Karma- The Test Runner, for Automated Testing of Web Based Applications

Pranay Dutta[1], Ashok Verma[2], Jitendra Prithviraj[3]
Student of M.E (Software Systems)[1], Professor & HOD (CSE)[2], Asst . Prof (CSE) [3]
Gyan Ganga Institute of Science & Technology
Jabalpur

Abstract - **Software testing plays a vital role in verifying software functionality and avoid bugs in code. Automated tests using code instead of conducting manual tests can reduce the amount of tedious work during the development process by huge amount and will definitely improve the software quality. The most widely adopted language for developing the web applications is JavaScript, the obvious reason is its very dynamic nature. Ironically compiler's could not grab errors like a variable name which has been misspelled or an object invoking a non existing method . In order to catch the aforesaid errors the system developers have to actually sprint the code. In view of the above testing is absolutely necessary.The test runner Karma, helps the web application developers to be more prolific and efficient by making automated testing simpler and faster. Karma has been successfully tested on many avenues, including software giants such as Google and YouTube.The motto describes the intend and execution of Karma, and the logic behind them.**

Keywords : Karma, Software Testing, JavaScript[11].

I. INTRODUCTION

The wide diffusion of Internet has produced a significant growth of the demand of Web-based applications with more and more strict requirements of reliability, usability inter-operability and security. Due to market pressure and very short time-to-market, the testing of Web-based applications is often neglected by developers, as it is considered too time-consuming and lacking a significant payoff. This depreciable habit affects negatively the quality of the applications and, therefore triggers the need for adequate, efficient and cost effective testing approaches for verifying and validating them.

During code refactoring or execution of new features in software, errors often occur in existing parts. This may have a serious blow on the reliability of the system. Therefore it is a global practice to imply automated testing to verify the functionality of software in order to detect software flaws before they end up in a live setting. The most widely adopted language for developing the web applications is JavaScript [11]; the obvious reason is its very dynamic nature. Ironically compiler's could not grab errors like a variable name which has been misspelled or an object invoking a non existing method . In order to catch the aforesaid errors the system developers have to actually sprint the code. Therefore an automated testing tool is the absolute need of time.

This paper revolves around the automated Test Runner –Karma, which incessantly runs all the tests in backdrop (without off-putting the developer), every time any part of the file is changed. Eventually the productivity is improved and the faith of a developer in automated testing increases. The motto of this dissertation describes the intend and execution of Karma, and the logic behind them. Going ahead it describes various existing automated tools have been compared with Karma. Further the design & implementation of Karma has also been described in the subsequent chapters.

II. JAVASCRIPT & IT'S INEVITABLE APPLICATIONS

JavaScript[11] is definitely the most open programming language. JavaScript isn't the pinnacle of elegance. However, it is a very flexible language (high level language), has a reasonably elegant core, and enables to use a mixture of object-oriented programming and functional programming.

One important attribute of JavaScript is its dynamic behavior, any developer can create things very quickly, especially in the beginning phase of a project. Further it is also flexible and productive. The dynamic behavior comes with its own confrontations.

There is no static typing, which makes type checking and IDEs support hard. Convention for structure of a project are not settled yet and therefore projects typically end up with very messy codebase. Additionally, there are many inconsistencies between different browsers, especially when it comes to dealing with Document Object Model (DOM).

In order to overcome the aforementioned shortcomings, the following measures have been taken:

• Model View Controller frameworks (eg. AngularJS) are helping with overall application structure and rising the level of abstraction by offering bigger building blocks.
• Web Browsers are made more smart and witty.
• The freshly evolved languages that eventuallycompiles to JavaScript (eg. CoffeeScript,) are improving the syntax and adding features to the language.
• The inconsistencies between different browsers are fixed by providing higher level APIs(as an intergrated part of Libraries eg. JQuery[13]).
• The experience-based techniques used in different IDE's are being improved for dealing with dynamic languages and bringing refactoring tools for easier coding .
• Various other new languages that compiles to JavaScript (eg. TypeScript, Dart) are bringing static types into the language.

III. DESCRIPTION OF THE PROBLEM

Web applications run in a web browser. This means that in order for the developer to test if a code change did not break the system, he needs to open the browser, load the app and visually check if it works. This is the core issue - the workflow. The missing compiler is not that big of an issue, because the JavaScript virtual machines in web browsers have typically very quick bootstrap and so running the code can be faster than compiling C++ or Java code. The core issue is the workflow - the developer has to switch context from the editor to the browser, reload the page and visually check the app, probably even open the web inspector and interact with the app to get it into a specific state.

