# CoffeeScript & Jasmine Testing - Tool Support for Cloud9 IDE

Tobias Pfeiffer, Tobias Metzke

Hasso-Plattner-Institute, Potsdam, Germany

**Abstract.** Proper application testing leads to higher quality code on the long term. Thus, development environments encouraging and supporting programmers in software testing clearly benefit the quality of the produced code. This paper investigates the capabilities of web-based IDEs in terms of software testing support and integration. An example IDE has been explored by an prototype application written in CoffeeScript using the TDD methodology. Furthermore, the shortcomings discovered have been countered by proper tool support, integrated in the IDE. The tools and their benefits described in this paper contribute to a more encouraging web-based IDE that could serve as a replacement for traditional development environments in the near future.

## 1 Introduction

Applications that used to be on desktops only can now be found in browsers with all their former functionality at hand. What is more, the capabilities of the world wide web are used to enrich these systems even more.
Software development environments are likely to follow this trend as well. In this paper, we investigated the *Cloud9 Online IDE* [1] and its functionality. We chose this specific web-based IDE because it already had a good reputation among different development communities and was recommended by Trevor Burnham [20]. To explore the capabilities of the IDE, we implemented an example application with the help of the IDE.
The Cloud9 IDE is mainly directed at JavaScript developers. Thus, JavaScript could be the choice of language for implementing the prototype application. However, we chose *CoffeeScript* [4] for the implementation, as it compiles into JavaScript, provides syntactic simplification over JavaScript. It also enabled us to investigate the flexibility of the IDE when it comes to using relatively new languages.
Furthermore, the investigation focused on the support for the Test-Driven Development (TDD) [2] work flow[1] by the IDE. As described by Jeff Langr [3], writing tests in TDD methodology most often results in tests of higher quality and a cleaner code base. This is why we favor writing our applications in TDD

[1] TDD is based on the thought that the tests should be written first, e.g. before the implementation. This way every line of code written in a TDD project should be tested.

methodology where applicable. We therefore regard the support of the TDD cycle as a quality measure of an IDE.
The insights gained by the research are discussed in this paper. As a our main contribution, we added tool support for the IDE based on the shortcomings found in this investigation. We described the new tools and displayed their benefits. This was done by comparing the work flows used before and after the tools were plugged into the Cloud9 IDE.
This paper first gives an overview of work related to ours in the following section. Afterwards, Section 3 describes the insights gained by investigating the Cloud9 IDE with the help of an example application. After presenting the new tool support in Section 4, this paper concludes with a brief summary in Section 5.

## 2 Related Work

Web-based software development environments are getting increasingly popular, which can be depicted from their sheer number [1], [6], [7]. This trend most likely has its roots in the fact that more and more desktop applications are migrated to the web. Many of these IDEs work just like desktop applications, however there are other IDEs like Lively Kernel [5] leveraging the web-based concept more. These are more unlike regular desktop IDEs.
However the development environments tend to become in-browser application. Thus they need to compete with traditional desktop-based IDEs.
All of the online IDEs have in common that they are accessible from everywhere with nothing but a browser. They do not have to be set up and can be used from whatever computer while providing basically the same user experience. Depending on the concept of the IDE they even take load of the pc that is used for development, as the programs are executed on servers and not on the local machine. Collaboration features can also often be found in web-based IDEs.
There are also downsides compared to classic IDEs. Web-based IDEs need to be constantly connected to the Internet which is a well-known handicap. Another common drawback is that online IDEs are often not as feature rich as traditional desktop-based IDEs. This might be due to technological limitations and the fact that they are not as mature as their desktop siblings.
The support for the language of our choice, CoffeeScript [4], differs greatly between different IDEs and editors. There are plug-ins for nearly every commonly used programming environment [8]. Syntax highlighting is supported by nearly all of them. However automatic compilation is not a feature of most of them. To the knowledge of the authors no IDE provides an approximate line mapping between JavaScript and CoffeeScript as of now, since this is a hotly debated topic in the community[2].
Good support for testing applications is also an important topic for conventional IDEs - most of them include a test-runner or test-runners can easily be downloaded as a plug-in. For instance this is the case for RubyMine [9] and

[2] Issue at the CoffeeScript repository discussing the need for line mappings `https://github.com/jashkenas/coffee-script/issues/558`

Eclipse [10]. The need for good JavaScript testing testing support has also recently been addressed by JetBrains [11].

