

Comparison of modern web frameworks

Nikita Klyukin



Author(s) Nikita Klyukin	
Degree programme BITE	
Report/thesis title Comparison of modern web frameworks	Number of pages and appendix pages
<p>Nowadays there are a lot of different technological solutions available for people. You can create a web application using a CMS (Content Management System) that requires almost no programming knowledge while at the same time you can do everything almost from the scratch with ASP.NET or ROR. Sometimes it is hard to decide what technology would benefit your project most and this research will try to solve this problem.</p> <p>There are several web frameworks out there. But some of them are more popular than the rest. Ruby on Rails has been on top for a while now. It's one of the most popular frameworks out there. And for good reasons. Ruby is a simple language to learn. RoR almost works out of the box. All you need is a server and you are good to go.</p> <p>However Node.js based frameworks, such as Express.js, have recently been gaining popularity. Node.js is not a complete out of the box solution. It does require some extra tools to get it rolling such as an MVC framework (Express.js) and a few other depending on the application definition. But it still has its own appeals. It uses JavaScript as its main language for example.</p> <p>This research reviews these frameworks and afterwards compares them according to a comparison model based on a set of criteria defined for this specific research.</p> <p>After doing this research I can conclude that Ruby on Rails is a better choice for developers seeking a fast and easy way to start their application development. It is much more developer-friendly and does not require the developer to be a framework guru to use it to its maximum potential. However Node.js (Express.js and others) proved to be much more flexible and an excellent choice for advanced developers.</p>	
Keywords Web framework, Ruby on Rails, Node.js, JavaScript, Express.js, CMS, development	

Table of contents

1	Introduction	1
1.1	The what.....	Error! Bookmark not defined.
1.2	The how.....	Error! Bookmark not defined.
2	Theoretical framework.....	4
2.1	Relevance of the research and comparison model.....	4
2.2	Application definition	6
2.3	Comparison parameters	6
2.4	Framework 1	8
2.5	Framework 2	9
3	The actual comparison.....	11
3.1	Javascript.....	Error! Bookmark not defined.
3.2	Rails functionality	11
3.3	Community	12
3.4	Scaffolding	13
4	Development.....	13
4.1	Rails	14
4.1.1	The setup	14
4.1.2	Basic functionality implementation	15
4.1.3	Finishing the app	17
4.2	Node.js	21
4.2.1	The Setup	21
4.2.2	Basic functionality implementation	23
4.2.3	Finishing the app	30
5	Conclusion.....	33
5.1	Parameters comparison.....	33
5.2	The summary	37
5.3	The learning outcome	38
6	References	39
7	Appendices	40

1 Introduction

1.1 Objective

Web technologies are advancing every day. Seems that only yesterday you had to call a restaurant to book a table and today all you have to do is open their website or mobile app and press a few buttons. Web applications today are pretty much at the same levels of functionality that only the normal apps used to have.

We all are using a lot of different services that are only available to us thanks to the advancements that were made in the IT industry and the web sector specifically. And it doesn't seem like the technological train is planning to slow down any time soon. Many people see a lot of potential in this field and are joining it. Those who are providing us with web solutions are web developers and they are using different tools to do so.

If you ever would try to become a web developer you will face a rather tricky choice to make. Which technology is best to use for this specific project? And the most frustrating problem is the fact that the speed at which technology is evolving makes it almost impossible to find an absolute solution to any given problem.

Many companies when in need of a web/mobile application face a huge variety of different ways to accomplish that task. There are countless options out there. You can select to outsource that kind of a task to an IT company that will do the project for you. Or you could assemble your own team of developers and have it done your way under your own supervision. In any case you would end up looking for the best way to accomplish the task in hand.

This research focuses on providing a look at the surface of web development. The main objective is to provide a reasonable advice on selecting a web development framework based on a developer's needs. This research reviews these frameworks and afterwards compares them according to a comparison model based on a set of criteria defined for this specific research.

1.2 Methodology

So in order to understand this research we first need to establish the boundaries of it. Since there are numerous frameworks out there you cannot define the one single best one. Depending on your project you will always be able to find pros and cons of the frameworks that are only valid for that specific task. And there is no guarantee that the tool that you used for project A will be as good for project B.

So first of all there should be an application definition. Two applications that are similar in terms of functionality but are done using different tools. This way we will have a fair ground

for comparison of the frameworks and the results will provide the data required to make an educated decision when selecting one of the frameworks.

In order to select 2 frameworks first thing to do would be to create a number of requirements for those frameworks. First of all they should be well known and widely used, since this kind of tools should have a reasonable amount of proper documentation and literature for them. Secondly they should be capable of doing similar things, because otherwise the comparison would be rather pointless. And finally it would help a lot if both of them would be already in some sort of a rivalry.

By doing some research I found what web technologies the big companies are using for their web needs. I found out that companies like Shopify, Indiegogo and Groupon are using Ruby on Rails while companies like Google, PayPal, Walmart and LinkedIn use node.js for their web application needs. I also came across a few statistical resources that provide rating for frameworks based on their popularity. One of the most reliable sources for such data is repository hosting web service. GitHub is considered by most developers to be the go-to repository provider. According to <http://github.info> by 4th quarter of 2014 such languages as Ruby and JavaScript made it to top 6. If you look back to 2013 these 2 languages were the second and the first languages in this rating accordingly. Of course these languages, especially JavaScript, are used a lot for other purposes as well. So I had to double check if frameworks based on them are in fact still popular. According to <http://hotframeworks.com/#rankings> Ruby on Rails sits on the 2nd place while Express.js sits on 7th position. Since both of them are positioned at top 10 spots and are both popular and able frameworks I consider them the perfect choice for this comparison.

Ruby on Rails, a web application development framework written in the Ruby language, has been a really popular framework for years. As I mentioned above, big companies use it and it serves their needs perfectly.

Express.JS, a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. Node.js is getting more and more attention as the time goes on. Many developers enjoy using it since you have your entire backend done in JS and as long as you know it you will learn to use it really fast.

It is important to note that Node.js even with the help of Express.js is nowhere near in terms of features to where Ruby on Rails is. According to the frameworks website “Express.js is a minimalistic node.js framework that provides MVC capabilities to the node.js environment” (expressjs.com). When you look at capabilities of Ruby on Rails however you get a whole list of features. According to the official Ruby on Rails website “Ruby on Rails is an opinionated Framework that makes you adhere to its way of doing things out-of-the-box. From the routing system to ActiveRecord, Rails assumes you build your apps in a particular way.” (rubyonrails.org)

So in order to be clear when this document will be talking about node.js it will also include a number of other tools such as express.js into consideration.

All of the information for these frameworks will be gathered from sources such as libraries and online forums that discuss these topics.

My personal favorite is stackoverflow.com. It is a question and answer site for professional and enthusiast programmers. It contains many discussions about different programming problems. And the topic of comparing multiple web frameworks is not something that nobody else discusses. There are multiple questions related to these problems. Even the comparison of RoR and Node.js is not something new to stack overflow forums. However you always must remember that online forums do not guarantee you 100% accurate facts. Literally anyone can log in and type something completely weird. I am very critical to the points made by other people on the Internet. I always look for multiple discussions related to the problem in question and compare them. Looking for discussions where different people state the same thing is a very good way to gain a larger pool of information. But no matter what information I get from online resources I always remember the boundaries and restrictions that apply to that kind of sources. Whenever you take a statement from a forum you always need to find arguments that support that opinion.

In order to decide whether a specific discussion is trustworthy you need to check a number of things. First of all, if we are talking about stack overflow, we need to check the person's reputation and validity of his other contributions. Secondly we must take the information that he provides and apply it to our own experience. This way you can ensure that the knowledge obtained from stack overflow is worthwhile.

There are numerous discussions that could be found on stackoverflow discussing this specific thing – what should a developer use Rails or Express? And all of them are giving rather insightful details on both frameworks.

According to “Clarifications about Rails and Node.js”, a discussion started by a senior member of stackoverflow it all depends on your application's needs.

“There aren't enough differences between Node.js and Rails for it to practically matter. A lot of what Node.js can do can be pulled off in Rails with things like EventMachine and Pusher. So unless you are really familiar with Rails' limitations, and know you'll be pushing the boundaries, you'd be hard pressed to make something a seasoned Rails developer couldn't do.” (Clarifications about Rails and node.js, stackoverflow.com)

Basically this person proposes that even though node.js does provide a lot of flexibility it might not really matter unless you know that Rails will not be enough for you application.

