5-2016

Which language(s) are best (for web development)

Jackson A. Stone
University of Tennessee at Chattanooga, ffr569@mocs.utc.edu

Which language(s) are best?
(For Web Development)

By
Jackson Stone

Expected Graduation: Spring 2016, BS in CPSC: Scientific Applications

Honors Thesis
The University of Tennessee at Chattanooga
Honor's College

Examination date: 3/28/16

Professor David Schwab
Project Director

Professor Katherine Winters
Examiner

Dr. Jonathan Mies
Examiner

Dr. Greg O'Dea
Honor's College Representative

Abstract

This thesis is intended to shine light on the complex nature of the online software ecosystem, in the hopes that it may help students who wish to pursue web development determine how to best spend their time researching and learning varying technologies.

Table of Contents

Introduction

This thesis is intended to shine light on the complex nature of the online software ecosystem, in the hopes that it may help students who wish to pursue web development determine how to best spend their time researching and learning varying technologies. More specifically, this thesis is meant to compare and contrast several, often times competing, different ways of constructing a website on the internet. For the sake of the nontechnical individuals reading this, the thesis will contain frequent analogies to help the understanding of technical jargon and concepts. To begin with, a useful analogy for thinking about different technologies on the internet will be that of organisms in an ecosystem.

Every website is more or less unique. However, websites are all made of the same fundamental bits. Much like how all living things have DNA and a metabolism, all websites have shared core components. Each of these websites is built using some sort of language or languages, and for the sake of this paper it will be useful to think of these languages as a way to taxonomically group websites. Much like how all forms of life can be grouped into a handful of kingdoms, all modern day websites are a result of a handful of languages. Also, like biology, many of these languages share a common ancestor, and have deviated from each other at some point in the past. Some languages have managed to develop their own strategy to maintain an edge, while others have been unable to adapt to the changing landscape (in this case the Internet), and have faded away into extinction.

Over time, as the demands and expectations of internet users has increased, many languages have grown in complexity and have become more specialized in their tasks. Often

times these specialized versions of a language will become a higher level abstraction of the root language. These language abstractions are usually not as flexible as the original, but able to do a specialized task much more efficiently. These language abstractions are often called frameworks. This process continues until one framework is so specialized it is not adequate to create most of a site, and several language frameworks technologies become co-dependent for the creation of websites. These language co-dependencies are often called webstacks.

So to summarize, all websites are made from one or more languages. These languages usually share common ancestors, but have branched off of each other at some point in their past, some earlier than others. Some languages are on the rise while others are on the decline. Often times a given language has branched into specialized applications of that language called frameworks. These specialized frameworks will frequently group together in symbiotic relationships with other frameworks in order to create a holistic webstack.

In order to try and make some sense of this crazy jungle, this thesis will first discuss the basic components all modern websites share, the anatomy of a typical website. It will then discuss the origins of several of the most popular root languages on the internet today, and discuss their history up to today. From there it will discuss some of the resulting frameworks and webstacks that are currently in widespread use and show useful metrics to gauge longevity and trajectory of these technologies. Then it will discuss my process of constructing an identical website using each of the aforementioned webstacks, showing

language syntax examples for comparison. I will also run a denial of service (DoS) attack on each of these sites, and gather metrics on each website and how it holds up against the attack, as a way to simulate scalability of the technology.

In conclusion, the reader should have a basic understanding of what is out there in the Internet Ecosystem, the niche that each language is currently filling, the expected trajectory and performance capability of a given technology, and can make an informed decision on what technology to use for a given problem or for a desired website.

The Anatomy of a Webpage

Every website uses Hypertext Markup Language (HTML) which defines the webpage's text, and the structure of content blocks within the browser window. To view the HTML making up any webpage, if you are using Google Chrome, you can simply right click on any webpage, and select "inspect element". This action will display the code your browser is using to generate the webpage you see. All modern websites also use Cascading Style Sheets (CSS). CSS, as its name implies, provides styling to the content within a page such as images, width and height, font family, color, and animations. HTML and CSS combine to form everything you see within your browser. HTML is the infrastructure, while CSS is the furnishings. Below is a very simple example of each. (Table 1)

HTML	CSS	End result

`<h1>Hello world</h1>`
`<p>This is a test</p>`
	`h1 {`
	`  color:red;`
	`  font-size:16px;`
	`}`
	`p {`
	`  color:blue;`
	`  font-size:12px;`
	`}`
		Hello world
		This is a test

Table 1.

Though it may seem hard to believe, this is what makes up the entirety of what you *see* on the internet. However, many sites are not simply arrangements of text and images, they have an interactive component to them as well, like a Google search bar, or a drop down menu. This is where JavaScript (JS) comes in. To continue the infrastructure and furnishing analogy, JavaScript is the electrical work and appliances in the building. It is not required for a building to have electricity to be a functional building, but chances are you do not want to spend a large amount of time there if it does not. It is the same with JavaScript and websites. It is a language that can directly change a given web page's HTML and CSS on the fly. When one scrolls down their Facebook feed, JavaScript is changing the HTML on the screen to contain your new posts as they stream in. In some cases this means very little actual HTML or CSS are sent by a server when you visit a site, but it is instead all added in by JavaScript, like a self-assembling house. Table 2 shows an example of this in JavaScript. Notice how the end result is the same. This is because JavaScript is generating identical CSS and HTML as in Table 1.

JavaScript file	End result

`var d = document;`
`var header = d.createElement("h1");`
`var para = d.createElement("p");`
`var root = d.getElementsByTagName("html")[0];`
`header.innerHTML = "Hello World";`
`para.innerHTML = "This is a test";`
`header.style.color = "red";`
`header.style["font-size"] = "16px";`
`para.style.color = "blue";`
`para.style["font-size"] = "12px";`
`root.appendChild(header);`
`root.appendChild(para);`
	Hello world
	This is a test

Table 2

You may be asking why anyone would ever want to go through so much work for so little and the answer is they wouldn't. In practice it would generally not be practical to structure a site like this entirely out of plain JavaScript. It is a lot of work for relatively little result. However, moving forward it is important to know that JavaScript can insert and mold HTML and CSS, as well as communicate to servers. Though one would likely not use plain JavaScript, frameworks have been created for JavaScript to specialize it for creating HTML and CSS which this thesis will elaborate on in later sections.

These three languages: HTML, CSS, and JavaScript (HCJS) are how the web developers package the things that are sent to users. Everything they send, with the exception of static assets like pdfs and images, is described in these three languages. (Any discrepancies one experiences between different internet browsers is a result of some browsers choosing to interpret these three languages in subtly different ways.)

Although a combination of these three languages is the end result seen by a visitor to a website, this is only half of the story. More often than not, a user will expect a "custom" webpage experience, such as logging into their Facebook profile and looking at their unique newsfeed. Frequently, every user has a unique webpage sent to them. It may have the same general format for everyone, but the content is customized for each user. It would be massively impractical to hand make a unique webpage for each user that plans to visit a site, and it would be far too much data to send it all to each user and subsequently filter it. How this magic is able to take place is with *server-side programs*.

Server and Browser Interactions

When dealing with websites, there are two major sides. The first is *client-side* which is composed of what we have discussed up to this point, namely HCJS. The other half is *server-side*. This is what Google has running on their endless server-racks in San Jose, trying desperately to find what a user is looking for as fast as possible when they input a search. Figure 1 shows a possible interaction with a Google webpage.

