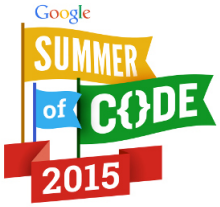
ÃL'cole Nationale SupÃL'rieure de Techniques AvancÃL'es Bretagne

Summer Internship Report

Web development in Lua programming language

Improvements to Sailor framework during Google Summer of Code



Author:
Etiene da Cruz Dalcol

Supervisor:
Dr. Olivier Reynet

September 27, 2015

Contents

1 Abstract

Lua is a very fast and powerful scripting language that can be easily embeddable. For that reason, it has gained a niche in the game development industry. Lua has great potential and incredible benchmarks. Despite being also an excellent tool as a general purpose language to develop robust applications, its use in web developments needs to be more widespread.

Sailor was invented to increase the ecosystem of web development in Lua but it is still in early development. The purpose of this work is to turn Sailor into a more mature software by adding new features and improving existing ones such as adding automated tests and improving the use of Lua instead of Javascript when programming for the browser.

ResumÃľ

Lua est un langage trÃĺs rapide et puissant qui peut Ãłtre embarquÃľ facilement. Pour cette raison, il a gagnÃľ une place importante dans l'industrie de dÃľveloppement de jeux. Lua a des potentiels et benchmarks incroyables. En dÃľpit d'Ãłtre aussi un excellent outil en tant que langue d'usage gÃľnÃľral pour dÃľvelopper des applications robustes, son utilisation dans le dÃľveloppement web devrait Ãłtre plus gÃľnÃľralisÃľe.

Sailor a ÃľtÃľ inventÃľ pour augmenter l'ÃľcosystÃĺme du dÃľveloppement web en Lua, mais il est encore dans un stage de dÃľveloppement prÃľcoce. Le but de ce travail est de transformer Sailor dans un logiciel plus mature en ajoutant de nouvelles fonctionnalitÃľs et l'amÃľlioration de ceux existants, tels que l'ajout de tests automatisÃľs et d'amÃľlioration de l'utilisation de Lua lieu de Javascript lors de la programmation pour le navigateur.

2 Introduction

2.1 Google Summer of Code

Google Summer of Code (GSoC) is a global program that connects students with open source, free software and technology-related organizations. During the a 3 month period on the summer, the students get familiarised with open source projects, work with the community and write code.

Google identifies open source projects and organizations that will receive funding and participate on the program. The organizations will provide mentors to guide students during the program. Students submit projects to the organizations, who rank them. Organizations may suggest a list of ideas for projects. Once Google defines how many student slots for projects are allocated to an organization, the organization decides which students and projects are accepted and pair them with a mentor.

While most students come from a Computer Science and Software Engineering background, this is not mandatory and the educational area of participants can be very wide.

The program is centered on some goals:[1]

1. Get more open source code written and released for the benefit of all.
2. Inspire young developers to begin participating in open source development.
3. Help open source projects identify and bring in new developers.
4. Provide students the opportunity to do work related to their academic pursuits during the summer: "flip bits, not burgers."
5. Give students more exposure to real-world software development (for example, distributed development and version control, software licensing issues, and mailing list etiquette)."

There are midterm and final evaluations, and the code completed must be submitted to GSoC's website by the end of the program. All development happens online, Google does not provide an office space and there's no requirement to travel.

Since the start of the program, in 2005, over 8500 successful students have participated, from over 109 countries, with 8000 mentors making 55 million lines of code.

2.2 LabLua

LabLua was one of the organizations selected to participate in the 2015 version of Google Summer of Code. It is a research lab at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) affiliated with its Computer Science Department. Its researches are on the field of programming languages, with an emphasis on the Lua programming language. The founder of LabLua, Prof. Roberto Ierusalimschy, is one of the creators of the Lua language.

LabLua proposed the following list of ideas to GSoC:

1. LuaRocks add-ons system

[1] *Google Summer of Code Student Guide.* `http://en.flossmanuals.net/GSoCStudentGuide/index/`. Accessed: 2015-09-25.

2. Port Lua Test Suite to NetBSD Kernel
3. Elasticsearch Lua client (elasticsearch-lua)
4. Add support for left recursion to LPeg
5. Switch Typed Lua from optional typing to gradual typing
6. Adapt CGILua SAPI launcher to explore all WSAPI features
7. Add support for WSDL generation to LuaSOAP
8. Add support for prepared statements in LuaSQL
9. Multi-CPU usage in VLC
10. Multi-CPU usage in wireshark
11. Develop a binary serialization format with support for dynamically types values and an RPC protocol for dynamically typed invocations based on this format.
12. Develop a library for Lua that allows Lua programs to access features provided by the platform's underlying operating system (OS) kernel, such as process control, network access, file system, event notification, etc.
13. Port an SDL-based C++ open source game to CÃľu

They received in total 6 slots, from which, four of them were filled by students working on ideas 2, 3, 7 and 13 and two of them were ideas proposed by students themselves. One of the student-proposed ideas was my project to improve Sailor, a web development framework I had been developing on my free time.

The project was mentored by Dr. Fabio Mascarenhas, professor at Universidade Federal do Rio de Janeiro. Since Google did not provide for office space or work placement, ENSTA Bretagne offered a placement during the summer to work on this project under the supervision of Professor Dr. Olivier Reynet.

2.3 The Lua programming language

> Lua is a powerful, fast, lightweight, embeddable scripting language.[2]

This description provided by lua.org does not fully grasp what's interesting about this language. Lua was created in a very specific context: Petrobras, a multinational energy and oil corporation headquartered in Rio de Janeiro, Brazil had interactive graphical programs for engineering applications. These programs needed some flexibility and they were used by non professional programmers. Lua was created to be very simple yet powerful, allowing the customisation of these softwares through scripting. Avoiding cryptic syntax and semantics, Lua is a very readable language with a short learning curve.