## 3 Working with the Cloud9 IDE

The Cloud9 IDE [1] is an online development environment, designed to support a wide range of programming languages including *Ruby* and *Python*. However, the execution and support of *JavaScript* applications with the help of *Node.js* [12] is its main purpose so far. Since the IDE is relatively new and in steady development, the number of features and supported languages is continuously evolving. The features include live collaboration, code deployment to services like *Heroku* [13], in-browser code execution and code completion. Furthermore, there also is an offline version of the IDE which can be installed and used locally without internet connection.
To investigate the potential of the Cloud9 IDE we developed an example application in CoffeeScript with the help of the IDE. The goal of this application was to recommend a set of articles that could interest the reader of a given article. The functionality of the application is further described in the following section.

### 3.1 Example Application

The example application *CoffeeRecommender* [27] is a recommendation engine based on the *Plista* contest[14]. We used it to investigate the features of the Cloud9 IDE. Based on the information provided by a given web article[3], this application finds related articles in a set of given articles[4]. The goal is to find the ones that appear most interesting to the reader of the given article. Figure 1 show the application, embedded in the work flow of its environment.
The application, written in *CoffeeScript* in the Cloud9 IDE, has been developed in TDD methodology. Therefore, we used the *Jasmine*[15] testing framework. Consequently, every new feature was first manifested in a set of tests that described the way it is used and the expected results. The work flow we went through for implementing a new feature is illustrated and discussed in the next section.

### 3.2 Work Flow

Following a Test-Driven Development work flow when implementing the example application, we went through the development work flow shown in Figure 2 most of the time. It describes the steps we had to go through when a code change led to failing tests. All of the steps shown in this figure are an abstraction of all the actions the Cloud9 user has to go through.
The 5 phases consist of the following actions:

[3] this article contains information such as: title, subtitle, category, first paragraph of the content and further more
[4] These articles are provided by the Plista team in a training step.

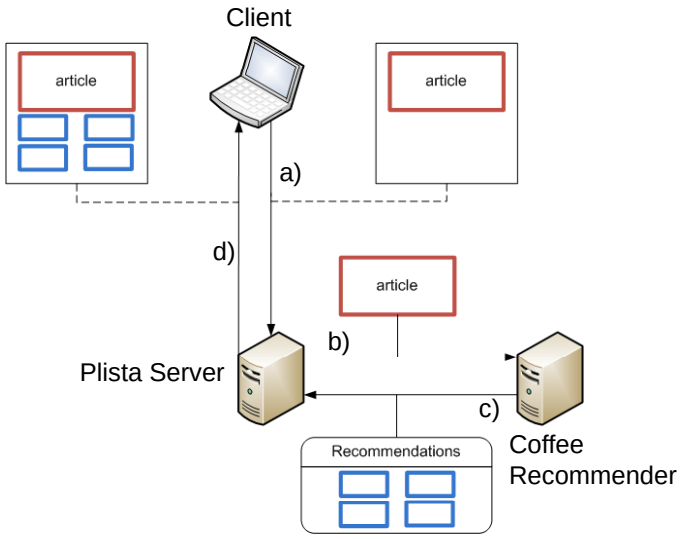


**Fig. 1.** The CoffeeRecommender embedded in its environment. This figure is an abstraction of the environment and leaves out intermediary entities. a) When a user chooses an article on a website the Plista server is asked to provide articles that can be recommended for the reader based on the requested article. b) The recommendation request is forwarded to the CoffeeRecommender. c) The application finds the best known matches for the given article and returns them to the Plista Server. d) The Plista server returns the recommendations that are then embedded in the website of the requested article.

**write code** The programmer writes tests and the feature code that should make these tests pass and saves the changes.

**type 'npm test' in console** The programmer runs the tests by switching to the console, typing this command and running all the tests present due to missing possibilities of only passing on single tests for execution.

**find error in console log** In the console output of the test results the programmer has to look for the error she produced with the code she wrote. Figure 7 in the Appendix shows an example of such an output. The time needed to find the corresponding line in that mass of lines can become a major barrier for testing the code at all.

**map JavaScript error to CoffeeScript** Since CoffeeScript is translated to JavaScript and then executed, the output deals with compiled JavaScript code. When the corresponding line is found in the console output it has to be mapped back to the written CoffeeScript code. Due to missing context and sparse information on the error in the console, the line has to be mapped manually by the programmer.