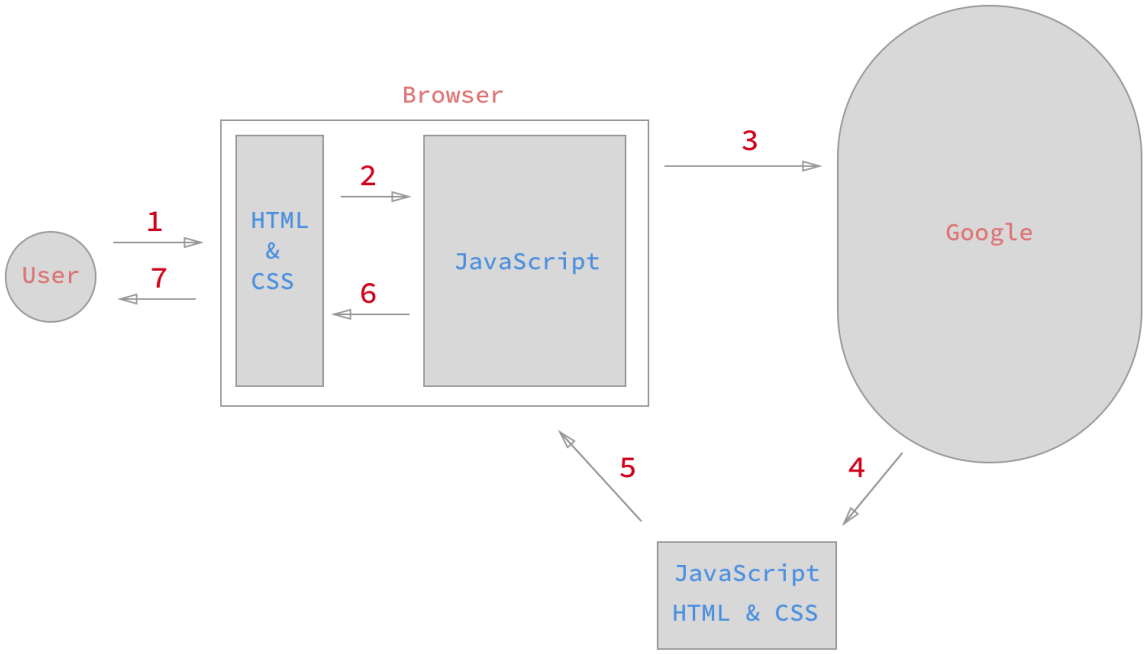


Figure 1

1. The user types something into their search bar, and the search bar is made from HTML.
2. When the user presses enter, JavaScript files take the text from the search bar and perform some preprocessing on the text, packaging it for the server.
3. The JavaScript files then send the user's search query to Google's servers over the internet, including a cookie to help Google identify who you are.
4. Google's servers process the search, and generate a new web page of results personalized to the user based on their search and cookie, in the form of new HCJS or data packets.
5. The browser replaces the old HCJS with the new code sent from Google's servers, or pipe the new data packets to the current JavaScript files

6. The freshly sent JavaScript then does some slight tweaks to the HTML and CSS, to make them just right for the user.
7. The browser then updates and displays the user's search results.

Though seemingly over-complicated for a simple search request, this chart is actually a drastic oversimplification of what is really going on, and is by no means the only way a server interacts with a user. This is to show however, that though HCJS is all that a user sees and interacts with, it is often the tip of the iceberg. Most web applications have a majority of their complexity on their servers, which are generating custom HCJS for their users. That is where the secret sauce for most web giants exist, and it is where there is the least amount of standardization.

Browsers force web developers to use HCJS, however, how a web developer decides to create and then send those files is totally up to the developer. HTML, CSS, and JavaScript files are only text files. They can therefore be generated by any language capable of processing and creating text files, which is most programming languages. (It is not uncommon for one language to generate another when dealing with web technologies.)

All of this to say, there are two main layers to any website, Client-side (HCJS), and Server-side (nearly any language a programmer chooses) which generates custom HCJS and fetches custom data for each user.

A webstack is a set of technologies designed to work together to make this process as easy as possible for a developer. As stated before webstacks can be a combination of several languages, and these languages work together to create the custom HCJS modern internet

users expect to see. This is why almost all websites on the internet are created with one of these stacks. So in discussing currently used technologies in the web world, one needs to talk about webstacks.

Webstacks to be discussed

The first webstack this thesis plans to delve into will be Ruby on Rails (RoR), which is one of the most popular webstacks to date using a combination of its predominant language Ruby, a dialect of JavaScript called CoffeeScript, an abstraction of HTML called html.erb, and an abstraction of CSS called SASS. The next webstack will be Laravel, a framework that uses PHP as its main language, one of the oldest server-side languages still in widespread use today. It also uses an abstraction of HTML called *blade.php*, and a CSS abstraction called LESS. (PHP is an acronym for PHP hypertext processor. The first P in PHP stands for PHP. In the computer science community irony is common, even among language creators.) Many aspects of Ruby on Rails influence can be seen in this framework. The last two webstacks this thesis will cover will be more recent webstacks that are composed almost entirely of JavaScript. The first is called the MEAN stack, and the second, Meteor. The last two represent a current shift in the philosophy of web-developers who feel many frameworks are bloated with complexity because of all the interactions happening between different languages, and attempts to streamline development by having everything in one language.

To understand these webstacks one needs to know some basic information about the main languages used in them. Therefore this thesis will briefly discuss the history and

intended use case for PHP, Ruby, and JavaScript, and then delve into the nitty-gritty of how they are each able to create a website, showing code snippets in each webstack. I have created four identical versions of a website in each of these stacks, and though they look the same, they share relatively little similarities in code. I will then briefly discuss their professed strengths and weaknesses, and show statistics on employability of these technologies.

RUBY

Language History

Ruby is the only language this thesis will discuss that was not created in the United States. It was developed in Japan by Yukihiro Matsumoto, or "Matz", as the Ruby community refers to him. His intent in creating the language was to have an object oriented scripting language. As defined by Oracle, object oriented means: "a method of programming based on a hierarchy of classes, and well-defined and cooperating objects." [2] In essence this means, while programming, one creates structured sets of data, called objects, which each contain certain methods or functions that can be performed on their stored data. The "scripting language" component, means the language is not compiled. In other words, the code is read dynamically, while it is being executed. To give an analogy, it is like performing a heist where no one knows what they are supposed to do until it is time to execute, only given knowledge of the current task. Because of this, the code can perform operations on itself, and be more flexible. In a way, it can improvise. These two seemingly conflicting concepts, one of rigid data structure, and another of dynamic and flexible code execution, are what make up Ruby.

Current Usage

Another distinguishing factor of Ruby, unlike JavaScript and PHP, is it is almost exclusively used in the context of its webstack, Ruby on Rails. JavaScript and PHP exist within a number of different contexts and webstacks and are fairly agnostic about how a programmer should structure their whole workflow. Ruby on the other hand, is less of a standalone language (though technically it has the capacity to be), and more of a syntactic component of Ruby on Rails, or "Rails" for short. [3]

For this reason, the advancement of Rails as a webstack, has steered the development of Ruby as a language. As a counter example, Laravel, the PHP webstack in discussion, must add multiple libraries (packages of code that give a programmer access to certain common use-case functions that are not built into the language) into the framework to make the magic happen. Also though many people may be programming in PHP, only a fraction will be using Laravel. Rails, on the other hand, has the advantage of having the full attention of the Ruby community. It has evolved into a webstack specific language.

Ruby on Rails

As mentioned before, the line between Ruby and Ruby on Rails is thin, because Ruby is almost exclusively used in the context of Rails. However, while Ruby introduces some interesting ideas as a language by combining Object Oriented design with a scripting language, Rails also introduces some new concepts into webstacks. One of the most notorious is the idea of "Convention over Configuration". In short, Rails enforces having a file structure in a particular layout (All your HTML.ERB layouts must go in App/views/layouts

folder for example.) In this way you do not need to tell Rails where you have stored any of your files. If you adhere to its rules, it will know where to find everything it needs. This also means a Ruby developer can more easily pick up where any other ruby developer has left off with their Rails app. The downside is less freedom in how you choose to structure your app's organization. Rails also enforces a Model-View-Controller architecture which is a way of separating concerns of an app between your data (models), business logic (controllers), and displays (views). There are several components to allow this kind of behavior.

HTML.ERB is the HTML abstraction Rails uses to generate HTML pages. Table 4 shows a basic example that loops through a set of data, generating all the HTML necessary to display the page as desired.

Shows all "Year" fields in a database. File: app/views/welcome/index.html.erb

```
<% @years.each do |year| %>
  <tr>
   <td><%= year.title %></td>
   <td><%= year.text %></td>
   <td><%= link_to 'Show', year_path(year) %></td>
   <td><%= link_to 'Edit', edit_year_path(year) %></td>
  </tr>
<% end %>
```

Table 4

Table 5 shows an example of code that would pass the "year" data to the code in table 4. In this example, table 4 is an example of a "view", while table 5 is the corresponding "controller".

Controller passes *@years* to the view. File: app/controllers/welcome_controller.rb

```
class WelcomeController < ApplicationController
 def index
  @years = Year.all
 end
end
```

Table 5

Table 6 is an example of the "model", which defines the structure of the year data. In this example, the database's default structure of rows and columns is used, so the file is mostly blank. (Also an *.RB* file)

Model tells controller what to fetch from database. File: app/models/year.rb

```
class Year < ActiveRecord::Base
end
```

Table 6

Most of the work is done based on naming conventions of the folders and files in the system. In other words, I never need to specify that the year model is in app/models/year.rb, it simply expects to find it there. This is the core of convention over configuration and this concept will be seen in Laravel as well, which is heavily influenced by Rails in terms of infrastructure.