[2] *The Programming Language Lua.* `http://lua.org/`. Accessed: 2015-09-25.

It's possible to have an idea about the simplicity of this language just by comparing the brevity of the Programming in Lua book versus books on pretty much any other language. This means you can have all of the language in your head. Its source code is also very succinct. As of version 5.3.1, described in about 23000 lines of Standard C, the whole distribution which includes documentation has only 276Kb. This allows easily porting Lua code to things that run standard ANSI C, even if they have typically low resources, and embedding Lua to applications without bloating them. The result is that Lua is a very widely used language, from micro controllers and washing machines to operating systems, power computers and video games.

Its power can be observed on its multi-paradigm extensible semantics and speed. For example, while Lua does not come with Object Oriented Programming out of the box, it is possible to prototype it using first class functions [3] and metatables [4]. Lua also performs way better than other popular programming languages on speed tests. A special dialect of Lua called LuaJIT is the fastest it can get while still being a dynamically typed scripting language:[5]

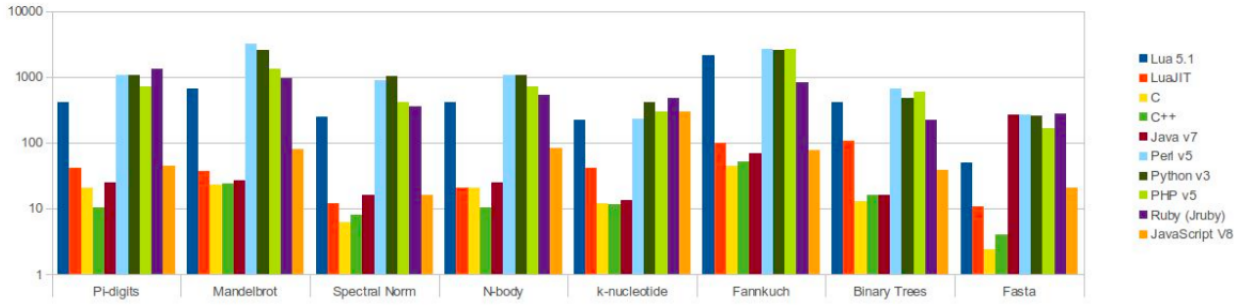


Figure 1: Speed comparison of popular script languages (less is better)

[3] Support of passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures. (*Wikipedia: First Class Function*. `https://en.wikipedia.org/wiki/First-class_function`. Accessed: 2015-09-25)

[4] Allows to change the behavior of a table. It is part of an extension mechanism which allows you to overload certain operations on Lua objects. (*lua-users wiki: Metamethods Tutorial*. `http://lua-users.org/wiki/MetamethodsTutorial`. Accessed: 2015-09-25) (Roberto Ierusalimschy. *Programming in Lua*. Departamento de Informatica, PUC-Rio, 2006, p. 117)

[5] Daniel Gruno. *Introducing mod lua*. `http://humbedooh.com/presentations/ACNA-mod_lua.odp`. Accessed: 2015-09-25, p. 8.

3 Analysis of the problem

Lua is greatly used as an embedded language and has found a niche in game development being the top language of choice for scripting in this area.[6] In the same time, it is still a great tool as a general purpose language but not much widespread in other domains such as web development. As of September 26th of 2015 Lua is used in less than 0.1% of websites whose server-side programming language we know.[7]

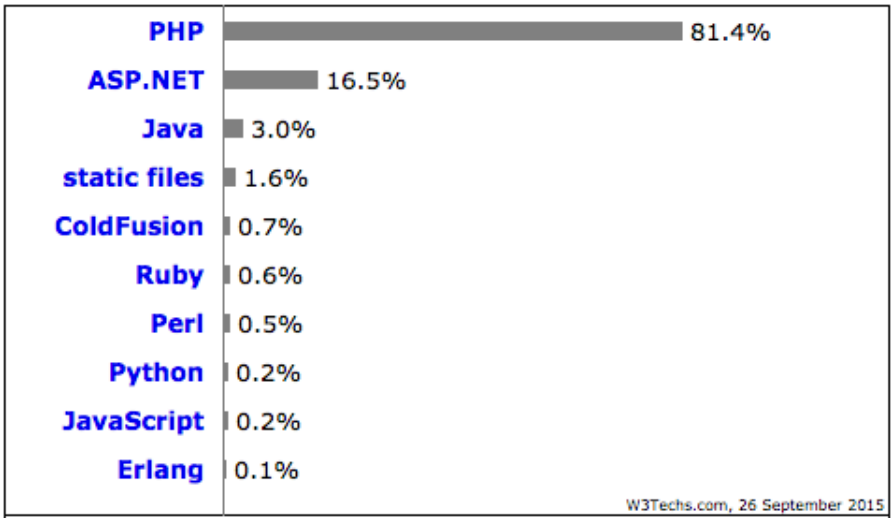


Figure 2: Usage of server-side programming languages for websites. Note: a website may use more than one server-side programming language

In an attempt to understand why this happens, I made an analysis of the context of the top used language: PHP. One may argue that PHP is not such a well-designed language. Then why is it the top language of choice for web development? Why is 82% of the web made of PHP websites? We could speculate that PHP's success is due to a good timing but, 20 years later, allegedly better programming languages have appeared and PHP's popularity continues to grow.[8]

[6] Mark DeLoura. *The engine survey: general results.* `http://www.satori.org/2009/03/the-engine-survey-general-results/`. Accessed: 2015-09-26. 2009.

