

Dissertation
By
Mindaugas Rinkunas
CI 301

Table Of Contents

Introduction	3
Changes Made	4
Removed	4
Added	4
JavaScript: As Foundation Of The Project	5
Mean Stack	6
AngularJS	7
ExpressJS	9
NodeJS	10
MongoDB	13
MVC Architecture	16
Security	17
Build Testing	19
Mocha	19
Chai	20
Travis	22
Representational State Transfer	24
Client Side Routing	25
Live Chat	28
Conclusion	30
Reference	33

Introduction

During three years of studying software engineering and visiting many different modules, gaining valuable insights that helped me to develop views on different concepts and techniques which can be applied when engineering anything from instructing a server to software applications and web platforms, these passing years revealed to me an emerging enthusiasm in learning various web technologies, their architectures, researching and utilising them seems to have strongest traction for my interest and attention, this is the main reason why i chose to seek knowledge to become full stack web developer. Full stack developer is capable on his own effort to develop a platform on the web, which includes: client side, server side and database back end development.

Reasons mentioned above have influenced my decision to choose the scope of the Green House project to put my self upon the challenge to build a platform with technologies that i have never used before and that should raise many problems i have never faced before which would put me into discomfort zone that i would have to overcome for Green House project to be successfull. Researching and applying new technologies, overcoming risen problems with developed solutions would grow me as an engineer and a full stack developer.

Green House project is a take on a familiar business model with main focus on the development process and robustness of the final product. The goal of entire project was not the development of the business model it self but rather the process of development which realised said business model as an online platform by utilising latest and most sought out technologies and security standards in web development industry.

Changes Made

In this section i will briefly list all the changes made to the original specification to provide an instant view of changes to the project. The reasoning for every change that was made is discussed thoroughly and in depth in other parts of this paper where it becomes more relevant in the context.

Removed

- ~~QUnit~~
- ~~Jenkins~~
- ~~Socket.io~~
- ~~ngRoute~~

Added

- + MochaJS
- +ChaiJS
- +Travis CI
- +Client-sessions
- +BcryptJS
- +UI-router

JavaScript: As Foundation Of The Project

JavaScript is one of the three core technologies when talking about world wide web production, the other two are CSS - styling language and HTML - markup of static web page content.

One of the biggest JavaScript advantages i consider to be is its approach as multi-paradigm language, code can be written in object oriented, imperative and functional styles and its always a choice of the developer which approach to use, unless of course developer is using a framework that dictates the approach of programming style when used to build components of your web project. However you could still be able to use any of the three styles when defining your own logic of the platform, so it can be a mix and match, but developers tend to go with the flow and develop in the programming style used in the actual framework to maintain the convention and readability. Another advantage is that there are immense amount of frameworks and technologies written in JavaScript, so to use any of them you don't need to learn a new language, the smaller learning curve is, the more time can be invested into development of the project. 'JavaScript is a programming language for the web, probably one of the popular and wildly used in the world right now.' Prechu (2016) and due to amount of technologies out there being so vast - with over a hundred frameworks and even more libraries out there, you can always choose the best solution to fit specific needs of your ongoing project and all of those technologies are written in one language, JavaScript.

Mean Stack

What is Mean stack? ‘The term MEAN stack refers to a collection of JavaScript based technologies used to develop web applications. MEAN is an acronym for MongoDB, ExpressJS, AngularJS and NodeJS. From client to server, to database, MEAN is full stack JavaScript.’ Jay Raj (2014). Further ahead will be discussed technologies of the MEAN stack that are used in Green House project, explaining how they contribute and will also discuss the thought process behind decisions i have made to use those technologies.

AngularJS

During the planning phase of the project i was looking for a framework with a good performance and great community support to ease up the learning curve, three client side frameworks were considered: AngularJS, EmberJS and BackboneJS. All three frameworks are well known and have made a name for them selves for being efficient in building maintainable single page web applications with minimal amount of code. When choosing a particular framework i kept two aspects in mind: performance and learning curve, the better the performance of the framework the better the platform will function and more easier learning curve is, the more time can be dedicated to development process of the platform, however, intention was not to sacrifice too much performance for the easier learning curve, they key was to find a balance between the two. AngularJS were chosen due to it being optimal in both aspects, when talking about performance it must be taken into consideration how fast angular script can load into browser and all that depends from framework size and bootstrapping speed.



According to an article by Uri Shaked where AngularJS, EmberJS and BackboneJS is compared technically - 'JavaScript assets are usually served minified and gzipped, so we are going to compare the size of the minified-gzipped versions. However merely looking at the framework is not enough. BackboneJS, despite being smallest (only 6.5kb), requires both UnderscoreJS (5kb) and jQuery (32kb) or ZeptoJS (9.1kb), and you will probably need to add some third party plug-ins to the mix.' Uri Shaked (2016), so all in all, size of AngularJS is 39.5kb where as BackboneJS with (jQuery and UnderscoreJS 43.5kb) or with (ZeptoJS and UnderscoreJS 20.6kb), EmberJS with (jQuery and Handlebars 136.2kb) which is quite large compared to other two frameworks. Decision was made to choose AngularJS over BackboneJS even though they go in par when taking size into

consideration, however Backbone is very minimalistic and overall has no structure to offer, therefore is probably more suited when intending to build your own framework from scratch. Angular has solid structure convention that you have to follow enabling anyone else who knows AngularJS to follow up in terms of development with no or little hassle, once convinced that AngularJS is optimal in performance when compared to the other two frameworks, checks were made in terms of size and activity of the community, that is also an important factor not always considered when picking a technology to work with, the size and activity of community not only suggests the quality and efficiency of the technology but it also means that there is more information online - tutorials, better documentation, encountered problems solved, etc. AngularJS is currently fifth most starred project on Github and number one starred project if compared to other single page web application frameworks out there, furthermore, according to google trends AngularJS related content searches generate hundred times more interest than EmberJS or BackboneJS. Please find the link [Google Trend \(2016\)](#) with more detailed information in reference section for the trend below.

