GENERATING REPORT DOCUMENTS FROM TEST ANYTHING PROTOCOL IN JAVASCRIPT

Georgi Bogdanov, Nikolay Pavlov, Asen Rahnev

Abstract. TAP, the Test Anything Protocol, is a direct text-based interface between testing modules in a test harness. The basic assumption behind TAP is that it is a standard format useful for separating the test execution and raw data generation from the test management and reporting. The protocol loosely allows joining a set of small testing tools (TAP Producers) in a language agnostic way in order to consume the generated output (TAP consumer) and generate a unified report of the test result. This article describes a TAP consumer NodeJS CLI essential tool designed to unify a distinct set of test results and predefined set of target test requirements into a single human readable report document intended for practical use by product owners, product shareholders, business analysts, project managers, etc.

Keywords: *TAP, NodeJS, CLI, Document Generation*

1. Introduction

Modern software development typically relies and heavily demands that new software naturally includes some proper form of tests, even more so for large-scale development. Modern solutions tend to become more segmented and modular based in order to allow component based structures and support from different teams or organizations, and easier maintainability, hence it is very likely that more than one test project will exist. Furthermore, a particular project might contain various testing libraries, which perform different tests types, such as functional testing and/or non-functional testing.

Numerous testing libraries or platforms frequently mean different result output and issues with the result analysis. Enforcing a single testing library or protocol, or solution is rarely possible when dealing with multiple software

Дата на получаване: 03.12.2018 г.
Дата на рецензиране: 29.03.2019 г.
Дата на публикуване: 17.05.2019 г.

languages and environments, a library or test harness that can fit anywhere is very hard to achieve and would probably be hard to master for all the teams involved in the development. There are tools facilitating testing cross-platform but they are typically not cross-language. This constitutes the unique problem of aggregating multiple test result into a single source of truth and creating a coverage report or stylized test acceptance coverage result for example.

TAP, the Test Anything Protocol is a highly parseable, human-readable, loosely specified format for reporting test results. It rose to popularity in the Perl community, with CPAN's Test family [1]. The protocol defines a philosophy that all parsers should follow [2]:

- Should work on the TAP as a stream (i.e. as each line is received) rather than wait until all the TAP is received.
- The TAP source should be pluggable (i.e. don't assume it is always coming from a Perl program).
- The TAP display should be pluggable.
- Should be able to gracefully handle future upgrades to TAP.
- Should be forward compatible.
  - Ignore unknown directives
  - Ignore any unparsable lines

In order to achieve tests result standardization and unification, TAP will be used as a medium between the different modules and testing harnesses. Utilizing this, we can develop a tool that will work as a TAP consumer and later on use the consumed data to generate test acceptance results document.

2. Requirements

The following tool requirements are provided:

- It must be platform independent and not require GUI
- It must be able consume TAP output from different producers
- It must be able to accept an additional user provided data
- It must be able to output stylized ctemplate with test acceptance results

2.1. Platform independence and text based interface

The users of the tool want be able to use the it on different platforms (Windows, Linux. Mac) because the modules they are testing and generating test results for will be on multiple platforms. Additionally, the place where applications and tests are ran will include servers or virtual environments where GUI is often excluded from the platform or system in order to conserve system resources.

NodeJS is a good solution because:

- It is a multiplication event driven JavaScript run-time. It can be easily installed on all the required platforms.
- Allows creating text-mode programs or command-line interfaces (CLI) which can be globally registered into a given operating system and used anywhere in that OS.
- Has existing integration with TAP consumers/producers via numerous libraries
- Strong ecosystem with hundreds of thousands of packages available for all manner of purposes

2.2. Must consume TAP output form different producers

NodeJS allows consuming stream output and reading file streams.

2.3. The additional user provided data

Users need to be able to provide external data related to the business logic or the templates for generating documents.

2.4. Stylized templates

Users have to be able to define and edit or create their own templates to fit specific needs.

3. Implementation and functionality

3.1. Defining common conventions in the different test suites and packages

The common conventions are defined to ease the test process information negotiation and handling so that this info can be easily processed and mapped or read. Defining the conventions has to conform and be aligned with the most familiar language employed in the component tests and possibly with the language/library that will handle the TAP consumption. TAP output is streamed in a way that will allow maximum useful data output and minimum loss of valuable test process information.

First, the internal test conventions and process were defined so that each test would utilize a standard method that generates and handles each test's meta information stream to the TAP output. This common process operates a minimal set of methods.