Also, to generate CSS, Ruby developers tend to use SASS. This is a language that compiles into CSS (Table 7). Notice that like Ruby, SASS uses white space in the place of "{" and "}". This is an example of convention over configuration present in the SASS language

itself. The core idea is if a programmer is doing what they are doing the *correct* way they need to type a lot less code. However one must adhere to the rules, otherwise the system will break.

SASS (character count: 116)	Resulting CSS (character count: 151)

`.navigation`
` ul`
`  margin: 0`
`  padding: 0`
`  list-style: none`
` li`
`  display: inline-block`
` a`
`  display: block`
`  padding: 6px 12px`
`  text-decoration: none`
	`.navigation ul {`
	` margin: 0;`
	` padding: 0;`
	` list-style: none;`
	`}`
	`.navigation li {`
	` display: inline-block;`
	`}`
	`.navigation a {`
	` display: block;`
	` padding: 6px 12px;`
	` text-decoration: none;`
	`}`

Table 7

In principle this comprises a very simplified list of Ruby on Rails components. There are volumes that could be written on the language Ruby alone, which is about 20 years old now, let alone all of Rails. Ruby is still under continual development as a computer dialect, going through a new release last December[4]. Rails too, now 10 years old, is about to release version 5 of the framework, which is still being maintained by its creator, David Heinemeier Hansson (known as DHH in the Rails community) and his team. They have just released the beta version 5 of the platform, which offers websocket support, a more recent technological innovation in the web to allow real-time data updates[5]. Websockets will come up again when discussing Meteor.

The image count will be the same across all the different versions of the website. However, the size of *doc* (HTML files), *js* (JavaScript files), and *css* (CSS files), will vary slightly based on the webstack in use. Also, what is not shown here is the amount of work being done by the cloud server to generate these pages. This will vary heavily based on the webstack.

PHP

(PHP: Hypertext Processor)

Language History

PHP was created by Rasmus Lerdorf in 1994. Initially PHP stood for Personal Homepage Page tools. PHP was later renamed to the recursive title of "PHP: Hypertext Processor", PHP being the first P in PHP. It was originally created to track visits to Lerdorf's resume site. Originally written in C, the toolkit grew and grew over time, acquiring a richer feature set. It eventually allowed for the data within databases to dynamically direct what a webpage presented. In June of 1995, the PHP source code was released, allowing the public to tweak and improve the language. The language began to get popular in the young web landscape, and in 1998, at least 60,000 domains had the distinct "PHP" headers in their webpages. This was approximately one percent of all websites at the time, all while the language was still chiefly being maintained by a single individual. [7]

Current Usage

Over time, more developers began to join in and cultivate the language, and by 1998 a reported ten percent of all web servers were running the language. PHP 5, the most current

version, is currently installed on tens of millions of web servers and is one of the leading server-side languages by the numbers. It is what much of Facebook, and Wordpress is built on, though several of the companies heavily using PHP are trying to move away from the aging language[8]. It was one of the first to enter the web game in a big way, and that has solidified it in the modern market, for better or for worse. The language was created by a hobbyist with a specific task in mind (track visits) and that has shown with time.

The common analogy with PHP is if you ask for a hammer, it will hand you a tool with claws for pulling out nails on both ends. The tool can be used to hammer in nails still, but it does not work the way you would expect it to work [9]. Debugging is also a large inconvenience in PHP. Because errors will frequently not show up until trying to see your webpage, and no PHP is run in the browser, one has to juggle between their console and the browser and try to figure out what has gone awry. PHP is one of the few languages where missing a line terminator ";" can still result in hours of searching for where you missed placing that single character. The webpage simply will not appear until the semicolon is in place.

However the language has a lot of momentum, arguably the most popular of which comes in the form of WordPress which makes up an estimated eighteen percent of all self-hosted websites on the internet, which is an enormous market share [10]. That means there are tens of millions of sites built on PHP, so the language will be in use for quite some time despite its eccentricities and inconsistencies. This also results in there being a number of people that have specialized in the oddball language.

Because of the talent density of the language there have been some attempts at creating webstacks that help alleviate much of the language's security flaws and ad-hoc structure. The framework this thesis uses as an example (though there are many PHP frameworks) is Laravel.

Laravel

Laravel was created by Taylor Otwell in 2011, so it is a relatively young framework compared to Rails, but it makes a good example for cutting-edge PHP development. Like Rails, Laravel incorporates a Model-View-Controller (MVC) architecture. In other words, there are separate files for the actual user interface or *UI* (views), the data structures (models) and the business logic (controllers)

Considering that PHP is a language that has been patched together over the years with, sometimes, dozens of ways to do the same thing, Laravel offers decisiveness with how to structure your application. Laravel also goes one step further than Rails. Laravel not only requires you to structure your app in a particular way but they also provide you with a virtual machine (VM) configured identically to the servers they will provide to you to host your application, for a fee. This allows a developer to test code on their local machine knowing it will operate in an identical way to when they push their code up to a cloud server offered by Laravel through *Forge,* a service the developers of Laravel offer. In addition to that, they offer *Laracasts* that allow a beginner to become competent with the technology starting off with minimal programming experience. They also have a suggested service for collecting online payments through *Cashier* and social sharing through *Socialite*.

In other words, a developer working in the Laravel webstacks, needs to make *no* decisions about which service to use to learn, publish, or cash in on their apps. Laravel makes all of those decisions for them from the start. This, depending on who one asks, is a great strength and weakness of the webstack. As an example, for the purposes of this thesis in testing Laravel's performance, I was required to publish my Laravel app through a service other than *Forge* (Amazon web service, in particular). Because the system is highly coupled to Forge for cloud hosting, this was difficult to do, and documentation on how to do this was practically non-existent. Nearly all of Laravel's tight-knit community is using the suggested services the Laravel officially supports. This means it is hard to go off of the well-treaded path when using Laravel.

Components of Laravel

Views: For view creation Laravel uses a Laravel specific template system called *Blade,* all of which have the file prefix "blade.php". All views are also in a particular directory in the predefined file tree structure, similar to Rails. Blade files in particular for *Year* objects go in "/resources/views/years/". Table 8 gives an example of a Blade file showing all of the year objects sent to it from a controller:

/resources/views/years/index.blade.php

```
@foreach($years as $y)
   <div class="year">
     <h1 id="{{$y->title}}">
      {{$y->title}}
     </h1>
     <p>{{$y->path}}</p>
  </div>
@endforeach
```

Table 8

The controller in question is a PHP file structured like the text in Table 9.

/app/http/controllers/YearsController.php

```php
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;
use DB;
use App\Year;

class YearsController extends Controller
{
 public function index()
 {
  $years = Year::all();
  return view('years\index', compact('years'));
 }
}
```

Table 9

The “Year::all();” command above gets the “Year” objects from the database, whose location is in the file that defines the *Model* which is another PHP file structured like Table 10.