Compare Search terms ▼

emberjs
Search term

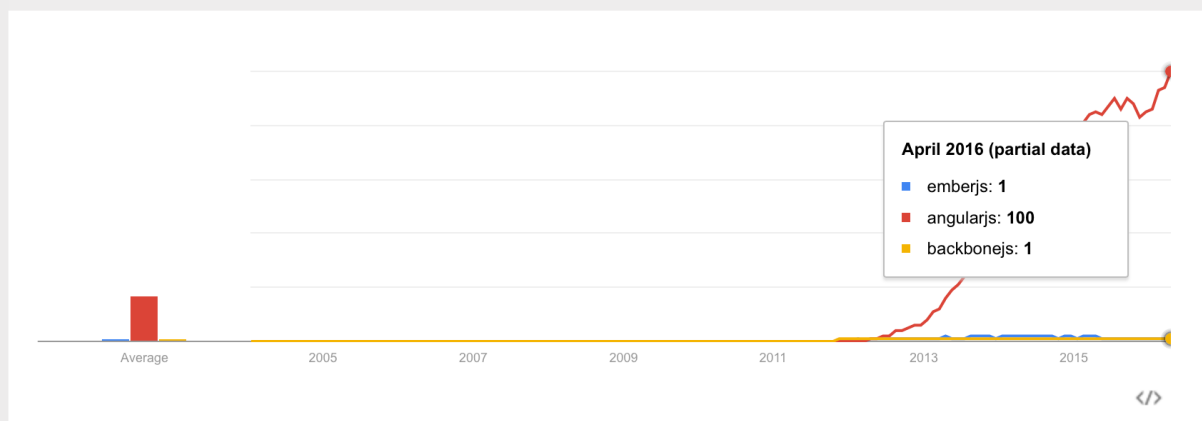
angularjs
Search term

backbonejs
Search term

+ Add term

Interest over time ?

☐ News headlines ? ☐ Forecast ?



ExpressJS

ExpressJS is NodeJS web applications framework (NodeJS will be discussed in a further section), that assists you in managing routes in a RESTful manner and organising your code adequately in to Model-View-Controller pattern on the server side, all though ExpressJS is not forcing a Model-View-Controller pattern when it is being used in a project, it is left for a developer to decide, however it certainly makes it convenient to follow MVC pattern by supporting over fourteen template engines such as Jade that can render templates from the server side to provide views for web applications and can be easily integrated to work with such databases as MongoDB, CouchDB or Redis.

NodeJS

NodeJS is a JavaScript runtime environment for building server side solutions. NodeJS is built on google V8 engine (Virtual Machine). NodeJS has been built with scalability in mind as a real-time system, 'As an asynchronous event driven JavaScript runtime, Node is designed to build scalable network applications. In the following "hello world" example, many connections can be handled concurrently. Upon each connection, the callback is fired, but if there is no work to be done, node is sleeping' NodeJS (2016).

```
const http = require('http')

const hostname = '127.0.0.1'
const port = 3000

const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain')
  res.end('Hello World\n')
})


server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`)
})
```

NodeJS can be very beneficial to the the project that needs to be a scalable network application which could support thousands of concurrent connections and even more beneficial if used as a part of MEAN stack, one language, easy to learn, easy to maintain and to follow the convention.

However, one could say that JavaScript is not equipped to handle sockets and network connections and that would be a solid argument, fortunately, NodeJS is written in C language that is fully capable on performing complex tasks mentioned above and JavaScript is ideal on sending instructions to C program instructing it on what needs to be

done. Due to this approach, written code is also reduced to a minimum, as displayed above in the example from <https://nodejs.org/en/about/>. Example above is a simple server side app, sending a “Hello World” text as a response to the client.

Another great advantage in NodeJS is that it is dead-lock free and you don't need to worry about it when building server side with NodeJS.

‘Almost no function in Node directly performs I/O, so the process never blocks. Because nothing blocks, scalable systems are very reasonable to develop in Node.’ Node  (2016).

Since from the beginning of project planning phase, Green House platform was intended to feature real time chat between users. What was needed is the solution that would allow easily cater tens of thousands concurrent connections of users. Essentially, this is why NodeJS was chosen for its capability to scale network applications. The difference between NodeJS and other server side multithreaded solutions such as Java, in lower level is that when a multithreading capable server side solution is used, it spawns one thread per connection and for example, that thread takes 2 megabytes of space in ram of the machine, lets consider that the said machine used for a server side has 16gb of ram, that would put a theoretical limit of more or less than eight thousand concurrent connections and since every connection has a thread assigned, it is capable of some heavier computation.

Green House platform would not perform heavy computations, but what it would need to be able to do is to maintain high amount of connections running concurrently and thats what NodeJS is capable of providing.