A. Objective

The main objective is to create a tool - a test runner, that helps web application developers to be more productive and effective by making automated testing simpler and faster. Further another goal of this paper is to promote Test Driven Development (TDD) . There are two essential prerequisites that the developer needs in order to successfully apply TDD:

• Testable code

• Testing environment

Writing testable code, is where AngularJS and the philosophy behind it come in. AngularJS guides developers to write loosely coupled code that is easy to test.

Karma applies to the second prerequisite - its goal is to bring a suitable testing environ- ment to any web developer. The overall setup and configuration has to be straightforward, so that it is possible to start writing tests in a few seconds and instantaneously see test results on every code change. The following features are also important.

- Resolving Cross Browser issues

There are many cross-browser issues and one of the goals of testing is to catch these issues. A cross-browser issue means that some browsers implement an API slightly differently or does not implement some features at all. Different browsers running on different Operating Systems (OS) can also have different bugs. That is why Karma needs to be able to execute the tests on any browser and even on multiple devices such as a phone or a tablet.

- Remote Control

The basic workflow of running the tests on every save should work automatically without any additional interaction. The developer should just save a file and instantly see test results, without leaving the text editor.For other tasks such as triggering the test run manually, Karma needs to be fully con- trolled from a command line. That will allow easy control from any IDE as well.

- Speed

Minimizing the space by allowing developers to run the tests on every file save. If the developer runs the tests every time a file is saved, it is easy to spot the mistake, because the amount of code written since the last run is small.

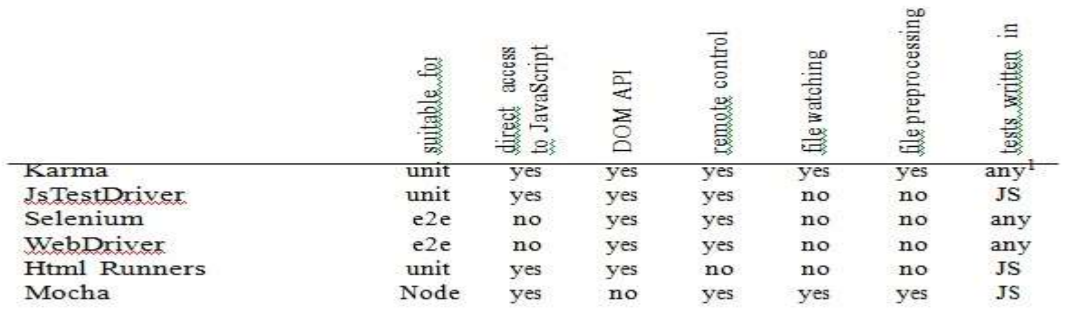
- Integration with IDEs and text editors

Karma needs to be agnostic of any text editor or IDE. It needs to provide basic function- alities such as watching source files and running the tests on every change or running only a subset of tests, without any support from the text editor or IDE.

B) Already Existing Automated Test Runners

The following sections describe existing solutions for testing web applications. Most of these solutions are suitable for either low level testing, such as Mocha or JsTestDriver, or high level testing, such as Selenium or WebDriver. Therefore these solutions are not necessarily competing - they are rather complementing each other.

- Mocha
- JsTestDriver[11]
- Selenium[2]
- WebDriver / Selenium 2
- Html Runners



	suitable for	direct access to JavaScript	DOM API	remote control	file watching	file preprocessing	tests written in
Karma	unit	yes	yes	yes	yes	yes	any[1]
JsTestDriver	unit	yes	yes	yes	no	no	JS
Selenium	e2e	no	yes	yes	no	no	any
WebDriver	e2e	no	yes	yes	no	no	any
Html Runners	unit	yes	yes	no	no	no	JS
Mocha	Node	yes	no	yes	yes	yes	JS

Comparison of existing solutions and their features

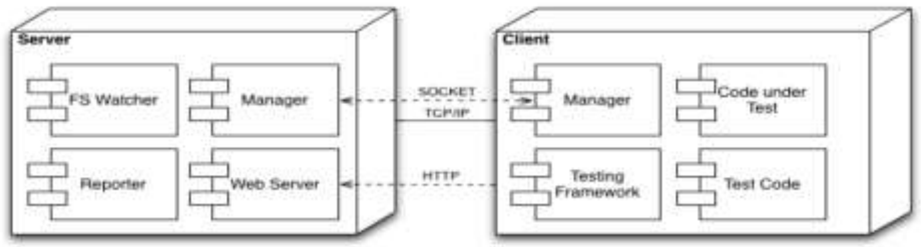
All these features share a single goal - a seamless workflow, which provides fast and reliable results, without distracting the developer. That is the core new feature that Karma brings into the web development.

