OAMK
OULU UNIVERSITY OF
APPLIED SCIENCES

Khoa Bui

BLOGGING PLATFORM WITH JAVASCRIPT STACK

BLOGGING PLATFORM WITH JAVASCRIPT STACK

Khoa Bui
Thesis
Spring 2017
Bachelor of Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Bachelor of Information Technology

Author: Khoa Bui
Title of the bachelor's thesis: Blogging Platform with JavaScript Stack
Supervisor: Lauri Pirttiaho
Term and year of completion: Spring 2017 Number of pages: 55 + 2 appendices

The aim of this bachelor's thesis was to study the application of JavaScript in web development, the advantages and drawbacks as if developing a simple web application (a blog platform).

In the scope of this project, the author used ReactJS, Flux and SCSS in the Frontend development, NodeJS, ExpressJS, MongoDB, Mongoose, JSON Token in the Backend development, and Webpack and Babel in the Workflow development. Additionally, the development technologies also included NGINX, Vagrant and Webstorm (an IDE for JavaScript development).

And in the end of the project, there is a clearer overview on an alternative development stack with primarily JavaScript programming languages and as a result, a working blogging platform with JavaScript.

Keywords: Web Development, ReactJS, NodeJS, MongoDB.

PREFACE

This Bachelor's Thesis is a part of my study in the field Information Technology at Oulu University of Applied Sciences, School of Engineering, Kotkantie Campus, during the Spring of 2017. The primitive objective of the project was to study and extend the knowledge in understanding JavaScript web development stack while implementing a simple blog platform with technologies.

I would like to express my thankful and respectful gratitude to Mr Lauri Pirttiaho for the all the help and support in both technical and documentation parts of the project.

Oulu, 12.4.2017

Khoa Bui

CONTENTS

ABSTRACT 3
CONTENTS 5
VOCABULARY 7
1 INTRODUCTION 9
2 TECHNOLOGICAL INTRODUCTION 10
2.1 MongoDB 10
2.2 Redis 11
2.3 NGINX 11
2.4 ReactJS and Flux Architecture 12
2.5 SCSS and SMACSS 14
3 INITIAL SETUP 16
3.1 Remote Server Setup 16
3.1.1 NGINX 17
3.1.2 Let’s Encrypt. 18
3.1.3 PM2 20
3.2.1 Vagrant and Virtual Box 22
3.2.2 Project folder structural setup 22
3.2.3 Setup version control 23
3.3.1 Webpack 24
3.3.2 Utilization of modern technology with SCSS and ES6 25
3.3.3 Basic folder structure: 26
4 BACKEND DEVELOPMENT 27
4.1 INITIAL SETUP AND STRUCTURE 27
4.1.1 Babel compiler setup 27
4.1.2 ExpressJS application structure 28
4.2 ROUTING 29
4.2.1 Response 29
4.2.2 Access control 30
4.2.3 Middleware: 31
4.2.4 JSON Web Token 32
4.2.5 CRUD application: 33

4.2.6 Naming convention 34
4.3 DATABASE 35
4.3.1 Introduction 35
4.3.2 Schema 35
4.3.3 Schema Reference 37
4.4 EVENTS 38
4.5.1 Static content caching 39
4.5.2 Database caching 40
4.5.3 General strategy 40
4.5.4 Cache management 41
5 FRONTEND DEVELOPMENT 44
5.1 ReactJS and Flux architecture 44
5.1.1 Frontend Routing 44
5.1.2 Components 46
5.1.3 Flux architecture 49
5.1.4 Markdown Editor 51
5.2 Style Implementation 53
5.2.1 Application structure 53
5.2.2 Layout 55
5.2.3 Browsers support: 57
6 CONCLUSION 59
REFERENCES 60

VOCABULARY

Listed below are the abbreviations used in the whole document and their respective full form.

Frontend	the development of presentation layer which contains mostly user interaction
Backend	the development of data access layer which contain database and business logic defined by owner
JavaScript	the programming language which is used in this application, both frontend and backend
CSS	Cascade Styles Sheet is style programming language which allow developer to shape the user interface of a web application
CSS3	the latest version of CSS
HTML	Hyper Text Markup Language, a programing markup for frontend side to instruct how the user interface show appear.
MongoDB	object-oriented database, which is classified as NoSQL (None Sequel Database)
Redis	An open-source, in-memory structure store, used as database.
Vagrant	an open-source program used for virtualization
WordPress	an open-source blogging platform
PHP	a programming language in backend development
MySQL	an open-source relational database structure
Webpack	an open-source web bundler which bundling all static content in frontend development