**find line in CoffeeScript file** Often, the two lines the programmer is interested in tell which file of the own project fails highest in the stacktrace and

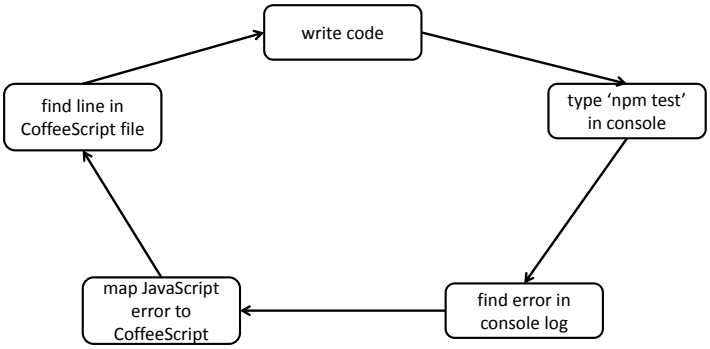


**Fig. 2.** Work Flow in Cloud9 IDE when following TDD with CoffeeScript and Jasmine framework

what the error is. The programmer then navigates to the failing file. Unfortunately, the lines given in the console output correspond to the lines in the translated JavaScript file. Identifying the corresponding line in the CoffeeScript file with the help of the given error message can take a lot of time with a growing code base. This again increases the risk of creating a barrier to testing the code base at all.

Ideally, the work flow to implement a new feature would be highly supported by an IDE to enable fast enrichment of the developed application. Unfortunately, when following TDD and using CoffeeScript in the Cloud9 IDE the programmer does not get the support needed for rapid development. How this has been improved by proper tool support by us is shown in Section 4.

### 3.3 Benefits

Despite the shortcomings in the testing work flow, an online IDE like the Cloud9 IDE provides a series of benefits. First of all, an online development environment allows access to the project independent from a specific computer. Provided with a computer and an internet connection, a programmer can instantly work on the code base from basically everywhere. What is more, collaborative work on the code in real-time becomes possible. Since the newest update of the Cloud9 IDE, programmers can share their code with others, grant write access and code collaborative in real-time. Although this feature is neither yet stable nor working precisely, it underlines the high potential online IDEs have. Furthermore, the IDE already supports the execution of code written in Python, Ruby and JavaScript. Regarding the steady development of the IDE in the past, it will presumably support even more languages for instant execution in the future. Finally, Cloud9 is an open source application [16] hosted on *GitHub* [17]. Programmers can develop plug-ins to extend the functionality, change existing code and contribute to the enhancement of the IDE.

### 3.4 Drawbacks

Apart from the benefits, there are also drawbacks to the online IDE. Defects in usage mainly correspond to the missing functionality mentioned in Section 3.2. To sum these up, there was no support for testing our files with the Jasmine test framework, CoffeeScript was not supported and had to be integrated based on a workaround [18]. Furthermore, developers have to switch frequently between the code editor and the console to identify the errors in the code. The missing CoffeeScript support also leads to a missing access to the compiled JavaScript. This would be helpful for identifying the line in the CoffeeScript file where the error occurred. Apart from these defects, it is not possible to use external plug-ins in the hosted online version of the Cloud9 IDE. Only the functionality provided can be used, external plug-ins are only available in the offline version. Thus, the user is limited to the provided functionality when using the online version.
Besides these shortcomings in the usage of the IDE itself and in the support of the testing work flow, there are drawbacks to the plug-in development for the Cloud9 IDE as well. These are presented in Section 4.4, after the implementation of new tool support was displayed in the following section.

## 4 Enhancing the Cloud9 IDE

We wanted to mitigate the problems of our work flow with the Cloud9 IDE as described in section 3.2. Mainly we focused on the following problems:

- insufficient support for the TDD cycle
- lack of support for CoffeeScript

One of the biggest inconveniences for us was the disconnection between the error displayed in the error log and the actual error in the CoffeeScript file. The error references the compiled JavaScript code and not the original CoffeeScript code. It is time-consuming to guess where an error might have originated in the CoffeeScript file, search it and fix it. Such an error log is displayed in figure 7.
Therefore we had to make the compiled JavaScript available. Furthermore we wanted to provide a mapping between the CoffeeScript and the JavaScript code. Our approach for this problem is described in section 4.1. Moreover we aimed at providing a better support for testing with the framework of our choice, Jasmine [15]. The implementation and functionality is described in section 4.2. The result of our work can be best seen in the new work flow resultant from our plug-ins, which is explained in section 4.3. Finally our experiences when working on a plug-in for the Cloud9 IDE are summarized in section 4.4.

### 4.1 The Livecoffee Plug-In

One of the widely mentioned concerns about CoffeeScript is that it compiles to JavaScript and runs as JavaScript. This problem is most apparent with development tools such as a debugger, as you write your code in one language and

debug it in another language. One of our gravest problems during the development of the *CoffeeRecommender* was that errors occurring during execution or testing only mentioned the JavaScript line number. Without direct access to the compiled JavaScript we could only guess where the error originated. So simple access to the compiled JavaScript would be a plus. The more complete solution however is an approximate line mapping between JavaScript and CoffeeScript. We solved these problems by extending an already existing plug-in.

