Brine

Release 0.13.0-SNAPSHOT

Apr 22, 2020

User Guide

A Cucumber based DSL for testing REST APIs.

CHAPTER 1

How to Read This Documentation

Brine's documentation is broken into a few major sections where the descriptions here should help indicate which section is the best resource for a particular type of need. If there are questions which are not answered after consulting the appropriate section of documentation, then please file a documentation bug.

User Guide The User Guide provides an introduction to the use and underlying concepts of Brine. The guide also includes a reference list of what is exposed by Brine. The User Guide therefore acts as a high level overview to the standard functionality provided by Brine.

Specification Brine specifications are the living, definitive reference for the Brine DSL, and should be able to answer any specific questions about the behavior of the provided DSL (or any other behavior which is not runtime-specific). The specification should therefore be able to answer any lower level questions about standard Brine behavior but is unlikely to provide guidance in terms of practical application of that functionality.

Articles Articles will be used to provide more targeted information about using Brine. This will include techniques, ideas, and practices around using Brine, how to use Brine with other technologies, and more in-depth explorations of specific use cases.

1.1 Introduction

1.1.1 Motivation

REpresentational State Transfer APIs expose their functionality through combinations of fairly coarse primitives that generally revolve around the use of transferring data in a standard exchange format (such as JSON) using HTTP methods and other aspects of the very simple HTTP protocol. Tests for such an API can therefore be defined using a domain specific language (DSL) built around those higher level ideas rather than requiring a general purpose language (the equivalent of scripted *curl* commands with some glue code and assertions). This project provides such a DSL by using select libraries integrated into Cucumber, where Cucumber provides a test-oriented framework for DSL creation.

1.1.2 Sample Usage

The general usage pattern revolves around construction of a request and performing assertions against the received response.

```
When the request body is assigned:
  """
  {"first_name": "John",
   "last_name": "Smith"}
  """
And a POST is sent to `/users`
Then the value of the response status is equal to `201`
And the value of the response body is including:
  """
  {"first_name": "John",
   "last_name": "Smith"}
  """
```

1.1.3 Key Features

Request Construction and Response Assertion Step Definitions The most essential feature is that which is menthoed above: the ability to issue constructed requests and validate responses. The other features largely supplement this core functionality to support a wider range of use cases.

Variable Binding/Expansion To support cases where dynamic data is in the response or is desired for the request, values can be bound to identifiers which can then be expanded using Mustache templates in your feature files.

Type Conversion and Coercion Different types of data can be expressed directly in the feature files or expanded into variables by using the appropriate syntax for that type. A facility it coerce types to allow more intelligent comparison of calues is also provided. This allows for implicit conversion of simpler values (normally strings) when paired with an operand of a richer type (such as a date/time).

Resource Cleanup Tests are likely to create resources which should then be cleaned up so that the system can be restored to its pre-test state. Steps to facilitate this are provided.

Common Client Behavior Clients can be configured to adopt standard behavior such as adding headers to satisfy security requirements.

Definition of Reusable Actions Rather than executing actions immediately they can be stored for subsequent evaluation. This is currently used to support polling behavior to allow for delayed convergence (such as when using eventually consistent systems), but the mechnism can be extended to provide a range of functionality.

1.2 Installation

1.2.1 Using Docker

A Docker image is provided which can be used to execute any test suite that doesn't require additional dependencies (it could also assist with projects that do have additional dependencies, but that is undocumented and may be better addressed by a custom image).

This can be done most simply by bind mounting the directory of feature files to the `/features` directory within the container, such as:

```
docker run -v /my/test/suite:/features mwhipple/brine:0.13-ruby
```

The `CUCUMBER_OPTS` environment variable can be used to pass additional arguments to cucumber. The `FEATURE` environment variable specifies the container path to the feature file if for some reason a value other than the default `/features` is desired.

1.2.2 Using the Ruby Gem

Brine is published as brine-dsl on rubygems. The latest version and other gem metadata can be viewed there. Brine can be used by declaring that gem in your project `Gemfile` such as:

```
gem 'brine-dsl', "~> #{brine_version}"
```

where `brine_version` is set to the desired version, currently '0.13'.

Brine can then be "mixed in" to your project (which adds assorted modules to the `World` and loads all the step definitions and other Cucumber magic) by adding the following to your `support/env.rb` or similar ruby file:

```
require 'brine'
World(brine_mix)
```

Select pieces can also be loaded (to be documented). With the above, feature files should be able to be written and executed without requiring any additional ruby code.

1.3 Tutorial

To demonstrate typical usage of Brine, let's write some tests against http://myjson.com/api (selected fairly arbitrarily from the list at https://github.com/toddmotto/public-apis).

1.3.1 Setting BRINE_ROOT_URL

Brine expects steps to use relative URLs. The feature files specify the behavior of an API (or multiple APIs), while the root of the define where to reach that API, so this should align with a natural separation of API behavior and location. More practically an API is likely to exist across various environments such as local, qa, stage, and production; having a parameterized root for the URLs eases switching between these while encouraging inter-environment consistency.

When all tests are to be run against the same root url, the value of the root can be specified with the environment variable *`BRINE_ROOT_URL`*. This can be set when running Cucumber with a command such as:

```
BRINE_ROOT_URL=https://api.myjson.com/ cucumber
```

or by any other means that populates the environment appropriately.

A personally preferred approach is to have per-environment make files and include the desired file(s) into the main make file with a line such as:

1.3.2 A Basic GET

Most tests will involve some form of issuing requests and performing assertions on the responses. Let's start with a simple version of that pattern: testing the response status from a GET request.

```
Feature: Absent resources return 404s.

  Scenario: A request for a known missing resource.
    When a GET is sent to `/bins/brine-absent`
    Then the value of the response status is equal to `404`
```

1.3.3 A Write Request

For POST, PATCH, and PUT requests you'll normally want to include a request body. To support this, additional data can be added to requests before they are sent.

```
Feature: A POST returns a 201.

  Scenario: A valid post.
    When the request body is assigned:
      """
      {"name": "boolean-setting",
       "value": true}
      """
    And a POST is sent to `/bins`
    Then the value of the response status is equal to `201`
```

See also:

Steps *Request Construction*

1.3.4 Test Response Properties

http://myjson.com/api returns the link to the created resource which is based off of a generated id. That means the exact response cannot be verified, but instead property based testing can be done to verify that the data is sane and therefore likely trustworthy. In this case we can check that the `uri` response child matches the expected pattern.

```
Feature: A POST returns a link to the created resource.

  Scenario: A valid post.
    When the request body is assigned:
      """
      {"name": "boolean-setting",
       "value": true}
      """
    And a POST is sent to `/bins`

    Then the value of the response status is equal to `201`
    And the value of the response body child `uri` is matching `/https://api.myjson.
↪com/bins/\w+/`
```

1.4 Concepts

1.4.1 The use of `s

Backticks/grave accents are used as *parameter delimiters*. It is perhaps most helpful to think of them in those explicit terms rather than thinking of them as an alternate *quote* construct. "Quoting" may invite the assumption that the

parameter value is a string value while the type transformation allows for alternative data types.

`s were chosen as they are less common than many other syntactical elements and also allow for the use of logically significant quoting within parameter values while hopefully avoiding the need for escape artistry.

1.4.2 Selection and Assertion

As tests are generally concerned with performing assertions, a testing DSL should be able to express the variety of assertions that may be needed. Because these are likely to be numerous, it could easily lead to duplicated logic or geometric growth of code due to the combinations of types of assertions and the means to select the inputs for the assertion.

To avoid this issue the concepts of selection and assertion are considered separate operations in Brine. Internally this corresponds to two steps:

1. Assign a selector.
2. Evaluate the assertion against the selector.

In standard step use this will still be expressed as a single step, and dynamic step definitions are used to split the work appropriately.

For example the step:

```
Then the value of the response body is equal to `foo`
```

will be split where the subject of the step (`the value of the response body`) defines the selector and the predicate of the step `is equal to `foo`` defines the assertion (which is translated to a step such as `Then it is equal to `foo``).