[7] Q-Success. *Usage of server-side programming languages for websites.* `http://w3techs.com/technologies/overview/programming_language/all`. Accessed: 2015-09-26.

[8] Q-Success. *Historical yearly trends in the usage of server-side programming languages for websites.* `http://w3techs.com/technologies/history_overview/programming_language/ms/y`. Accessed: 2015-09-26.

	2010 1 Jan	2011 1 Jan	2012 1 Jan	2013 1 Jan	2014 1 Jan	2015 1 Jan	2015 26 Sep
PHP	72.5%	75.3%	77.3%	78.7%	81.6%	82.0%	81.4%
ASP.NET	24.4%	23.4%	21.7%	20.2%	18.2%	17.1%	16.5%
Java	4.0%	3.8%	4.0%	4.1%	2.7%	2.8%	3.0%
static files							1.6%
ColdFusion		1.3%	1.2%	1.1%	0.8%	0.7%	0.7%
Ruby	0.5%	0.5%	0.6%	0.5%	0.4%	0.6%	0.6%
Perl		1.1%	1.0%	0.8%	0.6%	0.5%	0.5%
Python	0.3%	0.3%	0.2%	0.2%	0.2%	0.2%	0.2%
JavaScript			<0.1%	<0.1%	0.1%	0.1%	0.2%
Erlang						0.1%	0.1%

Figure 3: Historical yearly trends in the usage of server-side programming languages for websites

This is what Jeff Atwood, stackoverflow co-founder, calls "The PHP singularity".[9] According to him, the faults of PHP don't matter because it has a huge ecosystem and easy deployment and the only solution to this is to build compelling alternatives who are as pervasive and as easy to setup and use. In his blog, he calls out the community of programmers to push other options forward.

Having some knowledge of the PHP ecosystem as a beginner and intermediate user, I considered this an interesting challenge. As Lua is a well-designed, fast and easy-to-learn language, I believe it is a perfect candidate for the job in terms of quality. But how about it's ecosystem?

3.1 The Lua ecosystem

3.1.1 General ecosystem

Despite being a successful 20-year old language, Lua's community and amount of libraries is small. This is an issue that is being addressed, specially in the recent years. The Kepler Project[10] (2006-2009) was a project developed with the goal to make a platform for web development in Lua.[11] While it did not accomplish its ultimate goal, it produced some of the modules that were essential for the later development of the ecosystem. One of these modules that deserves a highlight is LuaRocks,[12] developed by Hisham Muhammad to deploy Kepler modules. LuaRocks is now the package manager for Lua modules, acting as a one-stop repository that significantly reduced the entry barrier for Lua as an application development language. Pierre Chapuis, in his 2013 talk "State of the Lua Ecosys-

[9] Jeff Atwood. *The PHP Singularity.* `http://blog.codinghorror.com/the-php-singularity/`. Accessed: 2015-09-26. 2012.

[10] *Kepler 1.1.1.* `https://github.com/keplerproject/kepler`. Accessed: 2015-09-27.

[11] Andre Carregal and Tomas Guisasola. *CGILua: Building Web Scripts with Lua.* `http://keplerproject.github.io/cgilua/`. Accessed: 2015-09-27.

[12] Hisham Muhammad. *LuaRocks: the package manager for Lua modules.* `https://luarocks.org`. Accessed: 2015-09-27.

tem"[13] suggested modules should be even easier to find. He created a website called Lua toolbox,[14] aimed at ranking, endorsing and classification of Lua modules. In the meantime, MoonRocks, a tool aimed at easy uploading and hosting of modules[15] was created by Leaf Corcoran, also in 2013. This project has recently merged with LuaRocks.

All these efforts have been producing results and the ecosystem is growing steadily:[16]