**The old livecoffee plug-in** The "livecoffee" plug-in by Tane Piper[19] is recommended in the CoffeeScript book [20]. It is open source and shows the compiled JavaScript output and provides an approximate line matching as its core features. Unfortunately the plug-in is unmaintained, as it has not been updated in nearly a year. We had to make some adjustments in order for it to work with the new version of Cloud9.
The livecoffee plug-in works entirely on the client-side. It compiles the current CoffeeScript file using the CoffeeScript compiler. As this is part of our plug-in it is also a potential source of errors. This way it could happen that the code is compiled with a different version when executed than the one that we use to show the JavaScript. Furthermore it is possible that new constructs will be introduced in future CoffeeScript versions. If the CoffeeScript library in our plug-in is not kept up to date, then this would result in errors.

**Enhancing the functionality** We extended the existing functionality with our fork of the plug-in [25] in three different ways. We improved the approximate line matching, provided a visual aid for the line matching and enabled other plug-ins to use part of the functionality of this plug-in through loose coupling.
The approximate line matching was really approximate as it just took the current line and jumped to the same line in the JavaScript file. Unfortunately line matching is a broader but important problem. Due to the lack of viable alternatives at this point a new CoffeeScript compiler is being built, which will provide source mappings [21]. As this project is not yet ready to be used we decided to leverage the rich open source ecosystem a bit more.
We incorporated the *CoffeeScriptLineMatcher* library [22] written by Steve Howell. This library matches blocks of CoffeeScript code to the corresponding blocks of JavaScript code. It does so by going through the code looking for statements, which can be determined to definitely be the same, like if statements or assignments.
The visual aid works by coloring the matching block of lines of both the CoffeeScript and the JavaScript red as it can be seen in figure 3. Moreover the plug-in provides instant feedback, when the approximate line matching is enabled. Every time the position of the cursor in one of the windows is changed the current line in the other window is adjusted appropriately and the blocks are highlighted again. Refer to the screen cast [28] for more information.
Also we enabled other plug-ins to use the functionality of the livecoffee plug-in. The coupling is very loose since you can not expect the plug-in to be loaded

as it is not a default plug-in of Cloud9. We introduced the *livecoffee_show_file* event, which has to contain a line property. If the livecoffee plug-in is loaded, then it will open its window and go to the specified line in the JavaScript output window, highlight the block and also go to the same line in the matching CoffeeScript file. Other plug-in authors may use this functionality to enhance their support for CoffeeScript. We used it in our Jasmine test panel in order to go to the block in the CoffeeScript file which matches an error in the compiled JavaScript.


**Fig. 3.** The livecoffee plug-in in use with the *itemStorage* class of the CoffeeRecommender project

### 4.2 The Jasmine Plug-In

The goal for our Jasmine plug-in [26] is to make Test-Driven-Development with the Jasmine testing framework and CoffeeScript as comfortable as possible. Therefore we have integrated a panel into Cloud9, whose structure is similar to the one of the new test panel in Cloud9.

**The panel** The panel is one of the common panels on the left hand side in the IDE. Here all recognized testing files are shown. In order to do this we rely on a convention: All test files are assumed to be in the *spec* folder of the project or in a sub-folder of the *spec* folder. In addition they should end in *.spec.coffee*, which

is a convention introduced by Jasmine itself.
Whenever tests have been executed then this panel can be used to see every single test case, with its description, and whether it passed or failed. Figure 4 is a screen shot of the panel with some failing tests.

