Acceptance testing using Fable

The Fable project Lars Wirzenius Daniel Silverstone

0.2-64-g5c15f80

Abstract

Fable is a tool that supports acceptance testing in two ways: it is a way to write and run automated acceptance tests, and also presents the acceptance test suite to non-expert readers as a human-readable text document.

This document explains Fable, its architecture, its input language, and specifies the acceptance criteria for Fable.

Contents

1 Beware **2**

2 Introduction **3**
2.1 A fairy tale of acceptance testing . . . 3
2.2 Motivation for Fable . . . 4

3 Requirements **5**

4 Fable architecture **6**

5 The Fable input language **8**
5.1 Markdown files . . . 8
5.2 Bindings . . . 10
5.3 Step implementations . . . 11

6 Acceptance tests for Fable **12**
6.1 Shared files between scenarios . . . 12
6.1.1 b.yaml—Bindings file . . . 12
6.1.2 f.py—Python file with functions . . . 12
6.2 The simplest possible scenario . . . 12
6.2.1 simple.md—markdown file . . . 13
6.3 A scenario step fails . . . 13
6.3.1 fail.md—markdown file . . . 13
6.4 Two scenarios in the same markdown file . . . 14
6.4.1 two.md—markdown file . . . 14
6.5 Two scenarios, one has a failing step . . . 14
6.5.1 onefails.md—markdown file . . . 15
6.6 Two scenarios, both have a failing step . . . 15
6.6.1 twofail.md—markdown file . . . 16

Chapter 1

Beware

This document describes a future that will be, not current status quo. Fable is its initial, active development period, and we don't want to update this document every time we make a change. Thus, be aware that parts of this document is speculative and may describe a version of Fable that does not yet exist.

Chapter 2

Introduction

Fable is a tool for acceptance testing of software. It helps describe acceptance criteria for all stakeholders in a software project, and also for computers. Such criteria may come from paying clients, users, the developers, their employers, or elsewhere.

Fable is specifically meant for describing automated acceptance tests. It takes a two-pronged approach, where it lets developers write automated tests for all the acceptance criteria they have, and runs the tests. On the other hand, Fable also produces a standalone document, which describes the automated tests all stakeholders, including non-technical ones, such as project managers and clients.

More concretely, Fable helps developers specify the automated acceptance tests so that the tests can be executed, but also understood without requiring programming knowledge to understand.

Fable is meant to be used to produce a document to facilitate communication between various shareholders of the software being developed.

Fable's overall working principle is that the tests are implemented and documented in a number of source files, which by two Fable tools. One tool produces a PDF or HTML document, for humans to read. The other produces a test program, which executes the acceptance tests.

2.1 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.

"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 than what you really, really wanted.

2.2 Motivation for Fable

Keeping track of requirements and acceptance criteria is necessary for all but the simplest of software projects. Having all stakeholders in a projects agree to them is crucial, as is that all agree how it is verified that the software meets the acceptance criteria. Fable aims to provide 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 develop the system in the first place ("developer")

All stakeholders need to understand the acceptance criteria, and how the system is evaluated against them. 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 Fable approach 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 Fable 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 Fable output document to be good and helpful, writing it will require effort and skill. No tool can replace that.

Chapter 3

Requirements

This chapter lists requirements for Fable. These requirements are not meant to be testable as such. For more specific, testable acceptance criteria, see the later chapter with acceptance tests for Fable.

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

UnderstadableTests Acceptance tests should be possible to express in a way that's easily understood by non-programmers.