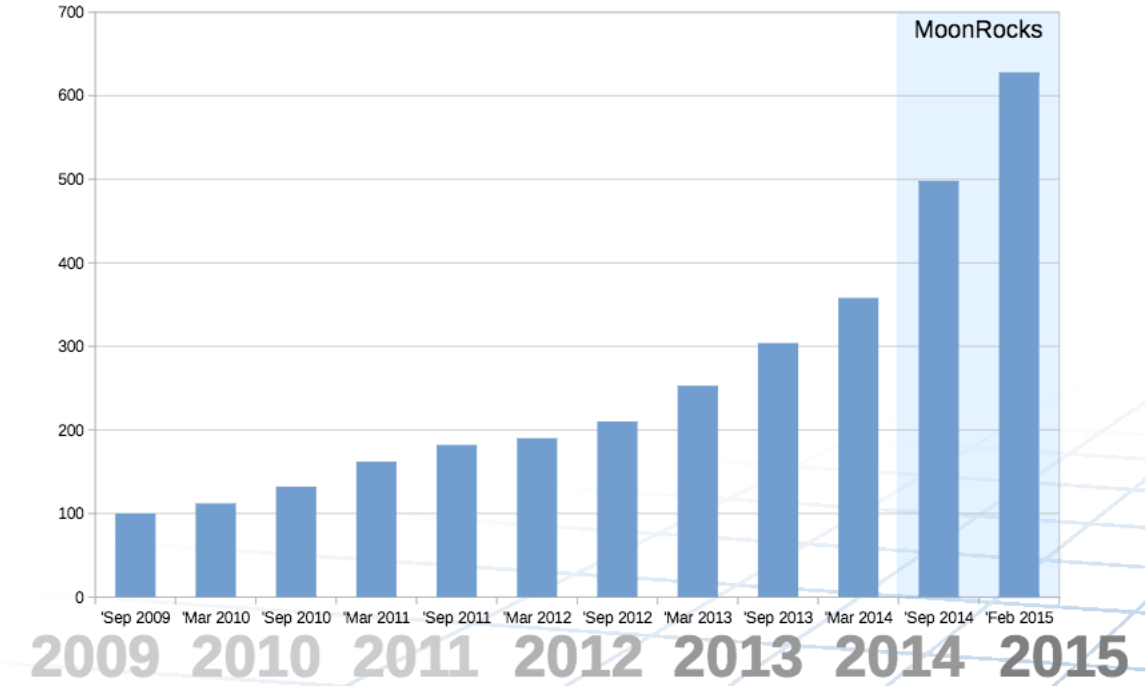


Figure 4: Recent growth of the repository

3.1.2 Web servers supported

Lua can run on a variety of web servers. Apache has a module to run lua called mod_lua, Nginx has a distribution called open resty that allows to run Lua too. This is great news, because they are the top two web servers used.[17] ThereâĂŹs Xavante, which is a web server written in Lua and allows a really quick deploy, as well as plenty of others options like Mongoose, Lwan or Lighttpd.

[13] Pierre Chapuis. *State of the Lua Ecosystem.* `http://files.catwell.info/presentations/2013-11-lua-workshop-lua-ecosystem/`. Accessed: 2015-09-27. 2013.

[14] Pierre Chapuis. *Lua Toolbox.* `https://lua-toolbox.com/`. Accessed: 2015-09-27.

[15] Leaf Corcoran. *Moonrocks: A command line tool for uploading and installing from the public Lua module hosting site, MoonRocks.* `https://github.com/leafo/moonrocks`. Accessed: 2015-09-27.

[16] Hisham Muhammad. *LuaRocks, fostering an ecosystem of Lua modules.* `http://hisham.hm/papers/talks/hisham_luarocks_fosdem2015.pdf`. Accessed: 2015-09-27. 2015.

[17] Q-Success. *Usage of web servers for websites.* `http://w3techs.com/technologies/overview/web_server/all`. Accessed: 2015-09-26.

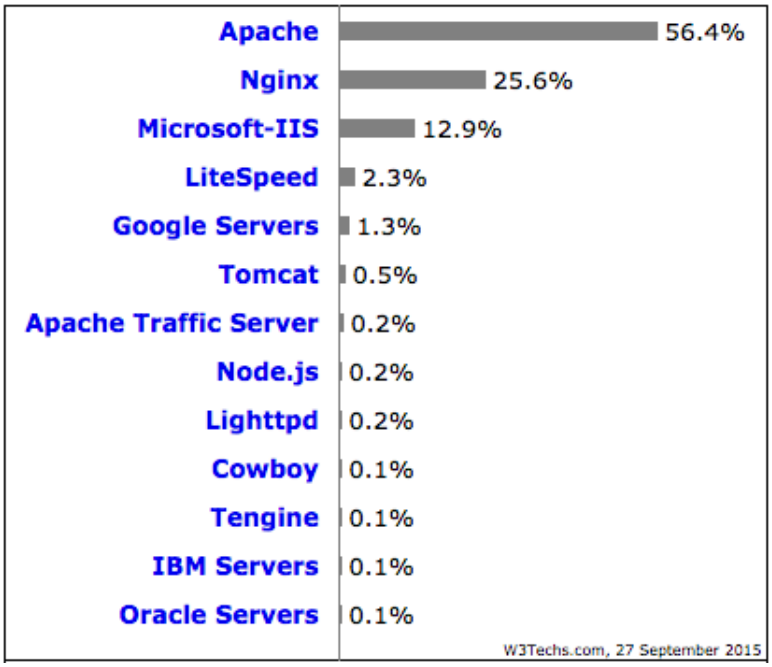


Figure 5: Usage of web servers for websites

3.1.3 Existing tools and frameworks

Orbit:

Orbit is the longest established framework the Lua community has, but it's not in active development anymore. It's very lightweight and follows an MVC architecture.

Can be installed through LuaRocks: Yes

Documentation: Very succinct but sufficient

License: GPL

Website: `http://keplerproject.github.io/orbit/`

Luvit:

Luvit adopts a different kind of approach from Orbit. It's designed to be a port of Node.js to Lua. It accomplishes this task very well, claiming to be 2 to 4 times more performant and making great memory savings. However, for those that are not already proficient in writing applications with Node.js, it can be very complex to use as its documentation is not yet prepared to receive full beginners in the web development world. It's very powerful, it's in intense development and it is maybe the most popular Lua web development framework, but still very little known outside the community.

Can be installed through LuaRocks: No

Documentation: Insufficient

License: Apache 2.0

Website: `https://luvit.io/`

Lapis:

Lapis is a very interesting framework. It's not exactly in Lua, but in MoonScript, a language that compiles to Lua like CoffeeScript compiles to JavaScript. You can still, however, use Lua to write your application. It's in very active development by a dedicated maintainer. It does not provide an MVC architecture out of the box but it's possible to prototype one.

Can be installed through LuaRocks: Yes

Documentation: Extensive and informative

License: MIT

Website: `http://leafo.net/lapis/`

Other projects worth mentioning:

TurboLua | Apache 2.0: `http://www.turbolua.org/`

Tir | BSD: `http://tir.mongrel2.org/`

Ophal | AGPL 3.0: `http://ophal.org/`

4 Proposal

As seen at the ecosystem analysis, while Lua already provides a set of libraries and tools for web development, none of them has reached a somewhat wide-spread use in the web development community. And while the Lua ecosystem seems to be flourishing nicely, this set is still considerably small comparing to the amount of tools offered in other programming languages. Having this in mind, I considered that building a new tool for web development in Lua was not a redundant task. As a follow-up to Atwood's call for action, I decided to push and advocate Lua as a promising platform and simply create more options. In the worst-case scenario I would learn a lot of things about the Lua programming language and about web development.