Another discussion, “Node.js/Express.js or Ruby on Rails for an ABSOLUTE beginner [closed]” has a similar discussion going on. They are saying that Express.js is a framework that focuses on a minimal set of features while Rails tries to be the all-together framework. However when it comes down to Node.js in general people in this discussion

agree that Rails is the easier way to go at the moment since it currently has a very large and established community behind it.

“When you asked about Javascript and RoR in your previous question, you effectively asked about Node.js and RoR.

If you are building a commercial/enterprise level application, stick with RoR. There are way more resources, bigger communities, relatively stable releases, and you can easily find good developers with RoR experience.

Express.js is like Sinatra for Ruby. Neither is as advanced as Rails, preferring to be more 'basic' and providing a minimal feature set rather than try to be an 'everything at once' framework that Rails is.” (Node.js/Express.js or Ruby on Rails for an ABSOLUTE beginner [closed], stackoverflow.com)

There are, of course, more discussions out there, but mostly they all are focusing on RoR's ease of use and on Node.js/Express.js's flexibility.

Now that I have acquired enough opinions on the topic the next step is to put both frameworks to the test. After all the information is gathered and the theory is there it's time to begin developing the solutions.

2 Theoretical framework

In order to compare 2 frameworks I will need to define a comparison model. The article from ACM digital library “A comparison model for agile web frameworks” is a perfect example of a way to compare web frameworks and it will be used as a backbone for developing my own comparison model.

2.1 Relevance of the research and the comparison model

Today there is no such discipline as web framework comparison. There are some similar ones, but most of them are rather young still. It is a big problem for many companies and individual developers today. How do you choose which framework to use or learn? Why would you choose one over another? Many researches have attempted to answer these questions. But most of them agree that it is essential to find a proper way to compare web framework. Here is an opinion from an article, published a couple years ago:

“Nowadays, there is a wide range of available frameworks and an evaluation is required to determine which is the most suitable for a particular application. Using an inappropriate framework for a system leads to increase development cost and time and reduces software quality.” (A comparison model for Agile Web Frameworks, 2008)

But first we must answer what is a web application framework (WAF)? According to “A comparison model for Agile Web Frameworks”:

“Web application frameworks (WAF) are defined as a set of classes that make up a reusable design for an application or, more commonly, one tier of one application. This specialization in tiers (persistence, web flow, ...) has led to its classification due to its proliferation. Agile web frameworks are defined as full stack web frameworks for developing an application. In contrast with web application frameworks, they are not specialized in one layer, but offer the full stack.” (A comparison model for Agile Web Frameworks, 2008)

There are multiple frameworks available and they all could be suited to do one task or another. But since it is rather hard to find a decent comparison of 2 given frameworks, especially when it comes down to details many web development projects end up being not as efficient as they could have been. As a result of such projects people lose money and time. And sometimes the entire project fails and they have to go back the beginning. A comparison model for Agile Web Frameworks article provides great insight on why web framework comparison is indeed a relevant thing today. It also gives a really well organized comparison model that is based on a set of aspects that are then used to evaluate each framework.

” The goal of the comparison model is facilitating the evaluation of agile web frameworks. This is achieved by a fine-grained characterization of agile web frameworks through an evaluation criteria that is based on the definition of a set of parameters given the common views of the frameworks. Parameters are grouped into a set of aspects shown on figure 1, which summarize general features and issues found in web application development. The aspects are domain description and persistence, presentation, security, usability, testing, service orientation, component orientation and adoption...” (A comparison model for Agile Web Frameworks, 2008)

Here is the list of categories taken out of the context:

- Domain Description and Persistence
- Component orientation
- Presentation
- Service orientation
- Usability
- Security
- Adoption
- Testing

Each aspect has a number of parameters. These parameters are later on used to compare each framework. For example for domain description these parameters are:

“Domain description parameters:

- Are data migrations built-in in the development process?
- Is schema database automatically inferred from domain definition?
- Is domain definition automatically inferred from schema data?
- Does it support validation?
- Does it support transactions?” (A comparison model for Agile Web Frameworks, 2008)

This seems like a perfectly sound way to compare frameworks. Since there is no standard for web framework comparison this method might not be ideal. Or maybe it is ideal. I am not qualified to answer this question, but since the authors of the article made that method work I see no reason why I cannot build my comparison based on it. And this is exactly how I am going to compare Ruby on Rails with Express.js (among a number of other tools).

2.2 Application definition

Selecting application to develop is not as simple as it might seem. This application must be capable of demonstrating some basic capabilities of the 2 frameworks. On the other hand it should be a simple application that is both easy to develop in a short period of time and it should only demonstrate the basic things. A blogging application seems to fit this definition pretty well. It is a common type of an application. If you google for “blogging platform” you will get more than 3 million results. Blog’s functionality is pretty straightforward. It is an application that fits the basic Create, Use, Update and Delete (CRUD) model requirements. However it is still possible to demonstrate some advanced features of the framework in hand while doing a simple application such as a blog.

The application will be capable of simplest operations – posting, editing and deleting the blog posts as well as creating and modifying comments. There will be no security, authorization or authentication. These applications are going to be a basic demonstration of their respective technologies.

2.3 Comparison parameters

So should I use Ruby on Rails or Node.js (+ express.js, mongodb, etc.) for my development needs? I am going to answer this question by taking (and adopting) criteria from the ACM article and then developing a simple application to demonstrate the technologies. It is important to mention that this specific application is just a tiny demonstration of the

frameworks' capabilities so it will not be possible to draw extensive conclusions on the frameworks in general. The comparison will only touch the surface of them and hopefully will provide reasonable starting point for a complete beginner at web development. This is a general comparison and not a full head to head of the 2 frameworks.

As I mentioned in chapter 2.1 the structure for comparison will be similar to the one ACM article uses. The comparison will be based on 2 layers. Firstly there will be a couple of categories of comparison. Each of those aspects will contain a number of parameters that will be the key to comparing the frameworks.

In order to obtain the categories I will first need to look at what other resources point out. According to the ACM article mentioned above there are numerous important categories to consider. However since my comparison is not focusing on such details it would not be a great idea to just take the article's model and use it here without modification. Some stackoverflow discussions provide reasonable basis for categories of comparison. For example in discussion named "Main pros and cons of Ruby on Rails and Express.js" there is a number of different topics mentioned. Many people compare things like community support, presentation layer, project kickoff time and the amount of time to complete the application versus the flexibility it provides. (Main pros and cons of Ruby on Rails and Express.js, stackoverflow.com)

After analyzing this discussion I have used the ACM article model to create the list of categories together with their respective parameters:

- Installing the framework to your local machine
 - How many things are required to be installed?
 - How fast can all of them be found and installed?
- Scaffolding
 - Is scaffolding supported?
 - How fast can you get a "hello world" application running?
 - How is the out-of-the-box UI?
- The application
 - Amount of time used to get to know the framework
 - Amount of time used to develop the app
 - Amount of code that has been written
 - Overall functionality
- Features head to head
 - Both applications have similar features implemented?
 - What features are missing (if any)?
 - Was it equally easy to implement similar features?
- Community

- Amount of time required to obtain required setup information
- Amount of guides available for each tool
- Validity of the information that is provided by these resources
- Is community actively developing its framework?
- Availability of different 3rd party tools

2.4 Framework 1

First framework that I will be using is Ruby on Rails. It is considered to be one of the most complete open-source frameworks out there. It is capable of doing a lot of good things and it does them well. It has a predefined way of building the applications and it does things right out-of-the-box. Here is a list of some features included out of the box (www.rubyonrails.org):

- Supports most of databases, plug and play
- Good engines for MVC
- Test Driven Development friendly
- Great documentation

Here is the basic definition taken from the official website of the framework:

“Rails is a web application development framework written in the Ruby language. It is designed to make programming web applications easier by making assumptions about what every developer needs to get started. It allows you to write less code while accomplishing more than many other languages and frameworks.”(www.rubyonrails.org accessed on 14/2/2015)

Let's dig a bit deeper into what exactly is RoR.

“What is Ruby?”

Before we ride on Rails, let's know a little bit about Ruby, which is the base of Rails.

Ruby is the successful combination of:

- Smalltalk's conceptual elegance,
- Python's ease of use and learning, and
- Perl's pragmatism

Ruby is

- A High Level Programming Language
- Interpreted like Perl, Python, Tcl/TK.
- Object-Oriented Like Smalltalk, Eiffel, Ada, Java.
- Originated in Japan and Rapidly Gaining Mindshare in US and Europe.

Why Ruby?

