Acceptance testing using Saga

Lars Wirzenius Daniel Silverstone

Abstract

Saga is a tool that supports acceptance testing in two ways: it is a way to implement 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 Saga and its input language.

Contents

1 Introduction **2**

2 Saga architecture **3**

3 The Saga input language **4**
3.1 Markdown files 4
3.2 Bindings 5
3.3 Step implementations 6

Chapter 1

Introduction

Saga is a tool for acceptance testing of software. This means it helps software developers and teams to make sure their software fulfils the acceptance criterita for the software. Such criteria may come from users, the developers, their employers, or other elsewhere.

Saga is specifically mean for automated acceptance testing. It takes a two-pronged approach, where it lets developers implement automated tests for all the acceptance criteria they have, and runs the tests. On the other hand, Saga also produces a PDF file, which documents the automated tests for non-technical stakeholders.

More concretely, Saga helps developers implement and document their automated acceptance tests in a way that, at the same time, helps the developers automatically test their software, and write documention for the tests in way that doesn't require programming knowledge to understand.

Saga is meant to be a tool for developers, who use it to produce a document, which is meant to facilitate communication between various shareholders of the software being developed.

Saga's overall working principle is that the tests are implemented and documented in a number of source files, which Saga reads to execute tests, and to produce a PDF document for non-developer consumption.

Chapter 2

Saga architecture

Saga's architecture it to read input files, and produce two outputs. On the one hand, it outputs a program that executes the tests specified in the input files. On the other hand it outputs a human-readable document (as PDF), for communicating what is being tested.

The approach is that of a compiler. Saga doesn't act as an interpreter, which executes each test as it runs, and instead produces a program (or the source code of a program) which can execute all the tests. Instead, Saga produces a complete program, which performs all the tests, when it's run.

Saga will be able to produce the test program in various languages, probably using some form of templating that is partially under user control. This way, Saga can support, say, Python and Rust, and it's the user's choice which language they're most comfortable with.

Acceptance tests are expressed to Saga 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.

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

Chapter 3

The Saga input language

Saga input consists of three types of files:

- markdown file which document that acceptance tests; these are independent of the step implementation language
- binding files which bind specific scenario steps to their implementations; these are also independent of the implementation language
- scenario step implementations, which are implemented in a specific programming language (e.g., Python or Rust)

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

3.1 Markdown files

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

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

Any other code blocks are ignored by Saga.

Saga understands the full Markdown language, mostly by ignoring everything except its own code blocks and headings. It uses Pandoc to produce PDF files, and anything that Pandoc supports is OK to use.

Saga can treat 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 Saga to group code blocks into scenarios. All code blocks for the same scenario MUST be under the same heading, and there can be no sub-headings inside the scenario. It doesn't matter if the heading is chapter, section, subsection, or deeper, and different scenario headings can be at different levels, as long as each scenario has no subdivisions.

FIXME: Discuss whether it would be useful for Saga to support, say, PlantUML and Graphviz code blocks for graphs and stuff.

Within the Saga code blocks, Saga understands a special language, derived from Gherkin, as defined by the Cucumber testing tool. The language understood by Saga 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 whitespace characters are preserved
- logical lines define steps in a test scenario
- the meaning and implementation of the steps are defined by other Saga input files
- the keywords are: ASSUMING, USING, GIVEN, WHEN, THEN, AND, with meanings defined below; keywords can be written in upper or lower case, or mixes, Saga doesn't care

The keywords have the following meanings:

- ASSUMING—a condition for the scenario; all ASSUMING steps MUST be at the beginning of the scenario, and may not be preceded by other steps; if the condition fails, the scenario is skipped.

  This is meant to be used for things like 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, Saga can limit which scenarios run concurrently.

  For example, if several test 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 Saga 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 get maximal concurrency.)

- 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.

- 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".

3.2 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:

```
- given: a service
  function: start_service

- given: I am (name:\S+)
  match:
    name:
      type: string
      save: true

- when: I access the service
  function: access_service
  require:
    name: string
  produces:
    exit_code: int

- then: it's OK
  function: check_access_was_ok
  require:
    exit_code: int
```

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. Saga 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.

3.3 Step implementations

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

```
def start_service():
    ...

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

def check_access_was_ok(exit_code=None):
    assert exit_code == 0
```

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