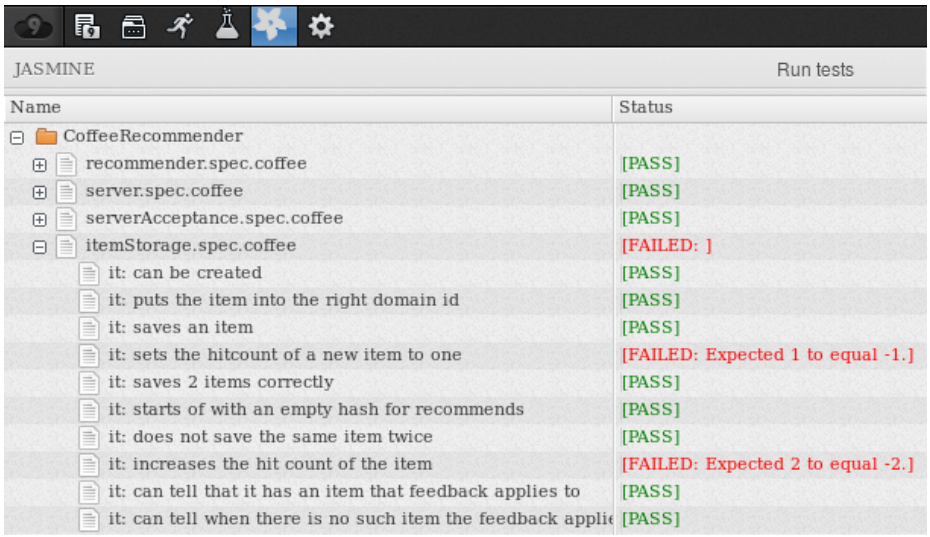


**Fig. 4.** The Jasmine panel after tests with 2 failures ran

**Executing Tests** We provide many different ways to execute tests. The distinct means are all present as they are more fitting for particular situations or developers. However the granularity with which you can execute tests is capped at the file they are defined in. So you can only execute test files separately not individual test cases. Following is a description of all the different ways you can execute tests with our plug-in at the moment:

**Automatically after file save** Tests are executed automatically when a file is saved according to naming conventions. So whenever a file called *implementation.coffee* is modified the plug-in will look for a file called *implementation.spec.coffee* and execute it if present. Similarly if you modify a test file itself, then after you save it the very same file will be executed. This behavior heavily favors Test-Driven-Development which relies on frequent feedback.

**Manually execute all tests via short-cut** You can press the *Ctrl + J* short-cut in order to execute all tests.

**Manually execute a test by double clicking** As it can be seen in figure 4 the different test files are shown in the Jasmine panel. Double clicking on any of those files will execute it.

**Manually execute a selection of tests with the Run-Button** You can select a sub-set of test files to run by clicking them while holding the *Ctrl* key. Afterwards you can click on the *Run Tests* button in order to execute this sub selection of tests.

**Locating the error in the CoffeeScript file** We provide an aid for the programmer to find the line of code in the CoffeeScript files that caused the error. Often when trying to locate an error programmers tend to look at the stack trace and search for the first line that is from their project. Then they go there in order to investigate the problem further. The Jasmine plug-in basically does the same. It parses the stack trace and looks for the first occurrence of a file belonging to the current project, e.g. not a library. Then the loose coupling with the livecoffee plug-in is used to map the so found line of the compiled JavaScript to the matching block of the CoffeeScript. The file is opened and the cursor already placed in the right line, so the programmer can get right back to work. Of course this functionality is only accessible when the livecoffee plug-in is loaded.
Developers may access this feature by right clicking on the specific test case that failed and select *Show Error in Coffee*. The error message is displayed in our panel as well (see figure 4). If the programmer should need additional information, then he can still check out the complete error log including all stack traces on the Cloud9 console.

**How the plug-in works** The plug-in itself is client-side, which means that it is executed in the environment of the browser. This is how the user interface and the communication with the livecoffee plug-in works.
However in order to execute tests we need to execute code on the server-side. This process is more complicated and a simplified version is described in figure 5. On the server side the tests are executed with the version of *jasmine-node* [23] that the given project has installed. It is assumed that *npm* [24] was used to install jasmine-node. This was a conscious choice as we do not want to introduce additional dependencies to the Cloud9 IDE. Moreover we do not want to force users to use a specific version of jasmine-node.
With the execution of tests being as it is right now, the only way to get a hold of the test results is parsing the console output generated by jasmine-node on the server-side. This results in some conventions and liabilities of the plug-in. It is a convention that the first *describe* block of a spec-file should begin with the name of the spec file. Otherwise there is no way for us to determine which tests belong to which file. Moreover the parsing can be considered rather brittle, if the *jasmine-node* plug-in ever significantly changes the format of their verbose output, then our plug-in will likely break.
It would be a better solution to accompany the client-side jasmine plug-in with a server-side plug-in. That way the server-side plug-in could run the tests and