ReactJS	frontend JavaScript framework developed by Facebook Inc.
DOM Document Object Model	a programming interface for HTML, XML and SVG documents
ITCSS	a methodology to develop CSS and their preprocessor language
SCSS and LESS	open-source programming languages for higher level of CSS which later compiles to CSS
JSON	JavaScript Object Notation
HTTP	Hypertext Transfer Protocol
UI	User Interface
MIT	Massachusetts Institute of Technology
Git	Version Control Software
Github	a web-based Git repository hosting service
NPM	NodeJS Package Manager
ES6	EcmaScript Language Specification standard

1 INTRODUCTION

In recent years, the development and accessibility of the Internet have been dramatically increased. Roughly forty percent of the world population has access to the internet service (1.). More and more daily access to the internet has provoked the increasing natural demand to share contents and ideas to the world wide web. Among the publishing platforms, WordPress is known for being the most popular publishing tool (43.).

WordPress, an open source platform to publish and manage content, is mostly based on web (They did release the Calypso desktop program, which is based on JavaScript as a version for their admin panel (2.)). In a web application, WordPress, despite of its popularity, uses a traditional LAMP stack (which comprises of Linux, Apache, MySQL and PHP) for the backend and the frontend is flexible depending on the customization of a developer and a user.

The LAMP stack is a very affordable and efficient technology (21.). However, in the raising of new technologies, this thesis proposes a way to create and study a platform for publishing a blog platform with JavaScript both frontend and backend.

The objective of this project is to demonstrate a new workflow for JavaScript development in the web development and put the theory into building a blogging platform system. The application allows a user easily publish their story and post it via a web-based admin panel. It also allows a developer to maintain the application at ease. The application has a rich frontend workflow which is built with Webpack, PostCSS, and ReactJS. It uses ITCSS as the methodology behind the CSS development. In the backend, the application applies the MongoDB as a persistent mechanism, and ExpressJS as the framework based on NodeJS. It also uses Redis as a caching strategy for a better performance.

2 TECHNOLOGICAL INTRODUCTION

The technologies used in this application are quite new and trendy. It will clarify the thesis if the technologies are quickly introduced.

2.1 MongoDB

MongoDB is one of the most famous member in the family of NoSQL (40.). The raise of NoSQL is due to the demand gathering and storing massive information every day and and the fact that the data keeps growing not only horizontally but also vertically, which is quite unpleasantly to develop for a traditional SQL database to implement and update (41.). NoSQL has provided a wide range of databases,both structured and unstructured types. The flexibility and diversity they offer for many implementation is very promising (22.).

Initially released by MongoDB Inc in the year 2009 (3.), MongoDB is the most famous solution among NoSQL. MongoDB is classified in document oriented database types, which contain the NoSQL database type which stores, retrieves, and manages document-oriented information. MongoDB is classified as the same group with SimpleDB (Amazon), RethinkDB, CouchDB (Apache), and DocumentDB (Microsoft) (4.).

Currently, MongoDB Inc supports eight different products for MongoDB: MongoDB Server, MongoDB Atlas, MongoDB Cloud Manager, MongoDB Ops Manager, MongoDB Drivers, MongoDB Compass, MongoDB Shark Connector, and MongoDB BI Connector (5.).

Differentiating from traditional databases such as MySQL or SQL server, MongoDB does not have any relational constraints explicitly in across their documents. The relationship can be defined by a user by their own server programming languages. This feature helps a database structure to be flexible for alteration in the future because of the need of rapid data development in the technical world. (41.)

2.2 Redis

Categorized in a key-value database amongst the NoSQL family, Redis also known as the REmote DIctionary Server (6.), is a database structure which is open-source, networked, in-memory, and stores keys with an optional durability. Redis, simply put, maps keys to a value. However, more than a regular memory storage, it also supports other type rather than sorely strings (7.):

- Lists of strings
- Sets of strings (collections of non-repeating unsorted elements)
- Sorted sets of strings (collections of non-repeating elements ordered by a floating-point number called score)
- Hash tables where keys and values are strings
- HyperLog: Logs used for approximated set cardinality size estimation.

With these features, Redis is usually used as a caching mechanism in a web application due to its speeds, and durability.

Redis currently supports many programming languages, such as NodeJS, Java, C/C++, PHP, Python, Scala, Ruby, Go, and Haskell.

2.3 NGINX

NGINX is a free, open-source, high-performance HTTP server and a reverse proxy, as well as an IMAP/POP3 proxy server. NGINX is known for its high performance, stability, rich feature set, simple configuration, and low resource consumption (8.).