The support this the assertion steps will always follow a pattern where the subject resembles `the value of ...` and the predicate always resembles `is ...`. Learning the selection phrases, the assertion phrases , and how to combine them should be a more efficient and flexible way to become familiar with the language rather than focusing on the resulting productions.

The chosen approach sacrifices eloquence for the sake of consistency. The predicate will always start with a variation of "to be" which can lead to awkward language such as `is including` rather than simply `includes`. The consistency provides additional benefits such as consistent modification; the "to be" verb can always be negated (to `is not`), for example, rather than working out the appropriate phrasing for a more natural sounding step (let alone the logic).

One of the secondary goals of this is that assertion step definitions should be very simple to write and modifiers (such as negation) should be provided for free to those definitions. As assertion definitions are likely to be numerous and potentially customized, this should help optimize code economy.

Selection Modifiers

In the pursuit of economical flexibility Brine steps attempt to balance step definitions which accommodate variations while keeping the step logic and patterns fairly simple. Selection steps in particular generally accept some parameters that affect their behavior. This allows the relatively small number of selection steps to provide the flexibility to empower the more numerous assertion steps.

Traversal

Selection steps can normally target the root of the object specified (such as the response body) or some nodes within the object if it is a non-scalar value (for instance a child of the response body). This is indicated in the *Step Reference* by

the *[TRAVERSAL]* placeholder. `child` *EXPRESSION* or `children` *EXPRESSION* can optionally be inserted at the placeholder to select nested nodes as described in *traversal*.

Negation

Negation is indicated in the *Step Reference* by the presence of a *[not]* or semantically equivalent placeholder. To negate the step the literal text within the placeholder should be included at the indicated position.

1.4.3 Handling Nested Elements

Using brine should provide easy to undertand tests. Given:

```
When the request body is assigned:
  """
  {"name":"Jet Li",
   "skills":"Being the one, Multiverse-homicide"}
  """
And a POST is sent to `/people`
```

A check on skills could follow up, with the response returning the created object inside an object with `data` and `links` sub-objects (Hypermedia API):

```
Then the value of the response status is equal to `201`
And the value of the response body child `data.skills` is a valid `Array`
And the value of the response body child `data.skills` is including:
  """
  "Multiverse-homicide"
  """
And the value of the response body child `data.skills` is including:
  """
  "Being the one"
  """
```

The above example uses child comparison against type and value, and verifies multiple elements from the body. This can be useful if your response contains HATEOAS (Hypermedia As The Engine Of Application State) links. The end goal is that anyone reading the specification will be able to ascertain without Cucumber or DSL knowledge what the intent is.

If order can be guaranteed then checks could be combined into a simpler format:

```
Then the value of the response status is equal to `201`
And the value fo the response body child `data.skills` is a valid `Array`
And the value of the response body child `data` is including:
  """
  {"skills":["Being the one", "Multiverse-homicide"]}
  """
```

Todo: This should also be supported through pending set equality assertions.

On a more serious note, the above could also be used to verify business logic such as for medical professionals working with large insurers or healthcare where the line-items usually have to be sorted by price descending.

1.4.4 Resource Cleanup

All test suites should clean up after themselves as a matter of hygiene and to help enforce test independence and reproducibility. This is particularly important for this library given that the systems under test are likely to remain running; accumulated uncleaned resources are at best a nuisance to weed through, and at worst can raise costs due to heightened consumption of assorted resources (unlike more ephemeral test environments).

Brine therefore provides mechanisms to assist in cleaning up those resources which are created as part of a test run. A conceptual hurdle for this type of functionality is that it is very unlikely to be part of the feature that is being specified, and therefore should ideally not be part of the specification. Depending on the functionality (and arguably the maturity) of the API, most or all of the cleanup can be automagically done based on convention. There are tentative plans to support multiple techniques for cleaning up resources based on how much can be implicitly ascertained. . . though presently there exists only one.

Step indicating resource to DELETE

If the API supports DELETE requests to remove created resources but it is either desirable or necessary to specify what those resource PATHS are, a step can be used to indicate which resources should be DELETEd upon test completion.

See also:

Steps *Resource Cleanup*

1.4.5 Actions

Brine offers the ability to define a bundle of _actions_ which can be later evaluated.

Restricted Official Usage

This functionality could be used to support a wide range of functionality, but functionality will be added to the core library conservatively to address actual issues encountered or specific cases identified by opened issues. Reservation to add such features is due to YAGNI with an additional concern that some of that functionality could dillute the focus of this library.

Such functionality which is not offered by the official library can leverage the _actions_ feature and be implemented with a fairly small amount of code; more information will be provided through Articles.

Supported Functionality

Polling

For any system which may perform background work or uses a model of eventual consistency there may be a delay before the expected state is realized. To support such cases Brine supports the concept of polling. Polling allows the definition of a set of actions which will be repeated until they succeed or until some duration of time expires (at which point last failure will be returned).

For example a code block such as:

```
When actions are defined such that
  When a GET is sent to `/tasks/{{task_id}}/status`
  Then the value of the response body child `completed` is equal to `true`
And the actions are successful within a `short` period
```

will repeatedly issue a request to the specified status endpoint until the resource indicates it is completed. The indendation is not required but may help readability. It is important that any such actions definition is *closed* (something is done with the actions such as the outdented polling step above), otherwise the system will just continue to collect actions.

Specifying Duration

As mentioned above, polling will be bound by a duration withiin which the actions must be satisfied.

Specifications should represent the contract with customers, and therefore any delay captured in the specification should correspond to what is guaranteed to clients.

If the system under test has a duration within which it is guaranteed that the tested state must be realized then such time should be in the specification and parsed from that file; such parsing is not currently supported so an issue should be opened if it is desired. In other cases the duration should be specified using an appropriately fuzzy term (such as *short*) which can be passed as a parameter to the test execution. The durations can be defined using environment variables of the format `BRINE_DURATION_SECONDS_${duration}`; for the above a setting such as *BRINE_DURATION_SECONDS_short=5* would poll for 5 seconds. In addition to not polluting the specification with what may not belong there the use of such looser terms allows for values to vary to accommodate differences between environments or deployments.

Specifying Polling Frequency

A reasonable default value for the interval between polling attempts will be set: currently 0.25 seconds. If for any reason it is desired to change this time then a new value can be provided as the `BRINE_POLL_INTERVAL_SECONDS` environment variable.

Currently all polling will use the same global setting for the polling interval. If there is a desire to have finer control, then open an issue (most likely support for per-duration overrides would be added).

Note: The interval will affect the precision of the polling duration. With the numbers in the example above a naive view would assume that the intervals will fit neatly into the duration with a maximum of $(5.0/0.25 + initial)$ 21 attempts, but each execution will take some time, and a sleeping thread will be activated in *no less than* the time requested. Therefore the polling will not align with the duration and the interval also determines how much the effective polling duration deviates from that requested. The values should be adjusted/padded appropriately to allow for such slop (anticipate $duration +/- interval$). Precise matching of durations is non-trivial and outside the scope of this project.

1.5 Traversal

The language exposed by Brine is flat but the data returned by the server is likely to include deeper data structures such as objects and collections. To allow selection within such structures a **traversal** language is embedded within some steps which will be indicated by the use of the `TRAVERSAL` placeholder.

The traversal language consists of a selected subset of JsonPath.

1.5.1 JsonPath Subset

A subset of JsonPath functionality will be officially supported that is believed to cover all needed use cases without requiring deep familiarity with JsonPath. This may lead to more numerous simple steps in place of fewer steps that use

unsupported expressions. The simpler steps should result in specifications that are clearer: both in terms of being more readable and also having more precisely defined logic. Brine is designed to be ported to multiple runtimes and only the selected subset will be supported across those runtimes (which should facilitate porting to any runtimes for which a more complete JsonPath implementation does not exist). Any expressions not listed here will not be disallowed and will work if handled by the JsonPath implementation, but are not officially supported.

