "foo.bib",
      "bar.bib",
      "image.gif"
    ],
    "binding_files": [
      "b.yaml",
      "other.yaml"
    ],
    "function_files": [
      "f.py",
      "other.py"
    ],
    "bibliographies": [
      "foo.bib",
      "bar.bib"
    ],
    "files": [],
    "scenarios": []
}
```

4.11 Embedded files

Subplot allows data files to be embedded in the input document. This is handy for small test files and the like.

Handling of a newline character on the last line is tricky. Pandoc doesn't include a newline on the last line. Sometimes one is needed—but sometimes it's not wanted. A newline can be added by having an empty line at the end, but that is subtle and easy to miss. Subplot helps the situation by allowing a `add-newline=` class to be added to the code blocks, with one of three allowed cases:

- no `add-newline` class—default handling: same as `add-newline=auto`
- `add-newline=auto`—add a newline, if one isn't there
- `add-newline=no`—never add a newline, but keep one if it's there
- `add-newline=yes`—always add a newline, even if one is already there

The scenarios below test the various cases.

4.11.1 Extract embedded file

This scenario checks that an embedded file can be extracted, and used in a subplot.

given file **embedded.md**
when I run sp-docgen **embedded.md** -o **foo.html**
then file **foo.html** exists
and file **foo.html** matches regex /**embedded\.txt**/

File: **embedded.md**

```
---
title: One embedded file
...

~~~{#embedded.txt .file}
This is the embedded file.
~~~
```

4.11.2 Extract embedded file, by default add missing newline

This scenario checks the default handling: add a newline if one is missing.

given file **default-without-newline.txt**
then **default-without-newline.txt** ends in one newline

File: **default-without-newline.txt**

```
This file does not end in a newline.
```

4.11.3 Extract embedded file, by default do not add a second newline

This scenario checks the default handling: if content already ends in a newline, do not add another newline.

given file **default-has-newline.txt**
then **default-has-newline.txt** ends in one newline

File: **default-has-newline.txt**

```
This file ends in a newline.
```

4.11.4 Extract embedded file, automatically add missing newline

Explicitly request automatic newlines, when the file does not end in one.

given file **auto-without-newline.txt**
then **auto-without-newline.txt** ends in one newline

File: **auto-without-newline.txt**

```
This file does not end in a newline.
```

4.11.5 Extract embedded file, do not automatically add second newline

Explicitly request automatic newlines, when the file already ends in one.

given file **auto-has-newline.txt**
then **auto-has-newline.txt** ends in one newline

File: **auto-has-newline.txt**

```
This file ends in a newline.
```

4.11.6 Extract embedded file, explicitly add missing newline

Explicitly request automatic newlines, when the file doesn't end with one.

given file **add-without-newline.txt**
then **add-without-newline.txt** ends in one newline

File: **add-without-newline.txt**

```
This file does not end in a newline.
```

4.11.7 Extract embedded file, explicitly add second newline

Explicitly request automatic newlines, when the file already ends with one.

given file **add-has-newline.txt**
then **add-has-newline.txt** ends in two newlines

File: **add-has-newline.txt**

```
This file ends in a newline.
```

4.11.8 Extract embedded file, do not add missing newline

Explicitly ask for no newline to be added.

given file **no-adding-without-newline.txt**
then **no-adding-without-newline.txt** does not end in a newline

File: **no-adding-without-newline.txt**

```
This file does not end in a newline.
```

4.11.9 Fail if the same filename is used twice

given file **onefiletwice.md**
when I try to run sp-docgen **onefiletwice.md** -o **onefiletwice.html**
then exit code is non-zero
and file **onefiletwice.html** does not exist

File: **onefiletwice.md**

```
---
title: Two embedded files with the same name
...

```{#filename .file}
This is the embedded file.
```

```{#filename .file}
This is another embedded file, and has the same name.
```
```

4.11.10 Fail if two filenames only differ in case

given file **casediff.md**
when I try to run sp-docgen **casediff.md** -o **casediff.html**
then exit code is non-zero
and file **casediff.html** does not exist

File: **casediff.md**

```
---
title: Two embedded files with names differing only in case
...

```{#filename .file}
This is the embedded file.
```
```

````
8
9  ```{#FILENAME .file}
10 This is another embedded file, and has the same name in uppercase.
11 ```
````

4.12 Steps must match bindings

Subplot permits the binding author to define arbitrarily complex regular expressions for binding matches. In order to ensure that associating steps to bindings is both reliable and tractable, a step must match *exactly one* binding.

File: **badbindings.yaml**

```
- given: a binding
  function: a_binding
- given: a (?:broken)? binding
  function: a_broken_binding
  regex: true
- given: a capitalised Binding
  function: os.getcwd
  case_sensitive: true
```

4.12.1 Steps which do not match bindings do not work

File: **nobinding.md**

````
---
title: No bindings available
bindings:
- badbindings.yaml
...
# Broken scenario because step has no binding

```scenario
given a missing binding
then nothing works
```
````

given file **nobinding.md**
and file **badbindings.yaml**
when I try to run sp-codegen --run **nobinding.md** -o **test.py**
then exit code is non-zero

4.12.2 Steps which do not case-sensitively match sensitive bindings do not work

File: **casemismatch.md**