NodeJS is single threaded server side solution and all connections served by one thread, this is ideal since it instantly removes the bottleneck of available thread/ram space ratio for concurrent connections. Every connection made for NodeJS is held in the event loop which essentially goes around every connection and serves it, in

this way as long as NodeJS thread does not get loaded with heavy computations. According to the blog Caustik(2012) where Caustik performed live testing simulation during which one million of concurrent connections was achieved consuming only 16gb of ram. That is why NodeJS is the right server side solution for Green House platform to support it as a real-time system which does not require to support CPU intensive operations.

MongoDB

MongoDB is a database classified as no sql, meaning - its not using traditional sql relational approach of storing data as tables. Instead it stores JSON like documents that are referred as BSON (binary JSON). BSON is simply an extended version of JSON that has added support for more data types while also making data to be more efficiently encoded and decoded with variety of different languages. GreenHouse platform data exchange between client side and server side is in JSON format, my intent was not to load server side with more computations that would be required to parse data from database and would slow down the server (retrieving it and reformatting data into JSON objects to send it to client), also to save more time for development of other aspects of the platform. I used MongoDB to solve this and no parsing was needed, it is great advantage that data exchange between server side and MongoDB is in

JSON format and this is an example of data retrieved from GreenHouse database.

```
[switched to db greenhouse
> db.listings.find().pretty()
[{
  "_id" : ObjectId("57101c2ae4b12588ffb7bc50"),
  "dateAdded" : "14/04/2016-23:39:38",
  "userFirstName" : "Mindaugas",
  "userId" : "57101b37e4b12588ffb7bc4e",
  "address" : "50 Upper North Street",
  "city" : "Brighton",
  "county" : "East Sussex",
  "postCode" : "BN1 3FH",
  "dateAvailable" : "25/06/2016",
  "price" : "1200",
  "paymentPeriod" : "PM",
  "propertyType" : "FLAT",
  "title" : "2 Bedroom Flat",
  "description" : "Greetings,\n\nReally nice flat, contact me for more information.\n\nRegards,\n\nMint",
  "__v" : 0
},
{
  "_id" : ObjectId("57101d56e4b12588ffb7bc51"),
  "dateAdded" : "14/04/2016-23:44:38",
  "userFirstName" : "Mindaugas",
  "userId" : "57101b37e4b12588ffb7bc4e",
  "address" : "20 Brighton Road",
  "city" : "Brighton",
  "county" : "East Sussex",
  "postCode" : "BN5 GT5",
  "dateAvailable" : "25/07/2016",
  "price" : "3500",
  "paymentPeriod" : "PM",
  "propertyType" : "HOUSE",
  "title" : "5 Bedroom House",
  "description" : "Hi,\n\nBig house for rent, if you have any questions, please dont hesitate to contact me.\n\nRegards,\n\nMint",
  "__v" : 0
},
{
  "_id" : ObjectId("57101f93e4b12588ffb7bc53"),
  "dateAdded" : "14/04/2016-23:54:11",
  "userFirstName" : "Mindaugas",
  "userId" : "57101b37e4b12588ffb7bc4e",
  "address" : "20 Road",
  "city" : "London",
  "county" : "London County",
  "postCode" : "GT5 GT5",
  "dateAvailable" : "25/06/2016",
  "price" : "1000",
  "paymentPeriod" : "PM",
  "propertyType" : "FLAT",
  "title" : "1 Bedroom Flat",
  "description" : "Good sized flat, newly renovated, enquiries are welcome.\n\nRegards,\n\nMint",
  "__v" : 0
},
{
  "_id" : ObjectId("5710a8924ef837f402e7c8d9"),
  "dateAdded" : "15/04/2016-09:38:42",
```

That saves us a lot of development time and avoid unnecessary computations that would otherwise slow down the server, since we don't need to parse data retrieved from database because data is stored almost exactly as it is retrieved from client side, only specific id record is added by default to serve as a unique key and a specific tag by the Mongoose to accommodate its own tracking. However using

MongoDB presented me with other issues, since i am using it as a database solution for Green House project first time and i come with from sql background thought through out the years in this course, non relational database concept is ~~new to me i didn't know~~ how to enforce constraints such as - make that record field cannot be left empty or how to declare datatypes, after doing some research i have discovered Mongoose, it is a modelling tool for MongoDB and it does the job well. Before data is stored, data types and constraints are enforced by the server side using Mongoose modelling tool. You can simply declare the structure and constraints of your collections in your MongoDB database like in this example from GreenHouse project source code below.

```
schemas.js
1 var mongoose = require('mongoose');
2
3 var Schema = mongoose.Schema;
4 var ObjectId = Schema.ObjectId;
5
6 var Listing = mongoose.model('listings', new Schema({
7   id: ObjectId,
8   dateAdded: { type: String, required: true },
9   userFirstName: { type: String, required: true },
10  userId: { type: String, required: true },
11  address: { type: String, required: true },
12  city: { type: String, required: true },
13  county: { type: String, required: true },
14  postCode: { type: String, required: true },
15  dateAvailable: { type: String, required: true },
16  price: { type: String, required: true },
17  paymentPeriod: { type: String, required: true },
18  propertyType: { type: String, required: true },
19  title: { type: String, required: true },
20  description: { type: String, required: true }
21 }));
22
23 var User = mongoose.model('users', new Schema({
24   id: ObjectId,
25   firstName: { type: String, required: true },
26   secondName: { type: String, required: true },
27   email: { type: String, unique: true, required: true },
28   password: { type: String, required: true }
29 }));
30
31 module.exports.Listing = Listing;
32 module.exports.User = User;
```

as displayed in the example above, all the records are declared as strings and are forced to contain an entry before insertion into the database.