Nowadays, many web applications, especially big players in social media application (46.), have to coopetic with the gigantic web traffic every day. NGINX, is designed to tackle the scalability issue and concurrency problem in a web application. Brilliantly storing the notorious C10K problem, NGINX has demonstrated the power to harness a high traffic web application (8.).

Through times, a web application has changed dramatically. It has transformed from a monolithic application to a dynamic web application which handles scalability problems (Figure 1.)

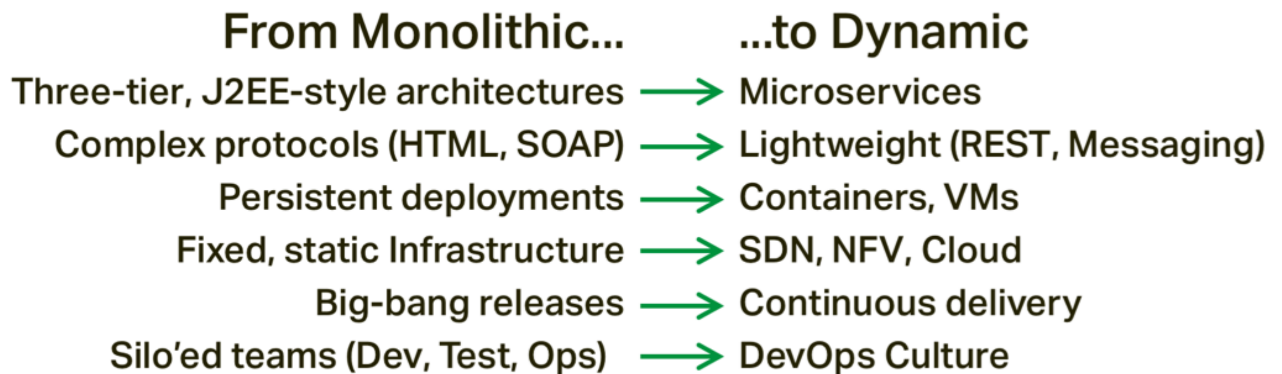

FIGURE 1. Monolithic and dynamic application transformation

2.4 ReactJS and Flux Architecture

ReactJS is an open-sourced JavaScript library, developed and maintained by Facebook Inc. Jordan Walke, a software engineer at Facebook, is credited for creating ReactJS as a component library for PHP (9.). ReactJS was first deployed in Facebook newsfeed in 2011 and later in 2012. It is also used in Instagram. Until 2013, ReactJS has been published and open-source at JSConf. (9.)

ReactJS has introduced the concept and implementation of Virtual DOM (9.). Virtual DOM is an innovative improvement in DOM. It acts as an abstract layer for DOM, which creates a drafted version of DOM tree for manipulation and later renders the actual DOM tree. The Virtual DOM approach provides a tangible solution for rich web application as it implements a constant and complex DOM update. By implementing Virtual DOM, ReactJS has optimized applications both the performance and maintaining and developing of applications(42.).

Differing from most JavaScript frameworks, ReactJS is considered as the view part of application and it is able to adapt to most framework such as Backbone(MVC), Knockout(MV*) and AngularJS(MVV) to boost their speed of existing applications.

Another famous feature comes along with ReactJS is JSX syntax. JSX is an XML-like syntax. A hybrid syntax allows a developer to blend JavaScript and HTML syntax together without touching an actual template. ReactJS supports both the

vanilla JavaScript syntax and the JSX syntax. Therefore, in most cases of ReactJS application, there are no HTML files, except the HTML files that the ReactJS components are mounted into.

Despite the flexibility of ReactJS to adapt to any environment and framework, Facebook developers also demonstrate their own architecture: Flux. Flux refused to be defined as a framework but rather a pattern. Designed to work with an unidirectional data flow in the ReactJS application, Flux minimizes the data flow round trips in a web application (10.).