/app/Year.php

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Year extends Model
{

}
```

Table 10

Table 10 shows code that implies there should be a “years” table in your database. Any controller file that subsequently uses “Year::all()” will know to check the database table named years when trying to fetch years. The Laravel specific system that allows for this to take place is named “Eloquent.” The above example is similar to how Rails defines models, with the exception of namespaces being necessary at the top of the file.

In terms of a CSS abstraction, Laravel gives developers a couple options, but the only one that works from the very start is LESS, also known as “{less}”. Like Rails with Sass, LESS compiles to CSS. Table 11 shows an example of LESS and the resulting CSS.

{less} (Character count: 68)	Resulting CSS (Character count: 82)

`#header {`
` color: black;`
` .navigation {`
`  font-size: 12px;`
` }`
` .logo {`
`  width: 300px;`
` }`
`}`
	`#header {`
	` color: black;`
	`}`
	`#header .navigation {`
	` font-size: 12px;`
	`}`
	`#header .logo {`
	` width: 300px;`
	`}`

Table 11

All of this together resulted in the lightest site by far. Due to all business logic of the site happening inside the server and practically none happening through JavaScript, fewer files need to be sent in response to user requests. This means the bottleneck for the load time of a site is determined more by the performance of the server, rather than a user's bandwidth. A full table of comparisons between the differing sites memory footprint can be seen in table 22.

JavaScript

Language History

JavaScript (JS), not to be confused with the similarly named common computer technology Java, has a complex history. First of all it was created in only ten days, in 1995 by Brendan Eich, who at the time was working at Netscape (modern day Mozilla.) After receiving

a trademark license from Sun (the owners of Java at the time), its name became JavaScript. The association with Java was desirable due to the widespread adoption of Java during that time, meaning the name carried an air of familiarity and dependability in the computer science community. The fledgling JavaScript language was then able to latch onto the rise of Java and ride on its coat tails.

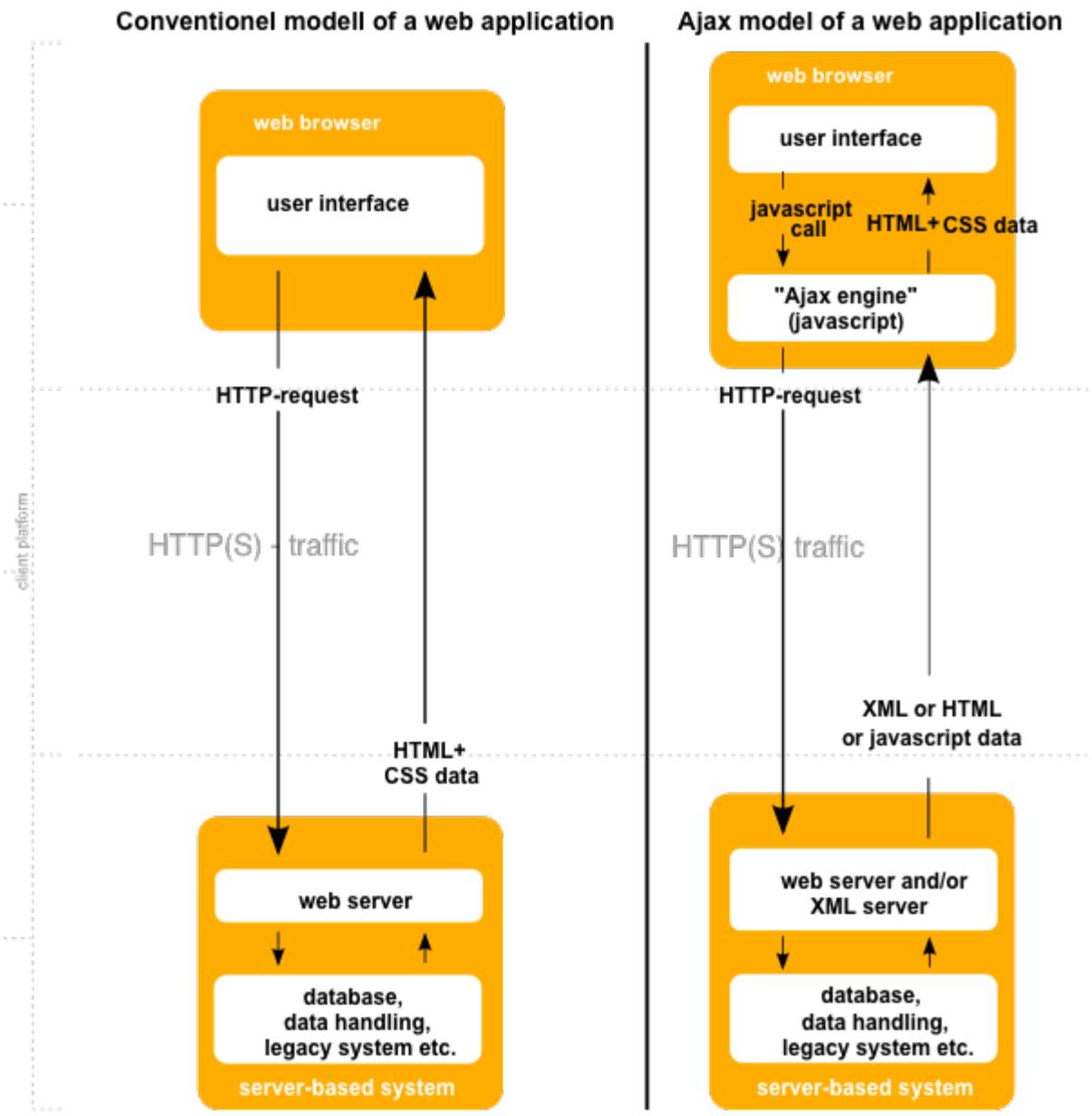
Creating a new language in only 10 days was an impressive technical feat, but the short development time left some serious gaps in JavaScript. A group called the European Computer Manufacturers Association (ECMA), a non-profit computer standards organization, began to standardize the language, ironing out several of its quirks, in late 1995. This early standardization is in opposition to PHP's overall lack of early organization. For this reason, JavaScript, though far from a perfect scripting language, is a much more consistent language than PHP [9]. How JavaScript became a part of every webpage we visit today is also an interesting piece of history.

The purpose of JavaScript back when it was first created was to manipulate Java Applets (A method of embedding Java Programs inside a browser) in the Netscape (now Firefox) browser. It was designed for people who did not want to spend weeks learning all of the intricacies and complexities of Java. JavaScript abstracted away most of that complexity behind a simple and flexible scripting language. Rather than compiling into static executing code, JavaScript was able to change how it executed on the fly, unlike its popular counterpart Java, making it easier to experiment with. This lead to JavaScript being used for much more that Java Applet manipulation. Developers began using it for a number of things, like

manipulating images and the text that appeared on the screen. With Netscape being the only browser with a JavaScript engine, it had a monopoly on this kind of dynamic behavior [1]. What followed is known as the “Browser Wars” and it is arguably the best thing that ever happened to JavaScript.

Browsers are constantly in tight competition for market share of internet users. The main opposition of NetScape’s browser at the time was Internet Explorer. Not to be outdone by Mozilla, Microsoft implemented their own JavaScript engine, one that claimed to be much faster. An arms race followed. By the time Google joined the fight with Chrome, all three major companies had some of their best engineers working to make their JavaScript engines airtight and faster than the competition. Tens of millions of R&D dollars, and hundreds of top notch engineers’ best efforts were put into the little language until it became the fine-tuned powerhouse it is today, ubiquitous across all web pages.

Over time, two major additions came to the language. The first was AJAX (Asynchronous JavaScript and XML) calls. This allowed JavaScript functions to ask a server for more data without having to visit a new page. This is something most internet users take for granted, but it is a relatively new invention (2004-2005) [12]. The change this caused can be shown in Figure 2.



By DanielSHaischt, via Wikimedia Commons - https://commons.wikimedia.org/wiki/File%3AAjax-vergleich.svg, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=29724785

Figure 2

Current Usage

The introduction of AJAX meant that instead of sending whole web pages in a big batch over the internet, one could just send data, and allow the JavaScript that was sent

earlier to make the web page on behalf of the server. In this way one can distribute the computational burden between users. Also after the initial load, limited bandwidth is not as limiting. Data heavy interactive web applications (like Google Maps) became possible.

Employability

JavaScript began to take on more and more of the complexity of web applications. Over time, more and more of an application's muscle was moved into JavaScript files that ran in the user's browser. To cope with this rise in JavaScript complexity, more JavaScript engineers had to be trained and hired to handle the demand, leading to JavaScript being the most popular language today in terms of job offerings, and community size (Table 14) when compared to any other programming language. This also proved to the industry that JavaScript was capable of scaling and handling enterprise level problems. What followed was that industry leaders started to write all of their application, client-side and server-side, in JavaScript.

Language	Jobs with language as keyword on Glassdoor as of Feb. 2016[1]	Package counts as of Feb. 2016[2]	Stackoverflow users using language[3]	Github popularity rank[4]
JavaScript	57,772	npm: 245,487 Bower: 49,452	54.5%	#1
Ruby	16,352	Rubygems: 114,958	8%	#3
PHP	15,575	Packagist: 87,433	27.9%	#4

Table 12

1.https://www.glassdoor.com/index.htm
2.http://www.modulecounts.com/, https://rubygems.org/search?utf8=%E2%9C%93&query=
https://packagist.org/statistics
3.http://stackoverflow.com/research/developer-survey-2015
4.http://githut.info/

Node.js

Node.js, also known as Node, is the server-side implementation of JavaScript, first created in 2008 by Ryan Dahl. It runs using the V8 engine, which is the same JavaScript engine used by the Google Chrome browser. In essence it makes a server into a faceless browser that executes JavaScript code continuously, waiting for requests from users. It is all asynchronous by default, meaning whenever I/O operations are taking place it can continue to run, while a chunk of code is pushed onto a worker, which waits to hear back from the database. In other words, there is no downtime. The "Event Loop", which is the name of the JavaScript mechanism that makes this behavior possible, is always spinning. Ruby and PHP can imitate this behavior if a few hoops are jumped through, but with Node, it is the default behavior. For this reason Node Servers are known for scaling well, and are gaining

popularity. As proof of Node's popularity, its community package system is already the most popular in the world, nearly twice that of the second most popular, Java [13]

The advent of Node also meant developers that had been programming client-side scripts (front-end developers) could start to do server-side programming (back-end developers). All web-centric companies were already staffing JavaScript developers, because one needed them to create most websites. After Node.js, conceivably, companies only needed to heavily hire JavaScript developers, which were already plentiful due to preexisting demand.

As a result of this relatively new JavaScript ubiquity, several "Pure" JavaScript stacks have come into being. This thesis discusses two such stacks. The first, MEAN, is a combination of several very popular NPM, or Node Package Manager, packages, used in conjunction with one another. The last, Meteor, is perhaps the most ambitious of the webstacks covered so far, which blurs the line between Client and Server altogether.

MEAN

The MEAN stack is chiefly a combination of four things: MongoDB, ExpressJS, AngularJS, and of course, NodeJS. This is the most flexibly structured of all the webstacks covered in this thesis, being composed at its very base of four decoupled packages.

MongoDB is in a family of databases called "NoSQL". The name NoSQL is somewhat purposefully inflammatory of SQL databases, which are by far the most common family of databases. To give an analogy, an SQL database, like MySQL, is anatomically structured like a several paged Excel spreadsheet. It is a large collection of named columns called "attributes"

and rows called “entries”. This is then broken up further into tables. Each table can be thought of as holding a single type of entry. The problem with this kind of database is if your data is not conveniently structured in a two dimensional way.

For example, say one wanted to make a meaningful database of classes at UTC. A “class” has many students enrolled, it may also be offered several different times. To structure this data in a SQL database, one would require several tables. One would be the class table itself. Each class entry would have an ID column, as well as entries like “name” etc. Then there would be a student table, each with their own ID and other pertinent information about the student. Then one needs a table of “offerings” which stores a class ID, and a date for when the class is offered. Then one needs a table of “enrollments” which stores, for each student in each offering of each class a pair of student IDs and offering IDs. This is not an uncommon scenario in SQL database architecting. The web of tables explodes rather quickly for a complex system.

This is where NoSQL, or non-relational databases come into the picture. Rather than creating a separate table for each type of object, and relationship between each object, one can embed objects inside of each other. That may sound complicated, but it is really the way most humans already think of things.

To demonstrate the previous example, to create the above “class” database in a NoSQL database like MongoDB, one would need only two data types in their database, student and class. They would be structured as shown in Table 13

Student	Class
_id: (The student's unique ID) name: … (other stuff about student)	id: (class ID) offerings:[{ time: 9:00am, students_enrolled:[list of student IDs] }, { time: 11:00am, students_enrolled:[list of student IDs] },], other stuff about class…

Table 13

Each class data point contains a list of offerings that each in turn contain a list of student IDs. This makes the database more intuitive, because it can be organized in the way one already thinks about it. Lists within lists are OK in a Non-relational database. Put simply, your data structure does not need to be flat. The data is formatted into JSON (JavaScript Object Notation), which, ironically, is a prevalent data structure in many programming languages. And as the name implies, it has the same syntax as Objects in JavaScript. In other words, MongoDB stores data as JavaScript Objects. Even the database is structured like the JavaScript.

In order to secure one's database against injection of attributes that are undesirable, a typical package to use along with MongoDB is called Mongoose. This checks everything that

is entering the database to ensure it meets the desired structure of JSON. Table 16 shows an example of a year model in Mongoose.

model.js