Ruby is becoming popular exponentially in Japan and now in US and Europe as well. Following are greatest factors:

- Easy to learn
- Open source (very liberal license)
- Rich libraries
- Very easy to extend
- Truly Object-Oriented
- Less Coding with fewer bugs
- Helpful community”

(<http://www.tutorialspoint.com/ruby-on-rails/rails-introduction.htm> Ruby on Rails introduction, accessed on 14/2/2015)

2.5 Framework 2

So we looked RoR now it's time to look at node.js.

A common misconception is that Node.js is a framework by itself. Sometimes you can even find evidence of people calling it a language. Both of those opinions are rather popular but equally wrong. Node.js is an application runtime environment. It allows you to write server side code in JavaScript. Considering that node.js cannot really compete with RoR unless you give it a few friends to work with it would be better to talk about comparing a number of modules together with node.js vs. RoR. As a matter of fact the comparison of RoR vs. Node.js is pointless unless you add a module or two to node.js since they are not exactly the same thing.

There is a great blog entry from habrahabr.ru about Node.js and Ruby on Rails. According to it:

“There are a number of really cool things about node.js. Two of them make node an awesome platform for web. Firstly it's the engine. V8 is a very fast. Approximately 8 times faster than Python. Another thing is the fact that node.js is async-driven. All the requests can be performed together. A single server can handle a lot more than most other servers.

Here is a list of benefits that node.js provide us with:

- Its web server is capable of handling big numbers of connections right out of the box
- It is gaining popularity and a lot of libraries from other languages are being ported to node.js
- Is great for real-time web applications” (Habrahabr/Sca, Node.js vs Ruby on Rails, habrahabr.ru)

Here is the basic definition that you can find on wiki:

“Node.js is an open source, cross-platform runtime environment for server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, Linux, FreeBSD, and others.

Node.js provides an event-driven architecture and a non-blocking I/O API that optimizes an application's throughput and scalability. These technologies are commonly used for real-time web applications.

Node.js uses the Google V8 JavaScript engine to execute code, and a large percentage of the basic modules are written in JavaScript. Node.js contains a built-in library to allow applications to act as a Web server without software such as Apache HTTP Server or IIS.” (Wikipedia, accessed on 14/2/2015)

Before we dive deeper into Node.js we need to understand something. Node.js in fact is NOT a framework (as was defined before).

First of all Node.js is a web server with the low-level routing and administration capabilities, written in JavaScript in order to asynchrony and high speed. And in order for this comparison to be fair node.js will need some friends. Express.js and MongoDB could be one option. So let me explain what those things are as well.

Express.js is, according to its official web page:

“Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.”

(www.expressjs.com Express, accessed on 14/2/2015)

Neither node.js nor express.js come with a database so we will need to pick one for it.

“MongoDB (from humongous) is a cross-platform document-oriented database. Classified as a NoSQL database, MongoDB eschews the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas (MongoDB calls the format BSON), making the integration of data in certain types of applications easier and faster. Released under a combination of the GNU Affero General Public License and the Apache License, MongoDB is free and open-source software.”(www.mongodb.org, About MongoDB, accessed on 14/2/2015)

3 The actual comparison

3.1 Environments

Let's take a look at the actual development environments for web-applications. Node.js attracts with its asynchronous front and back end JavaScript shell while with Rails a developer is required to be dealing with SQL, Ruby AND JavaScript. But after examining both languages I came to a conclusion that even though it seems that the fact of single language usage is a huge advantage it actually is one of the most overrated advantages. Ruby is not that complicated to learn. All you really need is to read a bunch of articles and look at a bunch of examples. According to the habrhabr blog post: "With Ruby there is no "wacky" stuff with callbacks that you so often see in JavaScript because of its non-blocking architecture (which actually is considered a huge plus under conditions of high concurrency. Also sometimes it is really good to have predefined syntax for namespaces, classes, and modules." (Habrahabr/Sca, Node.js vs Ruby on Rails, habrahabr.ru) Basically it is up to the developer's preference.

3.2 Rails functionality

Node.js + Express.js, according to expressjs.com, provide routing together with controllers, views and helpers. However it is nowhere close to what Rails provides.

Rails come right out-of-the-box with integrated ecosystem, completely functional MVC (Model-view-controller (MVC) is a software architectural pattern for implementing user interfaces. (Model-View-Controller, Wikipedia)) and a number of useful features that are rather straight forward in their use (info page, rubonrails.org).

Here is an example lineup that I found in the habrahabr blog post:

"Here are a couple examples of functions: `url_for` and `image_tag`. `url_for` is an integrated function that always gives back url of its data object (products for example) and `image_tag` always takes the correct url of an image from the asset pipeline." (Habrahabr/Sca, Node.js vs Ruby on Rails, habrahabr.ru)

According to official [express.js](http://expressjs.com) about page express is a minimalistic mvc framework that focuses on basic functionality. But of course all of it is possible in Express.js to do everything that was mentioned above. However it will require a whole lot of work with modules and helpers. According to stackoverflow discussion "Best modules for Node.js for beginners" the main examples would be yeoman.io, mongoose, monGOD and [Jade](http://jade).

Also another thing that Rails has in its pocket is the fact that you do not need to worry about data models. Yes, in Node.js such thing as [Mongoose](http://mongoose) can handle data but even with its help it is pretty hard to find the same levels of control over data that RoR's data

handler provides says the author of Node.js vs Ruby on Rails from habrahabr.ru. And it also is available from the application level, module level, and controller or view level. However it is worth noting that all of these features have drawbacks. “Very often a developer will find himself just lucky to guess the correct way of doing one thing or another and quite often it is not the most efficient way. Also when something goes wrong and brakes down at times it is really hard to identify the reason for the problem because of special names for variables in rails.” (Habrahabr/Sca, Node.js vs Ruby on Rails, habrahabr.ru) With Node.js you only get what you wrote yourself with minimal distortions.

3.3 Community

Node.js had a great start with NPM (<https://www.npmjs.com/>) but even the most popular packages have bugs and the amount of concurrency is sky high. Some module will simply refuse to work with others for some reason nobody can explain. In the end any developer will find himself in an attempt to glue a broken vase with superglue because you never can be sure whether the package A will work properly with package B already installed. My personal experience with NPM modules however turned out to be really good. I have used multiple npm solutions and all of them worked just fine. Of course my use of them was not extensive and even though, to my knowledge, no problems in terms of one package not working with another one never happened I still have to take other, more experienced users’ opinions into consideration and many of stackoverflow users say that it is a common issue to come across modules that are not working with each other.

On the side of Rails however is one of the largest repositories of free libraries and tools. Also a worthy mention – The Ruby Toolbox (<https://www.ruby-toolbox.com/>) that provides information on relative popularity of tools separating them based on functionality. I have personally used it and I found it really helpful. The insight on most popular tools helps to make a more educated decision when selecting one for myself.

Many modules that come out-of-the-box with Rails you will have to get by hand in Node.js. Things like connection with database, profiling, caching, migration, data models, pipelines, routing, console commands, testing, etc. Even the favicon is in place from the start. Of course all it is trivial but it’s all required for proper work and sometimes it’s just getting old to look at all those messages about conflicting modules in Node.js that for some reason just do not work with each other.

Above all there is a huge amount of guides, books and examples on Rails. Way more than there are available for Node.js. Just take a look at Railscasts (<http://railscasts.com/>) with its 300+ videos. But it’s also a good idea to keep in mind that Rails is changing pretty fast and many books have already lost their actuality and some tasks have received their solu-

tions after the publication of those books and guides. As a result it became a good practice to look up information that is at least 1 year or less old (using “< 1 year old” selector in google for example). The best place to start searching for an advice is Rails Guides (<http://guides.rubyonrails.org/>).

Another way to compare the amount of data available for these technologies is to look them up on google. A search of “node.js guide” provides about 1 410 000 results. A search of “express.js guide” returns 32 500 000 results. A search of “ruby on rails guide” however provides only 1 090 000 results. Even though ruby on rails has been around for much longer the amount of work the community puts into node.js and its frameworks is just incredible. With this much community support Node.js is becoming better every day.

3.4 Scaffolding

Now let's assume that we do have equally equipped frameworks, databases and everything is great. How quick can a person get the application running?

According to my own experience Node.js requires way more time to write and combine everything (once again because of 3rd party plugins and absence of strict structure). In Rails in order to get a fully loaded RESTful interface all you need is a single command and you will have the correct controller, view and model set up and waiting for you in their proper places.

Most of the time this kind of a thing take about 1 minute in Rails and about 1 hour in Node.js. So in order to make this comparison even on the scaffolding front another tool that Node.js will require is a scaffolding plugin. One of the best web's scaffolding tools available today is yeoman.io (<http://yeoman.io/>). It provides an ability to scaffold complete projects or useful parts. With this tool Node.js will be able to use a small set of commands to generate a fully functional scaffolding for a blog.