````
---
title: Case sensitivity mismatch
bindings:
- badbindings.yaml
...
# Broken scenario because step has a case mismatch with sensitive binding

```scenario
given a capitalised binding
```
````

given file **casemismatch.md**
and file **badbindings.yaml**
when I try to run sp-codegen --run **casemismatch.md** -o **test.py**
then exit code is non-zero

4.12.3 Steps which match more than one binding do not work

File: **twobindings.md**

````
---
title: Two bindings match
bindings:
- badbindings.yaml
...
# Broken scenario because step has two possible bindings

```scenario
given a binding
```
````

given file **twobindings.md**
and file **badbindings.yaml**
when I try to run sp-codegen --run **twobindings.md** -o **test.py**
then exit code is non-zero

4.12.4 List embedded files

The `sp-meta` command lists embedded files in its output.

given file **two-embedded.md**
when I run sp-meta **two-embedded.md**
then output matches /**foo.txt**/
and output matches /**bar.yaml**/

File: **two-embedded.md**

````
---
title: Two embedded files
````

```
...

~~~{#foo.txt .file}
~~~

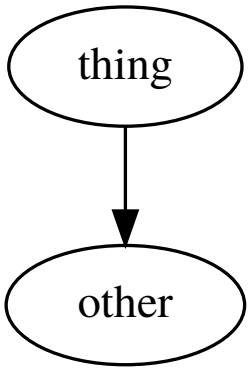
~~~{#bar.yaml. .file}
~~~
```

4.13 Embedded graphs

Subplot allows embedding markup to generate graphs into the Markdown document.

4.13.1 Dot

Dot is a program from the Graphviz[11] suite to generate directed graphs, such as this one.



The scenario checks that a graph is generated and embedded into the HTML output, not referenced as an external image.

given file **dot.md**
and file **b.yaml**
when I run pandoc --filter sp-filter **dot.md** -o **dot.html**
then file **dot.html** matches regex /**img src="**/

The sample input file **dot.md**:

File: **dot.md**

```
This is an example Markdown file, which embeds a graph using dot markup.

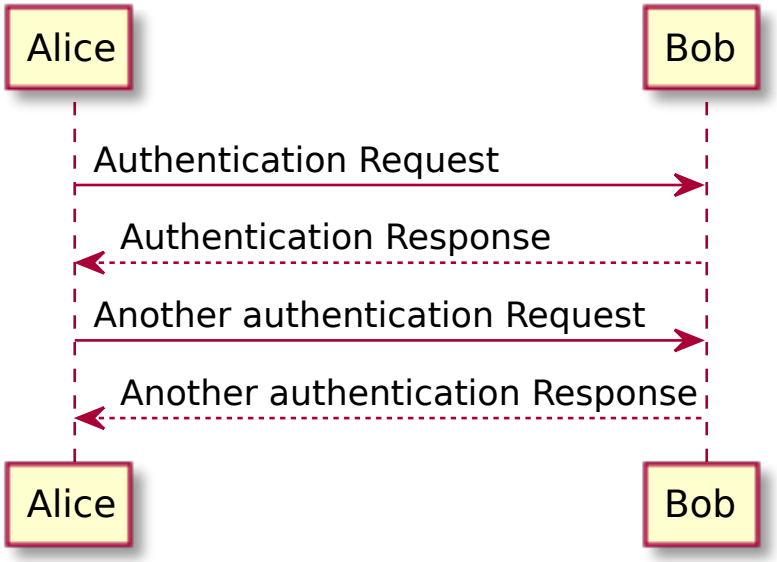
~~~dot
digraph "example" {
```

[11] http://www.graphviz.org/

```
thing -> other
}
~~~
```

4.13.2 PlantUML

PlantUML[12] is a program to generate various kinds of graphs for describing software, such as this one:



The scenario below checks that a graph is generated and embedded into the HTML output, not referenced as an external image.

given file **plantuml.md**
and file **b.yaml**
when I run pandoc --filter sp-filter **plantuml.md** -o **plantuml.html**
then file **plantuml.html** matches regex /**img src="**/

The sample input file **plantuml.md**:

File: **plantuml.md**

```
This is an example Markdown file, which embeds a graph using
PlantUML markup.

~~~plantuml
@startuml
Alice -> Bob: Authentication Request
Bob --> Alice: Authentication Response
```