```
var mongoose = require('mongoose'),
  Schema = mongoose.Schema;

  var YearSchema = new Schema({
    poi: [{
      title: String,
      text: [String],
      additions: [String]
    }],
    title: String,
    languges: [String]
  });

  var YearModel = mongoose.model('Year', YearSchema);
```

Table 14

Code shown in table 14 is all one needs to create a “Year” object in their database that contains a list of pois (Points of interest), that each contain titles, a list of differents texts, and a list of additions. It also contains a “title” field for a title of the year, and a list of languages. The last line attaches the Schema to the model of Year in MongoDB. In a SQL database, this same data structure would take at least 5 different tables. So though it seems complex, compared to the status quo, it is a drastic improvement in coder efficiency. Also, after pulling the data from a database, if one is coding in JavaScript no parsing is necessary because the

data is already formatted to the shape of a JavaScript object. One can directly begin using the data like any other JavaScript variable.

ExpressJS handles routing. In other words, when you visit a site with a URL like: "https://www.mysite.com/about/jackson" Whatever site is dishing out the "mysite.com" domain name, has their server listening for what's added at the end of the URL. It sees at the end of "mysite.com" is the text "/about/jackson". Express sets up listeners for these strings and will return the appropriate pages based on this string. If nothing is added at the end, the server will usually have a default homepage setup.

In ExpressJS, a file to handle the above URL looks something like what is shown in Table 15.

app.js

```
var app = express();
index = function(req, res){
  res.render('/views/index');
};
app.get('/about/jackson', index);
```

Table 15

When a user asks for the URL "https://www.mysite.com/about/jackson" ExpressJS will fire the *index* function which will render the index page stored in the views folder. In Rails and Laravel, this "routing" is built into the webstack. Because MEAN is more of a set of commonly loosely grouped packages rather than a cohesive webstack, ExpressJS is what provides this functionality.

Also because the whole application is JavaScript already, most of the view behavior is defined in client-side JavaScript. This is done in a front-facing JavaScript Framework called AngularJS. Unlike Laravel and Rails, that have a Model-View-Controller architecture server-side, with a MEAN stack application, the server only worries about responding to API calls and updating the database, and all the other logic is run in the client's browser.

AngularJS was developed by Google, and offers a number of organizations for JavaScript development that are not normally present. Using the Ajax calls described earlier, it creates dynamic web applications that after the initial page visit, only need bits and pieces of data from the server to generate subsequent pages. This can result in a slow initial load time because the server sends all logic at the start, but all subsequent interactions with the application are generally much faster. Table 16 includes the JavaScript used to make an Ajax call to the server to fetch some data, and the HTML below it that renders each "year" object returned from the NoSQL database. Notice how no parsing is necessary to use that data returned from the database.

yearcontroller.js

```
myApp.controller('years', ['$scope', '$http', function($scope,
$http){
  $http.get('/data').then(function(result){
    $scope.years= result.data;
  });
}]);
```

index.html

```
<div ng-repeat="year in years">
{{year}}
</div>
```

Table 16

Also, if the computational complexity of the application is great enough, this can allow a service provider to distribute the computational load of the app. In other words, if one were to submit a profile photo to a Laravel server, the cropping of that photo would likely take place on the server, increasing your server fees for an adequately sized application. On the other hand a heavily client-side JavaScript application, can have a user's computer use its own resources to crop the photo before it is sent to the MEAN server. As a user this may only be a couple second delay on their end, but the server does not have to do any of the work. This approach tends to scale much better because of each visitor to the site effectively donates their machine to run application logic. Their computational resources grow with each new visitor. All their servers have to do is send a user the scripts to run.

In terms of the size of the site, a MEAN stack website is larger in the initial size of packages delivered (conventionally), because it sends the index.html page, styling, *and* JavaScript files that handle dynamic page generation. Data is then fetched from the server on a need-by-need basis. So at least two round trips are needed to generate the page. The JavaScript then needs a couple frames to re-render the page for the user to fit the new data that has come in. The payoff however, is shown when there is a multi-page website, and users jump from page to page. All subsequent calls to the database are just data fetches. Frequently, no new HTML or CSS needs to be fetched, as is the case in many server-side rendering stacks.

Meteor

There are however unique downfalls to the MEAN stack. Mainly, unlike Laravel, and Ruby on Rails, there are many different flavors of MEAN stack, and no one's two MEAN stacks are identical in terms of packages being used, and version numbers. Because MEAN is a loosely coupled set of open sourced Node packages, not maintained by any central authority, the likelihood is that your stack is a unique snowflake among all other servers on the internet. This means when debugging, resources that fit one's exact needs are hard to come by because the community is so fragmented. Some use Angular 1.4, while others have decided to opt for Angular 2, which is not backwards compatible and has some pretty substantial syntactic differences. Also a package like ExpressJS may one day simply lose its contribution base in a matter of months because the package is supported by volunteers. When building something of scale or enterprise level, this can be a deal breaker. MEAN stack applications do not age well, because the 4 technologies are widely evolving on different trajectories, and there is no central authority ensuring backwards compatibility.

That is where Meteor comes into play. Meteor is an open sourced webstack, but where it differs from stacks like the MEAN stack is all of its major components (with a couple small exceptions like routing) are built in-house by a single for-profit company (Meteor Development Group) that is well funded [14]. The company earns revenue through offering development support and deployment automation and scaling through a service called Galaxy. The whole system is built on NodeJS, and offers a unique approach to Client vs. Server style applications.

Being a centralized JavaScript solution, Meteor is able to do things other webstacks cannot. As an example, Meteor can run the same code on the server and the client. An example of this is shown in Table 17.

Example of how Meteor separates client, server, and code meant to run on both.