This idea was the beginning of Sailor, a framework for fast development of web applications in Lua I started developing in 2014. In the same way as Lua, Sailor is free software and written under the MIT License. Its applications are structured in a model-view-controller (MVC) architecture. Like other web development frameworks, such as Ruby on Rails, it is designed to make the development process faster by making some assumptions and conventions (not requiring configuration for everything) and encouraging principles like DRY (Don't Repeat Yourself).

Its ultimate goal is to be very easy to use while still being a high quality tool. Sailor offers many possibilities of setup to its users, can be used for building projects of any size in such a friendly language that is Lua and allows the use of Lua in the client-side (browser) as well, eliminating the use of JavaScript if desired.

This last point is a really interesting aspect of this project. Writing code in Lua to run in the browser is allowed through an integration of Sailor with a Lua-to-Javascript virtual machine. While it will be less performant, there are other advantages. Usually, a beginner web developer needs to learn at least 4 types of languages for building a complete dynamic website, including those that are not considered programming languages: a server side language of choice, JavaScript for the client side, HTML / CSS and SQL. Being able to use the same language for both the server and the client side reduces the entry barrier and improves maintainability and reusability of the code. In addition to this, with the beginning of the development of Web Assembly[18] in April 2015, the possibility to run Lua natively on browsers may be attained in the foreseeable future, which is very promising.

Sailor Characteristics as of version 0.3 - Jupiter

Features: Routing, ORM, Validation, Themes and Layouts, Bootstrap integration, Sessions, Cookies, friendly URLs, auto generator of models and CRUDs, easy form module, Lua in the browser, LuaRocks setup

Lua version compatibility: 5.1 and 5.2

Operating Systems: Windows, Mac or Linux

Web servers: Apache with mod_lua, OpenResty distribution of Nginx, Xavante, Lwan ou Lighttpd.

[18] Luke Wagner and Seth Thompson. *WebAssembly: Development of WebAssembly and associated infrastructure.* `https://github.com/WebAssembly`. Accessed: 2015-09-27.

Databases: MySQL, PostgreSQL, SQLite and other databases supported by the LuaSQL library.

Lua-to-JS virtual machine: Lua5.1.js (a 5.1 coding style is required for the client side)

Dependencies:

1. lua >= 5.1, < 5.3
2. datafile >= 0.1
3. luafilesystem >= 1.6.2
4. valua >= 0.2.2
5. lbase64 >= 20120807
6. cgilua >= 5.1.4, < 5.2
7. xavante >= 2.3
8. wsapi-xavante >= 1.6.1

Licence: MIT

Website: `http://sailorproject.org`

4.1 The Summer Project Proposal

Sailor is a work in progress with a small codebase that is growing gradually. The project submitted to Google Summer of Code was focused on developing a new feature and improving an existing one:

1. Implementing a test suite
2. Improving the usability of Lua at the client side

The results would significantly increase the overall quality and usability of Sailor. In addition, Sailor's participation in GSoC could allow it to get more traction, which would be beneficial for both Sailor and the Lua community as it introduces some fresh blood into the current Lua web development scene.

4.2 Proposed development schedule

May 25th - June 5th
: Researching how other frameworks use their test suites

June 6th - June 15th
: Researching and testing existent test Lua modules

June 16th - July 1st
: Either integrating an existing test module with Sailor or developing a new one

July 2nd - July 16th
Testing, bug fixing and documenting

July 17th - July 23rd
Researching and testing Lua to JavaScript VMs. E.g. MoonshineJS

July 24th - August 6th
Improving current way to manipulate DOM from Lua and load Lua modules to be used on client side.

August 7th - August 16th
Testing, bug fixing and documenting

August 17th - August 21st
Polishing and making sure nothing was missed

5 The development

5.1 Implementing a test suite

Testing is a crucial part of software development. It is so important that there are development techniques that focus on that, such as Test-driven development (TDD). In TDD, first, an initially failing test for a function is written, and then a function that passes the test is developed.

Writing tests is useful not only for assuring that the software does what it is meant to do, but also that it will still do what its meant in different environments, with different inputs, within a reasonable time. By writing automated tests, there's a documentation of which cases the software is being tested against and the tests can be rerun when changes are made to the software or the environment to ensure that the expected results are still obtained.

As of version 0.3 Jupiter, Sailor did not provide an easy functionality to write automated tests to its applications.

The first step about implementing a test suite into Sailor was to investigate how this is done on different frameworks. A number of frameworks presented a functionality to test the applications made with them and insightful documentation such as Yii (PHP)[19] , Laravel (PHP)[20] , Lapis (Lua)[21] , Ruby on Rails (Ruby)[22] and Flask (Python)[23] .

It was observed that the most common forms of application testing in web development had:

1. Fixtures: the ability to load sample data into a test database
2. Unit tests: the ability to test very specific pieces of code such as model methods
3. Functional tests, also called integration tests: the ability to test broader pieces of code and how they work together in the flux of the application use, such as the execution of a controller action

There was little to no variation in how unit tests were done but what was defined as their functional tests varied a lot. Some frameworks mocked up a request, some frameworks launched a test server and made actual HTTP requests. Some frameworks provided even more complex testing, in a third category, for deeply testing the interface and included, for example, the simulation of mouse clicks in a web page.

For the purpose of this project, it was decided that mocking a request and executing Sailor's internal functions for running controller actions would be enough.