4 Development

Here comes the next part – the development phase. First of all – what exactly is the application that will be developed? It is a simple blogging platform. Everything in this phase will be done using Windows 8.1 pro. The code editor of my choice is Sublime text 3 with a few add-ons. All the development phases are going to be from the scratch, including the basic installation of Rails and Node.js. Let's begin.

4.1 Rails

4.1.1 The setup

First of all I need a working version of Rails. In order to get one the easiest way is to use Railsinstaller (<http://www.railsinstallfest.org>). All you need to do is download an exe file and press install. In a matter of seconds the installation is complete and you end up with a fully functional Ruby on Rails.

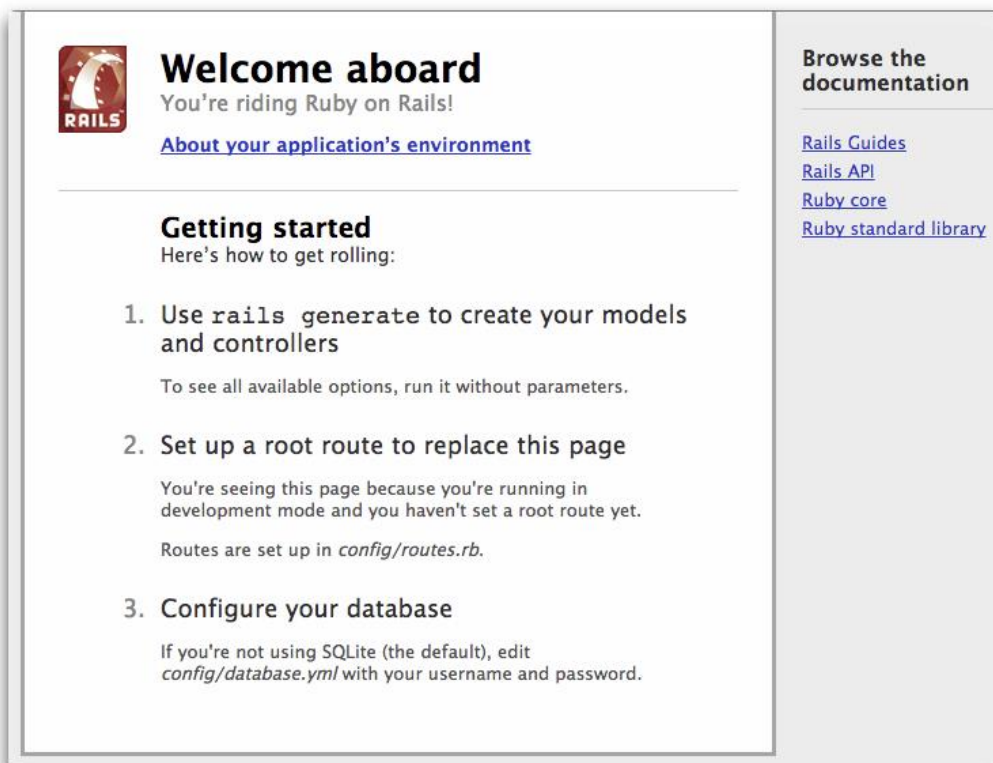
Now that Rails is up and running on my machine the next step is to create a project. In command prompt with a single line you can generate a new application where you can start installing all the required dependencies:


```
rails new rails_blog
```

After a few moments the basic application is up and running for us. Just to check that everything is working I am using a simple chain of commands to first get into the folder with the application and then run the application:

```
cd rails_blog  
rails server
```

Now when I open my browser at <http://localhost:3000> I get this:



 **Welcome aboard**
You're riding Ruby on Rails!

[About your application's environment](#)

Getting started
Here's how to get rolling:

- 1. Use rails generate to create your models and controllers**
To see all available options, run it without parameters.
- 2. Set up a root route to replace this page**
You're seeing this page because you're running in development mode and you haven't set a root route yet.
Routes are set up in `config/routes.rb`.
- 3. Configure your database**
If you're not using SQLite (the default), edit `config/database.yml` with your username and password.

Browse the documentation

- [Rails Guides](#)
- [Rails API](#)
- [Ruby core](#)
- [Ruby standard library](#)

This means that everything has been installed correctly and this is a simple Rails application running on my local machine.

4.1.2 Basic functionality implementation

```
rails generate scaffold Post title:string body:text
```

This command does a few things at once. It asks rails to generate a scaffolding for a resource that I want to call Post with 2 attributes: a title that is a string and a body element that is a text.

```
invoke active_record
create db/migrate/20130422001725_create_posts.rb
create app/models/post.rb
invoke resource_route
route resources :posts
invoke scaffold_controller
create app/controllers/posts_controller.rb
invoke erb
create app/views/posts
create app/views/posts/index.html.erb
create app/views/posts/edit.html.erb
create app/views/posts/show.html.erb
create app/views/posts/new.html.erb
create app/views/posts/_form.html.erb
invoke helper
create app/helpers/posts_helper.rb
invoke assets
invoke coffee
create app/assets/javascripts/posts.js.coffee
invoke scss
create app/assets/stylesheets/posts.css.scss
invoke scss
create app/assets/stylesheets/scaffolds.css.scss
```

This is rails generating the scaffolding. An important file was generated among many others – a migration file that should be located at “db/migrate/1234567890_create_posts.rb”. The numbers depend on the date the file was generated. It contains ruby code that rails will be using in order to manage how the data is stored in the database:

```
class CreatePosts < ActiveRecord::Migration
  def change
    create_table :posts do |t|
      t.string :title
      t.text :body

      t.timestamps
    end
  end
end
```

This code creates a table Posts and this table has 2 columns, a title column and a body column. Now this must be applied to the database:

```
rake db:migrate
```

Once this command is completed the rails server has a new page at <http://localhost:3000/posts> :

Listing posts

Title Body

[New Post](#)

Without much work it is already possible to create, edit and delete blog posts:

New post

Title

Body

[Back](#)

4.1.3 Finishing the app

Validation implementation

Next step is adding some basic functionality like a validator. In order to enforce the user to enter a title the presence validation method must be used. This method must be added to the post.rb file that is located at “app/models/post.rb”:

```
class Post < ActiveRecord::Base
  validates presence of :body, :title
end
```

If you attempt to edit out the title this will happen:

Editing post

1 error prohibited this post from being saved:

- Title can't be blank

Title

Some improvements to the UI

The next step is to make these things look a bit prettier. Firstly – show post page. It is located at “app/views/posts/show.html.erb”. After opening this file in sublime text and a bit of tinkering I ended up with this:

```
<p id="notice"><%= notice %></p>

<h2><%= link_to_unless_current @post.title, @post %></h2>
<%= simple_format @post.body %>

<%= link_to 'Edit', edit_post_path(@post) %> |
<%= link_to 'Back', posts_path %>
```

Of course this is not best design ever but it is still a small but noticeable improvement to the basic view.

Next thing is to make the blog listing look better than it does at the moment. In order to do that it is a common thing to use a Rails partial (“partial” is a reusable part of HTML code). In order to make all the listings and the single blog pages look similar first thing to do is to create a file at “app/views/posts/_post.html.erb”. The underscore in front of the filename lets Rails know that this file is a partial. I take this code:

```
<h2><%= link_to_unless_current @post.title, @post %></h2>
<%= simple_format @post.body %>
```

that was in the show.html.erb file and put it into the _post.html.erb file. Now all the currently present “@post” must be changed to just “post” instead. Now _post looks like this:

```
<h2><%= link_to_unless_current post.title, post %></h2>
<%= simple_format post.body %>
```

Now the partial must be put into the show view. So now the show.html.erb needs this line added:

```
<%= render partial: @post %>
```

Making the entire thing look like this:

```
<p id="notice"><%= notice %></p>

<%= render partial: @post %>

<%= link_to 'Edit', edit_post_path(@post) %> |
<%= link_to 'Back', posts_path %>
```

Doing just that won’t actually change the index page look so some changes must be done to index.html.erb file. The table that was added there earlier should be removed and replaced by the partial so it would be used instead making the file look like this:

```
<h1>Listing posts</h1>

<%= render partial: @posts %>

<%= link_to 'New Post', new_post_path %>
```

Database model and routing for comments

In order for a blog to be complete it should have a comments section. The backend should be created first. A command passed to the command prompt:

```
rails generate resource Comment post:references body:text
```

This will generate the resource.

```
rake db:migrate
```

And this will update the database.