Ideally, a web application would be resolved around the single flow structure as shown in figure 2 below:

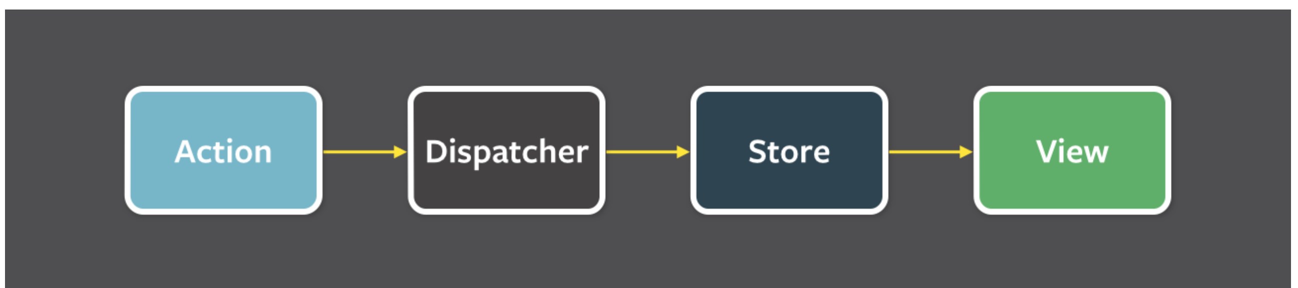

FIGURE 2. Regular frontend application flow

However, as the application grows bigger and more complex, it is inevitable to have a scenario where: the view triggers more than one action and recursively, the actions also happen any time in the view part and ultimately, the application flow will be generally depicted as shown in figure 3 below:

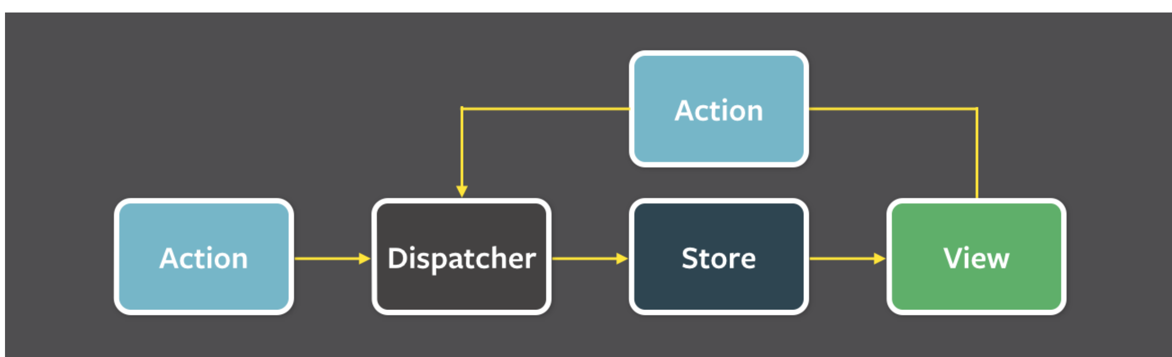

FIGURE 3. Flux application glow

Flux structures are composed of Action, Store, Controller-View and Dispatcher. Every Flux application has a single dispatcher to manage the data flow. It is the registry of callbacks into the stores and it does not have any business logic. Similar to the Model in the MVC architecture, Stores contains data and logic of application and every time the data of the application has changed, it emits a change event to a controller-view. The dispatcher API provides methods that allows us to dispatch an event to the stores and pass a payload of data as the parameter, which an action can be called. The creation of action is wrapped into a semantic helper method which sends the action to the dispatcher. The controller-view usually contains the built-in methods in ReactJS library, such as setState, forceUpdate.

The Flux pattern has many implementations, but in this project, Alt library is utilized.

2.5 SCSS and SMACSS

CSS is a very essential component in web development. CSS extends the implementation of web application via predefined rules which instruct a browser to render to an end user. Through many upgrades and implementation, in the newest version, CSS is quite extensive to implement frontend features. Nevertheless, due to the lack of programming language features, such as inheritance, variables and basic loops, it is not yet considered a programming language and regularly, the development encounters unnecessary duplicated code blocks.

Fortunately, SCSS and LESS are the most popular solutions to address these issues (44.). Categorized as a pre-processor CSS, they transform the abstract syntaxes of CSS to actual CSS codes. Amongst the two candidates, according to the author’s experience, SCSS is preferred as SCSS is much faster and because it has many plugins and libraries, famously mentioned as Compass. SCSS actively supports many basic features of programming languages:

- Variables: SCSS supports different variables both in the predefined and undefined form.

- Nested: SCSS supports a nested CSS syntax block, creates a clearer structure to CSS and implements many CSS design philosophies, such as ITCSS, OOCSS, SMACSS.
- Mixins: Mixins helps to prevent a DRY (Do not Repeat Yourself) problem in CSS as they create defined CSS code blocks, which can be later reused across the application. Mixins also support arguments.
- Loops: SCSS supports three basic operations with loops: while, for, each.
- Selector inheritance: SCSS supports the DOM inheritance via a syntax @extend syntax.