IV. DESIGN

The major factor that were considered for the the design and implementation were:

• speed,
• reliability,
• testing on real browsers and
• seamless workflow.

The overall architecture model of the system is client-server with a bi-directional communication channel between the server and the client. Typically, the server runs on the developer's local machine. Most of the time, the client runs on the same physical machine, but it can be anywhere else, as long as it can reach the server through HTTP**.**



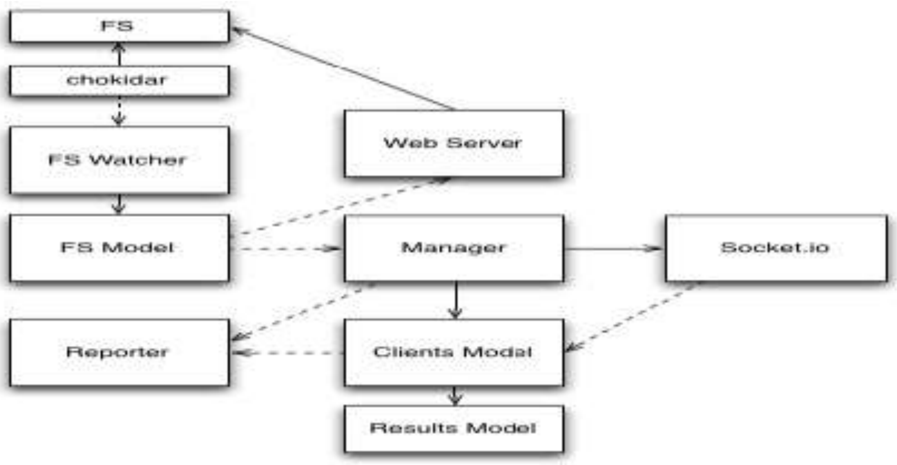
The high level architecture

A) Experimental Setup & Implementation

Karma is implemented in JavaScript. The server part runs on Node.js (Node) and the client part runs in a web browser. Implementation of the two major parts of the system - the server and the client - and how they communicate are described as under:

<u>Server</u>

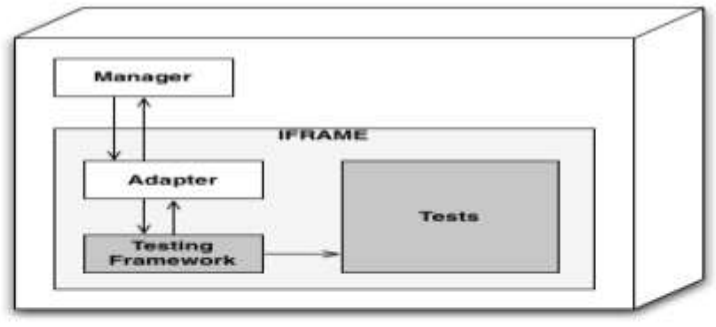
The below mentioned figure shows all the server components and how they communicate. The solid lines represent direct method invocation, the dashed lines represent communication through events**.**