Now it is time to let Rails know that the Posts will potentially have many Comments. The class `post.rb` located at “`app/models/post.rb`” should be modified. A simple line “`has_many :comments`” inside the class will provide that functionality:

```
class Post < ActiveRecord::Base
  attr_accessible :body, :title
  has_many :comments
  validates_presence_of :body, :title
end
```

Now in order to complete the backend for the comments the configuration of their url must be done. All the configurations are saved in a file “`config/routes.rb`”.

An update to this file is in order:

```
QuickBlog::Application.routes.draw do
  resources :posts do
    resources :comments, :only => [:create]
  end

  # root :to => 'welcome#index'
end
```

Now the urls are in order an action must be created, that will create the comments. The file to modify is located at “`app/controllers/comments_controller.rb`” and should be modified to look like this:

```
class CommentsController < ApplicationController
  def create
    @post = Post.find(params[:post_id])
    @comment = @post.comments.create!(params[:comment])
    redirect_to @post
  end
end
```

Now the backed for the comments is done the final thing to do is to put the comments into the HTML view.

In order to display any comments that have been submitted for a post and allow users to submit them file “app/views/posts/show.html.erb” must be modified:

```
<p id="notice"><%= notice %></p>
<%= render :partial => @post %>
<%= link_to 'Edit', edit_post_path(@post) %> |
<%= link_to 'Back', posts_path %>
<h2>Comments</h2>
<div id="comments">
  <%= render :partial => @post.comments %>
</div>
<%= form_for [@post, Comment.new] do |f| %>
  <p>
    <%= f.label :body, "New comment" %><br/>
    <%= f.text_area :body %>
  </p>
  <p><%= f.submit "Add comment" %></p>
<% end %>
```

Also another file must be created. It will be called _comment.html.erb at “app/views/comments/_comment.html.erb” with this content in it:

```
<%= div_for comment do %>
  <p>
    <strong>
      Posted <%= time_ago_in_words(comment.created_at) %> ago
    </strong>
    <br/>
    <%= comment.body %>
  </p>
<% end %>
```

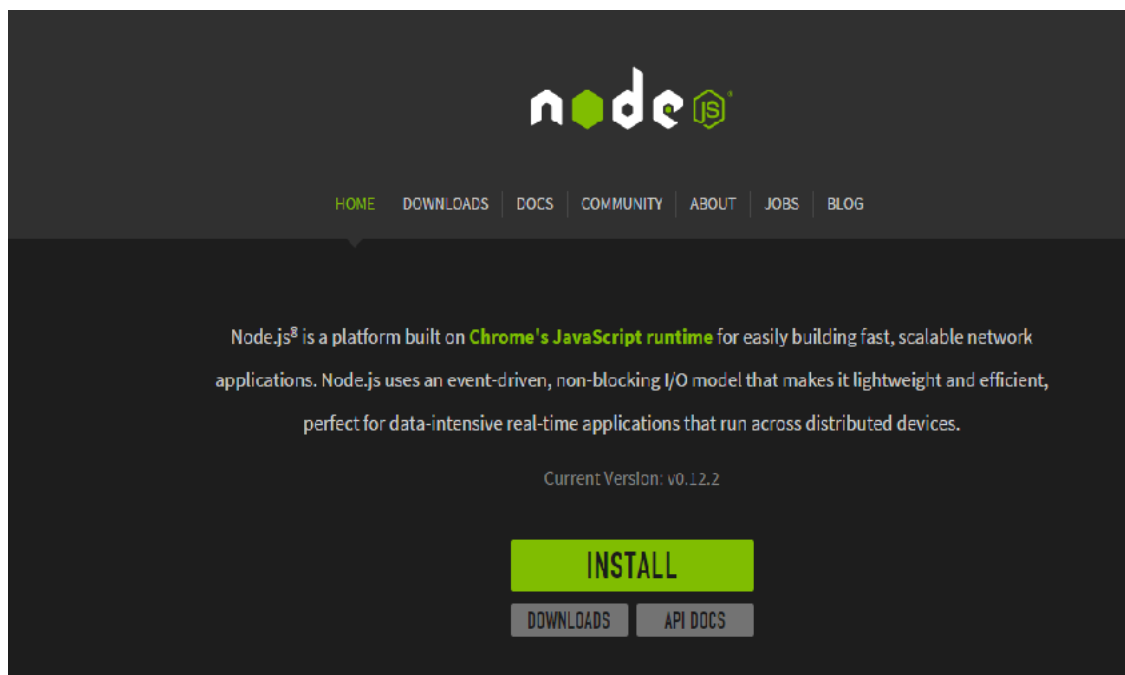
And now the comments are working.

There you have it. A simple blogging application with just the basic functionality done with Ruby on Rails.

4.2 Node.js

4.2.1 The Setup

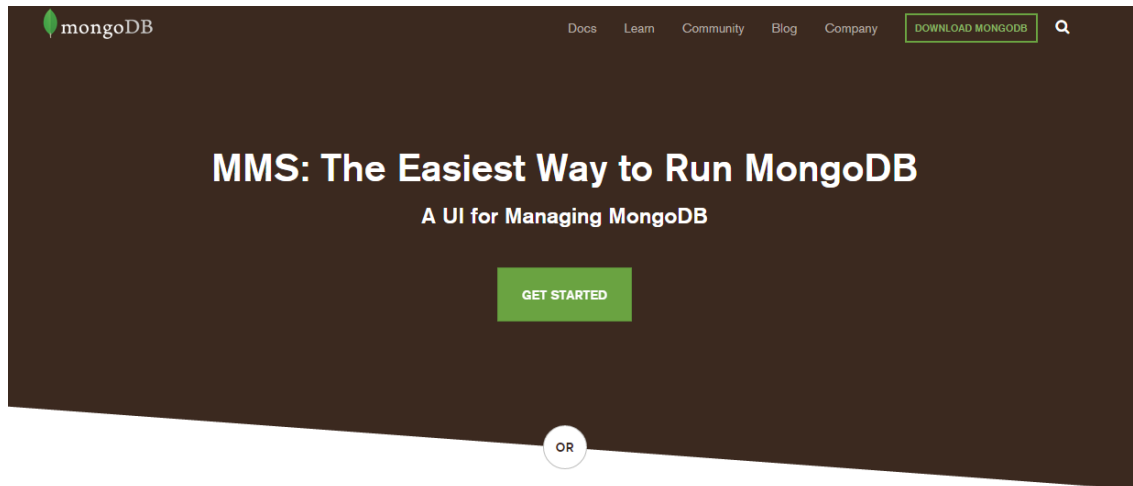
Just like in Rail the first thing to do is to install the pre-requisites. In case with node there is a bit more to it though. First of all the node.js itself. This one is pretty simple. At the official website of node.js (<https://nodejs.org/>) there is a download link (see a screenshot below). All that is need to be done is pressing the green install button and follow the instructions there:



For the purposes of this exemplary application everything is done to run along with v0.4.10 of node.

Now while the Rails installer has SQLite preinstalled Node does not come with a pre-installed DB so it requires a separate installation. The database of choice in this case is mongoDB. Installing mongoDB is as simple as downloading an installer and running it. It is free and available to anyone from the official website of mongoDB

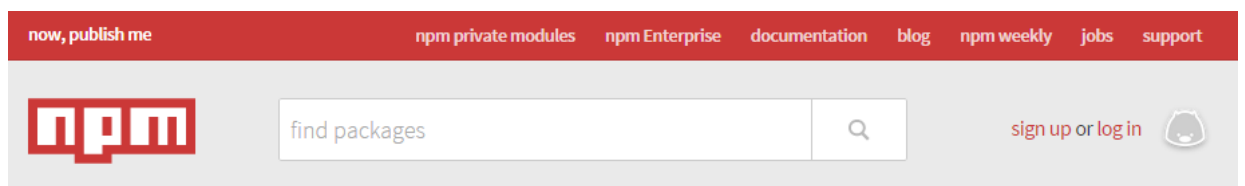
(<http://www.mongodb.org/downloads>):



Download and Run MongoDB Yourself


[Current Release](#) [Previous Releases](#) [Development Releases](#)


And last but not least – NPM. It is also pretty important for the development of web applications with Node.js. It is the package manager for node.js. Once again in order to install the npm you have to visit the official download webpage (<http://www.npmjs.com>) and download the installer for your own OS:



npm is the package manager for **javascript**.

 **141,643** total packages

 **19,314,653** downloads in the last day

 **304,339,848** downloads in the last week

 **1,235,237,153** downloads in the last month

Now that the environment is ready it is time to get to work.

First of all, since Node.js is not a framework we will need to get a hold of one for it. From the console a simple command:

```
npm search
```

Will show all the packages that you can install. After seeing that everything is working it is time to get the Express.js on our machine.