In the case with selenium tests for example, just two methods were used, one to setup the test description and one more to setup the test standard output.

```
let testDescription = {
  objective: 'Create and run example',
  preConditions: 'Web Browser',
  configuration: '',
  duration: '15 seconds',
  coverage: 'Dashboard',
  note: ''
}

let steps = [
  'step_1:Click navigation',
  'step_2:Click Create button',
  'step_3:RQOD_4881'
]

function tapOutputDescription (testDescription, t) {
  for (const propertyName of testDescription) {
    if (testDescription.hasOwnProperty(propertyName)) {
      t.log(`${propertyName}: ${testDescription[propertyName]}`)
    }
  }
}

function tapOutputStep (steps, next, t) {
  t.log(steps[next])
  return next++
}
```

To setup the description in the beginning of each test we manipulate a simple object as template and it is populated with a key-value pair and this information is later be handled by the TAP consumer CLI tool.

For the steps output during test execution, we use a predefined key for steps and increasing number sequence to show a human-readable indication, as a value we use either a brief functional description provided by QA, developer, business-analyst or a unique ID key that adheres to a requirement defined by Product owner, client, business-analyst or QA.

The main process of executing each test then outputs the test name, test description meta data and then on each relevant step, the corresponding steps' key-value output information. This way the test goal reach and value of completion is clear and transparent and easy to inspect, when the tests are run. It is easy to detect the tests percent of successful completion. If the test was ran without any issues or errors, the full steps output will be visible and if there is an issue somewhere inside the test execution it will be easier to spot and identify, AND easier to fix.

When errors in the test execution are detected earlier during that test definition or execution this saves time and development effort. Defining a better general output from the tests allows for non-developer or non-technical personnel to read and understand if there are any issues, and this improves the time required for fixing. A clear and transparent process saves time in communication handling.

3.2. Handling test suites without standard TAP output

In some cases when the test suite does not have TAP standardized output it has to be simulated using available output from the test suites platform (eg. Postman).

In Postman for example, similar test meta description and steps execution were used, but in this case an output method was added so that TAP output could be simulated and stored into a file for example.

```
  var jsonData = pm.globals.get('testDescription')
  tests["objective: " + jsonData.objective] = true // debug message
  tests["preConditions: " + jsonData.preConditions] = true // debug
message
  // or
  pm.test("objective: " + jsonData.objective, true) // debug message
  pm.test("preConditions: " + jsonData.preConditions, true) // debug
message
```

The TAP output has to be gathered/read later by the TAP consumer, in most cases the output can be read as a stream and processed simultaneously, but in cases when we have to simulate TAP output the data is often read from a file and concatenated to the stream data.

3.3. NodeJS CLI interfacing

NodeJS is extremely useful for creating command-line applications [3][4]. In order to speed up the time for implementation and instead of reinventing the wheel a number of third-party packages designed to help work with the command line was selected for use:

- chalk – for colorizing the output

- clui – additional visual components
- clear – to clear the console
- Inquirer – a collection of sophisticated interactive command line user interfaces.
- Stream – for working with streaming data
- tap-merge – merge multiple TAP streams
- tap-parser – parsing test anything protocol

In order to comply with the TAP Philosophy the CLI tool is developed so that it can accept both stream from multiple sources and file input. The user that runs the CLI tool is responsible for providing the arguments that point to a file and/or a stream and issuing the correct commands or additional argument to initialize interfacing with the tool. Executing of the TAP consumer can be attached either after each test is ran to directly consume it is results, or as a separate process executed by the user.

The application is created so that it can accept XML or CSV file formats as data input used to match test description and unique ID keys and include supplementary information for either the test and it is coverage or for the template and the final document which will be generated and presented to the end users.

3.4. Templating

For the final document output the user can provide a specific template file to be used. The user can either point to the new template using the provided interface commands or specify it as an argument to command prompt call to the application. If the user does not provide any specific template to be used, by default, the application provides a predefined Handlebars HTML template with an explicit definition of available variables.

The initial design of the default template was selected so that it will mimic the official test results documents presented to the clients and product owners. The fundamental concept is that the users will rarely have to alter the output style and design.

Handlebars was picked as first template language of choice because it delivered enough flexibility in the template, allows creating semantic templates efficiently, the templates can be essentially logic free (where the code and view are isolated) helping to preserve the separation of concerns principle, and it provides partials and helper methods defined in the code to enable better parsing of the input data [5]. Another significant factor was that It is using a well know syntax for the development teams of the software components and applications.

```
handlebars.registerHelper("testOk", v => {
  return new handlebars.SafeString(
```

```
      v
        ? '<span  style="background-color:yellow;">NOK</span>'
        : '<span  style="background-color:#42f448;">OK</span>'
    );
});
handlebars.registerHelper("breaklines", function(text) {
    text = handlebars.Utils.escapeExpression(text);
    text = text.replace(/(\r\n|\n|\r)/gm, "<br>");
    return new handlebars.SafeString(text);
});

handlebars.registerHelper("testReportProperty", (property, v) => {
    return module.exports.extractPropertyFromSystemOut(property, v);
});

handlebars.registerHelper("transliterateOutput", (property, v) => {
    return module.exports.transliterateOutput(property, v);
});
```

The originally manually created test acceptance result documents were made with Microsoft Word and the generated documents based on handlebar templates were a little different when compared, so a second templating format was added to support specific needs of some of the end users.