Investigating if there were already libraries that provided tests in Lua was not difficult. There aren't many around. Busted[24] is one of the top Lua libraries used with this purpose, it is in active

[19] *The definitive guide to Yii: Testing.* `www.yiiframework.com/doc/guide/1.1/en/test.overview`. Accessed: 2015-09-27.
[20] *Testing Laravel - The PHP framework for artisans.* `http://laravel.com/docs/5.1/testing`. Accessed: 2015-09-27.
[21] Leaf Corcoran. *Lapis: testing.* `leafo.net/lapis/reference/testing.html`. Accessed: 2015-09-27.
[22] *A guide to testing Rails applications.* `http://guides.rubyonrails.org/testing.html`. Accessed: 2015-09-27.
[23] *Testing Flask applications.* `flask.pocoo.org/docs/0.10/testing/#testing`. Accessed: 2015-09-27.
[24] Olivine Labs. *Busted: Elegant Lua unit testing.* `http://olivinelabs.com/busted/`. Accessed: 2015-09-27.

development and well maintained, so it seemed like an obvious choice.

Before starting the integration of Sailor with Busted, some modifications to Sailor were necessary:

1. Multiple databases
   As of version 0.3, Sailor allowed the configuration of a database to be used by the application. This was changed to allow defining multiple database configurations and change environments to facilitate changing databases for testing.
2. Validation toggle
   Normally, before saving a model, its attributes were validated. Now it is possible to turn off the validation if desired. This is useful for loading fixtures into the test database regardless if they are following the normal validation rules or not since testing with faulty inputs might be desired. model:save() -> model:save(validate) the default is true
3. Adding count method to model
   This modification was not strictly necessary since the counting of entries in a table could be done manually, but this would facilitate making assertions and reduce the amount of code written for tests.
4. Renaming and restructuring Sailor's binary
   When Sailor was installed, it installed a binary called sailor_create, useful for rapidly creating new blank applications. Now that it would gain a new utility, it was renamed to simply sailor, being able to receive two commands: create and test.

```
sailor create "My Application"
cd my_application
sailor test
```

5.1.1 Busted integration

A new module was added to Sailor, called tests. As of now it contains two important functions:

```
test.request(path, data, additional_headers)
```

Makes a mockup request to a certain path of a Sailor application.

path: *string*. The path you want to make a request to, such as a controller or controller/action. Example: 'user/view'

data: *table*. A table containing some data you want to send to the request, such as get or post. Example: get = id = 1

additional_headers: *table*. A table containing additional headers you may want to send. Example: ACCEPT = 'application/json'

This function will return a result table with the following fields:

res.status: *number*. The status of the response.

res.body: *string*. The body of the response.

res.headers: *table*. Any headers out that were set.

res:redirected(path): *function*. A function that receives an internal Sailor app path and sees if the request was redirected there.

```
test.load_fixtures(model_name)
```

Loads tests fixtures into the database.
model_name: *string*. The name of the model to be loaded.
Returns a table with the objects created.

A new function was also added to the form module.

```
form.ify(object)
```

object: Sailor model object.
Returns a table of attributes of this model as they would be if sent via POST through a Sailor form. This is useful for mocking a POST request.

Sailor applications now will come with a new directory called tests. It is organised in the following structure:

```
fixtures/
unit/
functional/
bootstrap.lua
helper.lua
```

The fixtures/ directory will contain the fixture files. In Sailor, they are Lua files with the same name as the respective models that they are testing. They must return a table with the sample data. It is important to note that loading the fixtures will truncate existing data so it is important to configure the separate database for running tests on /conf/conf.lua.

Example fixture for testing an User model with two attributes, username and password:

```
-- /tests/fixtures/user.lua

return {
    {
        username = 'joao',
        password = '123456'
    },

    {
        username = 'maria',
        password = '1234'
```

```
    }
}
```

The bootstrap.lua file contains code that the user may wish to be executed before running their tests. It is useful, for example, to load the fixtures:

```
--/tests/bootstrap.lua
...
local t = require "sailor.test"
t.load_fixtures('user')
```

The helper.lua file should be used to write functions that will be shared among different tests, like a helper library.

The unit/ directory will contain the unit tests files. The test scripts must follow a format specified by Busted. More features and details can be found at BustedâĂŹs website. A basic flow includes calling a describe function passing a description and a callback function that calls one or more it functions tests, also with a description and a callback containing specific tests and assertions. A very simple test could be written as follow:

```
-- /tests/unit/user_test.lua
describe("Testing User model", function()
    local User = sailor.model('user')
    local fixtures = require "tests.fixtures.user"

    it("should create a new user", function()
        local count_before = User:count()
        -- Counting current users
        local u = User:new(fixtures[1])
        -- Creating one more user with one of the fixtures
        assert.is_true(u:save(false))
        -- Asserting that it saves
        assert.is_equal(User:count(),count_before+1)
        -- Asserting that the count increases by one
    end)
    -- ... more tests
end)
```

The functional/ directory contains functional tests. They must follow the same format specified by Busted as the unit tests. Here is an example of a functional test script, containing two tests and four assertions, using Sailor:

```
-- /tests/functional/user_controller_test.lua
describe("Testing User Controller", function()
    local User = sailor.model('user')
    local test = require "sailor.test"
    local form = require "sailor.form"
```

```
    local fixtures = require "tests.fixtures.user"

    it("should open index", function()
        local res = test.request('user/index')
        -- Getting the response a mock request to
        --  the index of the controller
        assert.same(200,res.status)
        -- Asserting that the status of the response is ok
        assert.truthy(res.body:match('View all'))
        -- Asserting that the page contains a string 'View all'
    end)

    it("should create a new user", function()
        local count_before = User:count()
        -- Counting current users
        local res = test.request(
                  'user/create',
                  {post = form.ify(fixtures[1])}
                )
        -- Posting some user to the 'user/create' page of your app
        assert.same(Category:count(), count_before + 1)
        -- Asserting that the number of users increased by 1
        assert.is_true(res:redirected('user/index'))
        -- Asserting that the page redirected you to the index
    end)
end)
```

5.1.2 Running tests

To run tests all that's needed to be done is to go to the app's directory and type sailor test or sailor t on command line. It is also possible to pass some flags that are accepted by Busted.