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

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

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

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

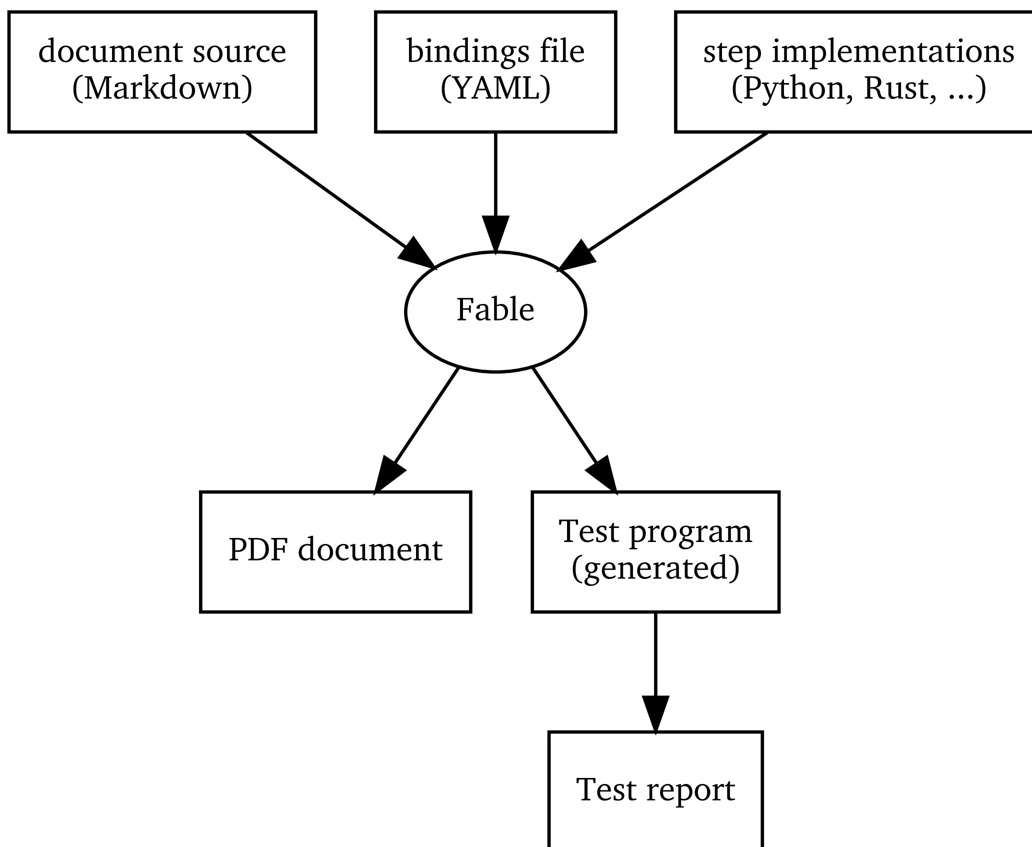
FastTestExecution Executing the acceptance tests should be fast.

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.

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 Fable. 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.

Chapter 4

Fable architecture



Fable reads input files, and produces two outputs. On the one hand, it outputs a program that executes the tests specified in the input files. The person running Fable then runs the test program to get a test report, with results of each of the test scenarios. On the other hand it outputs a human-readable document (as PDF or HTML), for communicating what is being tested.

Fable is able to produce the test program in various languages, using a templating system to make it simple to add support for more languages. Fable comes with support for Python and Rust. It's the user's choice which language they're most comfortable with.

Acceptance tests are expressed to Fable in the form of test scenarios, in which a sequence of actions are taken, and then the results are checked. If the checks fail, the scenario fails.

Fable runs the scenarios concurrently (but see the USING keyword), within the constrains of hardware resources. If Fable determines it doesn't have all the resources to run all scenarios at once, it will run fewer, but randomly chosen scenarios concurrently, to more likely to detect unintentional dependencies between scenarios.

Note that Fable or the scenarios it runs aren't meant deploy the software, or start services to run in the background.

Chapter 5

The Fable input language

Fable input consists of three types of files:

- one or more markdown files which document the acceptance tests
- a binding file (in YAML) which binds scenario steps to their implementations
- scenario step implementations, which are implemented in some programming language (e.g., Python or Rust); Fable will combine this code with some scaffolding provided by Fable itself

The input files for a simple acceptance test suite for Fable would be divided into three files: `foo.md`, `foo.yaml`, and `foo.py` (assuming step implementation in Python).

5.1 Markdown files

The Fable input language is markdown files using fenced code blocks with backticks. The code blocks must indicate that they contain Fable language:

````
```fable
given a service
and I am Tomjon
when I access the service
then it's OK
```
````

Code blocks for the PlantUML or GraphViz dot markup languages are also supported. These can be used for generating embedded images in the documented produced by Fable.

Any unfenced code blocks (indented code blocks) are ignored by Fable, as are fenced code blocks using unknown content types.

The Fable code generator understands the full Markdown language, by ignoring everything except headings and its own code blocks. The Fable document generator uses Pandoc to produce standalone files, and anything that Pandoc supports is OK to use.