In order to obtain maximum conformity with Microsoft Word, the file format used was docx templates in combination with “docx-templates” [6] NodeJS library. With the following combination, the users can write their templates in a well-known format simply using placeholders in the document and then have the application replace these placeholders with the proper contents. The decision to use docx templates with placeholders or queries in them was made because most of the libraries for NodeJS available do not adequately support all of Microsoft Word features and It would have been harder to generate specific documents using code alone.

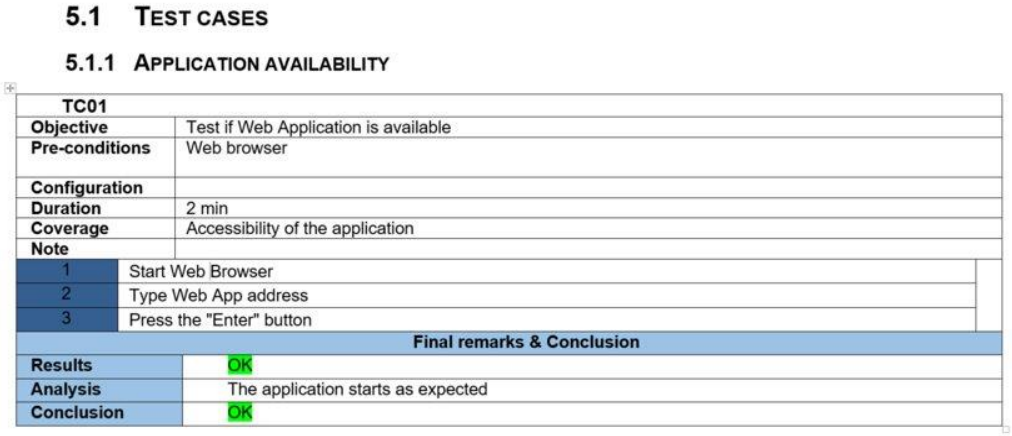
3.5. Output

The ultimate output of the processed templates obtain a finished document that can be viewed by all users and targeted audience. The supported output formats include:

- PDF – handlebars and docx templates are converted to PDF output file;

- HTML – handlebars template is outputted as a file;
- DOCX – supported using handlebars and docx templates;
- Email* – html contents email using handlebars template stylized for emails (has specific CSS support compared to normal browser viewed html).

With the above-mentioned output formats, the needs of a diverse audience of end-users are satisfied.