MVC Architecture

First time i got introduced to patterns was during my first year in university by attending CI101 Mike Smith module and the very first pattern i learned was Model View Controller pattern. The concept was taught thoroughly during lectures. Assignment for CI101 i have chosen was an ATM simulation of which architecture was a Model View Controller and encouraged me to to research more on the pattern. CI101 module gave me a good foundation on understanding what are the patterns, it made me realise how patterns can be beneficial and depending on project specifics the right architecture should always be used when building software or online platform/system related product.

Coincidentally, Green House project also applies Model View Controller pattern for its architecture.

Model View Controller pattern separates complex logic into modules and establishes universal pattern of communication between them and this is the case in Green House platform as well, client side contains views and controllers while server side contains model (business logic). Not only does this benefits the system by reducing clutter produced in source code over time, but also makes the source code parts more easily reusable and entire build of the system highly modular - developer can make changes in different parts of the platform or software source code without breaking the code as long as modules that are changed still submits to established pattern of communication (sends out and retrieves data as expected).

Team of developers also greatly benefits from this pattern. Multiple teams are able to work on different aspects of the platform - UI

developers can work on Views while business logic developers are working on model.

I am confident now, that Green House platform carries clean code which is modular and highly maintainable.

Security

During development, i started to place more and more effort and attention in developing effective security for the platform. Just by setting up a password in the database and comparing it on the server side for authenticity is not enough. Due to two major weak parts i have discovered in GreenHouse platform during development process, that being said, additional security technologies had to be researched and implemented.

First weak part - To track sessions I'm using cookies and those cookies are stored on a browser, and anyone with the knowledge “where to click” can easily check the contents of the stored cookie, contents like: login and password. That would be disastrous as this kind of information leak would compromise users account especially if user used a public machine to login into the platform. Therefore cookies had to be encrypted for safer and more reliable storage in the browser. I also took a human error into consideration, that sometimes when users are logged in on publicly accessible machines they forget to logout after finishing their work and close the browser. It would be ideal if server would invalidate the stored cookie on the browser after browser has been closed.

By researching on these problems i have came across a library called node-client-sessions (there is a link in a reference with more in depth information) that covers these areas of problems and i have implemented this library into GreenHouse project. The library, node-client-sessions provides highest standards of security and uses AES-256 cipher to encrypt data which is reviewed and tested by NSA

(National Security Agency) and approved by United States government to be used to encrypt data which requires highest level of clearance to be accessed, in other words - top secret data. AES cipher is also an ISO/IEC 18033-3:2010 published standard in an official ISO catalogue, ISO (2015).

Second weak part - During development i realised that i cannot consider database to be 100% resistant towards attacks that would compromise data and looking at the cases such as - major company like eBay data leak (BBC) 2014, made me to consider in securing data on database server side it self, how could sensitive information like user passwords be protected. After researching i have discovered that sensitive information can be hashed before inserting into database and that there are more than one library that can be a great assistance in doing that. I have used Bcryptjs to secure data on a database itself, therefore if data leak occurs, attacker gets meaningless data, because sensitive data like passwords is hashed and attacker cannot access accounts based on stolen data.

Bcryptjs is based on Blowfish cipher that was presented at USENIX in year 1999, Wikipedia(2015) and held its ground in the industry for one and a half decade, it is still used by this day.

Another beneficial aspect that contributes greatly to security is that Bcryptjs is resilient to brute force attacks and the way this is achieved is Bcryptjs being an adaptive function which can in turn increase amount of iterations over time to make it self slower when some one starts repeatedly and rapidly testing your system against different values to try and guess the right one. That would provide more time for owners of the targeted server to detect said attack and prevent it.

Build Testing

During the planning phase i have chosen QUnit as projects testing library, however, when it came to development i encountered numerous problems. While QUnit fits well for client side testing it does not offer much on its own to test server side, there is however a plugin that allows us to mock http requests and responses for the client side and can be used for parallel development of front end and server side at the same time. Front end developers could build client side on mocked data while server side developers are building server functionality. But in this case i needed to test server side specifically, because if server side is buggy, severe consequences might occur while using GreenHouse platform. Therefore i brought in Mocha testing framework paired up with Chai assertion library for NodeJS, these two tools are powerful when it comes to testing, below i will explain how i solved risen issues with an aid of tools mentioned above.

Mocha

Mocha is a very flexible testing framework that runs developers designed tests and outputs results in a command prompt. The main reason behind the choice of Mocha for testing was its flexibility. You can pair it up with any assertion library, meaning that depending on what you are testing you can choose different approach in regards of logic used in the tests. Which in this case i needed to test server side's functionality and the problem was servers security that i wrote which prevented me from effectively testing the routes.

Every time a request was made to the server side for sensitive information like user profile, server would check for a valid session of the cookie stored in a browser, which means that sending login and password is not enough. With every request i also need to provide

valid cookie data for server to evaluate and send back sensitive data requested if evaluated session holds. That is where Chai assertion library was used.