1.5.2 Cardinality

Each traversal expression will select *all* matching nodes which is therefore represented as a collection. Often, however, only a single node is expected or desired. Therefore the traversal expression will also be accompanied by a phrase which defines the expected cardinality, normally `child` and `children`. `children` will *always* return an array corresponding to all matched values while `child` will return the first matched value. When the targeted values is one or more arrays then `children` would return an array of such arrays whereas `child` would return the first such array. The distinction is therefore about the number of children matching the traversal path and not the type of such matched values: an single array should be retrieved with `child`, extracting an array of specific attribute collected from an array of objects should be done with `children`.

1.5.3 Expressions

`.`*`KEY`* Access the child named *`KEY`* of the target node. The leading **`.`** can be omitted at the start of an expression.

`.[`*`INDEX`*`]` Access the element of the array at index *`INDEX`*.

`.[`*`FROM`*`:`*`TO`*`]` Access a slice of the array containing the elements at index *`FROM`* through *`TO`* (including both limits). TODO: Are both really included? A half-open interval seems more standard.

1.6 Environment Variables

The primary channel for tuning Brine's behavior is through environment variables. Environment variables are used since they can be consistently used across a wide range of technologies and platforms, and align with the ideas espoused by The Twelve Factor App.

`BRINE_VAR_${name}`
: User defined variables can be provided to a Brine execution by assigning them to an environment variable with a name of `BRINE_VAR_`*`$name`*. Such variables will be available in the binding environment with the name specified (without the `BRINE_VAR_` prefix) and can therefore be used in template expansions. For example an environment variable set such as `BRINE_VAR_account_id=123` could then be used in a step such as:

```
When a GET is sent to `/accounts/{{account_id}}/resources`
```

This would result in a request being sent to */accounts/123/resources*. Note that the the prefix is stripped but no form of case folding or other normalization is done: therefore the name referenced in the template and the environment variable with the prefix removed should match exactly.

`BRINE_ROOT_URL`
: The location of the default API being tested. Requests will be sent to URLs of the pattern *`${BRINE_ROOT_URL}${PATH}`* where *`PATH`* is provided by the feature steps. Including path segments as part of the prefix in `BRINE_ROOT_URL` is currently **not** supported, only non-path components should be specified (*`scheme`*`://`*`hostname`*`[:`*`port`*`]`). For example if an API has a context root path of `v1` where all paths are similar to `http://www.example.com/v1/...`, then `v1` cannot be added to `BRINE_ROOT_URL` and **must** be included in each feature step (or another workaround could be implemented). This has not yet presented itself as a problem; if it does then please file an issue and a solution can be implemented. A caveat to considering this an issue is that it may lead to specifications that are less expressive; any path prefix is likely necessary for

calling the API and implicit prefixes may lead to the specification becoming a less useful definition of how to use the API (simple, consistent usage is unlikely to breed confusion, but attempting to DRY up multiple common prefixes may obfuscate the underlying API).

BRINE_LOG_HTTP
Output HTTP traffic to stdout. Any truthy value will result in request and response metadata being logged, a value of `DEBUG` (case insensitive) will also log the bodies.

BRINE_LOG_BINDING
Log values as they are assigned to variables in Brine steps.

BRINE_LOG_TRANSFORMS
Log how parameter inputs are being transformed.

BRINE_DURATION_SECONDS_${duration}
How long in seconds *Polling* will be done when *`duration`* is specified.

BRINE_POLL_INTERVAL_SECONDS
The amount of time to wait between attempts when *Polling*.

1.7 Step Reference

1.7.1 Request Construction

Specification

The requests which are sent as part of a test are constructed using a builder.

When a *`METHOD`* is sent to `*`PATH`*` As every request to a REST API is likely to have a significant HTTP *`METHOD`* and *`PATH`*, this step is considered required and is therefore used to send the built request. This should therefore be the *last* step for any given request that is built.

When the request body is assigned: The multiline content provided will be assigned to the body of the request. This will normally likely be the JSON representation of the data.

When the request query parameter `*`PARAMETER`*` is assigned `*`VALUE`*` Assign *`VALUE`* to the request query parameter *`PARAMETER`*. The value will be URL encoded and the key/value pair appended to the URL using the appropriate `?` or `&` delimiter. The order of the parameters in the resulting URL should be considered undefined.

When the request header `*`HEADER`*` is assigned `*`VALUE`*` Assign *`VALUE`* to the request header *`HEADER`*. Will overwrite any earlier value for the specified header, including default values or those set in earlier steps.

When the request credentials are set for basic auth user `*`USER`*` and password `*`PASSWORD`*`
Assign HTTP Basic `Authorization` header. Will overwrite any earlier value for the `Authorization` header including those set in earlier steps.

1.7.2 Client Configuration

These steps modify the client in a way that will impact all requests sent, including any that are built and sent or issued during resource cleanup.

Specification

Given the client sets the header `*`HEADER`*` to `*`VALUE`*` Include the header *`HEADER`* with value samp:*{VALUE}* on each sent request.

1.7.3 Resource Cleanup

Specification

See also:

Concept *Resource Cleanup*

When a resource is created at `*PATH*` Mark *PATH* as a resource to DELETE after the test is run.

1.7.4 Assignment

Specification

When `*IDENTIFIER*` is assigned `*VALUE*` Assigns *VALUE* to *IDENTIFIER*.

When `*IDENTIFIER*` is assigned a random string Assigns a random string (UUID) to *IDENTIFIER*. This can be useful to assist with test isolation.

When `*IDENTIFIER*` is assigned a timestamp Assigns to IDENTIFIER a timestamp value representing the instant at which the step is evaluated.

When `*IDENTIFIER*` is assigned the response *(body|status|headers) [TRAVERSAL]* Assigns to IDENTIFIER the value extracted from the specified response attribute (at the optional traversal path).

1.7.5 Selection

Specification

See also:

Selection and Assertion *Selection and Assertion*

Then the value of the response *(body|status|headers) [TRAVERSAL]* is *[not]* Select the specified response attribute (at the optional traversal path) of the current HTTP response.

Then the value of the response *(body|status|headers) [TRAVERSAL]* does *[not]* have any eleme
Select any (at least one satisfying) element from the structure within the specified response attribute (at the optional traversal path).

Then the value of the *(body|status|headers) [TRAVERSAL]* has elements which are all
Select all elements from the structure within the specified response attribute (at the optional traversal path).

1.7.6 Assertion

Specification

See also:

Selection and Assertion *Selection and Assertion*

Then it is equal to `*VALUE*` Assert that the selected value is equivalent to *VALUE*.

Then it is matching `*VALUE*` Assert that the selected value matches the regular expression *VALUE*.

Then it is including `*VALUE*` Assert that the selected value includes/is a superset of *VALUE*.

Then it is empty Assert that the selected value is empty or null. Any type which is not testable for emptiness (such as booleans or numbers) will always return false. Null is treated as an empty value so that this assertion can be used for endpoints that return null in place of empty collections; non-null empty values can easily be tested for using a step conjoined with this one.

Then it is of length `*VALUE*` Assert that the value exposes a length attribtue and the value of that attribute is *VALUE*.

Then it is a valid `*TYPE*` Assert that the selected value is a valid instance of a *TYPE*. Presently this is focused on standard data types (initially based on those specified by JSON), but it is designed to handle user specified domain types pending some minor wiring and documentation. The current supported types are:

- `Array`
- `Boolean`
- `DateTime`
- `Integer`
- `Number`
- `Object` - JSON style object/associative array
- `String`

1.7.7 Actions

Specification

See also:

Actions *Actions*

Given actions are defined such that Collect subsequent steps as actions to be later performed.

Then the actions are *[not]* successful within a `$*DURATION*` period Repeatedly attempt to perform collected actions over the course of *DURATION*. Succeed if the actions are performed successfully, fail if the duration expires without a successful performance.