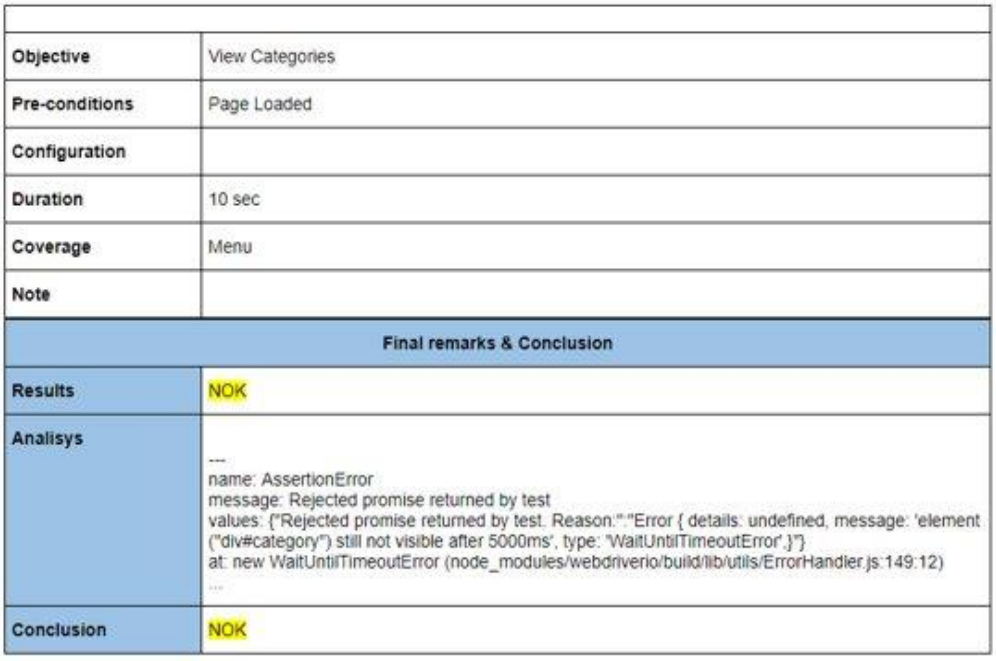


5.1 TEST CASES

5.1.1 APPLICATION AVAILABILITY

TC01	
Objective	Test if Web Application is available
Pre-conditions	Web browser
Configuration	
Duration	2 min
Coverage	Accessibility of the application
Note	
1	Start Web Browser
2	Type Web App address
3	Press the "Enter" button
Final remarks & Conclusion	
Results	OK
Analysis	The application starts as expected
Conclusion	OK

Figure 1. DOCX Output



Objective	View Categories
Pre-conditions	Page Loaded
Configuration	
Duration	10 sec
Coverage	Menu
Note	
Final remarks & Conclusion	
Results	NOK
Analisys	--- name: AssertionError message: Rejected promise returned by test values: {"Rejected promise returned by test. Reason:":"Error { details: undefined, message: 'element ("div#category") still not visible after 5000ms', type: 'WaitUntilTimeoutError',}"} at: new WaitUntilTimeoutError (node_modules/webdriverio/build/lib/utils/ErrorHandler.js:149:12) ...
Conclusion	NOK

Figure 2. HTML Output

4. Conclusion

The final NodeJS CLI tool for generating test acceptance results report was tested and carefully verified to comply with the required functional requirements

and its implementation was used to generate real production report documents. The resulting use of the application will save developers, business analysts, quality assurance engineers and product owners many hours in creating documentation and verifying that all tests are running properly and in compliance with the defined requirements. The development of the tool will continue by expanding it further with more templates and options, further analysis of it is value, benefits and its use in projects will be performed in the future.

Acknowledgements

This paper is partially supported by project FP17-FMI-008 of the Scientific Research Fund of Plovdiv University "Paisii Hilendarski", Bulgaria.

References

[1] Test Anything Protocol | Node Tap, c2015–2019, https://www.node-tap.org/tap-format/

[2] TAP Philosophy - Test Anything Protocol, c2015–2019, https://testanything.org/philosophy.html

[3] Writing Command-Line Applications in NodeJS, Peter Benjamin, 2015 https://medium.freecodecamp.org/writing-command-line-applications-in-nodejs-2cf8327eee2

[4] Build a JavaScript Command Line Interface (CLI) with Node.js, Lukas White, Michael Wanyoike, 2018, https://www.sitepoint.com/javascript-command-line-interface-cli-node-js/

[5] Generate a Resume in DOCX and HTML at the Same Time, Mike Bybee, 2015 https://mikebybee.com/blog/generate-resume-in-docx-and-html-at-the-same-time

[6] Word documents, The Relay Way. Bridging Word templates, GraphQL queries and JavaScript snippets, Guillermo Grau, 2018 http://guigrpa.github.io/2017/01/01/word-docs-the-relay-way/

Faculty of Mathematics and Informatics

Plovdiv University "Paisii Hilendarski"

236 Bulgaria Blvd, 4003 Plovdiv, Bulgaria

E-mail: georgi.bogdanov@outlook.com, nikolayp@uni-plovdiv.bg, asen@uni-plovdiv.bg

ГЕНЕРИРАНЕ НА СПРАВКИ ОТ ПРОТОКОЛА TEST ANYTHING ЧРЕЗ JAVASCRIPT

Георги Богданов, Николай Павлов, Асен Рахнев

Резюме. Протоколът Test Anything Protocol (TAP) е пряк текстов интерфейс между тестови модули в тестова система. Основната идея на TAP е да бъде стандартен формат, който спомага за разделянето на изпълнението на тестовете и генерирането на първични данни от управлението на тестовете и изготвяне на справки. Протоколът позволява слабо свързване на малки инструменти за тестване (TAP Producers) по независим от езика начин, да се консумира произведения резултат (TAP consumer) и да се генерират унифицирани справки за резултатите. Тази статия описва TAP Consumer инструмент на NodeJS, проектират да обедини определено множество от тестови резултати и предефинирани тестови изисквания в една обща справка, която да може да бъде разчетена лесно от човек и предназначена за ръководители на продукт, ключови участници в проекта, бизнес анализатори, ръководители проекти и др.