Installing it is as easy as:

```
npm install express -g
```

After express is installed it is now possible to start doing the magic and get the blog rolling.

4.2.2 Basic functionality implementation

Since in this case we are going to deal with a document oriented database (“A document-oriented database is a computer program designed for storing, retrieving, and managing document-oriented information, also known as semi-structured data.” Wikipedia, http://en.wikipedia.org/wiki/Document-oriented_database, accessed on 17/4/15) and not with a relational database (“A relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model as invented by E. F. Codd, of IBM's San Jose Research Laboratory.” Wikipedia, http://en.wikipedia.org/wiki/Relational_database_management_system, accessed on 17/4/15) there is no reason to look at table structure of the DB or even the relations between records within those tables since there is only 1 datatype needed in this application:

```
{
  _id: 0,
  title: '',
  body: '',
  comments: [{
    person: '',
    comment: '',
    created_at: new Date()
  }],
  created_at: new Date()
}
```

This datatype provides a decent foundation and it will suit the needs of this application perfectly. However it should be mentioned that this is not the only way and there are numerous other configurations that could be used for this purpose.

In express an application (usually) contains a call to configure, followed by a couple of method calls that provide declarations to routes and decide what happens to requests that match the route following by a call to a listen. So in case of a simple express application it could be written just like this:

```

// Module dependencies.
var express = require('express');

var app = express.createServer();

// Configuration
app.configure( function() {
});

// Routes
app.get('/', function(req, res) {
    res.send('Hello World');
});

app.listen(3000);

```

It is a single route declaration that works with GET requests to the address “/” from the browser and will return a simple text and a response code of 200 to the client side. Now this is just a simple app with almost simplest code a person can write. However express can automatically build out an application template for the developer. It will even pick a styling and templating engine by itself.

If you run these commands:

```

mkdir blog
cd blog
express -c stylus
npm install -d

```

Firstly, a directory for the blog will be created.

Secondly, Jade templating engine as well as stylus css engine will be used for generating the application.

And thirdly, npm will download and locally install all the required dependencies (for this application).

Now the only thing left to do is put the application file into the blog folder under “app.js”.

After that executing this command:

```
node app.js
```

Will start the node server and at “localhost:3000” there will be the good old “Hello world”.

The web server is now setup and it is possible to proceed to the next stage of the app development.

The layout, the data providers and the views

Most express applications are usually following a pretty similar layout:

```
express          /* The top level folder containing the app */
|-- app.js       /* The application code itself */
|-- lib          /* Third-party dependencies */
|-- public      /* Publicly accessible resources */
|   |-- images
|   |-- javascripts
|   |-- stylesheets
|-- views       /* The templates for the 'views' */
```

This is not a strict layout, it is possible to do it pretty much any way a person desires but a basic layout looks like this.


When it comes to returning and updating the data it is considered a good practice to have a persistent approach. First of all a backbone must be create. It should be a starting version just to get everything running. Later some other layers will be added (See Appendix A).

The above code is saved into a file name articleprovider-memory.js that is located in the same folder as app.js that was created earlier providing the application with the required low level functionality in for of data providers.

The next in line is some small modifications to app.js file (See Appendix B).

With this bit of code now if you were to run the app you would get the object structure of the 3 blog posts that were put into the memory provider to start working with:

Blog



[Post one](#)
Body one

[Post two](#)
Body two

[Post three](#)
Body three

Nothing fancy, but it's a start.

The views

After creating the basic way of storing and reading data the next step is to add an ability to display and create the data properly. First thing to do here is to create a way to view all the blog articles. The express command that was executed previously to bootstrap this application has provided a layout and an also the index page for itself.

Even though the layout file is good the way it is the index page must be adjusted as follows:

```
h1= title
#articles
- each article in articles
  div.article
    div.created_at= article.created_at
    div.title
      a(href="/blog/"+article._id)!= article.title
    div.body= article.body
```

And the routing code for "/" will be adjusted like this:

```
app.get('/', function(req, res){
  articleProvider.findAll( function(error, docs){
    res.render('index.jade', { locals: {
      title: 'Blog',
      articles: docs
    }
  });
});
```

The routing rule changes provide the application with the functionality of replying to all the requests that are headed "/"s way to be replied with a render of all the articles the data provider knows about using the index.jade template.

Now a little bit of styling would be due. The default layout provides the application with a basic stylesheet right away. It has been populated in "public/stylesheet/style.styl". The stylesheet is a default version and needs some tweaking:

```

body
  font-family "Helvetica Neue", "Lucida Grande", "Arial"
  font-size 13px
  text-align center
  text-stroke 1px rgba(255, 255, 255, 0.1)
  color #555
h1, h2
  margin 0
  font-size 22px
  color #343434
h1
  text-shadow 1px 2px 2px #ddd
  font-size 60px
#articles
  text-align left
  margin-left auto
  margin-right auto
  width 320px
.article
  margin 20px
  .created_at
    display none
  .title
    font-weight bold
    text-decoration underline
    background-color #eee
  .body
    background-color #ffa

```

First post

After some basic styling it is now time to get to business. In order to make a post it is in order to have a form to do so. This application is in no need of any complex forms, just a simple one will suffice. Also, 2 new routes must be created to accept the post data and another to return the form.

The form code looks like this:

```

h1= title
form( method="post")
  div
    div
      span Title :
      input(type="text", name="title", id="editArticleTitle")
    div
      span Body :
      textarea( name="body", rows=20, id="editArticleBody")
    div#editArticleSubmit
      input(type="submit", value="Send")

```

And the routes to app look like this:

```
app.get('/blog/new', function(req, res) {
  res.render('blog_new.jade', { locals: {
    title: 'New Post'
  }
});

app.post('/blog/new', function(req, res){
  articleProvider.save({
    title: req.param('title'),
    body: req.param('body')
  }, function( error, docs) {
    res.redirect('/')
  });
});
```

And just like that there is a simple form to add a new post:

New Post

Title : test blog

123123132

Body :

Send

And if I press send the post is added to the list:

Blog

[Post one](#)

Body one

[Post two](#)

Body two

[Post three](#)

Body three

[test blog](#)

123123132

From temporary in-memory JSON store to highly scalable mongoDB store.

In order to make the data persistent through the restarts of the node there must be an installation of node-mongodb-native dependency. It will provide access to mongoDB. NPM will be up to the task, as always. The dependencies of almost all node applications are included within a JSON file in the root of the app, package.json. The npm works with this file natively.

In order to add a dependency this file must be created. It will look something like this:

```
{
  "name": "application-name"
, "version": "0.0.1"
, "private": true
, "dependencies": {
    "express": "2.4.3"
  , "stylus": ">= 0.0.1"
  , "jade": ">= 0.0.1"
  , "mongodb": ">= 0.9.6-7"
  }
}
```

After saving this file it must be executed:

```
npm install -d
```

Now it is time to open the data provider file and make it mongoDB friendly: see Appendix 3 and 4.

And some changes to app.js are due as well: see appendix 5, 6 and 7.

4.2.3 Finishing the app

Displaying an individual article

Displaying an individual article is pretty much the same thing as displaying the list of them all. So in terms of code it's mostly building on top of what is already done with some changes and saving it to a new file.

Another thing that is needed is a new route to let the article to be referenced by a URL.

First of all the index page must be updated:

```
h1= title
#articles
- each article in articles
  div.article
    div.created_at= article.created_at
    div.title
      a(href="/blog/"+article._id.toHexString())!= article.title
    div.body= article.body
```

And the page that is showing a single blog entry:

```
h1= title
div.article
  div.created_at= article.created_at
  div.title= article.title
  div.body= article.body
  - each comment in article.comments
    div.comment
      div.person= comment.person
      div.comment= comment.comment
  div
    form( method="post", action="/blog/addComment")
      input( type="hidden", name="_id", value=article._id.toHexString())
      div
        span Author :
          input( type="text", name="person", id="addCommentPerson")
      div
        span Comment :
          textarea( name="comment", rows=5, id="addCommentComment")
      div#editArticleSubmit
        input( type="submit", value="Send")
```

The new route:

```

app.get('/blog/:id', function(req, res) {
  articleProvider.findById(req.params.id, function(error, article) {
    res.render('blog_show-final.jade',
      { locals: {
        title: article.title,
        article: article
      }
    });
  });
});

```

And last but not least the stylesheet:

```

body
  font-family "Helvetica Neue", "Lucida Grande", "Arial"
  font-size 13px
  text-align center
  text-stroke 1px rgba(255, 255, 255, 0.1)
  color #555
h1, h2
  margin 0
  font-size 22px
  color #343434
h1
  text-shadow 1px 2px 2px #ddd
  font-size 60px
#articles
  text-align left
  margin-left auto
  margin-right auto
  width 320px
.article
  margin 20px
  .created_at
    display none
  .title
    font-weight bold
    text-decoration underline
    background-color #eee
  .body
    background-color #ffa
.article
  .created_at
    display none
  input[type =text]
    width 490px
    margin-left 16px
  input[type =button], input[type =submit]
    text-align left
    margin-left 440px
textarea
  width 490px
  height 90px

```

And now the articles can be viewed separately with just a click.