Chai

With Chai assertion library I was able to recreate sessions in the testing environment that allowed me to test routes that send back responses with sensitive information or execute actions that require authentication. In the example below is displayed one test case from GreenHouse testing source code.

```
it('Should send back profile data as long as user is logged in. On /profile GET', function(done){
  var agent = chai.request.agent(server.app)
  agent
    .post('/login')
    .send({ email: 'some@email.com', password: '12345' })
    .then(function () {
      return agent.get('/profile')
        .then(function (res) {
          res.should.have.status(200);
          res.should.be.json;
          res.body.should.be.a('object');
          res.body.should.have.property('redirect');
          res.body.should.have.property('user');
          res.body.user.should.have.property('id');
          res.body.user.should.have.property('firstName');
          res.body.user.should.have.property('secondName');
          res.body.user.should.have.property('email');
          res.body.redirect.should.equal('/#/profile');
          res.body.user.firstName.should.equal('Exo');
          res.body.user.secondName.should.equal('Back');
          res.body.user.email.should.equal('some@email.com');
          recipient = res.body.user.id;
          done();
        })
    })
});
```


In the beginning of the test we instantiate an agent that will call the login route sending valid login information and will retain a valid cookie that is sent back from the server after login data is accepted. Since agent now contains session information, he can call routes that will require validation of that session, in the case above we are

requesting profile information of an account that details were used to login. We expect response status to have a code of two hundred, we expect that response is in JSON format, we also expect it to be an object and then we thoroughly check that the object would have the right properties and those properties would have the right values. By using Mocha and Chai as Green House project's testing environment i have managed to get valid responses from performing requests that were challenged by the server side to provide valid session information which is not only login and password but also cookie session information that is created after login and password is validated.

Travis

During planning phase it was decided to use Jenkins continuous integration server, but this has been changed due to several reasons. While trying to setup Jenkins on personal machine, it proved to be time consuming and precious development time was being lost, another reason is some what poor documentation and community troubleshooting in comparison to Travis. When compared Jenkins to Travis, it is clear that Jenkins brings more benefits in terms of control, due to it being hosted on personal machine you can configure Jenkins to fit any needs and that would be highly beneficial for tremendous projects that are being developed by 10 or more developers, since a lot of update pushing might occur you expect an availability one hundred percent and pushed updates are tested immediately. Travis CI does not guarantee you that, since travis is an online hosted service it comes already preconfigured and when performing tests, they might be put in a queue among the tests of other users that are testing at the same time to maintain Travis servers performance, also a tester is on the mercy of Travis online availability. That being said, what this project needed is only one machine capable automatically run the test suite before every push to Github without the need of any configuration of a server for only one developer that might be pushing an update at any given time. Travis offered exactly that and saved a lot of time that was dedicated towards development of GreenHouse platform, once new update is pushed to Github, Travis tests updated build by using your own developed testing suite and outputs status of the test

performed as displayed in following example of GreenHouse platform's pushed update, message is also sent to a registered email with the status report as displayed below.

Arisato / GreenHouse

build passing

[Current](#)
[Branches](#)
[Build History](#)
[Pull Requests](#)
More options

✓ master Update

Commit 9801801
Compare 7b51187...9801801
Mindaugas authored and committed

#45 passed

Elapsed time 45 sec
3 days ago

Followed by a console output with more detailed information about the testing process. Below displayed example is a fraction to give an idea about travis reporting, since the whole capture of the console output would take over few pages.

```

528 Access Granted
529 { attachments: [],
530   alternative: null,
531   header:
532     { 'message-id': '<1460838997353.0.3065@testing-worker-linux-docker-0dd6cdf4-3366-linux-6>',
533       date: 'Sat, 16 Apr 2016 20:36:37 +0000',
534       from: '=?UTF-8?Q?Tester?= <greenhousepropertyltd@gmail.com>',
535       to: '=?UTF-8?Q?Exo?= <some@email.com>',
536       subject: '=?UTF-8?Q?In_regards_of_property_advertised_on_Green_House.?=',
537       content: 'text/plain; charset=utf-8',
538       text: 'Quick message. Please use this email for reply: test@email.com' }
539   ✓ Should send and email to ad owner if sender is logged in on the platform on /emailad/:id (1643ms)
540   { email: 'some@email.com', password: '12345' }
541   Email is correct.
542   Password is correct.
543   Access Granted
544   This is Doc 5712a2536f7e21f90bf9cf30
545   This is Session 5712a2536f7e21f90bf9cf30
546   ✓ Should delete specific users, specific ad if said user is logged in. On /remove/:id DELETE (236ms)
547   { email: 'some@email.com', password: '12345' }
548   Email is correct.
549   Password is correct.
550   ✓ Should logout user from the platform on /logout GET (250ms)
551
552
553   13 passing (4s)
554
555
556
557   The command "npm test" exited with 0.
558
559   Done. Your build exited with 0.

```

Representational State Transfer

During the development of GreenHouse platform i wanted to make the system capable for expansion. For example if i decide to release an app in the future, i didn't want to be forced to rewrite the server side explicitly for an app, instead i wanted to build a server side compatible not only with the browser but with other devices also. While i was researching for possible solution i kept stumbling all the time on a concept called RESTful. After more research, i quickly learned that ExpressJS framework that i use in this project supports and makes it easy to apply RESTful concept to the system. The way RESTful concept benefits the system is quite simple as it should be, since RESTful does simplify communication between two machines over the internet.

Routes provided by the server side sends back a specific resource as a response to a machine that issued a request on that route using one of most common “GET”, “POST”, “PUT” or “DELETE” HTTP methods. These methods tells the server if client request intends to read, create or delete something.

Clients request link structure has to match the route structure represented on server side to retrieve the right resource. Once this standard established on a server side and client they successfully communicate and exchange data, any device out there requesting link with the matching structure to the server side route will receive a resource corresponding to that route. This standard applies to GreenHouse platform server side and if an app will be developed in the future, it will be able to use the same server side that is used by the Green House browser platform when retrieving resources from the server.