Fable treats multiple Markdown files as one, as if they had been concatenated with the **cat**(1) utility. Within the logical file, normal Markdown and Pandoc markup can be used to structure the document in

any way that aids human understanding of the acceptance test suite, which the caveat that chapter or section headings are used by Fable to group code blocks into scenarios.

All code blocks for the same scenario must be grouped under a single heading. Sub-headings are permitted within a scenario, but the next heading at the same or a higher level will end the scenario. This allows for scenarios to begin at any level, but not to leak into a wider scope within the acceptance document. For example, a scenario which starts after a level 2 heading may have subdivisions marked with level 3 or below headings, but will end at the next level 2 or level 1 heading.

Within the Fable code blocks, Fable understands a special language, derived from Gherkin, as defined by the Cucumber testing tool. The language understood by Fable has the following general structure:

- each logical line starts with a keyword at the beginning of the line
- logical lines may be broken into physical lines, by starting the continuation lines with one or more space or TAB characters; the physical line break and white space characters are preserved
- logical lines define steps in a test scenario
- the meaning and implementation of the steps are defined by other Fable input files
- the keywords are: ASSUMING, USING, GIVEN, WHEN, THEN, and AND, with meanings defined below; keywords can be written in upper or lower case, or mixes, Fable doesn't care

The keywords have the following meanings:

assuming A condition for the scenario. If an ASSUMING step fails, the scenario is skipped.

This is used for skipping test scenarios that require specific software to be installed in the test environment, or access to external services, but which can't be required for all runs of the acceptance tests.

using Indicate that the scenario uses a resource such as a database, that's constrained and can't be used by all scenarios if they run concurrently. When scenarios declare the resource, Fable can limit which scenarios run concurrently.

For example, if several scenarios require uncontested use of the GPU, of which there is typically only one per machine, they can all declare "using the graphical processing unit", and Fable will run them one at a time.

(This is an intentionally simplistic way of controlling concurrency. The goal is to be simple and correct rather then achievee maximal concurrency.)

The actual management of resources belongs to the generated test program at runtime, not the Fable compiler.

given Set up the test environment for the action (WHEN). This might create files, start a background process, or something like that. This also sets up the reversal of the setup, so that any background processes are stopped automatically after the scenario has ended. The setup and cleanup must succeed, or the scenario will fail.

The cleanups are executed in the reverse order of the GIVENs, and they're done always, whether the scenario succeeds or not.

when Perform the action that is being tested. This must succeed. This might, for example, execute a command line program, and capture its output and exit code.

then Test the results of the action. This would examine the output and exit code of the program run in a WHEN step, or examine current content of the database, or whatever is needed.

and This keyword exists to make scenarios "read" better in English. The keyword indicates that this step should use the same keyword as the previous step, whatever that keyword is. For example, a step "THEN output is empty" might be followed by "AND the exit code is 0" rather than "THEN the exit code is 0".

The order of steps is somewhat constrainted: first any ASSUMING steps, then any USING steps, at least one WHEN must come before a THEN.

5.2 Bindings

FIXME: The binding specification needs thought. This is just a sketch.

Binding files match scenario steps to functions that implement them, using regular expressions. The bindings may also extract parts of the steps, and pass them onto the functions as parameters.

Binding files are YAML files, with lists of bindings, each binding being a dict. For example:

```
- define:
    name: string
    exit_code: int

- pattern: given a service
  function: start_service
  cleanup: stop_service

- pattern: given I am (?P<name\S+)
  function: set_name
  produces: [name]

- pattern: when I access the service
  function: access_service
  requires: [name]
  produces: [exit_code]

- pattern: then it's OK
  function: check_access_was_ok
  requires: [exit_code]
```

In the example above, the "I am" step extracts the name of the user from the step. It's type is declared, and the value is saved for use by a later step.

The "I access" step expects the name to have been set by a previous step. Fable will check that the name is set, and give an error if it isn't, before any scenario runs. If name is set, it is given to the function to be called as a function argument.

The "I access" step further sets the variable "exit_code", and the "it's OK" step expects it to be set.

5.3 Step implementations

Continuing the example from the previous section, the following Python code might implement the functions:

```
def start_service():
    ...

def set_name(**matches):
    ...

def access_service(name):
    ...
    return {
        'exit_code': 0,
    }

def check_access_was_ok(exit_code):
    assert exit_code == 0
```

With these bindings, Fable produces a Python program, which calls these functions in order, and passes values between them via function arguments and return values. The generated program will handle running scenarios concurrently, and taking care of USING constraints, and other resource constraints.