```
if (Meteor.isClient) {
//code here will only execute on the user's browser
}
if (Meteor.isServer) {
//code here will only run in the server.
}

//Code outside will run on both the server and browser.
```

Table 17

This allows for "optimistic UI" features out of the box. It also means you only need to write your validation once. The same code that validates a submission on the server, can be sent to the client. (If say I want usernames to have a minimum length of 8 characters, I do not need to check that on the browser and the server. I only need to write code for that once, and it can be run in both locations.) This also allows Meteor to have a unique advantage when it comes to its packages.

Because Meteor is a uniform JavaScript webstack, that guarantees certain structures of a developer's app. This means package authors are able to write packages that incorporate front and backend logic.

To illustrate what this means, for most applications, if one wanted to implement a login system, they would need a number of packages. They would need to have a way to store user data securely, either on their own server or a 3rd party. They would also need

some method of client-side hashing of passwords, and usage of cookies to maintain that logged in state. They also then need to make a module that allows users to actually input a login and password. Then they need a server-side system that can authenticate that the user is in fact logged in for the current session. This usually amounts to four different things needed to be implemented. If one wants to use packages to achieve this they will have to find four different packages that can handle all of this, and the packages need to be aligned in how they interact. This is rarely the case so some configuration and integration overhead will be incurred.

Because Meteor abstracts the server-client relationship largely behind the scenes, a single package can add all of this to your project. What previously would take a week to implement in the MEAN stack, in Meteor - if a good package is available - can take a matter of minutes to get up and running because package authors can be more confident is how a user has structured their app.

When an app is pushed to production, Meteor will by default compress and package all of your JavaScript in the most efficient way possible. Any Precompilers (CoffeeScript, TypeScript, ES6, {less}, SASS, Stylus, etc.) can all be used easily within the meteor ecosystem, because a single package can edit the innards of a Meteor app as is necessary to allow this kind of behavior. Packages really are plug and play, with minimal configuration and integration.

Another advantage to a Meteor app is that a nearly identical code base can be published to mobile platforms such as iOS, and Android. But because this thesis is about the web world, this will not be taken into consideration.

The default database for Meteor, like the MEAN stack, is MongoDB. However, instead of using Mongoose, which is an ExpressJS specific package, Meteor uses SimpleSchema. Figure 18 shows an example of creating a year object in SimpleSchema:

collections.js

```
let Years = new Meteor.Collection('years'); //creates DB entry

let POISchema = new SimpleSchema({ //is embedded in yearschema
 title: {type: String},
 text: {type: [String]},
 additions: {type: [String]},
});

let YearSchema = new SimpleSchema({
 poi: {type: [POISchema]},
 year: {type: String},
 languages: {type: [String]},
});

Years.attachSchema(YearSchema);
```

Table 18

Because all data object definitions go through the “Meteor.collection” method, packages are able to create object types in your database, reducing the amount of boilerplate code one has to write.

In terms of front-end templating of HTML, by default Meteor uses Blaze, though it supports a variety of different front-end frameworks. Table 19 shows an example of a Blaze view.

index.html

```
<template name="year">
  {{#each years}}
    <div class="year">
      <h1 id="{{year}}">{{year}}</h1>
      <div class="language-header">Languages Known:</div>
       {{#each languages}}
         <span class="bolder">{{{this}}}</span>
       {{/each}}
     </div>
  {{/each}}
</template>
```

Table 19

The above example will output each year, and each language associated with each year, that is passed to the template named “year”. A controller that pulls in the data to the “year” template is known as a helper, and the Table 20 shows an example of that.

helper.js

```
if (Meteor.isClient) {
  Template.year.helpers({
  years: () => {
   return Years.find({}, {sort: [['year', 'asc']]});
  },
 });
}
```

Table 20

Using the convention over configuration rule established by Ruby on Rails, this snippet of code knows to hand this data to the “year” template.

To allow a template to actually pull the data from the server, one needs to explicitly publish this data. This is done by a JavaScript file structured like Table 21:

later in collection.js