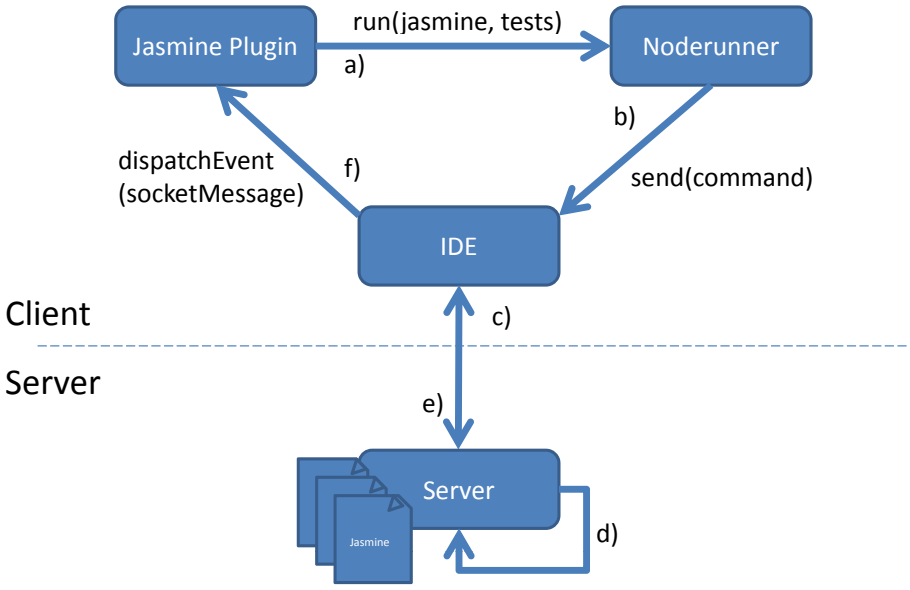


**Fig. 5.** A simplified version of the work flow needed to execute tests. At a.) the *noderunner* is instructed to execute the tests with Jasmine, he forwards this call to the IDE at b.). The IDE issues the command to the server at c.). Then the server executes jasmine with the specified test files at d.). The console output of this is sent back to the IDE via socket messages at e.). The IDE emits events with these messages which our plug-in registers a listener on in order to reassemble the console output at f.)

gather the results using a custom reporter[5]. The results gathered by the reporter could then be sent to the client-side plug-in as JSON objects. Due to the given limitations concerning time, man power and lack of documentation for Cloud9 server-side plug-ins and Jasmine reporters we chose to work with what we knew worked, although it might be a bit frail.

### 4.3 New Work Flow

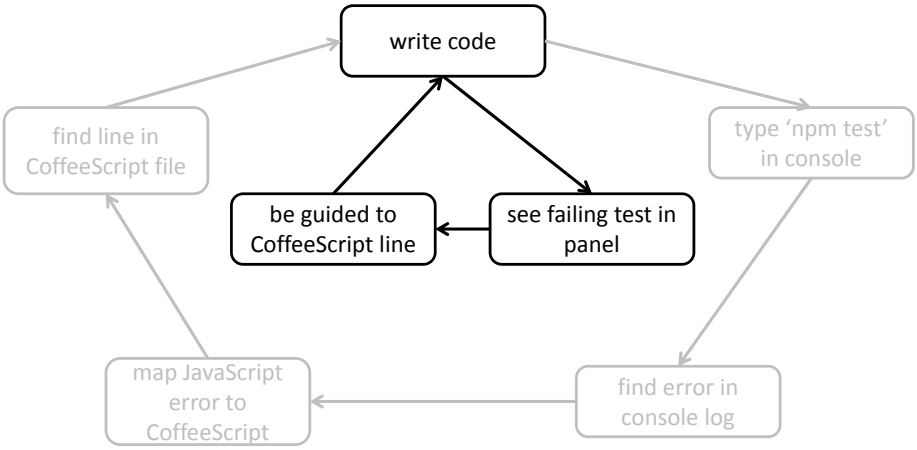


**Fig. 6.** The new work flow for testing with a failing test with the livecoffee plug-in and the jasmine plug-in in place - the old work flow can still be seen in gray

The usual Test-Driven-Development cycle with failing tests in Cloud9, as described in section 3.2, has been significantly shortened and thereby improved by our two plug-ins. The new work flow is depicted in figure 6. There are basically just the following three steps in the development cycle:

**write code** Write your code and then save it, this step remains unchanged.

**see failing test in panel** As soon as you save a file ideally the corresponding test-file is executed automatically, alternatively it is possible to run tests as described in section 4.2. Now the left hand panel shows whether all tests pass or if some failed. For failing tests the error message is displayed.

**be guided to CoffeeScript line** The user can right click on any failing test and select *Show Error in Coffee*. This results in the file of the project, where the error originated, being opened and the cursor to be placed in the line were the error occurred. Because of that the programmer is directly able to continue to write code to fix the bug.

[5] Jasmine offers the possibility to run with different reporters, to which the results are reported in detail.

When no error occurs during the testing cycle the circle is also shortened. The programmer just sees that all the tests pass and can continue to write code right away. This is opposed to switching to the console and typing *npm test* to run tests.