Client Side Routing

During development of the client side i quickly found out that ui design i was meant to implement cannot be implemented. Reason for this is Angular's ngRoute module. The way ngRoute works is, it maps developed html template to a specific route, so when the link pointing to a particular route is clicked, html template associated with that route is loaded. These templates can be loaded on demand into index.html, so whenever you click a different link another view is displayed on index.html without reloading the page into another page and that is a single page web application which stays in index.html but loads different views into that index.html on demand. However, design i had in mind is suppose to load two views at the same time - view of the listings stored in the system and login panel, once user logs in with the login panel, login panel view should be changed into welcome panel which indicates that user is logged in, without changing anything about the listings view. This can be achieved with ngRoute module loading only one view by simply creating two html templates: one template with listings and and login panel and a second template with listings and logged in confirmation panel, then just switch between the two when user logs in or logs out. This approach would have its downside - developer would have to write two almost identical html templates and that would be redundant, this would diminish modularity as you would have to create duplicate templates to put some sort of small twist in existing html template. We have to keep same html markup that defines listings in both templates just because another part is changing. I didn't like the process mentioned above, considered it to be redundant and realised that when the application grows while being developed, these issues will arise with other views as well and making views cluttered with multiple duplicate templates that would be hard to maintain. Because there is two templates with listings section in them, if we would like to make a

change to listings section we would have to introduce the same change in two different locations, this can increase depending on how many duplicate templates there are. Therefore i had to find a solution to this immediately before progressing further with the development of the platform.

To solve these issues i had to utilise ui-router module (ui-router is referenced in a reference section below), this module enables loading of multiple views at the same time by nesting them together. By using this module i have managed to load two different html templates into one page at the same time, listings template and login template. If user logs in into platform, only login panel changes into logged in confirmation panel and this time listings part haven't been touched since it is stand alone template now and we only needed to change the login panel into another view. Please find an example from the GreenHouse project source code of the utilised ui-router below.

```
app.js
1  var app = angular.module('app',['ui.router']);
2
3  app.config(['$stateProvider', '$urlRouterProvider', function($stateProvider, $urlRouterProvider){
4      $urlRouterProvider.otherwise("/login");
5
6      $stateProvider
7          .state('home', {
8              url: "/",
9              views: {
10                  "listingView": { templateUrl: "views/listing.html",
11                               controller: "listingsController" },
12                  "loginView": { templateUrl: "views/login.html",
13                               controller: "loginController" }
14              }
15          })
16          .state('login', {
17              url: "/login",
18              views: {
19                  "listingView": { templateUrl: "views/listing.html",
20                               controller: "listingsController" },
21                  "loginView": { templateUrl: "views/loggedin.html",
22                               controller: "loggedinController" }
23              }
24          })
25          .state('logout', {
26              url: "/logout",
27              views: {
28                  "profileView": { templateUrl: "",
29                               controller: "logoutController" }
30              }
31          })
32      })
33  })
```

As displayed in the image above there are two templates nested in one state and when one state switches into another only one template is changed. This way i managed to reduce the code needed to produce the views because duplicates has been eliminated and also reduce the web app physical size so it could be downloaded faster by a web browser.

Live Chat

Live chat between users of the platform is a major implementation. It effects client side and server side source code drastically. For this implementation you need to setup and configure sockets in both client side and server side, sockets need to be set up to listen for particular sockets and serve as a pipe for data stream between client side and server side or vice versa.

To really understand socket.io real-time engine i had to develop my own stand alone chat platform with chat server side. Building this stand alone chat i have gained valuable insights on how socket.io can be utilised to build real-time systems. However due to reasons explained below i haven't implemented live chat into the Green House platform.

Even though stand alone platform is written and ready for integration when tweaked a little to fit requirements of the platform, i didn't managed to implement live chat due to a risen challenge when time was of the essence, drawing closer to the date of submission.

The way AngularJS works is that every view template has its own controller attached to it and the only script that is allowed to load in a specific view template is the one thats declared in a controller of that particular view template or globally via index.html page by adding script tags with a source link. There are some script parts of socket.io that you need to include globally and as mentioned above, is easy to do via index.html page. The problem arises once you try to load your own developed script in a specific view template to work with globally declared socket.io library. As mentioned above, AngularJS only allows to load scripts coming from the attached controller of a template and AngularJS controllers does not allow dependency injection from external JavaScript files. From what i have managed to gather when researching, i understood that i would need to write my own versions of some parts of Angular's internal directive system and to do that i

would need a lot more time than i have to finish the platform.

Therefore decision was made to exclude live chat so that time could be dedicated to development of the rest of the platform functionality and the live chat would be delivered as a future update.

However, along with Green House platform i have included a stand alone chat platform which is fully functional and could be easily hosted online for a public chat purpose and also is ready for implementation as one-to-one private live chat in Green House platform, for your respectful evaluation. Live chat is developed but due to unexpected and time consuming problem mentioned above, couldn't been implemented. Although i intend to deliver the implementation of the live chat in Green House platform to showcase it in my portfolio and resume, but the implementation will have to be scheduled after graduation.



Conclusion

Through the duration of this project i have gained invaluable experience by researching and engineering Green House platform. Majority of this project's technologies i have utilised are for a first time and therefore substantial amount of research took place to assist me in understanding their advantages and disadvantages, how to use those technologies and most importantly, when to use them and when not to.