```
Meteor.publish('years', function(){
  return Years.find();
});
```

Table 21

This means anyone who visits the webpage has access to all years within the database. This also has the added benefit of have real time updates. If an administrator added a new year to the database, that new year would automatically appear on all users’ pages without them even having to hit refresh. This has a performance cost associated with it, because it means Meteor has a websocket connection with every user using the site, but it has the added benefit that all data being shown is the latest data to show. No extra configuration is needed on the part of the developer on this. It is Meteor’s default behavior.

Measurements

Resource Usage

Each of the sites, though identical in appearance, sent slightly different types of files in different proportions in order to generate the end result. Table 22 shows a comparison between the various sites. This was measured using Google Chrome's YSlow extension.

Components	Ruby on Rails (Ruby)	Laravel (PHP)	MEAN stack (JavaScript)	Meteor (JavaScript)
All	3531 KB	3155 KB	3522 KB	8536 KB
HTML	16 KB	14 KB	4.2 KB	7 KB
JavaScript	385 KB	132.8 KB	466.9 KB	5455 KB
CSS	135 KB	148.2 KB	143.2 KB	151 KB
Images	2923 KB	2923 KB	2923 KB	2923 KB
Favicon	72 KB	0 KB	0 KB	0 KB

Table 22

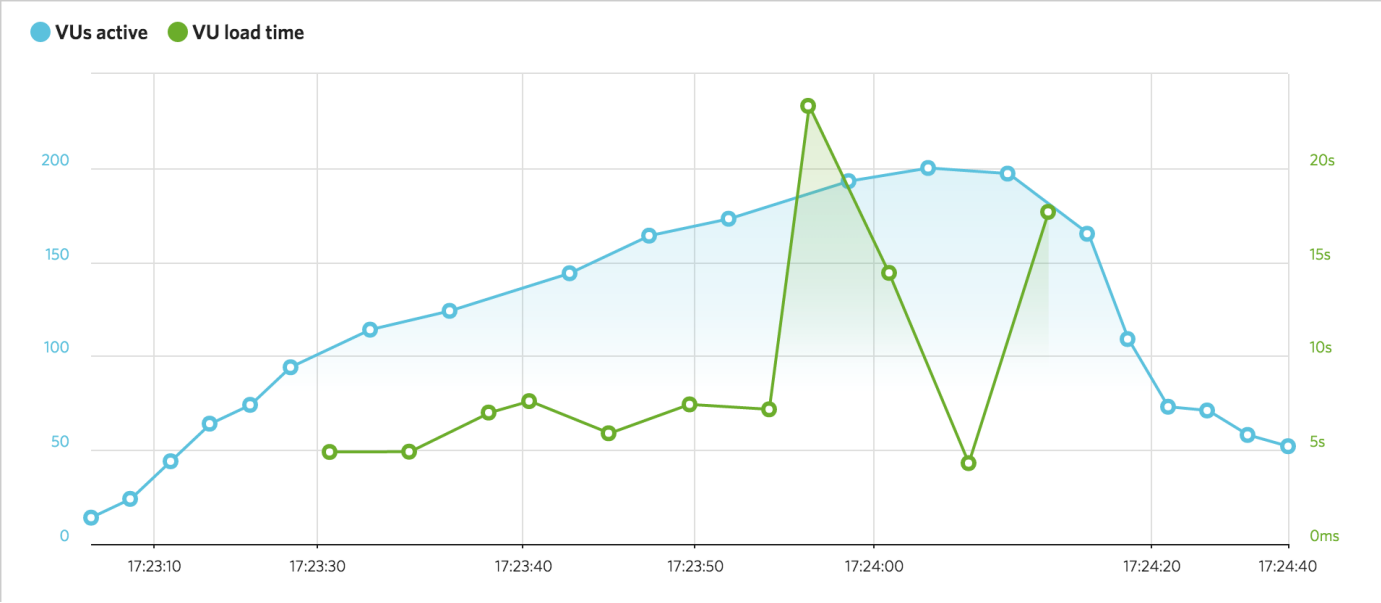
DoS Attack

To give each framework a run for their money, I deployed an identical site built in each corresponding stack, and deployed them on an identical server spec (AWS EC2 t2-micro cloud instance). I then used an online service centralized in Sweden called LoadImpact (https://loadimpact.com/) that sent 200 virtual users to each server within the span of a minute. Given that each of these websites was being dished out of Amazon's smallest single

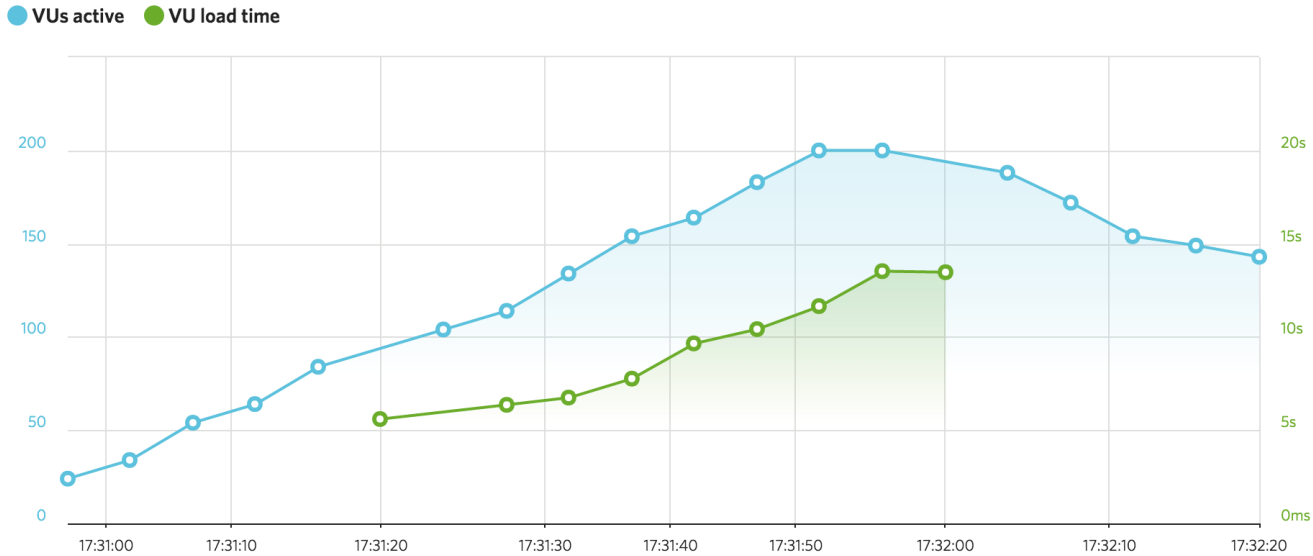
core servers, this was plenty to see how the stack handles itself under pressure. LoadImpact then recorded load times for each virtual user. It then returned the results in a graphical format. On the next two pages are the results of that experiment. The results are unexpected to say the least.

Results

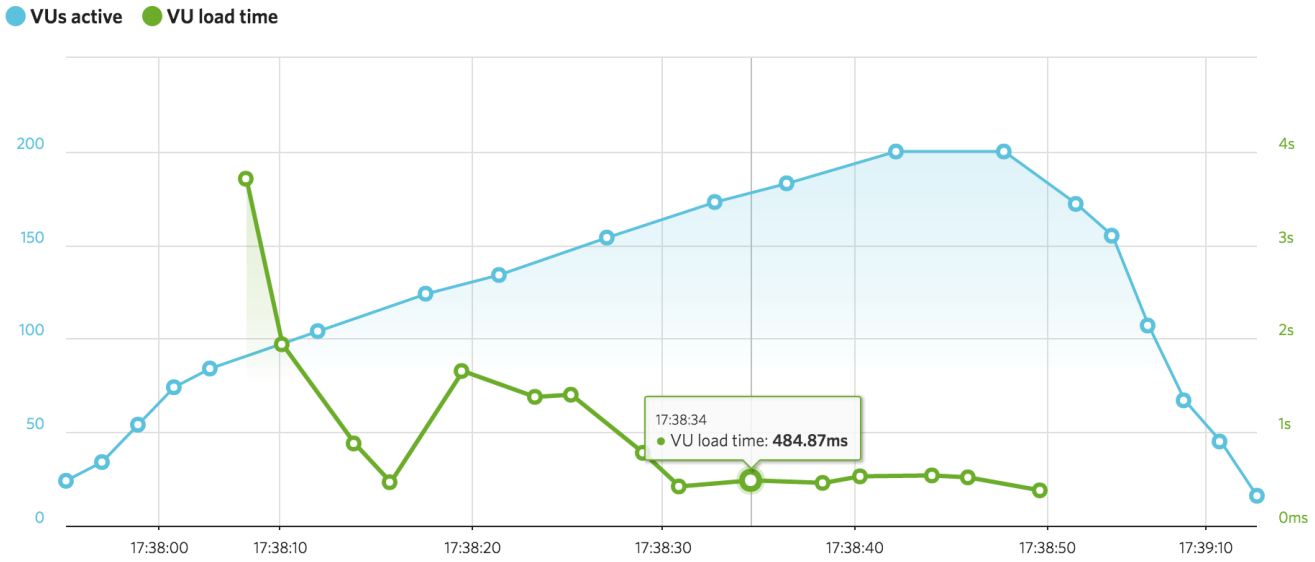
Ruby on Rails (Ruby) - 3273 reqs, 274.39 MB



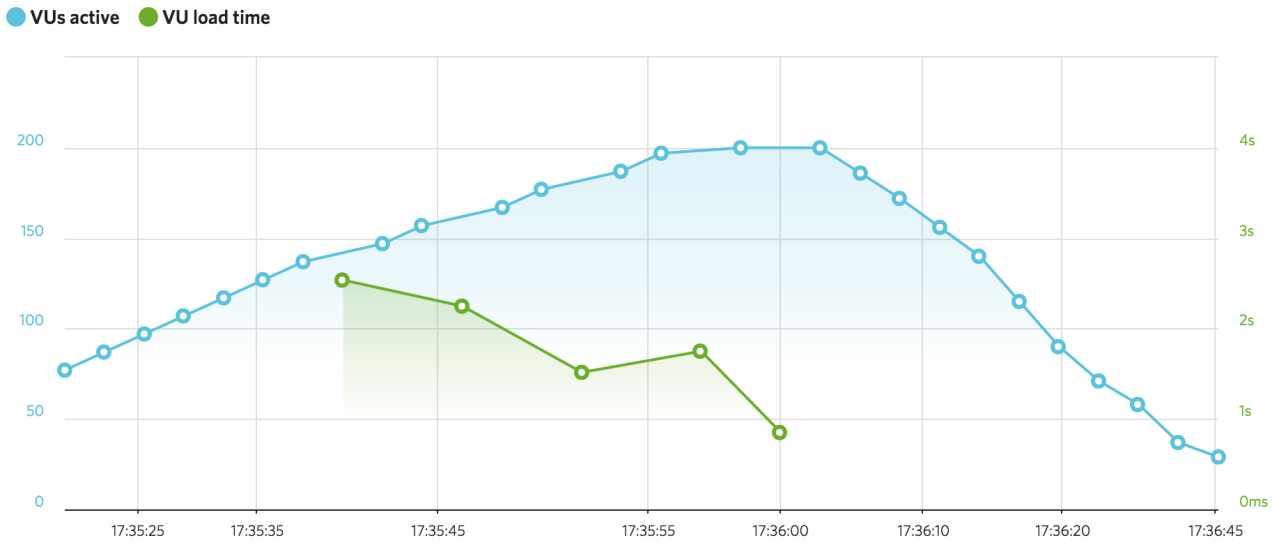
Meteor (JavaScript) - 10019 reqs, 993.38 MB



Laravel (PHP) - 3686 reqs, 939.56 MB



MEAN Stack (JavaScript) - 3210 reqs, 348.83 MB



Laravel by far performed the best of any of the other stacks. PHP is not known for its performance, however there are several explanations for why the results varied to the degree they did. (By more than a factor of 10!)

1. By nature the internet is fickle and yields irregular results. It is very possible AWS simply had a performance hiccup while I executed the previous three tests. Given how a vast amount of the load time should have been images, which in all cases were over fifty percent of the site's memory footprint, a x10 speed is unexpected to say the least given that all sites had identical images.
2. Laravel out-of-the-box does some caching on behalf of the developer, meaning if a similar page is continually requested from users, it saves the page and does not re-render it, but sends it to the new user. All stacks have the capacity to do this, but it is not built in from the start.
3. No optimization was done on the part of any of the stacks. For example, Meteor was never published in production mode, because the AWS server did not support that functionality. As a result the site was not nearly as preformant as it would be otherwise.
4. The small websites are also not indicative of what a real web-app may be like. Most production websites are not single, static page applications.

However, all that aside, with the least amount of effort put towards optimization, based on this single experiment, it appears Laravel wins the prize.

Conclusion

The web is a wild and crazy place for a developer. A technology only invented a few decades ago, the Internet, has spawned one of the most fantastic and diverse realms of enquiry and craftsmanship. In some sense it's all as simple as some HTML, CSS, and JavaScript, but in another sense, it's not about those technologies at all. The languages in use on servers are as numerous as they are diverse. Countless webstacks in circulation each have different fundamental philosophical approaches to the task of web development. They each have certain strengths and goals, and each has within it a nugget to learn from.

Although this thesis gave an overview into four very different webstacks, it has only scratched the surface of this topic. There are several other important technologies such as Google's GWT [15] (a full-stack Java framework), Microsoft's ASP.NET (which is built on C#), and an up and coming webstack called Phoenix [16] (a webstack built on a dialect of Erlang called Elixir - a functional language unlike anything covered in this paper). Clearly there is a wide array of ways to code for the web. In terms of personal exploration and growth, the options are near limitless.

This thesis has given the reader an overview of the types of webstacks that exist today in order to springboard them into further web technology based studies. Though employment leans heavily toward technologies such as JavaScript, ASP.NET, and Java as of 2016, if one looks hard enough, one can find a business doing development in just about any technology. Therefore, if an individual wishes to pursue a career in web development, the

most important question to answer is “which language(s) do I enjoy the most?”. The only way to answer this question is to explore.

Definitions

API: Application Program Interface. How a business exposes their server-side programs to be used by other programs. Example: When a user logs into a Spotify account using a Facebook account, Spotify is utilizing Facebook’s API.

App: Short for application, which is a general term for a full software product. Could be a website or a mobile application based on context.

Asynchronous: While the code waits on input or output, the rest of the program continues to run. This gives a rise in performance.