1.8 Specification

The behavior of Brine is itself specified using Cucumber specs; these are intended to provide examples for all of the provided steps, including representations of all offered behavior and handling of edge and corner cases. Additionally the specifications are intended to facilitate porting of Brine to new runtimes as the same specifications should be able to be used across those runtimes.

1.8.1 Request Construction

```
Feature: Basic Request Construction
  A simple request with a specified method and path can be sent.

  Scenario Outline: Basic URL
    When a <method> is sent to `/anything`
    Then the value of the response body child `method` is equal to `<method>`

  Examples:
```

(continues on next page)

(continued from previous page)

```
    | method |
    | GET    |
    | POST   |
    | PATCH  |
    | DELETE |
    | PUT    |
```

```
Feature: Assigning a Request Body

  Scenario: Attach a basic body.
    When the request body is assigned:
      """
      {"request": 1}
      """
    And a PUT is sent to `/anything`

    Then the value of the response body child `json` is equal to:
      """
      {"request": 1}
      """

  Scenario: Attach a template body.
    Given `val` is assigned `foo`
    When the request body is assigned:
      """
      {"request":"{{val}}"}
      """
    And a PUT is sent to `/anything`

    Then the value of the response body child `json` is equal to:
      """
      {"request": "foo"}
      """
```

```
Feature: Adding Query Parameters

  Scenario: A single parameter is appended to the URL.
    When the request query parameter `foo` is assigned `bar`
    And a GET is sent to `/get`

    Then the value of the response body child `args` is equal to:
      """
      {"foo": "bar"}
      """

  Scenario: Multiple parameters are appended to the URL with proper formatting.
    When the request query parameter `foo` is assigned `bar`
    And the request query parameter `baz` is assigned `1`
    And a GET is sent to `/get`

    Then the value of the response body child `args` is equal to:
      """
      {"foo": "bar",
       "baz": "1"}
      """
```

(continues on next page)

(continued from previous page)

```
  Scenario Outline: Values are encoded appropriately.
    When the request query parameter `foo` is assigned `<input>`
    And a GET is sent to `/get`

    Then the value of the response body child `args` is equal to:
      """
      {"foo": "<input>"}
      """

  Examples:
    | input          |
    | bar & grill    |
    | + +            |
    | (imbalance))   |

  Scenario Outline: Parametes are added regardless of HTTP method.
    When the request query parameter `foo` is assigned `bar`
    And a <method> is sent to `/anything`

    Then the value of the response body child `args` is equal to:
      """
      {"foo": "bar"}
      """

  Examples:
    | method  |
    | POST    |
    | PUT     |
    | DELETE  |
```

```
Feature: Adding Headers

  Scenario: A new header with a single value is added to request.
    When the request header `foo` is assigned `bar`
    And a GET is sent to `/anything`

    Then the value of the response body child `headers` is including:
      """
      {"Foo": "bar"}
      """

  Scenario: Default headers are present in requests.
    When a GET is sent to `/anything`

    Then the value of the response body child `headers` is including:
      """
      {"Content-Type": "application/json"}
      """

  Scenario: Default headers can be overridden.
    When the request header `Content-Type` is assigned `text/plain`
    And a GET is sent to `/anything`

    Then the value of the response body child `headers` is including:
      """
      {"Content-Type": "text/plain"}
```

(continues on next page)

(continued from previous page)

```
    """

  Scenario: Array header values are added to requests.
    When the request header `X-Array` is assigned `[1, 2, 3]`
    And a GET is sent to `/anything`

    Then the value of the response body child `headers` is including:
      """
      {"X-Array": "1, 2, 3"}
      """

  Scenario: The last set value for a given header wins.
    When the request header `foo` is assigned `bar`
    And the request header `foo` is assigned `baz`
    And a GET is sent to `/anything`

    Then the value of the response body child `headers` is including:
      """
      {"Foo": "baz"}
      """

  Scenario Outline: Header is added regardless of HTTP method.
    When the request header `Foo` is assigned `bar`
    And a <method> is sent to `/anything`

    Then the value of the response body child `headers` is including:
      """
      {"Foo": "bar"}
      """
  Examples:
    | method  |
    | POST    |
    | PUT     |
    | DELETE  |
```

```
Feature: Cleared After Sent
  After a request is sent, any values that were added to that request
  are cleared and will not be present in subsequent requests.

  Scenario: Request body is cleared.
    When the request body is assigned:
      """
      {"request":1}
      """
    And a PUT is sent to `/anything`
    And a PUT is sent to `/anything`

    Then the value of the response body child `json` is empty

  Scenario: Request parameter
    When the request query parameter `foot` is assigned `bar`
    And a GET is sent to `/anything`
    And a GET is sent to `/anything`

    Then the value of the response body child `args` is empty
```

```
Feature: Adding Basic Auth

  Scenario: A new header with expected value is added to request.
    When the request credentials are set for basic auth user `user` and password␣
↪`pass`
    And a GET is sent to `/anything`

    Then the value of the response body child `headers` is including:
      """
      {"Authorization":"Basic dXNlcjpwYXNz"}
      """
```

1.8.2 Client Configuration

```
Feature: Adding Extra Headers

  Scenario: A header is added to the request on basic construction.
    When the client sets the header `Foo` to `Bar`
    And a GET is sent to `/anything`

    Then the value of the response body child `headers` is including:
      """
      {"Foo": "Bar"}
      """

  Scenario Outline: A header is added to subsequent requests.
    Given the client sets the header `Foo` to `Bar`
    And a GET is sent to `/anything`

    When a <method> is sent to `/anything`
    Then the value of the response body child `headers` is including:
      """
      {"Foo": "Bar"}
      """
  Examples:
    | method |
    | GET    |
    | POST   |
    | PUT    |
    | DELETE |

  Scenario: Each header assignment is idempotent.
    Given the client sets the header `Foo` to `Bar`
    And the client sets the header `Foo` to `Bar`
    And a GET is sent to `/anything`

    Given the client sets the header `Foo` to `Bar`
    When a GET is sent to `/anything`
    Then the value of the response body children `headers.Foo` is equal to:
      """
      ["Bar"]
      """

  Scenario: Multiple header assignments can be combined.
    Given the client sets the header `Foo` to `Bar`
```

(continues on next page)

(continued from previous page)

```
    And the client sets the header `Blah` to `Baz`
    And a GET is sent to `/anything`

    When a GET is sent to `/anything`
    Then the value of the response body child `headers` is including:
      """
      {"Foo": "Bar",
       "Blah": "Baz"}
      """
```

1.8.3 Resource Cleanup

```
@pending @171
Feature: Resource Cleanup
  Resources created during testing can be marked for deletion.

  Scenario: Successful Basic Deletion
    Given expected DELETE sent to `/some/path`

    When a resource is created at `/some/path`

  Scenario: Returned 4xx
    Given expected response status of `409`
    And expected DELETE sent to `/some/path`

    When a resource is created at `/some/path`

  Scenario: Success Upon Retry
    Given expected response status sequence of `[504, 200]`
    And expected DELETE sent to `/some/path`

    When a resource is created at `/some/path`

  Scenario: Unreached Success
    Given expected response status sequence of `[504, 504, 504, 200]`
    And expected DELETE sent to `/some/path`

    When a resource is created at `/some/path`
```

1.8.4 Assignment

```
Feature: A Parameter
  An identifier can be assigned the value of the provided parameter.

  Scenario: Simple assignment.
    Given `foo` is assigned `bar`
    When the request body is assigned `{{ foo }}`
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is equal to `bar`
```

```
@pending @170
Feature: A Random String
```

(continues on next page)