Example:

```
cd my_app
sailor test --verbose
```

There should be an output like this:

```
************************
24 successes / 0 failures / 0 errors / 0 pending : 0.04076 seconds
```

5.2 Improving the usability of Lua at the client side

As of version 0.3 Sailor applications were shipped with a Lua-to-Javascript virtual machine called Lua5.1.js[25] developed by Alexander Gladysh. This integration needed a major rework for two main

[25] Alexander Gladysh. *Lua 5.1, built with emscripten, with low-level API.* `https://github.com/logiceditor-com/lua5.1.js/`. Accessed: 2015-09-27.

reasons:

1. The JavaScript bridge is not part of the VM. It was developed separately by a contributor when it was integrated with Sailor and it was incomplete. Some basic JS functions were missing. This was an issue because it did not allow for a very good DOM manipulation using Lua and because I did not want Sailor to be responsible for keeping this bridge updated.

2. It was virtually impossible to use the Lua require function on the client. This was an issue because it went against the goal of sharing existent Lua libraries, which made enabling Lua to the run on the browser basically useless. It was possible, but to do it the Lua library needed to be copied to the root of the app and manually mapped in a JavaScript file, which was a huge hassle, a lot of work and did not work on the go. Later on, a contributor added a function that would serve a Lua file on a required path for it to be loaded by the virtual machine. However, this modification only worked on applications who ran on Apache server. This means this functionality was a mess and needed to be simplified.

Looking for alternatives, an interesting option appeared. It's called Moonshine[26] and was developed by Paul Cuthbertson while he worked at Gamesys. Moonshine is a lightweight Lua VM for the browser with a wonderfully built JS bridge that not only works nicely for manipulating the DOM but it is also able to communicate well with JavaScript libraries, such as JQuery.

Moonshine does not accept code strings, but Lua bytecode, which meant that the Lua files needed to be manually pre-compiled. Fortunately, Lua is able to compile itself and this issue was circumvented by using the a combination of the loadstring() and the string.dump() functions on the Lua code read from the view files.

```
s = string.dump(assert(loadstring(s)))
```

Moonshine, however, had an issue with requiring Lua modules. It was also necessary that the file was previously pre-compiled and made available at the application's directory. It was a smaller hassle than Lua5.1.js because it did not needed to be manually mapped in a JS file, but it was still inconvenient. Fortunately, when contacting Moonshine's support, Cuthbertson was very helpful and provided a new functionality in Moonshine, which allowed bytecode to be loaded by vm.preload() function. With this in hands, Sailor could now automatically look for the library on its package.path, load it, convert to bytecode and send to the virtual machine.

Examples:

1. Manipulation of the DOM

```
<div id="app"></div>
<?lua@client

local app = window.document:getElementById('app')
```

[26] Paul Cuthbertson and Gamesys Limited. *Moonshine: A lightweight Lua VM for the browser.* `http://moonshinejs.org/`. Accessed: 2015-09-27. 2013.

```
print(app.textContent)
app.textContent = 'lets go'

window:alert('This code was written in Lua')

?>
```

2. Accessing Javascript functions and passing callbacks

```
<script>
function myJSFunction(msg){
    console.log(msg);
}

function myJSFunctionReceivesCallback(callback){
    callback();
}
</script>

<?lua@client
window:myJSFunction('Calling a Javascript function from Lua')

local function lua_callback()
    print('This is printed from a Lua function being called in JS')
end

window:myJSFunctionReceivesCallback(callback)

?>
```

3. Exporting Lua modules to the browser

   Remember that this code will run on the browser and some Lua modules won't make sense being used in this context! Attention: this feature is still under tests.

```
<?lua@client
local valua = require "valua"
-- If you installed Sailor, valua, our valuation module
--  was installed as a dependency

local v = valua:new().len(3,10)
print(v('Geronimo'))
-- true
?>
```

4. Accessing Javascript modules such as JQuery

```
<script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
<script>
function JQObj(s){
    return $(s);
    // This is necessary because the $() syntax will error on Lua
}
</script>

<div id="app"></div>

<?lua@client
local app = window:JQObj('#app')
app:html('This will be the new content of the div')
-- .html() is a JQuery function.
-- Please observe that in Lua we will use the ':' notation
?>
```

After the implementation of the Moonshine integration, replacing the Lua5.1.js virtual machine, a new issue was found. Since Moonshine received bytecode and the bytecode was generated by the server side execution, the Lua version accepted by Moonshine needed to be the same as the one running on the server side. Moonshine is compatible with Lua 5.1 Even though code written in Lua 5.1 style can be interpreted by LuaJIT, this would still pose a problem. Unfortunately, LuaJIT based servers would generate a completely different bytecode, unacceptable on the virtual machine. LuaJIT is a very performant dialect of Lua and the default Lua version when running Sailor applications on the OpenResty distribution of Nginx. LuaJIT couldn't simply be left aside.

Nonetheless, Cuthbertson was working on a new Lua-to-Javascript virtual machine called Starlight.[27] Starlight has all the benefits Moonshine has and accepts input code as simple strings to be loaded on the VM. When testing, however, it seemed to consume more memory. Seeing that not all Sailor users would run LuaJIT based servers, it was decided that Sailor should be compatible with multiple virtual machines to translate the Lua code. The developer is to be allowed to configure their applications to use their favorite VM. Even though it is in early development, as Starlight is the single VM that works on all situations needed by Sailor, it is the default. Later on, compatibility with another VM called Lua.vm.js[28] was also added. A comparison of all virtual machines supported by Sailor can be seen in the following table:

[27] Paul Cuthbertson. *Starlight: A Lua to ES6 transpiler.* `https://github.com/paulcuth/starlight`. Accessed: 2015-09-27.