### 4.4 Extending the Cloud9 IDE with plug-ins

Our experience when extending the Cloud9 IDE has been mixed. We were able to provide our functionality solely through plug-ins, without modifying any code that we have not written ourselves. This is a sign of a very good plug-in system. However we found the documentation of the plug-in system and the IDE itself lacking. There is documentation for the *Ajax.org Platform* [29] and the *Ace Editor* [30], but documentation for plug-ins and other functionality is missing. Documentation for writing plug-ins is seemingly being worked on right now [31]. However, plug-in authors can only look at the already existing plug-ins and try to figure out what they do and how they do it so far. There are many undocumented assumptions about properties of plug-ins that have to be fulfilled in order for the plug-in to work correctly. Moreover, the IDE is constantly being developed and evolving, which is good. Unfortunately even core APIs, like opening a file, seem to be changed. This results in big compatibility issues with newer and older Cloud9 versions. It is also discouraging, that third-party plug-ins can not be used in the hosted Cloud9 edition. However the developers seem to be eager to include plug-ins in the official Cloud9 repository, which is very pleasant. Especially the missing documentation hinders the development of plug-ins for the Cloud9 IDE, so we hope that the maintainers will provide appropriate documentation soon. We hope that this will result in more plug-ins being written for Cloud9 extending its functionality even further.

## 5 Conclusion

Testing your application, preferably with TDD, can increase the quality of the code basis tremendously. Thus, development environments that encourage TDD and support the programmer in the steps through the testing cycle can be of great benefit. In this paper we showed how insufficiently a programmer is supported in the testing cycle when using CoffeeScript and the Jasmine testing framework in the Cloud9 IDE. However, when providing proper tool support, the developer can intuitively work on his application and is steadily supported in the testing process. Since there are already good responses to the tools [28], [32], [33], [34], we believe the tools will soon be part of the default Cloud9 experience in the near future. With the tools presented in this paper and the steady development the Cloud9 IDE is going through, we suggest this IDE has a great chance to serve as a replacement for some of today's desktop IDEs in the near future. Developers should begin to think about a shift from classic desktop development environments to web-based IDEs and exploit the benefits of real-time collaboration and global team based development.

## References


1. Ajax.org. Cloud9 IDE. http://c9.io.
2. Beck, Kent. Test-Driven Development By Example. Addison-Wesley, 2003.
3. Langr, Jeff. Test-Driven Development: A Guide for Non-Programmers. Pragmatic Bookshelf 2011. http://pragprog.com/magazines/2011-11/testdriven-development.
4. Ashkenas, Jeremy. CoffeeScript: a little language that compiles into JavaScript. http://coffeescript.org/.
5. HPI Software Architecture Group. Lively Kernel: An Explorative Authoring Environment. http://www.lively-kernel.org/.
6. ShiftCreate. ShiftEdit: The Online IDE. http://shiftedit.net/.
7. WIODE. wiode: Web Installed Open Development Environment. http://www.wiode.org/.
8. CoffeeScript Wiki. Text editor plugins. https://github.com/jashkenas/coffee-script/wiki/Text-editor-plugins
9. JetBrains. RubyMine: The Most intelligent Ruby and Rails IDE. http://www.jetbrains.com/ruby/
10. Eclipse Foundation. Eclipse IDE. http://www.eclipse.org/
11. JetBrains. Webstorm & PhpStorm Blog: JavaScript unit testing support. 2011. http://blog.jetbrains.com/webide/2011/10/javascript-unit-testing-support/
12. Joyent, Inc. Node.js: a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. http://nodejs.org/.
13. Heroku, Inc. Heroku: cloud application platform. http://www.heroku.com/.
14. Plista. Join the plista Contest: search for the best performing live recommender algorithm. http://contest.plista.com/.
15. Pivotal Labs. Jasmine: A behavior-driven development framework for testing JavaScript code. http://pivotal.github.com/jasmine/.
16. Ajax.org. Cloud9 IDE: Open Source Javascript IDE Code Repository. https://github.com/ajaxorg/cloud9.
17. Github Inc. Github: Social Coding. https://github.com/.
18. Cloud9 IDE Support. Create a CoffeeScript / Node.js Project. http://cloud9ide.zendesk.com/entries/20559696-create-a-coffeescript-node-js-project.
19. Piper, Tane. Live CoffeeScript: An extension for Cloud9 IDE for CoffeeScript functionality. https://github.com/tanepiper/cloud9-livecoffee-ext
20. Burnham, Trevor. CoffeeScript : Accelerated JavaScript Development. Pragmatic Bookshelf, 2011.
21. Ficarra, Michael. CoffeeScript II. https://github.com/michaelficarra/CoffeeScriptRedux.
22. Howell, Steve. CS/JS Code Browser. https://github.com/showell/CoffeeScriptLineMatcher.
23. Hevery, Misko. Jasmine testing framework for node.js. https://github.com/mhevery/jasmine-node.
24. Schlueter, Isaac. Node Package Manager: A dependency management system for node.js. http://npmjs.org/.
25. Metzke, Tobias, Pfeiffer, Tobias and Piper, Tane. Live CoffeeScript: An extension for Cloud9 IDE for CoffeeScript functionality. https://github.com/PragTob/cloud9-livecoffee-ext