(continued from previous page)

```
  An identifier can be assigned a random string with decent entropy.

  Scenario: Several unique variables.
    Given `v1` is assigned a random string
    And `v2` is assigned a random string
    And `v3` is assigned a random string
    When the response body is assigned `[ "{{ v1 }}","{{ v2 }}","{{ v3 }}" ]`
    Then the value of the response body does not have any element that is empty
    And the value of the response body child `.[0]` is equal to `{{ v1 }}`
    And the value of the response body children `.[1:2]` does not have any element␣
↪that is equal to `{{ v1 }}`
    And the value of the response body child `.[2]` is not equal to `{{ v2 }}`
```

```
Feature: A Timestamp
  An identifier can be assigned a current timestamp.

  Scenario: Newer than some old date.
    Given `v1` is assigned a timestamp
    When the request body is assigned `{{ v1 }}`
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is greater than `2018-06-
↪17T12:00:00Z`

  Scenario: Values increase.
    Given `v1` is assigned a timestamp

    When `v2` is assigned a timestamp
    And the request body is assigned `{{ v2 }}`
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is greater than or equal to `{{␣
↪v1 }}`

    When `v3` is assigned a timestamp
    And the request body is assigned `{{ v3 }}`
    Then the value of the response body child `data` is greater than or equal to `{{␣
↪v1 }}`
    And the value of the response body child `data` is greater than or equal to `{{␣
↪v2 }}`
```

```
Feature: Response Attribute
  An identifier can be assigned a value extracted from a response attribute.

  Scenario: Response body.
    Given the request body is assigned `foo`
    And a PUT is sent to `/anything`
    When `myVar` is assigned the response body child `data`
    And the request body is assigned `{{ myVar }}`
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is equal to `foo`

  Scenario: Response body child.
    Given the request body is assigned `{"response": "foo"}`
    And a PUT is sent to `/anything`
    When `myVar` is assigned the response body child `json.response`
    And the request body is assigned `{{ myVar }}`
    And a PUT is sent to `/anything`
```

(continues on next page)

(continued from previous page)

```
    Then the value of the response body child `data` is equal to `foo`

  Scenario: Response body children.
    Given the request body is assigned `{"response": "foo"}`
    And a PUT is sent to `/anything`
    When `myVar` is assigned the response body children `json.response`
    And the request body is assigned `{{{ myVar }}}`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is equal to `["foo"]`

  Scenario: Response header.
    Given the request query parameter `test` is assigned `val`
    And a GET is sent to `/response-headers`
    When `myVar` is assigned the response headers child `test`
    And the request body is assigned `{{ myVar }}`
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is equal to `val`
```

```
Feature: Assign conventional environment variables.

  Scenario: Conventional environment variable is present in binding.
    When the request body is assigned `{{from_env}}`
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is equal to `Read from␣
↪environment`
```

1.8.5 Selection

```
Feature: Response Attribute
  A response attribute can be selected for assertion.

  Scenario: Response body.
    When a GET is sent to `/base64/SFRUUEJJTiBpcyBhd2Vzb21l`
    Then the value of the response body is equal to `HTTPBIN is awesome`

  Scenario: Response status.
    When a GET is sent to `/status/409`
    Then the value of the response status is equal to `409`

  Scenario: Response headers.
    When the request query parameter `foo` is assigned `bar`
    And a GET is sent to `/response-headers`
    Then the value of the response headers is including:
      """
      {"foo":"bar"}
      """

  Scenario: Response header child.
    When the request query parameter `foo` is assigned `bar`
    And a GET is sent to `/response-headers`
    Then the value of the response headers is including:
      """
      {"foo":"bar"}
      """
    Then the value of the response headers child `foo` is equal to `bar`
```

```
Feature: Any Element
  Assertions can be done against any element of a structure.

  Scenario: Nested list in response body
    When the request body is assigned:
      """
      ["a", "b", "c"]
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` does have any element that is␣
↪equal to `a`
    And the value of the response body child `json` does not have any element that is␣
↪equal to `d`

  Scenario: Map matches entries
    When the request body is assigned:
      """
      {"a": 1, "b": 2}
      """
    #Equality will match keys
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` does have any element that is␣
↪equal to `a`
    And the value of the response body child `json` does not have any element that is␣
↪equal to `d`

  Scenario: Spread nested lists
    When the request body is assigned:
      """
      [{"val":"foo"},{"val":"bar"}]
      """
    And a PUT is sent to `/anything`
    Then the value of the response body children `json..val` does have any element␣
↪that is equal to `foo`
    And the value of the response body children `json..val` does not have any element␣
↪that is equal to `other`
```

```
Feature: All Elements
  Assertions can be done against all elements of a structure.

  Scenario: List in response body
    When the request body is assigned:
      """
      ["a", "bb", "ccc"]
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` has elements which are all␣
↪matching `/\w+/`

  Scenario: Spread nested lists
    When the request body is assigned:
      """
      [{"val": "foo"},{"val": "foo"}]
      """
    And a PUT is sent to `/anything`
    Then the value of the response body children `json..val` has elements which are␣
↪all equal to `foo`
```

1.8.6 Assertion

```
Feature: Equal To
  It can be asserted that a value is equal to another value.

  Scenario: String in response body
    When the request body is assigned:
      """
      foo
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is equal to `foo`
    And the value of the response body child `data` is not equal to `foot`

  Scenario: Response Status
    When a GET is sent to `/status/404`
    Then the value of the response status is equal to `404`
    And the value of the response status is not equal to `200`

  Scenario: Object in response body
    When the request body is assigned:
      """
      {"foo": "bar"}
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is equal to:
      """
      {"foo":"bar"}
      """
    And the value of the response body child `json` is not equal to:
      """
      {"foo": "baz"}
      """

  Scenario: List in response body
    When the request body is assigned `[1, "foo", true]`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is equal to `[1, "foo", true]`
    And the value of the response body child `json` is not equal to `[1, "bar", true]`

  Scenario Outline: Objects must match completely
    When the request body is assigned `{"foo": "bar", "baz": 1}`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is not equal to `<comparison>`

  Examples:
    | comparison                          |
    | {}                                  |
    | {"foo": "bar"}                      |
    | {"foo": "bar", "baz": 1, "extra": 2} |
    | {"foo": "bar", "baz": 2}            |
```

```
Feature: Matching
  It can be asserted that a value matches another string or regex.

  Scenario: String in response body is matched against a regex.
    When the request body is assigned:
```

(continues on next page)

(continued from previous page)

```
      """
      http://www.github.com?var=val
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is matching `/github/`
    And the value of the response body child `data` is matching `/git.*\?.*=/`
    And the value of the response body child `data` is not matching `/gh/`
    And the value of the response body child `data` is not matching `/^github/`

  @pending @assertion_against_assignment
  Scenario: Regex in response body matched against a string
    When the request body is assigned:
      """
      /(.+)\1/
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is matching `blahblah`
    And the value of the response body child `data` is matching `boo`
    And the value of the response body child `data` is not matching `blah blah`
    And the value of the response body child `data` is not matching `blah`
```

```
Feature: Including
  It can be asserted that a value is a superset of another value.  It can be asserted␣
↪that a value is a superset of another value.  It can be asserted that a value is a␣
↪superset of another value.  It can be asserted that a value is a superset of␣
↪another value.

  Scenario: Basic object membership
    When the request body is assigned:
      """
      {"foo":"bar",
       "baz": 1,
       "other": "blah"}
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is including:
      """
      {"baz": 1}
      """
    And the value of the response body child `json` is not including:
      """
      {"missing":"value"}
      """
    And the value of the response body child `json` is including `other`
    And the value of the response body child `json` is not including `brother`
    And the value of the response body child `json` is not including `value`
```

```
Feature: Empty
  It can be asserted that a value is empty.

  Scenario: Empty body is empty.
    When the request body is assigned ``
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is empty

  Scenario: Whitespace-only body is empty.
```

(continues on next page)

