JasmineTests framework

- configuration - available to run different in different environments
- regression - available to run in previous versions, snapshots, history
- assertions - positive, negative testing
- easy-to-access - onsite and offsite
- easy-to-execute
- easy-to-write - through straight-forward json template
- logging (output target, verbose level, reporting (json)), validations
- synchronous
- chaining
- context
- pretests
    - running under different context
    - using “third-party” APIs (in our case using CCCAPI from MessagingAPI)
- repeat
- file upload
- pages
- anonymous
- scalability
- flexible
    - environment, usertype - define multiple environments in which to execute the same set of tests
    - testcase (forming various subsets)
    - specs folder
    - timeouts
- cleanup logs
- history - the requirement to run the tests that were available 2 months ago at the time of the previous release
- git integration - simultaneous development
- jenkins integration
- CORS - unresolved

pattern: challenge - experiments - results - future work

Related documentation:
https://cirius.atlassian.net/wiki/display/DG/Messaging+API+-+Integration+Test+Setup
https://cirius.atlassian.net/wiki/display/DG/Runscope+Strategy
https://cirius.atlassian.net/wiki/display/DG/How+to+run+specs+in+the+Jasmine+Test+Runner

The challenge

The company has to thoroughly exercise our API endpoints onsite and eventually offsite. The scope of testing comprises at least two products, large set of resources, more than three different environments (dev, staging, production, custom), a number of different usertypes, public calls, and multi-context dependencies. We need a framework that is scalable, flexible, extendable, easy to configure, easy to maintain, and persistent.

Experiments

During the course of finding the right solution for us, we explored various options and overcome a number of difficulties that in some combination still exist in third-party frameworks.

For example, we considered using Runscope product. However, there remained at least 4 arguments against it:

- We cannot define multiple environments in which to execute the same set of tests (based on our assumption that a bucket defines the environment and tests cannot be shared between buckets)
- We cannot keep history of test definitions (the way we currently do with source control)
- We may run into trouble trying to manage many tests in Runscope; we currently have about 5000 test cases and their number is growing. Although Runscope is pretty and has nice/well designed interface, we may overpower it by the sheer number of tests we have
- Esto has integrated the in-house testing framework with Jenkins and has resolved a major pain point of managing the execution and analysis of the tests (there is no longer an immediate need for replacement)

Results

Our solution is a generic and powerful testing framework built on top of frisby, jasmine, and node. It is an in-house solution that meets our needs but can be used at large by other software companies to test their API endpoints.

The following features have been implemented.

Ease of configuration

The framework consists of three layers - control, configuration, and test specifications. The configuration layer is defined per test harness. It allows us to define a set of parameters specific to a given environment. It can host multiple environments. The kind of parameters defined is open. The information is stored in a JSON format. It gets referenced in the specifications layer.

For our purposes, we store various guids, emails, and other environmental data that when referenced by a test gets injected in it during test execution. What allows us to run the same test suite in different environments is that we refer to equivalent parameters in different environments by the same name. It is up to the caller to choose the environment and inject the data into the active test.

We can also configure default API request arguments such as the API call endpoints, headers, and timeout in the configuration layer.

The configuration layer is isolated from the other two layers and can be created and maintained without knowledge of them. It requires no technical skills but knowledge of the product and the environments the product runs in.

Snapshots

The need to re-execute a test suite exactly as it was run at some point in the past is met by using source control. This allows us to run backwards compatibility checks on new versions of the API. It provides the QA team with a lot of control.

Ease and scalability of test specifications

The framework consists of three layers - control, configuration, and test specifications. The test specifications are a set of files. Each file defines a single test. The test is defined as a JSON object. The test may be as simple as defining a single API call. However, it is also quite flexible, and, in more complex scenarios, we can define pretests and we can chain tests through callbacks. This allows us to execute multiple API calls in one test supporting verification of workflows and reusability. Each test case follows the same pattern and it is short in its definition. It only needs to be aware of the API signature and any possible environmental parameters it needs to include in the API requests. Any response validation is also defined here.

Test specifications can be written in parallel. We currently have more than 5000 specifications defined in a number of folders. Placing them in folders allows for adding granularity to a test suite scope. For our purpose, granularity is also provided by adopting a naming convention that incorporates the resource name, the API call, the usertype, the expected response status, and JIRA case.

The test specification layer is isolated from the other two layers and can be created and maintained independently. It requires knowledge of the API but no special technical skills. A test template is provided that lets you define a new test in less than 2 minutes.

A test suite is executed by passing usertype, environment, and a test case filter, if any.

Logging

The logger is part of the control layer. It outputs the API request-response details to a target of the caller’s choice. There are presently four output targets - file, console, all, none. The caller specifies what they want to do during the test invocation/execution. They also have the option to specify explicitly the output file they want to use and the level of detail in the output. Once verbosity is flagged, runtime pointers are reported in the log.

Logs are JSON objects. They are easily available for further processing.

Presently, logging serves multiple purposes:

- reporting and persisting the results of a test suite execution
- data extraction - log files are created that contain specific parameters associated with the environment in which the test suite was run. For example, we store all guids returned as a result of a POST operation. They represent any new entities created in the environment. This data can then be used in IIS and database cleanup scripts.
- debugging - log files include detailed runtime information when the test suite is executed in verbose mode. The additional data is indispensable in debugging a test or the framework itself.

Control

The framework consists of three layers - control, configuration, and test specifications. The control layer is built on top of frisby. It comprises

- synchronous
- chaining
- context
- pretests
    - running under different context
    - using “third-party” APIs (in our case using CCCAPI from MessagingAPI)
- repeat
- file upload
- pages
- anonymous
- scalability - more products can be “hosted” immediately

Extendiblity

- git integration - simultaneous development
- jenkins integration - analysis and reporting

Future work

Cleanup scripts

Some of the tests add IIS sites which are only needed for the duration of the tests. They also create a lot of database artifacts. We want to automate the process of removing test data from the system. One step towards that is the collection of entity guids. Based on that information it is easy to create a batch file that removes IIS sites or updates the database. Another way to perform a cleanup correctly is through the API itself.

We have upgraded our logger to output these guids in a set of files associated with the test’s log file. The file names include the entity type. They are stored together with the test’s log. We plan to upload them to a shared network location in an automated fashion so that they are available for download no matter where and when the test was run. Once they are downloaded, they will be subject to some additional processing in order to complete the cleanup tasks. It is required that the cleanup process can be scheduled to run regularly and that it can be triggered manually as needed. Given the input we have the option to choose From and To dates to be cleaned. This will allow us to keep some testing data for longer as needed.

CORS

unresolved...