Chapter 6

Acceptance tests for Fable

6.1 Shared files between scenarios

The scenarios will use the files under different names.

6.1.1 b.yaml—Bindings file

```yaml
- given: precondition foo
  function: precond
- when: I do bar
  function: do
- then: bar was done
  function: was_done
```

6.1.2 f.py—Python file with functions

```python
state = {'done': False}
def precond(ctx):
    pass
def do(ctx):
    state['done'] = True
def was_done(ctx):
    assert state['done']
```

6.2 The simplest possible scenario

This tests that Fable 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 log file.

> *given* files *simple.md*, *simple.yaml* (from *b.yaml*), and *simple.py* (from *f.py*)

when I run ftt-docgen –pdf *simple.md*
then file *simple.pdf* exists

when I run ftt-docgen –html *simple.md*
then file *simple.html* exists

when I run ftt-codegen –run *simple.md*
then log file says scenario "*Simple*" was run
and log file says step "*given precondition foo*" was run
and log file says step "*when I do bar*" was run
and log file says step "*then bar was done*" was run
and program finished successfully

6.2.1 simple.md—markdown file

````
# Simple
This is the simplest possible test scenario

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

6.3 A scenario step fails

This tests that Fable can run handle a scenario step failing.

given files *fail.md*, *fail.yaml* (from *b.yaml*), and *fail.py* (from *f.py*)
when I run ftt-codegen –run *fail.md*

then log file says scenario "*Fail*" was run
and log file says step "*given precondition foo*" was run
and log file says step "then bar was done" failed
and program finished with an error

6.3.1 fail.md—markdown file

````
# Fail
This is a scenario that fails.

```fable
given precondition foo
then bar was done
```
````

6.4 Two scenarios in the same markdown file

This tests that Fable can run two successful scenarios in the same Markdown file successfully.

FIXME: This and all other tests that run more than one scenario should handle scenarios being run in random order, and concurrently.

> *given* files *two.md*, *two.yaml* (from *b.yaml*), and *two.py* (from *f.py*)
> *when* I run ftt-codegen –run *two.md*
>
> *then* log file says scenario "*First*" was run
> *and* log file says step "*given precondition foo*" was run
> *and* log file says step "*when I do bar*" was run
> *and* log file says step "*then bar was done*" was run
>
> *then* log file says scenario "*Second*" was run
> *and* log file says step "*given precondition foo*" was run
> *and* log file says step "*when I do bar*" was run
> *and* log file says step "*then bar was done*" was run
>
> *and* test program finished successfully

6.4.1 two.md—markdown file

````
# First
This is the simplest possible test scenario

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

# Second
This is another scenario.

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

6.5 Two scenarios, one has a failing step

This tests that Fable runs both scenarios, even if one has a failing step.

> *given* files *onefails.md*, *onefails.yaml* (from *b.yaml*), and *onefails.py* (from *f.py*)
> *when* I run ftt-codegen –run *onefails.md*

then log file says scenario "*First*" was run
and log file says step "*given precondition foo*" was run
and log file says step "*when I do bar*" was run
and log file says step "*then bar was done*" was run

then log file says scenario "*Second*" was run
and log file says step "*given precondition foo*" was run
and log file says step "then bar was done" failed

and program finished with an error

6.5.1 onefails.md—markdown file

````
# First
This is the simplest possible test scenario

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

# Second
This is another scenario. This fails.

```fable
given precondition foo
then bar was done
```
````

6.6 Two scenarios, both have a failing step

This tests that Fable runs both scenarios, even when both have a failing step.

given files *twofail.md*, *twofail.yaml* (from *b.yaml*), and *twofail.py* (from *f.py*)
when I run ftt-codegen –run *twofail.md*

then log file says scenario "*First*" was run
and log file says step "*given precondition foo*" was run
and log file says step "then bar was done" failed

then log file says scenario "*Second*" was run
and log file says step "*given precondition foo*" was run
and log file says step "then bar was done" failed

and program finished with an error

6.6.1 twofail.md—markdown file

````
# First
This is a failing scenario.

```fable
given precondition foo
then bar was done
```

# Second
This is another scenario. This fails.

```fable
given precondition foo
then bar was done
```
````