(continued from previous page)

```
    When the request body is assigned:
      """

      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is empty

  Scenario: Empty string is empty.
    When the request body is assigned `""`
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is empty

  Scenario: Non-empty string is not empty.
    When the request body is assigned `blah`
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is not empty

  Scenario: Quoted whitespace is not empty.
    When the request body is assigned `" "`
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is not empty

  Scenario: Empty arrays are empty.
    When the request body is assigned `[]`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is empty

  Scenario: Non-empty arrays are not empty.
    When the request body is assigned `[[]]`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is not empty

  Scenario: Empty objects are empty.
    When the request body is assigned `{}`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is empty

  Scenario: Non-empty objects are not empty.
    When the request body is assigned `{"foo":{}}`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is not empty

  Scenario: Null values are empty.
    When the request body is assigned `{"foo": null}`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json.foo` is empty

  Scenario: False is not empty.
    When the request body is assigned `{"foo": false}`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json.foo` is not empty

  Scenario: 0 is not empty.
    When the request body is assigned `0`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is not empty
```

```
Feature: Of Length
  It can be asserted that a value has a provided length.

  Scenario: String
    When the request body is assigned:
      """
      blah
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is of length `4`

  Scenario: Array
    When the request body is assigned:
      """
      ["foo", "blah"]
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is of length `2`

  Scenario: Map
    When the request body is assigned:
      """
      {"foo": "blah"}
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is of length `1`

  Scenario: Value without length attribute
    When the request body is assigned:
      """
      true
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is not of length `1`
    And the value of the response body child `json` is not of length `4`
    And the value of the response body child `json` is not of length `0`
```

```
Feature: A Valid
  It can be asserted that a value is a valid instance of a type.

  Scenario: String in response body is only a valid String.
    When the request body is assigned:
      """
      foo
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is a valid `String`
    And the value of the response body child `data` is not a valid `Number`
    And the value of the response body child `data` is not a valid `Integer`
    And the value of the response body child `data` is not a valid `Object`
    And the value of the response body child `data` is not a valid `Array`
    And the value of the response body child `data` is not a valid `Boolean`

  Scenario: Integer in response body is a valid Integer and Number.
    When the request body is assigned:
      """
      1
```

(continues on next page)

(continued from previous page)

```
    """
  And a PUT is sent to `/anything`
  Then the value of the response body child `json` is not a valid `String`
  And the value of the response body child `json` is a valid `Number`
  And the value of the response body child `json` is a valid `Integer`
  And the value of the response body child `json` is not a valid `Object`
  And the value of the response body child `json` is not a valid `Array`
  And the value of the response body child `json` is not a valid `Boolean`

Scenario: Quoted Number in response body is only a valid String.
  When the request body is assigned:
    """
    "1"
    """
  And a PUT is sent to `/anything`
  Then the value of the response body child `data` is a valid `String`
  And the value of the response body child `data` is not a valid `Number`
  And the value of the response body child `data` is not a valid `Integer`
  And the value of the response body child `data` is not a valid `Object`
  And the value of the response body child `data` is not a valid `Array`
  And the value of the response body child `data` is not a valid `Boolean`

Scenario: Empty Object in response body is only a valid Object.
  When the request body is assigned:
    """
    {}
    """
  And a PUT is sent to `/anything`
  Then the value of the response body child `json` is not a valid `String`
  And the value of the response body child `json` is not a valid `Number`
  And the value of the response body child `json` is not a valid `Integer`
  And the value of the response body child `json` is a valid `Object`
  And the value of the response body child `json` is not a valid `Array`
  And the value of the response body child `json` is not a valid `Boolean`

Scenario: Object in response body is only a valid Object.
  When the request body is assigned:
    """
    {"foo": 1}
    """
  And a PUT is sent to `/anything`
  Then the value of the response body child `json` is not a valid `String`
  And the value of the response body child `json` is not a valid `Number`
  And the value of the response body child `json` is not a valid `Integer`
  And the value of the response body child `json` is a valid `Object`
  And the value of the response body child `json` is not a valid `Array`
  And the value of the response body child `json` is not a valid `Boolean`

Scenario: Quoted Object in response body is only a valid String.
  When the request body is assigned:
    """
    "{"foo": 1}"
    """
  And a PUT is sent to `/anything`
  Then the value of the response body child `data` is a valid `String`
  And the value of the response body child `data` is not a valid `Number`
  And the value of the response body child `data` is not a valid `Integer`
```

(continues on next page)

(continued from previous page)

```
  And the value of the response body child `data` is not a valid `Object`
  And the value of the response body child `data` is not a valid `Array`
  And the value of the response body child `data` is not a valid `Boolean`

Scenario: Empty Array in response body is only a valid Array.
  When the request body is assigned:
    """
    []
    """
  And a PUT is sent to `/anything`
  Then the value of the response body child `json` is not a valid `String`
  And the value of the response body child `json` is not a valid `Number`
  And the value of the response body child `json` is not a valid `Integer`
  And the value of the response body child `json` is not a valid `Object`
  And the value of the response body child `json` is a valid `Array`
  And the value of the response body child `json` is not a valid `Boolean`

Scenario: Array in response body is only a valid Array.
  When the request body is assigned:
    """
    [1, "foo"]
    """
  And a PUT is sent to `/anything`
  Then the value of the response body child `json` is not a valid `String`
  And the value of the response body child `json` is not a valid `Number`
  And the value of the response body child `json` is not a valid `Integer`
  And the value of the response body child `json` is not a valid `Object`
  And the value of the response body child `json` is a valid `Array`
  And the value of the response body child `json` is not a valid `Boolean`

Scenario: true in response body is only a valid Boolean.
  When the request body is assigned:
    """
    true
    """
  And a PUT is sent to `/anything`
  Then the value of the response body child `json` is not a valid `String`
  And the value of the response body child `json` is not a valid `Number`
  And the value of the response body child `json` is not a valid `Integer`
  And the value of the response body child `json` is not a valid `Object`
  And the value of the response body child `json` is not a valid `Array`
  And the value of the response body child `json` is a valid `Boolean`

Scenario: false in response body is only a valid Boolean.
  When the request body is assigned:
    """
    {"foo": false}
    """
  And a PUT is sent to `/anything`
  Then the value of the response body child `json.foo` is not a valid `String`
  And the value of the response body child `json.foo` is not a valid `Number`
  And the value of the response body child `json.foo` is not a valid `Integer`
  And the value of the response body child `json.foo` is not a valid `Object`
  And the value of the response body child `json.foo` is not a valid `Array`
  And the value of the response body child `json.foo` is a valid `Boolean`

Scenario: null in response body is not any valid type.
```

(continues on next page)

(continued from previous page)

```
    When the request body is assigned:
      """
      [null]
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json[0]` is not a valid `String`
    And the value of the response body child `json[0]` is not a valid `Number`
    And the value of the response body child `json[0]` is not a valid `Integer`
    And the value of the response body child `json[0]` is not a valid `Object`
    And the value of the response body child `json[0]` is not a valid `Array`
    And the value of the response body child `json[0]` is not a valid `Boolean`

  Scenario: Selected Array child is a valid Array.
    When the request body is assigned:
      """
      {"val": [1, 2, 3]}
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json.val` is a valid `Array`

  Scenario: Selected Array child member is a valid String.
    When the request body is assigned:
      """
      {"val": [1, 2, 3]}
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json.val[0]` is a valid `Number`
    Then the value of the response body child `json.val[0]` is a valid `Integer`

  Scenario: Selected nested children are a valid Array.
    When the request body is assigned:
      """
      [{"val": 1},{"val": 2}]
      """
    And a PUT is sent to `/anything`
    Then the value of the response body children `json.val` is a valid `Array`

  Scenario: Selected nested children can be tested for type.
    When the request body is assigned:
      """
      [{"val": 1},{"val": 2}]
      """
    And a PUT is sent to `/anything`
    Then the value of the response body children `json.val` has elements which are␣
↪all a valid `Number`
    And the value of the response body children `json.val` has elements which are all␣
↪a valid `Integer`

  Scenario: Selected nested children can be tested for type when Arrays.
    When the request body is assigned:
      """
      [{"val": [1]},{"val": [2]}]
      """
    And a PUT is sent to `/anything`
    Then the value of the response body children `json.val` has elements which are␣
↪all a valid `Array`
```