Most challenging aspect of this project was to make all these technologies work and interact together by integrating them with one another and the most challenging part of the project was implementation of the server side, where i had to take into consideration how it will handle concurrent connections and compare it to Java server side solution by applying foundation of knowledge gained from CI346 Programming Languages, Concurrency And Client Server Computing by Mike Smith and also by further research of how NodeJS achieves concurrency in lower level.

If i would be doing this project again there are some aspects of the project that i would do differently, such as, when researching technology or a concept i would also do a research on how it synergises with other technologies already present in the projects spec, that would reduce occurring inconsistencies later in development process and causing to research for different solutions in the middle of development.

I would also like establish a different tier of architecture to be built upon. Currently the platform can be considered as two tier architecture because client side is on remote user machine and the server side with the database is located on the same machine. What i have learned in CI315 Object-Oriented Design And Architecture by Dr. Nour Ali is that i could increase scalability and make it more flexible if i would develop Green House project in three tier

architecture. What i would need to do is to separate server side and database in to different machines and develop a communication between them. That would benefit the system greatly because you could replicate different parts of the system more efficiently depending on the traffic, lets say if there are a lot of traffic going on through sockets (large amount of users using live chat), server side could be replicated with additional machines and thus accommodating the load or if we get a lot of property ad insertions, we could effectively replicate the database side with additional machines to deal with the traffic load of insertion operations performed on the database.

Reference

1. AngularJS (2010 - 2015) *HTML enhanced for web apps!*, [Online], Available: <https://angularjs.org> [15 July 2015]
A learning and documentation resource for AngularJS framework. Also provides tutorials on how to deploy and use the framework.
2. Bert Bos (1994 - 2015) *Cascading Style Sheets*, [Online], Available: <http://www.w3.org/Style/CSS/> [20 June 2015]
A learning and documentation resource for CSS3 (Cascading Style Sheets), also contains tutorials explaining on how to style web pages.
3. Body-Parser (2015) *Node.js body parsing middleware*, [Online], Available: <https://www.npmjs.com/package/body-parser> [22 August 2015]
A learning and documentation resource for body parser middleware, has plenty examples how use it to parse JSON objects on your server side.
4. Bootstrap (2011) *Bootstrap is the most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web*, [Online], Available: <http://getbootstrap.com> [27 June 2015]
A learning and documentation resource, with tutorials and examples on how to use Bootstrap framework, also has quite big exposition of websites with well known brands that used the Bootstrap framework in their projects.
5. ExpressJS (2015) *Fast, unopinionated, minimalist web framework for Node.js*, [Online], Available: <http://expressjs.com> [12 August 2015]
A learning and documentation resource for this framework, that has examples on how to deploy it and use it appropriately.

6. Florian Motlik (2013) *What is Jenkins? When and why is it used?*, [Online], Available: <https://www.quora.com/What-is-Jenkins-When-and-why-is-it-used> [12 October 2015]

An article written by Florian Motlik who is CTO and founder of 'Codeship', explaining what is Jenkins, when it is deployed and what advantages are there when using Jenkins.

7. Jenkins (2011) *An extensible open source continuous integration server*, [Online], Available: <https://jenkins-ci.org> [10 October 2015]

A learning and documentation resource, with usability concepts and examples on how to use Jenkins.

8. Jonathan Rasmusson (2015) *Agile In a Nutshell*, [Online], Available: <http://www.agilenutshell.com> [22 October 2015]

An article with rich explanation on what is and how is agile methodology adopted by a project, what advantages does it bring.

9. jQuery (2015) *jQuery, write less, do more*, [Online], Available: <https://jquery.com> [10 July 2015]

A learning and documentation resource on how to use jQuery to animate and build interactive web projects.

10. MongoDB (2015) *Launch your giant idea*, [Online], Available: <https://www.mongodb.org> [20 September 2015]

A learning and documentation resource of usability concepts, also on how to set up and use a mongoDB database.

11. Mongoose (2011) *elegant mongoldb object modelling for node.js*, [Online], Available: <http://mongoosejs.com> [17 August 2015]

A learning and documentation resource for Mongoose, how to use it to structure data on the server respectively.

12. Monjurul Habib (2013) *Agile software development methodologies and how to apply them*, [Online], Available: <http://www.codeproject.com/Articles/604417/Agile-software-development-methodologies-and-how-t> [23 October 2015]

An article explaining Agile Manifesto and many other different agile variations that broadly used in today's industries.

13. NodeJS (2015) *Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine*, [Online], Available: <https://nodejs.org/en/> [2 August 2015]

A learning and documentation resource for Node.js server scripting framework, accompanied with guides on how to use the framework.

14. QUnit (2015) *QUnit: a JavaScript unit testing framework*, [Online], Available: <https://qunitjs.com> [20 October 2015]

A learning and documentation resource with examples and tutorials on how to use QUnit framework for testing.

15. Socket.IO (2015) *Featuring the fastest and most reliable real-time engine*, [Online], Available: <http://socket.io> [5 September 2015]

A learning and documentation resource with easy and accessible tutorials for beginners.

16. W3C (2014) *HTML5*, [Online], Available: <http://www.w3.org/TR/html5/> [25 June 2015]

A learning and documentation resource, explaining HTML markup language and how to use it.

17. Wikipedia (2016) *JavaScript*, [Online], Available: <https://en.wikipedia.org/wiki/JavaScript> [5 Feb 2016]

Great source of information about JavaScript, its origin, role and importance in the world of web development and beyond.