Server components

Client

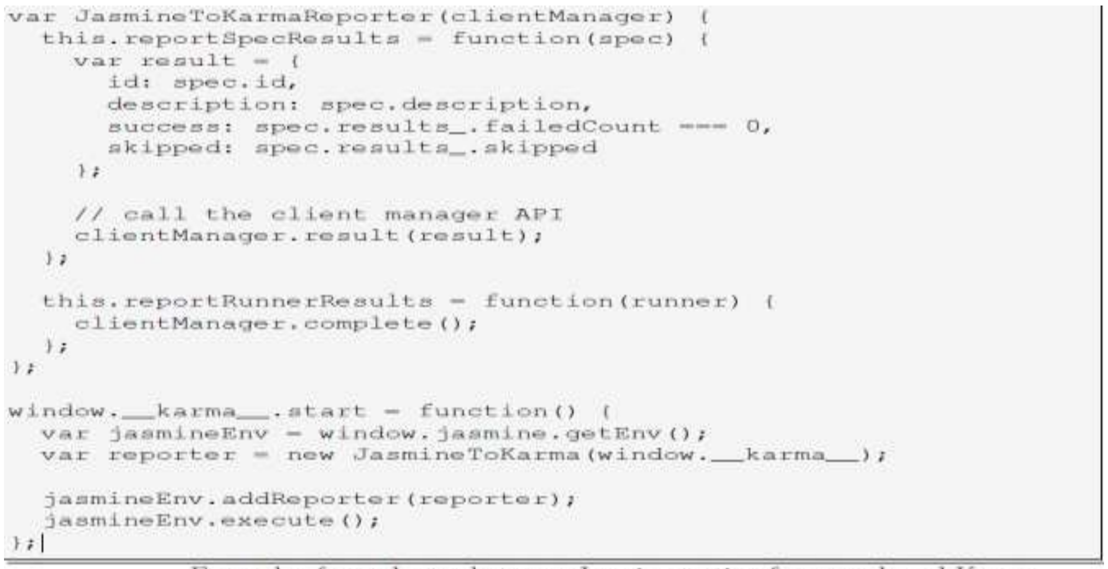
The client part of the system is where all the tests are actually executed. The client is more or less a web app, that communicates with the server through socket.io and executes the tests in an iframe.



Overall structure of the client

B) Communication Between Server and Client

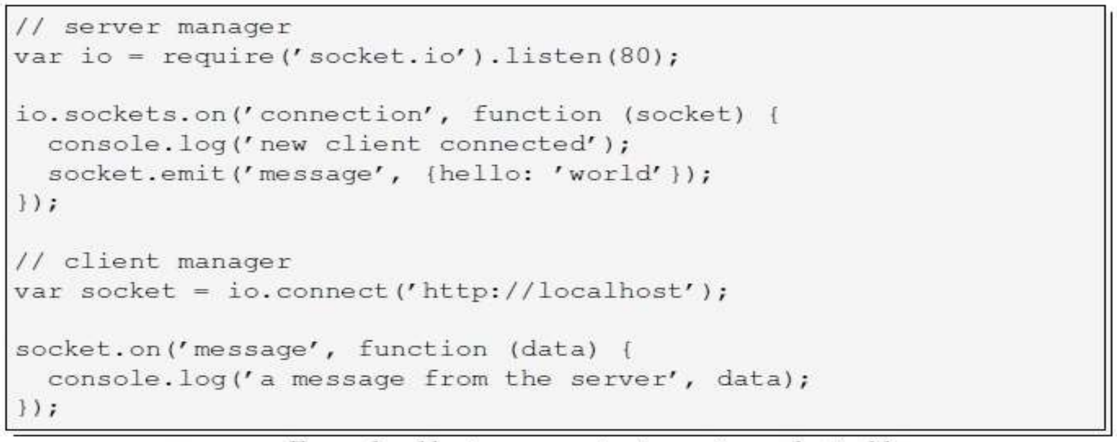
The communication between client and server is implemented on the top of socket.io library. The socket.io library provides an event based communication channel. The below mentioned code shows a simple example of such a communication - server manager emits events that client manager is listening on and vice versa. The socket.io library implements multiple protocols such as WebSocket, XHR polling, HTML polling, JSONP polling and uses the best one available on given web browser.



```
var JasmineToKarmaReporter(clientManager) {
  this.reportSpecResults = function(spec) {
    var result = {
      id: spec.id,
      description: spec.description,
      success: spec.results_.failedCount === 0,
      skipped: spec.results_.skipped
    };

    // call the client manager API
    clientManager.result(result);
  };

  this.reportRunnerResults = function(runner) {
    clientManager.complete();
  };
};

window.__karma__.start = function() {
  var jasmineEnv = window.jasmine.getEnv();
  var reporter = new JasmineToKarma(window.__karma__);

  jasmineEnv.addReporter(reporter);
  jasmineEnv.execute();
};
```

Example of an adapter between Jasmine testing framework and Karma

Client to Server Messages
The following table shows a list of implemented events between client and server -these messages are sent from client to server. Calling the client API methods
emits these events. Socket.io layer is responsible for encoding the data into JSON and sending them to the server. On the server, there is the server manager listening on these events and dispatching them to interested server components.



```
// server manager
var io = require('socket.io').listen(80);

io.sockets.on('connection', function (socket) {
  console.log('new client connected');
  socket.emit('message', {hello: 'world'});
});

// client manager
var socket = io.connect('http://localhost');

socket.on('message', function (data) {
  console.log('a message from the server', data);
});
```