1.8.7 Actions

```
@pending
Feature: Eventually
  Conditions which are eventually but may not be immediately satisfied can be tested.

  Scenario: A delayed response passes an eventual test.
    Given actions are defined such that
      When a GET is sent to `/bytes/5`
      Then the value of the response body is matching `/fdlb/`
    Then the actions are successful within a `short` period

  Scenario: A late response fails an eventual test.
    When the response is delayed 5 seconds
    And the response body is assigned:
      """
      {"completed": true}
      """

    Given actions are defined such that
      Then the value of the response body is equal to:
        """
        {"completed": true}
        """
    Then the actions are not successful within a `short` period

  Scenario: A late response passes a patient eventual test.
    When the response is delayed 5 seconds
    And the response body is assigned:
      """
      {"completed": true}
      """

    Given actions are defined such that
      Then the value of the response body is equal to:
        """
        {"completed": true}
        """
    Then the actions are successful within a `long` period

  Scenario: Action state is managed properly
    When actions are defined such that
      When `expected` is assigned `val`
      And the response body is assigned:
        """
        {"key": "val",
         "other": "blah"}
        """
      Then the value of the response body child `key` is equal to `{{expected}}`

      When `expected` is assigned `blah`
      Then the value of the response body children `other` is equal to `{{expected}}`

      When the response body is assigned:
        """
        {"key": "blah"}
        """
```

(continues on next page)

(continued from previous page)

```
      Then the value of the response body child `key` is equal to `{{expected}}`
    Then the actions are successful within a `short` period
```

1.8.8 Argument Transformation

```
Feature: Boolean
  An argument that could represent a boolean value will be transformed into a boolean␣
↪type.

  Scenario Outline: Docstring simple value.
    When the request body is assigned:
      """
      <input>
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is a valid `Boolean`
    And the value of the response body child `json` is equal to `<input>`

  Examples:
    | input      |
    | true       |
# This produces an empty PUT, should be handled by selection of assigned values
#    | false      |

  Scenario Outline: Inline simple value.
    When the request body is assigned `<input>`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is a valid `Boolean`
    And the value of the response body child `json` is equal to `<input>`

  Examples:
    | input      |
    | true       |
# This produces an empty PUT, should be handled by selection of assigned values
#    | false      |
```

```
@pending
Feature: DateTime
  An argument that could represent a date/time value will be transformed into a time␣
↪type.

  Scenario: Docstring datetime is serialized properly.
    When the request body is assigned:
      """
      2017-01-01T09:00:00Z
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is a valid `DateTime`
    And the value of the response body child `json` is equal to `2017-01-01T09:00:00Z`
    And the value of the response body child `json` is greater than `2017-01-
↪01T08:00:00Z`
    And the value of the response body child `json` is less than `2017-01-
↪01T10:00:00Z`
```

(continues on next page)

(continued from previous page)

```
  Scenario: Inline datetime is serialized properly.
    When the request body is assigned `2017-01-01T09:00:00Z`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is a valid `DateTime`
    And the value of the response body child `json` is equal to `2017-01-01T09:00:00Z`
    And the value of the response body child `json` is greater than `2017-01-
↪01T08:00:00Z`
    And the value of the response body child `json` is less than `2017-01-
↪01T10:00:00Z`

  Scenario: Value Comparison
    When `now` is assigned a timestamp
    And `then` is assigned `2017-01-01T12:00:00Z`
    Then the value of `{{ then }}` is less than `{{ now }}`

  @pending @170
  Scenario: Child Comparison
    When `now` is assigned a timestamp
    And the response body is assigned:
      """
      {"my_timestamp": "{{ now }}"}
      """
    Then the value of the response body child `my_timestamp` is greater than `2017-01-
↪01T12:00:00Z`
```

```
Feature: Integer
  An argument that could represent an integer will be transformed into an integer␣
↪type.

  Scenario Outline: Docstring simple value.
    When the request body is assigned:
      """
      <input>
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is a valid `Integer`
    And the value of the response body child `json` is equal to `<input>`
  Examples:
    |                input |
    |                    0 |
    |                   -0 |
    |                   10 |
    |                  -10 |
    |  123456789123456789 |
    | -123456789123456789 |

  Scenario Outline: Inline simple value.
    When the request body is assigned `<input>`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is a valid `Integer`
    And the value of the response body child `json` is equal to `<input>`

  Examples:
    |                input |
    |                    0 |
    |                   -0 |
```

(continues on next page)

(continued from previous page)

```
    |                  10 |
    |                 -10 |
    |  123456789123456789 |
    | -123456789123456789 |
```

```
Feature: List
    An argument that could represent a JSON list will be
    transformed into a list whose elements will be also be transformed.

  Scenario Outline: Docstring simple list.
    When the request body is assigned:
      """
      <input>
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is a valid `Array`
    And the value of the response body child `json` is equal to `<input>`
  Examples:
    | input                    |
    | []                       |
    | ["a", "b"]               |
    | ["a"  ,  "b" ]           |
    | ["  a", " b "]           |
    | [true, "false"]          |
    | [1,-3,"-5"]              |
    | ["foo,bar","baz"]        |
    | ["foo,bar,baz"]          |
    | ["foo\\"","bar"]         |
    | ["fo\\"o\\",bar","baz"]  |
    | [{"i":1},{"i":2}, "h"]   |

  Scenario Outline: Inline simple list.
    When the request body is assigned `<input>`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is a valid `Array`
    And the value of the response body child `json` is equal to `<input>`
  Examples:
    | input                    |
    | []                       |
    | ["a", "b"]               |
    | ["a"  ,  "b" ]           |
    | ["  a", " b "]           |
    | [true, "false"]          |
    | [1,-3,"-5"]              |
    | ["foo,bar","baz"]        |
    | ["foo,bar,baz"]          |
    | ["foo\\"","bar"]         |
    | ["fo\\"o\\",bar","baz"]  |
    | [{"i":1},{"i":2}, "h"]   |
```

```
Feature: Object
  An argument that could represent a JSON object will be
  transformed into an object whose elements will also be transformed.

  Scenario Outline: Docstring simple object.
    When the request body is assigned:
```

(continues on next page)

(continued from previous page)

```
      """
      <input>
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is a valid `Object`
    And the value of the response body child `json` is equal to `<input>`
  Examples:
    | input                                                  |
    | {}                                                     |
    | {"a":1}                                                |
    | {"foo":"bar", "num":1, "list": ["1", 2, true]}         |
    | {"foo": {"bar":{ "num":1, "list": ["1", 2, true]}}} |
    | {"foo": "\"list\": [\"1\", 2, true]"}                  |

  Scenario Outline: Inline simple object.
    When the request body is assigned `<input>`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is a valid `Object`
    And the value of the response body child `json` is equal to `<input>`
  Examples:
    | input                                                  |
    | {}                                                     |
    | {"a":1}                                                |
    | {"foo":"bar", "num":1, "list": ["1", 2, true]}         |
    | {"foo": {"bar":{ "num":1, "list": ["1", 2, true]}}} |
    | {"foo": "\"list\": [\"1\", 2, true]"}                  |

  Scenario: Object split over lines
    When the request body is assigned:
      """
      {
        "foo":"bar"
      }
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is equal to:
      """
      {"foo":"bar"}
      """
```

```
@pending
Feature: Quoted
  An argument that is quoted will be (not) transformed into
  into a string, regardless of any more specific data type the
  quoted value may resemble.

  Scenario Outline: Docstring simple value.
    When the request body is assigned:
      """
      <input>
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is a valid `String`
    And the value of the response body child `json` is equal to `<input>`

  Examples:
```

(continues on next page)