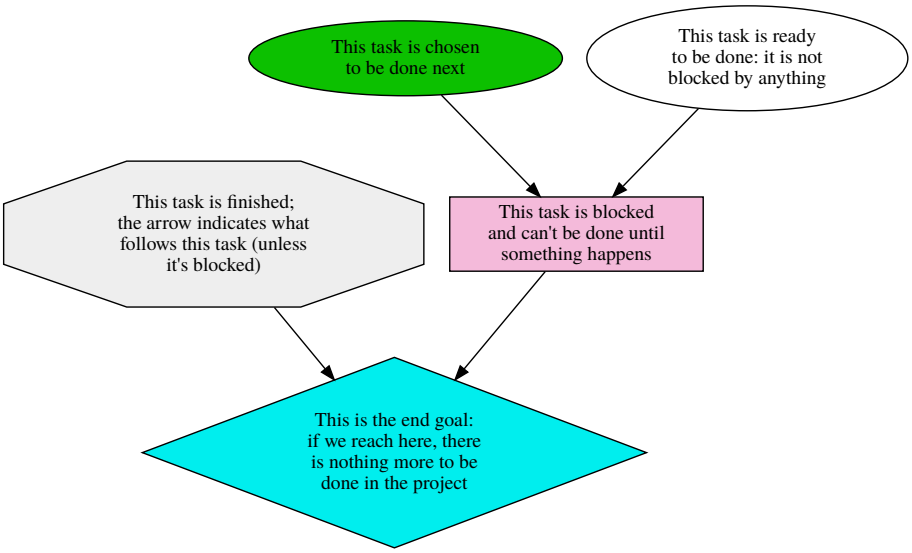
[12] https://plantuml.com/

```
Alice -> Bob: Another authentication Request
Alice <-- Bob: Another authentication Response
@enduml
~~~
```

4.13.3 Roadmap

Subplot supports visual roadmaps using a YAML based markup language, implemnted by the roadmap[13] Rust library. The library converts the roadmap into dot, and that gets rendered as SVG and embedded in the output document by Subplot.

An example:



This scenario checks that a graph is generated and embedded into the HTML output, not referenced as an external image.

given file **roadmap.md**
and file **b.yaml**
when I run pandoc --filter sp-filter **roadmap.md** -o **roadmap.html**
then file **roadmap.html** matches regex /**img src="**/

The sample input file **roadmap.md**:

File: **roadmap.md**

```
This is an example Markdown file, which embeds a roadmap.

```

[13] `https://crates.io/search?q=roadmap`

```
~~~roadmap
goal:
  label: |
    This is the end goal:
    if we reach here, there
    is nothing more to be
    done in the project
  depends:
  - finished
  - blocked

finished:
  status: finished
  label: |
    This task is finished;
    the arrow indicates what
    follows this task (unless
    it's blocked)

ready:
  status: ready
  label: |
    This task is ready
    to be done: it is not
    blocked by anything

next:
  status: next
  label: |
    This task is chosen
    to be done next

blocked:
  status: blocked
  label: |
    This task is blocked
    and can't be done until
    something happens
  depends:
  - ready
  - next
~~~
```

4.13.4 Class name validation

When Subplot loads a document it will validate that the block classes match a known set. Subplot has a built-in set which it treats as special, and it knows some pandoc-specific classes and a number of file type classes.

If the author of a document wishes to use additional class names then they can include a `classes` list in the document metadata which subplot will treat as valid.

given file **unknown-class-name.md**
and file **known-class-name.md**
and file **b.yaml**
when I try to run sp-docgen **unknown-class-name.md** -o **unknown-class-name.html**
then exit code is non-zero
and file **unknown-class-name.html** does not exist
and stderr matches /**Unknown classes found in the document: foobar**/
when I run sp-docgen **known-class-name.md** -o **known-class-name.html**
then file **known-class-name.html** exists

File: **unknown-class-name.md**

```
---
title: A document with an unknown class name
...

```foobar
This content is foobarish
```
```

File: **known-class-name.md**

```
---
title: A document with a previously unknown class name
classes:
- foobar
...

```foobar
This content is foobarish
```
```

4.14 Using as a Pandoc filter

Subplot can be used as a Pandoc *filter*, which means Pandoc can allow Subplot to modify the document while it is being converted or typeset. This can useful in a variety of ways, such as when using Pandoc to improve Markdown processing

in the ikiwiki[14] blog engine.

The way filters work is that Pandoc parses the input document into an abstract syntax tree, serializes that into JSON, gives that to the filter (via the standard input), gets a modified abstract syntax tree (again as JSON, via the filter's standard output).

Subplot supports this via the **sp-filter** executable. It is built using the same internal logic as Subplot's docgen. The interface is merely different to be usable as a Pandoc filter.

This scenarios verifies that the filter works at all. More importantly, it does that by feeding the filter a Markdown file that does not have a YAML metadata block. For the ikiwiki use case, that's what the input files are like.

given file **justdata.md**
when I run pandoc --filter sp-filter **justdata.md** -o **justdata.html**
then file **justdata.html** matches regex /**does not have a YAML metadata**/

The input file **justdata.md**:

File: **justdata.md**

```
This is an example Markdown file.
It does not have a YAML metadata block.
```

5 Extract embedded files

`sp-extract` extracts embedded files from a subplot file.

given file **embedded-file.md**
and file **expected.txt**
when I run sp-extract **embedded-file.md foo.txt** -d **.**
then files **foo.txt** and **expected.txt** match

File: **embedded-file.md**

```
---
title: Embedded file
...

~~~{#foo.txt .file}
This is a test file.
~~~
```

File: **expected.txt**

```
This is a test file.
```

[14] `http://ikiwiki.info/`