26. Metzke, Tobias and Pfeiffer, Tobias. Jasmine Test Panel for Cloud9: Making testing with Jasmine and CoffeeScript easier. https://github.com/PragTob/cloud9-jasmine-ext
27. Metzke, Tobias and Pfeiffer, Tobias. CoffeeRecommender: A recommendation engine written in CoffeeScript and NodeJS. https://github.com/PragTob/CoffeeRecommender
28. Metzke, Tobias and Pfeiffer, Tobias. CoffeeScript & Jasmine Support for Cloud9. http://www.youtube.com/watch?v=Iy49Ho1z-PQ
29. Ajax.org. Ajax.org Documentation. http://ui.ajax.org/#docs
30. Ajax.org. Ace API Reference. http://ace.ajax.org/api/
31. Cloud9 IDE Googlegroup. Cloud9 IDE Plugin Documentation. https://groups.google.com/forum/?fromgroups#!topic/cloud9-ide/-GKEtn2lM_8
32. Metzke, Tobias. Commit in Jasmine plug-in for the Cloud9 IDE. Commit comment by Ruben Daniels. https://github.com/PragTob/cloud9-jasmine-ext/commit/d74610498f2a89fb7930c93328fe2db496e525dd.
33. Jongboom, Jan. Tweet about Jasmine plug-in for the Cloud9 IDE. https://twitter.com/drbernhard/status/228856743835361282.
34. Pardee, Matt. Tweet about Jasmine plug-in for the Cloud9 IDE. https://twitter.com/matt_pardee/status/228869510491414529.

# Appendix

### A. Console Output - Error Log

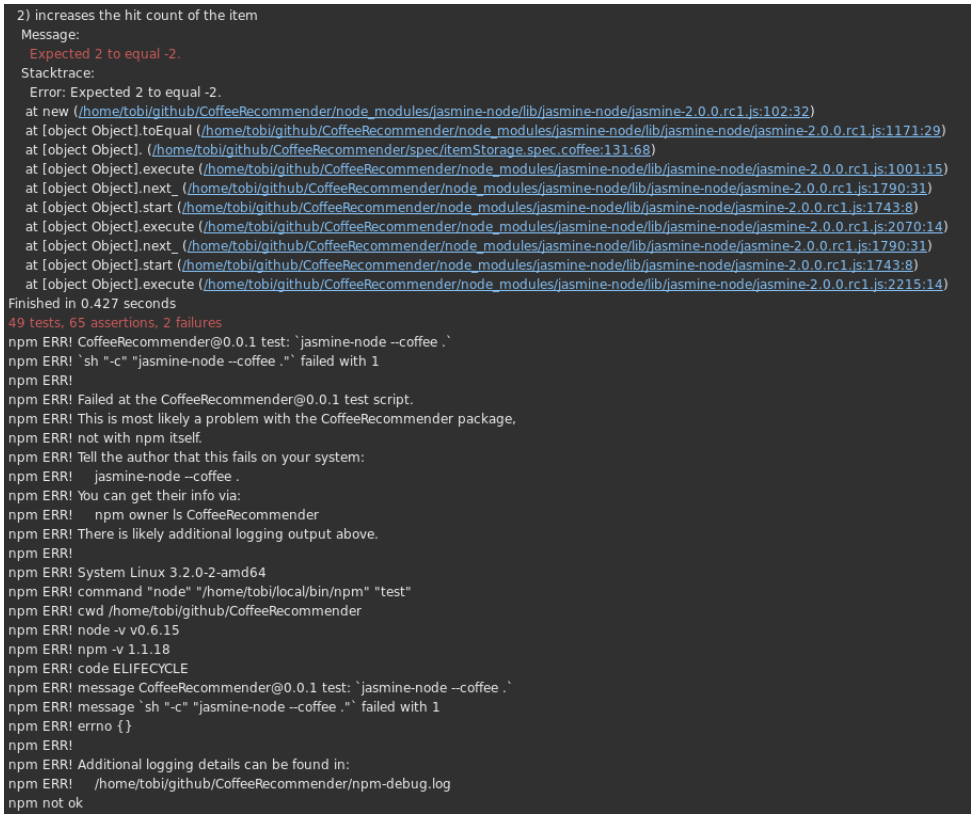


**Fig. 7.** Error Log of the Cloud9 IDE when tests fail