(continued from previous page)

```
    | input               |
    | "true"              |
    | ""123""             |
    | " -123 "            |
    | "["foo","bar"]"     |
    | "{"foo":"bar"}"     |

  Scenario Outline: Inline simple value.
    When the response body is assigned `<input>`

    Then the value of the response body is a valid `String`
    And the value of the response body is equal to `<input>`

  Examples:
    | input               |
    | "true"              |
    | "123"               |
    | " -123 "            |
    | "["foo","bar"]"     |
    | "{"foo":"bar"}"     |
```

```
@169
Feature: Regular Expression
  An argument that is enclosed in slashes (/) will be transformed into a regex.

  Scenario Outline: Docstring simple value.
    When the request body is assigned:
      """
      <input>
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is equal to:
      """
      <expected>
      """

  #Expecting Ruby stringification and using painful escaping
  Examples:
    | input             | expected                                          |
    | //                | ""(?-mix:)""                                      |
    | /\//              | ""(?-mix:\\\\\/)""                                |
    | /.*/              | ""(?-mix:.*)""                                    |
    | /"[[:alpha:]]?"/  | ""(?-mix:\\"[[:alpha:]]?\\")""                    |
    | /foo bar/         | ""(?-mix:foo bar)""                               |

  Scenario Outline: Inline simple value.
    When the request body is assigned `<input>`
    And a PUT is sent to `/anything`
    Then the value of the response body child `data` is equal to:
      """
      <expected>
      """

  #Expecting Ruby stringification and using painful escaping
  Examples:
    | input             | expected                                          |
```

(continues on next page)

(continued from previous page)

```
    | //                | ""(?-mix:)""                                    |
    | /\//              | ""(?-mix:\\\\\\/)""                             |
    | /.*/              | ""(?-mix:.*)""                                  |
    | /"[[:alpha:]]?"/  | ""(?-mix:\\"[[:alpha:]]?\\")""                  |
    | /foo bar/         | ""(?-mix:foo bar)""                             |
```

```
Feature: Template
  An argument that includes _{{ }}_ interpolation markers will be
  treated as a template and transformed into an evaluated version of
  that template using the current binding environment which will then
  also be transformed.

  Scenario Outline: Docstring single value template.
    When `bound` is assigned `<binding>`
    And the request body is assigned:
      """
      {{{ bound }}}
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is equal to `<expected>`
  Examples:
    | binding        | expected   |
    | true           | true       |
    | -452           | -452       |
    | ["a", 1]       | ["a", 1]   |

  Scenario Outline: Inline single value template.
    When `bound` is assigned `<binding>`
    And the request body is assigned `{{{ bound }}}`
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is equal to `<expected>`

  Examples:
    | binding        | expected   |
    | true           | true       |
    | -452           | -452       |
    | ["a", 1]       | ["a", 1]   |
```

```
Feature: Whitespace
  An argument that includes leading or trailing whitespace
  will be transformed so that such whitespace is removed
  and that value will also be transformed.

  Scenario Outline: Docstring simple value.
    When the request body is assigned:
      """
      <input>
      """
    And a PUT is sent to `/anything`
    Then the value of the response body child `json` is equal to `<expected>`
  Examples:
    | input              | expected     |
    |       true             | true         |
    |   123                | 123          |
    |  ["a"]                | ["a"]        |
```

(continues on next page)

(continued from previous page)

```
Scenario Outline: Inline simple value.
  When the request body is assigned `<input>`
  And a PUT is sent to `/anything`
  Then the value of the response body child `json` is equal to `<expected>`
Examples:
  | input              | expected     |
  |         true             | true         |
  |   123                | 123          |
  |  ["a"]                | ["a"]        |

Scenario Outline: Docstring value with a leading and trailing line.
  When the request body is assigned:
    """

    <input>

    """
  And a PUT is sent to `/anything`
  Then the value of the response body child `json` is equal to `<expected>`
Examples:
  | input              | expected     |
  |         true             | true         |
  |   123                | 123          |
  |  ["a"]                | ["a"]        |
```

1.9 Article List

Articles here will provide assorted information about using Brine.

The core Brine documentation (along with Brine itself) is focused on functionality which is expected to be very common across APIs and execution environments, so the documents here are intended to convey information addressing more specific concerns.

1.9.1 Assure Required Data is Available

Context

Ideally all tests should be as self-contained and isolated as possible; when writing functional tests, however, there are cases where this isn't feasible or possible. In some cases a system depends on another external system which is not a system that is under test and which (for whatever reason) cannot be easily worked with. In white box testing such a system would likely be represented by some form of test double, but this may be impractical and/or undesirable when testing a deployed system.

An example of such a system is user/account management which often incurs additional overhead to provision a new account. When testing a secured system valid accounts are needed for representative testing, but provisioning a new account may be difficult or outside the scope of the system that is being actively tested. If tested functionality involves enacting account-wide changes and the number of accounts is limited, then that is likely to unfortunately prevent complete test isolation.

Solution

In such cases a standard solution is to designate certain resources to be reused for certain tests. The term "assurances" is used here for verification of such resources…primarily because it starts with *a* which lends itself to relevant files being listed towards the beginning alphabetically in a given directory.

The goal of assurances is to specify conditions which are expected before other tests are to be run. Preferably the dependent tests should also explicitly declare the dependency but a significant solution for that is not established. Assurances therefore validate that preconditions are met; ideally if such preconditions can be established idempotently then the assurances can do so before the validation.

Assurances are NOT Tests

Assurances validate a state which is desired to be consistently retained within the system rather than being changed. This means that they should _not_ be used for tests as that would require state changes, nor should they clean up after themselves (as that would also imply a state change). If assurances are configured for a system which should also be tested, then appropriate tests should exist (including those that may validate any behavior relied upon by the assurance).

Consequences

As mentioned previously assertions help in cases where tests cannot be fully isolated, and therefore some known state must be established and reused across tests (and such state should *not* change). A practical reason for this is to allow for overlapping test executions. If tests are not fully isolated and test runs overlap whil state is being changed then tests may fail non-deterministically due to one test run pulling the state out from another. This in the simplest form can be a nuisance but it also effectively precludes the ability to speed up test runs through the use of parallelism/asynchronicity.

Todo: Enumerate drawbacks

Recipe

This can be done using standard cucumber tags. Assurances can be defined in designated samp:*assure_{description}.feature* files where each Feature is appropriately tagged:

```
@assure
Feature: Some preconditions are verified...
```

And then a Rake or similar task can be added to run those tagged features:

```
Cucumber::Rake::Task.new(:assure) do |t|
  t.cucumber_opts = "#{ENV['CUCUMBER_OPTS']} --tags @assure"
end
```

The task that runs the other tests then depends on the assure task:

```
task :invoke_cuke => [:assure] do
  #Run cucumber, potentially in parallel and likely with --tags `@assure`
end
```

This approach tests preconditions and will avoid running the rest of the tests if they are not (relying on standard Rake behavior). The assurances can also be run with different Cucumber behavior so that the full test suite can be more stochastic (randomized/non-serialized) while the assurances can be more controlled.

1.9.2 Intercepting HTTP Calls

Context

There may be cases where the request or reponse may need to be modified in some way before or after transmission, for example to add a custom header which is somehow specific to the testing configuration and therefore outside of anything that belongs in the specification.

Solution

This can be done through the registration of custom Faraday middleware on the Brine client. All of the registered middleware attaches to the generated connection through an array of functions in requester.rb. Each function accepts the connection object as its single parameter. The array of functions is exposed as `connection_handlers` and can be modified through getting that property from either the default `World` (for the default client) or from an appropriate `ClientBuilder` if creating custom clients and mutating it accordingly (setting the reference is not suported). You could just build your own client without the facilities provided by Brine.

Recipe

An example to add a custom header could be implemented as follows:

```
require 'faraday'
class CustomHeaderAdder < Faraday::Middleware
  def initialize(app, key, val)
    super(app)
      @key = key
      @val = val
    end

    def call(env)
      env[:request_headers].merge!({@key => @val})
      @app.call(env)
    end
  end
end

...

connection_handlers.unshift(proc do |conn|
  conn.use CustomHeaderAdder, 'x-header', 'I am a test'
end)
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

Symbols

B

E

G

T

W