Comments

Commenting is similar thing to everything done previous with just a single complexity that must be looked after.

In order to add comments to articles I am going to use the \$push update that will allow me to add an element after the end of an array property of an existing document atomically.

The styling and viewing was taken care of in the previous set of code however the routes are still to be provided.

The post handler:

```
app.post('/blog/addComment', function(req, res) {
  articleProvider.addToArticle(req.param('_id'), {
    person: req.param('person'),
    comment: req.param('comment'),
    created_at: new Date()
  }, function(error, docs) {
    res.redirect('/blog/' + req.param('_id'))
  });
});

app.listen(3000);
console.log("Express server listening on port %d in %s mode",
app.address().port, app.settings.env);
```

And a method to the data provider in order to make the change on the persistent level:

```
ArticleProvider.prototype.addToArticle = function(articleId, comment,
callback) {
  this.getCollection(function(error, article_collection) {
    if( error ) callback( error );
    else {
      article_collection.update(
        { _id:
article_collection.db.bson_serializer.ObjectId.createFromHexString(articleId)
},
        {"$push": {comments: comment}},
        function(error, article){
          if( error ) callback(error);
          else callback(null, article)
        });
    }
  });
};
```

And now the blogging app is complete.

5 Conclusion

5.1 Parameters comparison

Now that both blogging applications are completed the next step is to apply the parameters defined in chapter 2.3 to the applications and see what the result will look like.

Installing the framework to your local machine

Parameter	Ruby on Rails	Express.js + others
How many things are required to be installed?	A single installation file that installs all the required tools to the system.	Node.js, Express.js, yeoman.io and mongoDB all have to be downloaded and installed separately
How fast can all of them be found and installed?	Takes about 2-3 minutes total. Finding the installer is rather straight forward. You google it, download it and run it.	Requires you to know what you are doing. As long as you know what tools you need it takes 2-3 minutes per item to be found and installed. So in case of this application it was about 10 minutes total.

In terms of the setup both frameworks are rather easy to install but since Ruby on Rails has a complete installer available that with just 1 download lets you get the whole package while with express.js you are required to download everything you would like to use manually you will use a bit more time on setting up the environment for Express rather than for Rails. For a beginner it is rather easy to mess something up with node and express while RoR has a single installer that will do all the work for you. I would give this category to RoR.

Scaffolding

Parameter	Ruby on Rails	Express.js + others
Is scaffolding supported?	Yes. After installing RoR you get a complete package that includes scaffolding	Express.js does have a command to set up a skeleton for you application but it is nowhere near the level

		that RoR has it. In order to get to similar levels of scaffolding that RoR offers Node.js requires 3 rd party tools, such as yeoman.io
How fast can you get a "hello world" application running?	Roughly 15 minutes.	Roughly 45 mins without yeoman.io and about 25-30 mins with it.
How is the out-of-the-box UI?	Basic UI is not perfect but acceptable	There is no UI generated automatically by express.js while yeoman.io's UI is acceptable.

In terms of scaffolding Ruby on Rails out of the box capability is really amazing. It provides a nice and neat application set up while express.js only can provide a simple skeleton for its application. However if you take yeoman.io into consideration you get a completely different picture. It provide an even better setup for a huge amount of different applications from a blog to a portfolio application. While RoR comes equipet with scaffolding capabilities yeoman.io can at least match it. So this category could be considered both ways.

The application

Parameter	Ruby on Rails	Express.js + others
Amount of time used to get to know the framework (and its language)	Starting level: beginner. Only basic understanding of Ruby and Rails. It was a small learning curve since I had to learn a bit of ruby syntax. The language is rather easy to learn and the framework is pretty simple as well. After about an hour of playing with the language I could consider myself at a good enough level to continue.	Starting level: Intermediate. Decent level of Javascript, basic usage of the Node.js and Express.js. Understanding how Node.js works does not take long. Same goes for express.js. If you know Javascript you will have no problems following any instructions.

Amount of time used to develop the app	Roughly a week, considering that I was not working on it 24/7 and including the learning curve of Ruby/Rails.	Roughly a week. A few details were a bit harder to implement using express.js rather than RoR.
Amount of code that has been written	Most of the code was generated by the framework itself and most of the time I was just feeding commands to the console.	Without yeoman.io pretty much the entire application was written by me. From memory handling to UI I was doing everything by myself.
Overall functionality	All the planned functionalities were implemented without any particular problems.	Was harder to implement than with RoR mainly because there were much more things to take into consideration and also because there was much more code writing in general.

Node.js's biggest advantage by far is the fact that it is written completely in Javascript. A very popular language. Rails are built on ruby. Even though ruby does have a reasonable crowd it is still not nearly as big as the Javascript one is. When it comes down to the application development the ruby on rails, thanks to its structure that is predefined for you, is much easier for a beginner. However node.js provides a lot more flexibility. You decide everything in node.js world. The development is not hard structured and you are allowed to do it the way you feel is the best. Once again, it is a matter of preference. If you would rather have a pre-structured application for you – RoR is the way to go. If you want more freedom and flexibility – Node.js is.

Features head to head

Parameter	Ruby on Rails	Express.js + others
Both applications have similar features implemented?	Ruby on Rails application has 1 extra feature, validation for text input.	Express.js application is missing validation.
What features are missing	None.	Validation would require

(if any)?		usage of another library or a 3 rd party tool that I was not familiar with. Since the express.js application was already taking more time than originally planned this feature was released
Was it equally easy to implement similar features?	All features were rather easy implement. Mostly because they are all standard and they almost came right out of the box.	All features had to be done manually and most of the time required some extra work to be done in other areas of the application to make them work.

While it is possible to implement all of the features using either of these technologies you do have to consider the fact that it is much easier to implement things with rails. Especially if you are not an experienced developer RoR provides great starting ground. Yes, Node.js once again provides flexibility, but I think having a strict structure with a lot of helpers is much more important in this category.

Community

Parameter	Ruby on Rails	Express.js + others
Amount of time required to obtain required setup information	Almost no time. Amount of available information is huge.	Almost no time. Amount of available information is huge.
Amount of guides available for each tool	Searching google for “ruby on rails guide” will result in approximately 1 000 000 results.	Searching google for “express.js guide: will result in approximately 32 000 000 results.
Validity of the information that is provided by these resources	Most popular resources are constantly update and provide up to date guides.	Most popular resources are constantly update and provide up to date guides.
Is community actively developing its framework?	Yes. The Ruby on Rails is a huge community that has been around for a very long time and is still gaining new	Yes. Even though Node.js (and Express.js) are relatively new to the scene the community behind it is tak-

	followers.	ing huge steps and is already producing an enormous amount of quality content.
Availability of different 3rd party tools	Ruby on Rails has a huge library of gems and plugins.	Node.js had a huge success with npm at its launch. However it still has a long road ahead. There are a lot of different solutions available and sometimes it is hard to decide which one is best for your application.

Both frameworks receive amazing community support. RoR has been around for a long time and has a huge fan base and Node.js is moving forward with giant steps specifically because the community support has been so great.

5.2 The summary

So in this particular case both frameworks were up to task. However Ruby on Rails did provide an easier way to get everything setup. If you look at the parameters analyzed above you will see that the best part of using Ruby on Rails to develop a blog application is the fact that it provides a swift and simple way to get everything running while Node.js and Express.js (and others) provide a much more flexible environment. When building an application on top of Node.js you get to select everything from the get-go. Yes, you can reverse engineer Ruby on Rails and control everything there as well. But you don't have to while with Node.js you need to take care of everything by default. So you basically have to make a choice. Either you want to have an easier way with predefined functions and different helpers that are there for you from the start or complete freedom to do the app your way and control everything in the application.

I think that RoR is the framework of choice for a blogging application. As long as you can get over the learning Ruby language part you are ready to go and develop an applications at a much faster rate than while using Node.js.