18. Prechu (2016) *100+ JavaScript Frameworks For Web Developers*, [Online], Available: <http://www.cssauthor.com/javascript-frameworks/> [10 Feb 2016]

An article that lists over a hundred frameworks available, that are written in JavaScript.

19. Jay Raj (2014) *An Introduction To The MEAN Stack*, [Online], Available: <http://www.sitepoint.com/introduction-mean-stack/> [10 Feb 2016]

A brief introduction explaining what is MEAN stack and technologies used in MEAN stack, also with code samples.

20. Google Trend (2016) *Compare Interest Over Time*, [Online], Available: <https://www.google.com/trends/explore?hl=en-US#q=emberjs%2C%20angularjs%2C%20backbonejs&cmpt=q&tz=Etc%2FGMT-1> [10 Feb 2016]

Generated graph by google trend that displays ratio between searches being made in regards of AngularJS, EmberJS and BackboneJS.

21. Uri Shaker (2016) *AngularJS vs. BackboneJS vs. EmberJS*, [Online], Available: <https://www.airpair.com/js/javascript-framework-comparison> [10 Feb 2016]

Rich technical comparison of three well known frameworks.

22. Github (2016) *Search*, [Online], Available: <https://github.com/search?q=stars:%3E1&s=stars&type=Repositories> [10 Feb 2016]

A quick search displaying most starred projects on Github.

23. Pony Foo (2016) *JavaScript Developer Survey Results*, [Online], Available: <https://ponyfoo.com/articles/javascript-developer-survey-results> [5 Feb 2016]

Rich survey, exploring how developers tend to utilise JavaScript.

24. IEEE Xplore (2014) *JavaScript: The Used Parts*, [Online], Available: <http://ieeexplore.ieee.org.ezproxy.brighton.ac.uk/stamp/stamp.jsp?tp=&arnumber=6899250&tag=1> [5 Feb 2016]

A conference research document on how JavaScript programming language is used by the developers.

25. Wikipedia (2016) *MongoDB*, [Online], Available: <https://en.wikipedia.org/wiki/MongoDB> [7 Feb 2016]

Wikipedia page which offers information about MongoDB.

26. Github (2016) *mongodb/mongo*, [Online], Available: <https://github.com/mongodb/mongo> [7 Feb 2016]

Github repository containing MongoDB source code files, readme file also provides directions to richer information sources like documentation and how to learn and use MongoDB.

27. BBC (2014) *eBay faces investigations over massive data breach*, [Online], Available: <http://www.bbc.co.uk/news/technology-27539799> [15 Feb 2016]

BBC article about massive user data leak from eBay that affected about hundred and forty five million users.

28. Wikipedia (2016) *Advanced Encryption Standard*, [Online], Available: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#Security [10 Feb 2016]

A good source of information about encryption standards.

29. ISO (2015) *ISO/IEC 18033-3:2010*, [Online], Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=54531 [10 Feb 2016]

An entry of an official ISO catalogue introducing AES cipher as official ISO standard.

30. NPM (2015) *bcryptjs*, [Online], Available: <https://www.npmjs.com/package/bcryptjs> [11 Feb 2016]

Library link which can be used to access documentation of BcryptJS and download the library.

31. Wikipedia (2016) *bcrypt*, [Online], Available: <https://en.wikipedia.org/wiki/Bcrypt> [11 Feb 2016]

Source of information about what bcrypt is based on, its implementations and related algorithms.

32. Wikipedia (2016) *USENIX*, [Online], Available: <https://en.wikipedia.org/wiki/USENIX> [11 Feb 2016]

Wikipedia page with information about USENIX organisation and its achievements.

33. Mockjx (2016) *The jQuery Mockjax Plugin provides a simple and extremely flexible interface for mocking or simulating ajax requests and responses*, [Online], Available: <https://github.com/jakerella/jquery-mockjax> [20 Feb 2016]

A plugin that can be used in conjunction with QUnit to mock http requests and responses.

34. Mocha (2016) *mocha simple, flexible, fun*, [Online], Available: <https://mochajs.org> [20 Feb 2016]

Documentation and installation page for Mocha testing framework.

35. Chai (2016) *Chai is a BDD/TDD assertion library for node and the browser that can be delightfully paired with any javascript testing framework.*, [Online], Available: <http://chaijs.com> [20 Feb 2016]

Documentation and installation page for Chai assertion library.

36. Wikipedia (2016) *Presentational state transfer*, [Online], Available: https://en.wikipedia.org/wiki/Representational_state_transfer [5 Feb 2016]

Wikipedia entry with information about REST concepts.

37. Travis (2016) *Test and Deploy with Confidence*, [Online], Available: <https://travis-ci.org> [5 Feb 2016]

Travis homepage with documentation information and also where testing results are displayed.

38. Ui-router (2016) *The de-facto solution to flexible routing with nested views in AngularJS*, [Online], Available: <https://github.com/angular-ui/ui-router> [20 Feb 2016]

Ui-router Github page, where you can in depth documentation and tutorials on how to use this module in AngularJS.

39. Tomislav Capan (2013) *Why The Hell Would I Use Node.js? A Case-by-Case Tutorial*, [Online], Available: <https://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js> [7 Feb 2016]

A comprehensive tutorial also reviewing a practical side of NodeJS.

40. Caustik (2012) *Node.js w/1M concurrent connections!*, [Online], Available: <http://blog.caustik.com/2012/08/19/node-js-w1m-concurrent-connections/> [7 Feb 2016]

A blog featuring source code and test results of one million concurrent connections achieved.

41.