[28] Alon Zakai. *Lua.vm.js: The Lua VM, on the Web.* `https://kripken.github.io/lua.vm.js/lua.vm.js.html`. Accessed: 2015-09-27.

Table 1: Comparison of the integrated Lua to JavaScript virtual machines

	starlight	moonshine	lua51js	luavmjs
The code is pre-processed on the server and bytecode is sent to the JS VM		X		
The code is sent as a string to the JS VM	X		X	X
Compatible Lua version of the client written code	5.1	5.1	5.1	5.2.3
Works with Sailor on LuaJIT based servers, such as openresty	X		X	X
DOM manipulation	X	X	incomplete	X
Can require Lua modules	X	X	Only on Apache	
Can send call JS functions and send Lua callbacks	X	X		X
How to print "hi" to the console	print('hi')	print('hi')	js.console.log('hi')	print('hi')
How to pop an alert message with "hi"	window:alert('hi')	window:alert('hi')	js.window.alert('hi')	js.global:alert('hi')

6 References

6.1 Books

Ierusalimschy, Roberto. *Programming in Lua*. Departamento de Informatica, PUC-Rio, 2006.

6.2 Websites

A guide to testing Rails applications. http://guides.rubyonrails.org/testing.html. Accessed: 2015-09-27.

Atwood, Jeff. *The PHP Singularity*. http://blog.codinghorror.com/the-php-singularity/. Accessed: 2015-09-26. 2012.

Carregal, Andre and Tomas Guisasola. *CGILua: Building Web Scripts with Lua*. http://keplerproject.github.io/cgilua/. Accessed: 2015-09-27.

Chapuis, Pierre. *Lua Toolbox*. https://lua-toolbox.com/. Accessed: 2015-09-27.

— *State of the Lua Ecosystem*. http://files.catwell.info/presentations/2013-11-lua-workshop-lua-ecosystem/. Accessed: 2015-09-27. 2013.

Corcoran, Leaf. *Lapis: testing*. leafo.net/lapis/reference/testing.html. Accessed: 2015-09-27.

— *Moonrocks: A command line tool for uploading and installing from the public Lua module hosting site, MoonRocks*. https://github.com/leafo/moonrocks. Accessed: 2015-09-27.

Cuthbertson, Paul. *Starlight: A Lua to ES6 transpiler*. https://github.com/paulcuth/starlight. Accessed: 2015-09-27.

Cuthbertson, Paul and Gamesys Limited. *Moonshine: A lightweight Lua VM for the browser*. http://moonshinejs.org/. Accessed: 2015-09-27. 2013.

DeLoura, Mark. *The engine survey: general results*. http://www.satori.org/2009/03/the-engine-survey-general-results/. Accessed: 2015-09-26. 2009.

Gladysh, Alexander. *Lua 5.1, built with emscripten, with low-level API*. https://github.com/logiceditor-com/lua5.1.js/. Accessed: 2015-09-27.

Google Summer of Code Student Guide. http://en.flossmanuals.net/GSoCStudentGuide/index/. Accessed: 2015-09-25.

Gruno, Daniel. *Introducing mod lua*. http://humbedooh.com/presentations/ACNA-mod_lua.odp. Accessed: 2015-09-25.

Kepler 1.1.1. https://github.com/keplerproject/kepler. Accessed: 2015-09-27.

Labs, Olivine. *Busted: Elegant Lua unit testing*. http://olivinelabs.com/busted/. Accessed: 2015-09-27.

lua-users wiki: Metamethods Tutorial. http://lua-users.org/wiki/MetamethodsTutorial. Accessed: 2015-09-25.

Muhammad, Hisham. *LuaRocks, fostering an ecosystem of Lua modules*. http://hisham.hm/papers/talks/hisham_luarocks_fosdem2015.pdf. Accessed: 2015-09-27. 2015.

— *LuaRocks: the package manager for Lua modules*. https://luarocks.org. Accessed: 2015-09-27.

Q-Success. *Historical yearly trends in the usage of server-side programming languages for websites*. http://w3techs.com/technologies/history_overview/programming_language/ms/y. Accessed: 2015-09-26.

— *Usage of server-side programming languages for websites*. http://w3techs.com/technologies/overview/programming_language/all. Accessed: 2015-09-26.

— *Usage of web servers for websites*. http://w3techs.com/technologies/overview/web_server/all. Accessed: 2015-09-26.

Testing Flask applications. flask.pocoo.org/docs/0.10/testing/#testing. Accessed: 2015-09-27.

Testing Laravel - The PHP framework for artisans. http://laravel.com/docs/5.1/testing. Accessed: 2015-09-27.

The definitive guide to Yii: Testing. www.yiiframework.com/doc/guide/1.1/en/test.overview. Accessed: 2015-09-27.

The Programming Language Lua. http://lua.org/. Accessed: 2015-09-25.

Wagner, Luke and Seth Thompson. *WebAssembly: Development of WebAssembly and associated infrastructure.* https://github.com/WebAssembly. Accessed: 2015-09-27.

Wikipedia: First Class Function. https://en.wikipedia.org/wiki/First-class_function. Accessed: 2015-09-25.

Zakai, Alon. *Lua.vm.js: The Lua VM, on the Web.* https://kripken.github.io/lua.vm.js/lua.vm.js.html. Accessed: 2015-09-27.

7 Appendix

List of Figures

List of Tables