Even though I do think that ruby on rails is the better choice for blogs I want to make it perfectly clear that this was just a general comparison and by no means had it taken these frameworks to their maximum extend. If I was to go and create a short list of when to use node and when to use rails I would do some more research and come up with a list:

Use Ruby on Rails when:

- You have a well-defined app structure
- There is a presence of complex business rules and validation is required
- The amount of requests is not the decisive factor
- Administrative interface is required
- No specific database of choice defined

Use Node.js when:

- You are comfortable using a lot of APIs
- The application is a real-time web/mobile application
- Application is required to be capable to scale to a lot of concurrent request
- High flexibility of tools

All that said, there is no defined answer. It all depends on what kind of developers the company has and what are they willing to use. What are the application needs and what are people more comfortable with.

Both Rails and Node can provide the same results. I would argue that Rails is the better choice when the application need to be moving quickly. If things like prototyping and functional CRUD app are the things that are valued by a specific project. However a node.js git repository with some pre-grouped Node modules can be just as fast.

5.3 The learning outcome

During the process of writing this thesis I have learned a great amount of things about web development. I have discovered a huge amount of information about the process of selecting a web framework. I have learned that there are multiple frameworks that people are using constantly on a daily basis and there is no clear cut winner. Now I know that Ruby on Rails is capable of being a great tool for a beginner developer as well as a developer with 10 years of experience. I have been introduced to the process of not only developing an application but also documenting every step along the way. I have polished my research skills as well as refreshed my development ones. This thesis was a great learning experience and I hope that it will help in some way to people seeking advice on this topic.

6 References

Wikipedia, Web Developer, URL: http://en.wikipedia.org/wiki/Web_developer

Accessed: 2 May 2015.

José Ignacio Fernández-Villamor, Laura Díaz-Casillas, Carlos Á. Iglesias, A comparison model for agile web frameworks, 2008, ACM Publications, URL: http://ezproxy.haaga-helia.fi:2184/citation.cfm?id=1621087.1621101&coll=DL&dl=ACM&CFID=654480721&CF_TOKEN=89128493

Accessed: 2 May 2015.

Fabiano PS, When to Ruby on Rails, when to Node.js, URL:

<https://fabianosoriani.wordpress.com/2011/09/11/when-to-ruby-on-rails-when-to-node-js/>

Accessed: 16 May 2015

Dwayne Charrington, Should I Use Ruby on Rails or Node.js For My Next

Project/Startup?, URL: <https://hackhands.com/use-ruby-rails-node-js-next-projectstartup/>

Accessed: 10 May 2015

Habrahabr/Sca, Node.js vs Ruby on Rails, URL: <http://habrahabr.ru/post/218147/>

Accessed: 10 May 2015

Rails Casts, Ruby on Rails screencasts, URL: <http://railscasts.com/>

Accessed: 16 May 2015

Rails guides, Ruby on Rails guides, URL: <http://guides.rubyonrails.org/>

Accessed: 16 May 2015

Node.js, Node.js on the Road, URL: <https://nodejs.org/>

Accessed: 16 May 2015

Rae, Node.js vs Ruby on Rails, URL: <http://stackoverflow.com/questions/26160919/node-js-vs-ruby-on-rails>

Accessed: 16 May 2015

Common nodejs modules, URL: <http://stackoverflow.com/questions/29585911/common-node-js-modules>

Accessed: 16 May 2015

7 Appendices

Appendix 1

```
var articleCounter = 1;

ArticleProvider = function(){};
ArticleProvider.prototype.dummyData = [];

ArticleProvider.prototype.findAll = function(callback) {
  callback( null, this.dummyData )
};

ArticleProvider.prototype.findById = function(id, callback) {
  var result = null;
  for(var i =0;i<this.dummyData.length;i++) {
    if( this.dummyData[i]._id == id ) {
      result = this.dummyData[i];
      break;
    }
  }
  callback(null, result);
};

ArticleProvider.prototype.save = function(articles, callback) {
  var article = null;

  if( typeof(articles.length)=="undefined")
    articles = [articles];

  for( var i =0;i< articles.length;i++ ) {
    article = articles[i];
    article._id = articleCounter++;
    article.created_at = new Date();

    if( article.comments === undefined )
      article.comments = [];

    for(var j =0;j< article.comments.length; j++) {
      article.comments[j].created_at = new Date();
    }
    this.dummyData[this.dummyData.length]= article;
  }
  callback(null, articles);
};

new ArticleProvider().save([
  {title: 'Post one', body: 'Body one', comments:[{author:'Bob',
comment:'I love it'}, {author:'Dave', comment:'This is rubbish!}]},
  {title: 'Post two', body: 'Body two'},
  {title: 'Post three', body: 'Body three'}
], function(error, articles){});

exports.ArticleProvider = ArticleProvider;
```

Appendix 2

```
var express = require('express');
var ArticleProvider = require('./articleprovider-
memory').ArticleProvider;

var app = module.exports = express.createServer();

app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(require('stylus').middleware({ src: __dirname + '/public' }));
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});

app.configure('development', function(){
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true
}));
});

app.configure('production', function(){
  app.use(express.errorHandler());
});

var articleProvider= new ArticleProvider();

app.get('/', function(req, res){
  articleProvider.findAll(function(error, docs){
    res.send(docs);
  });
})

app.listen(3000);
```

Appendix 3

```
var Db = require('mongodb').Db;
var Connection = require('mongodb').Connection;
var Server = require('mongodb').Server;
var BSON = require('mongodb').BSON;
var ObjectID = require('mongodb').ObjectID;

ArticleProvider = function(host, port) {
  this.db= new Db('node-mongo-blog', new Server(host, port,
{auto_reconnect: true}, {}));
  this.db.open(function(){});
};

ArticleProvider.prototype.getCollection= function(callback) {
  this.db.collection('articles', function(error, article_collection) {
    if( error ) callback(error);
    else callback(null, article_collection);
  });
};

ArticleProvider.prototype.findAll = function(callback) {
  this.getCollection(function(error, article_collection) {
    if( error ) callback(error)
    else {
      article_collection.find().toArray(function(error, results) {
        if( error ) callback(error)
        else callback(null, results)
      });
    }
  });
};

ArticleProvider.prototype.findById = function(id, callback) {
  this.getCollection(function(error, article_collection) {
    if( error ) callback(error)
    else {
      article_collection.findOne({_id:
article_collection.db.bson_serializer.ObjectID.createFromHexString(id)},
function(error, result) {
        if( error ) callback(error)
        else callback(null, result)
      });
    }
  });
};
```

Appendix 4

```

ArticleProvider.prototype.save = function(articles, callback) {
  this.getCollection(function(error, article_collection) {
    if( error ) callback(error)
    else {
      if( typeof(articles.length)=="undefined")
        articles = [articles];

      for( var i =0;i< articles.length;i++ ) {
        article = articles[i];
        article.created_at = new Date();
        if( article.comments === undefined ) article.comments = [];
        for(var j =0;j< article.comments.length; j++) {
          article.comments[j].created_at = new Date();
        }
      }

      article_collection.insert(articles, function() {
        callback(null, articles);
      });
    }
  });
};

exports.ArticleProvider = ArticleProvider;

```

Appendix 5

```

/**
 * Module dependencies.
 */

var express = require('express');
var ArticleProvider = require('./articleprovider-
mongodb').ArticleProvider;

var app = module.exports = express.createServer();

// Configuration

app.configure(function() {
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(require('stylus').middleware({ src: __dirname + '/public' }));
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});

app.configure('development', function() {
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true
}));
});

```

Appendix 6

```

app.configure('production', function() {
  app.use(express.errorHandler());
});

var articleProvider = new ArticleProvider('localhost', 27017);

// Routes

app.get('/', function(req, res){
  articleProvider.findAll( function(error, docs){
    res.render('index.jade', {
      locals: {
        title: 'Blog',
        articles: docs
      }
    });
  });
});

app.get('/blog/new', function(req, res) {
  res.render('blog_new.jade', { locals: {
    title: 'New Post'
  }
});
});

app.post('/blog/new', function(req, res){
  articleProvider.save({
    title: req.param('title'),
    body: req.param('body')
  }, function( error, docs) {
    res.redirect('/')
  });
});

app.get('/blog/:id', function(req, res) {
  articleProvider.findById(req.params.id, function(error, article) {
    res.render('blog_show.jade',
      { locals: {
        title: article.title,
        article: article
      }
    });
  });
});
});

```

Appendix 7

```
app.post('/blog/addComment', function(req, res) {
  articleProvider.addToArticle(req.param('_id'), {
    person: req.param('person'),
    comment: req.param('comment'),
    created_at: new Date()
  }, function(error, docs) {
    res.redirect('/blog/' + req.param('_id'))
  });
});

app.listen(3000);
console.log("Express server listening on port %d in %s mode",
app.address().port, app.settings.env);
```