Client-side (Front-end): Aspects of a website that live in a browser. This includes HTML, CSS, and a certain kind of JavaScript. NodeJS is an example of JavaScript that is not client-side.

Codebase: The full body of source code. The sum of all program related text files making up an application.

Compiled: When source code has been converted to machine code, which are binary instructions that a computer can understand. The result is not human readable and it is the form most programs are distributed in. The result is so gnarly that it makes the program difficult nearly impossible to reverse engineer.

Cookie: A unique digital “finger print” that identifies a user for a server. Under normal circumstances, every request a user sends to a server, has a cookie attached.

CSS: Cascading Style Sheets. Used to decorate a web page’s content.

Data Packet: Small pieces of a message sent over the internet in a sequence in the form of binary information that is decoded when it arrives at its destination.

DoS Attack: Denial of service attack. When a server is intentionally flooded with requests so that it crashes or is unable to respond to genuine users trying to access the content.

Framework: A specialized abstraction of a computer language, usually good at a particular group of problems. Example: AngularJS is a JavaScript framework, created to make front-end JavaScript easier.

HCJS: HTML, CSS, and JavaScript in conjunction with one another.

HTML: Hypertext Markup Language. The language that content is displayed through on a browser.

JavaScript (JS): The Turing complete language that can run in a browser. It can be run on a server in the form of NodeJS.

Open Source: Code that is distributed through one of many potential publishing licenses with the intent of allowing others to build on the codebase. This code is not compiled before hand.

Package: A reusable and modular collection of programs that preform a particular function, that are intended to be used in a larger system. Commonly open source.

PHP: PHP hypertext processor. One of the oldest server-side technologies still in widespread use today.

Piped data: Data that is input into one program, and the subsequent output is used as input for another program, and so on.

Production: When a program is accessible by its intended audience, it is considered in "production".

Rails: Shorthand for the webstack Ruby on Rails.

Server-side (Back-end): Aspects of a website that live on a remote server. This is the aspect of the business that is generally kept secret. This can include nearly any language. This combine with client-side scripts make up a full website.

Server: A computer with a corresponding IP address that is "listening" for user requests, and responds to those requests using server-side programs.

Stack: Short for webstack.

Source code: The text that makes up a program. Ideally it is human-readable.

Web-socket: A way for a browser to stay in constant communication with a web server, taking in streams of flowing data. If data changes in the server, it is immediately pushed down to connected browsers that have a maintained web-socket connection.

Webstack: A group of technologies, typically composed of several languages and frameworks, that work in conjunction to create a website.

Sources

Citations

[1] "A Short History of JavaScript." *Web Education Community Group*. Web. 01 Apr. 2016. <https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript>.

[2] "Lesson 8: Object-Oriented Programming." Oracle.com. Web. 01 Apr. 2016. <http://www.oracle.com/technetwork/java/oo-140949.html>.

[3] "The History of Ruby." *Site Point*. 2014. Web. 01 Apr. 2016. <http://www.sitepoint.com/history-ruby/>.

[4] "Ruby." *Programming Language*. Web. 01 Apr. 2016. <https://www.ruby-lang.org/en/>.

[5] "Imagine What You Could Build If You Learned Ruby on Rails…." *Ruby on Rails*. Web. 01 Apr. 2016. <http://rubyonrails.org/>.

[6] "Ruby." *About*. Web. 01 Apr. 2016. <https://www.ruby-lang.org/en/about/>.

[7] "History of PHP and Related Projects." *PHP.net*. Web. 01 Apr. 2016. <http://php.net/manual/en/history.php>.

[8] "Why Hasn't Facebook Migrated Away from PHP?" - *Quora*. Web. 01 Apr. 2016. <https://www.quora.com/Why-hasn-t-Facebook-migrated-away-from-PHP>.

[9] "PHP: A Fractal of Bad design." *Fuzzy Notepad Atom*. Web. 01 Apr. 2016. <https://eev.ee/blog/2012/04/09/php-a-fractal-of-bad-design/>.

[10] "14 Surprising Statistics About WordPress Usage - ManageWP." *ManageWP*. 2014. Web. 01 Apr. 2016. <https://managewp.com/14-surprising-statistics-about-wordpress-usage>.

[11] Meyer, Eric A. CSS: The Definitive Gui<img src="https://lh3.googleusercontent.com/EWjISouKv8KAdJ58XNJDpcNy_yUk_BFhvyAmTnbyQ4aVcaKkywQ5NICo9JzjnDu6daNg=rw">

[12] "Ajax: A New Approach to Web Applications." *Adaptive Path » Ajax: A New Approach to Web Applications*. Web. 01 Apr. 2016. <https://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php>.

[13] "Node.js." *Node.js*. Web. 01 Apr. 2016. <https://nodejs.org/en/>.

[14] "People." *Meteor Blog RSS*. Web. 01 Apr. 2016. <https://www.meteor.com/people>.

[15] "Productivity for Developers, Performance for Users." *GWT Project*. Web. 01 Apr. 2016. <http://www.gwtproject.org/?csw=1>.

[16] "Phoenix." *Phoenix Framework*. Web. 01 Apr. 2016. <http://www.phoenixframework.org/>.

[17] Rao, Siddhartha. Sams Teach Yourself C in One Hour a Day. Indianapolis, IN: Sams Pub., 2012. Print. de. Beijing: O'Reilly, 2007. Print.

[18] Flanagan, David. JavaScript: The Definitive Guide. Sebastopol, CA: O'Reilly, 2006. Print.

[19] Chodorow, Kristina. MongoDB: The Definitive Guide. Beijing: O'Reilly, 2013. Print.

[20] Lerdorf, Rasmus, Kevin Tatroe, and Peter MacIntyre. Programming PHP. Sebastopol, CA: O'Reilly, 2006. Print.

Code Repository Links

PHP (Laravel): https://github.com/jacksonStone/LaravelPersonal

JavaScript (Meteor): https://github.com/jacksonStone/meteorPersonal

Ruby (Ruby on rails): https://github.com/jacksonStone/rubyPersonal

JavaScript (MEAN stack): https://github.com/jacksonStone/js-stack

Website Links (As of March 2016)

PHP (Laravel): http://ec2-52-87-211-140.compute-1.amazonaws.com/

JavaScript (Meteor): http://54.84.130.254/

Ruby (Ruby on rails): http://54.175.129.165

JavaScript (MEAN stack): http://52.23.200.234/