Example of basic communication using socket.io library

Event	Description
Register	the client is sending its id, browser name and version
result	a single test has finished
complete	the client completed execution of all the tests
error	an error happened in the client
info	other data (eg. number of tests or debugging messages)

Events emitted by client

Event	Description
info	the server is sending information about its status
execute	a signal to the client that it should execute all the tests

Events emitted by server
Further caching is done for optimizing the speed and Batching is done for multiple file changes

C) Testing

This section describes how Karma itself is tested, using testing onmultiple levels.The entire project has been developed using Test Driven Development practices since beginning. There are multiple levels of tests (~ 250 unit tests for the server side code, ~35 unit tests for the client side code and 12 end to end tests that test Karma as it is).

Server Unit Tests
Originally, server unit tests were written in Jasmine [38] and run using jasmine-node. The main motivation for rewriting them into Mocha was a better test runner (Mocha is better runner than jasmine-node) and better syntax for asynchronous testing.
Client Unit Tests
Unit tests for the client code are located in test/client/*. These tests are written using Jasmine as the testing framework. This code runs in browsers and therefore needs to be tested on real browsers. Of course, we use Karma itself to run these tests.

V. CONCLUSION & FUTURE WORK

The main contribution of this paper is a framework for automated testing of web applications. Karma has been proved that it is the most effective and swift way to automatically test web based applications. Karma has already proven itself to be a helpful tool, but there is still a lot of space for improvements. Some of the possible improvements are:

- More efficient file watching

Some amount of improvement is required in the chokidar library to use fs.watch as much as possi- ble, especially on Linux.

- Collecting Usage Statistics

Karma is being used by many users, but we do not have any exact numbers.Therefore collecting anonymous user data would help us understand the users better and make better decisions based on that knowledge.

- Integration with Browser Providers in the Cloud

There are services that provide browsers for testing. It would be very convenient if we had plugins for launching these remote browsers. Then, the developer could easily run the tests on almost any browser and any device, without actually having it.

- Parallel Execution

We can speed up the actual execution in the browser - by splitting the tests into smaller chunks and executing them in multiple browsers in parallel. Either on a single machine or using multiple machines**.**

REFERENCES

[1] The Ruby Toolbox: Code metrics, . URL https://www.ruby-toolbox.com/categories/ code_metrics. Accessed 2014-04-29.
[2] Selenium (software), . URL http://en. wikipedia.org/wiki/Selenium_%28software%29. Accessed 2014-05-19.
[3] Test automation, . URL http://en.wikipedia. org/wiki/Test_automation. Accessed 2014-05-21.
[4] Dorota Huizinga and Adam Kolawa. Automated defect prevention: best practices in software man-agement. Wiley-Interscience, IEEE Computer So-ciety, 2007. ISBN 9780470042120.
[5] Jeﬀ Kramer and Orit Hazzan. The role of ab-straction in software engineering. In Proceedings of the 28th international conference on Software en-gineering, pages 1017–1018. ACM, 2006.
[6] R. Lacanienta, S. Takada, H. Tanno, X. Zhang, and T. Hoshino. A mutation test based approach to evaluating test suites for web applications. Fron-tiers in Artificial Intelligence and Applications, 240, 2012.
[7] Upsorn Praphamontripong and Jeﬀ Oﬀutt. Ap-plying mutation testing to web applications. Sixth Workshop on Mutation Analysis (Mutation 2010), 2010.
[8] X. Zhang, T. Hu, H. Dai, and X. Li. Software devel-opment methodologies, trends and implications: A testing centric view. Information Technology Jour-nal, 9(8):1747–1753, 2010. ISSN 18125638.
[9] Selenium WebDriver - Documentation.<http://docs.seleniumhq.org/docs/03_webdriver.jsp> .
[10] Travis CI. <https://travis-ci.org/> .
[11] Google, Inc. JsTestDriver - Remote JavaScript console. <https://code.google.com/p/js-test-driver/>
[12] Joyent, Inc. Node.js. <http://nodejs.org> .
[13] The jQuery Foundation. QUnit: A JavaScript Unit Testing framework. <http://qunitjs.com/> .
[14] Pivotal Labs. Jasmine - A JavaScript Testing Framework. <http://pivotal.github.io/jasmine/> .