In our application, the thesis will introduce and apply the concept of CSS design philosophy. CSS design philosophies are architecture mindsets for designing CSS applications. There are many CSS design philosophies, such as SMACSS (Scalable and Modular Architecture for CSS), OOCSS (Object Oriented CSS), and ITCSS. In this project, the thesis decides to follow the SMACSS design philosophy.

SMACSS designs the application with the folder structure. It comprises of five basic folders (20.):

- Base: This folder contains the styles that define the basic appearance of application such as body, HTML, link elements.
- Layout: This folder contains rules that define how pages instructs themselves part by part
- Module: The main purposes of this folder is the reusable, modular parts of our design. They are the callouts, the sidebar sections, the product lists.
- State: This folder defines how the UI will be according to different state of applications
- Theme: Finally, this folder defines the customization that heavily relies on design of a specific application.

3 INITIAL SETUP

3.1 Remote Server Setup

Vagrant has handled most of the synchronization between a server and a local environment. Nevertheless, in order to deliver the web application toan end user, the remote server also required to have some special configurations.

First and foremost, to deploy the application to the public, it is very important to choose a Virtual Private Server (VPS) service that would fit the author’s need. There are literally several VPS service providers that meet the demands of a scalable, maintainable, secured and easily controlled via both a panel and command line access. However, after a deep research on both features and price, Digital Ocean (https://www.digitalocean.com) was chosen as the service provider in this application.

Also, in the matter of this application, the author also purchased a domain name: cungdangxua.com.

Next, the essential software in the application should be installed and tested properly. The sequence of actions is enlisted below:

- Installing NodeJS, MongoDB, Redis.
- Installing Git and setup authentication means to automatically check whenever the project is updated.
- Installing NGINX and testing if it is working correctly.
- Installing PM2 as a NodeJS process manager.
- Purchasing a domain, in this case point its DNS toward the Digital Ocean, so it can be used.
- Setup a secured connection(HTTPS) for the server with Let’s Encrypted.
-

To go to a further detail in each step would take redundant efforts while some of the tasks are trivial. Hence, the thesis would only be specific in the complicated ones.

3.1.1 NGINX

NGINX is a free, open-source, high-performance HTTP server and a reverse proxy, as well as an IMAP/POP3 proxy server. NGINX is known for its high performance, stability, rich feature set, simple configuration, and low resource consumption (8.).

After installing NGINX and checking if NGINX is working with the start and stop command line in remote instance, then the application has to configure the NGINX application setting so that it can be associated with the existing domain and the NodeJS application. For starter, NGINX settings reside in two directories: /etc/nginx/sites-enabled and /etc/nginx/sites-availables and both of these directories need a root permission to edit and update.

To associate a domain to the application, one should duplicate the old configuration blocks to a new one and name it as the domain name, in this case cungdanxua.com.

```
sudo cp /etc/nginx/sites-available/default /etc/nginx/sites-available/cungdangxua.com
```

Then update the configuration block must be updated with:

```
root /var/www/public;

server_name cungdanxua.com www.cungdanxua.com;

location / {

                        proxy_pass http://46.101.97.178:8080;
          proxy_http_version 1.1;
          proxy_set_header Upgrade $http_upgrade;
          proxy_set_header Connection 'upgrade';
          proxy_set_header Host $host;
          proxy_cache_bypass $http_upgrade;
                        # First attempt to serve request as file, then
                        # as directory, then fall back to displaying a 404.
                        try_files $uri $uri/ =404;
          }
```

Also, a symbolic link between the files must be created:

```
sudo ln -s /etc/nginx/sites-available/ cungdanxua.com /etc/nginx/sites-enabled/ cungdanxua.com
```

In this state, it is pointed as the domain. After the setting, the NGINX service is restarted, then the changes can be applied correctly.

3.1.2 Let's Encrypt.

Let's Encrypt is a free, automated, and open certificate authority developed and maintained by the non-profit Internet Security Research Group (ISRG) (11.).

Let's Encrypt is free and open source and actively develop. It is a versatile tool for this project to create an encrypted certificate, which can be used for the application to run on the secured protocol (HTTPS).

Following the official documentation, the remote instance is updated with a new software. However, once the software is installed, it is also advised to properly configure the server to apply to the encryption to an HTTP connection.

Let's Encrypt is equipped with a great plugin called Webroot. It is basically passed as an option to the root folder of the application. Therefore, it can generate to necessary files for the encryption parameters.

```
certbot-auto certonly -a webroot --webroot-path=/var/www/public -d example.com -d www.cungdanxua.com
```

After running this command line, it will generate four essential files:

- cert.pem: Your domain's certificate
- chain.pem: The Let's Encrypt chain certificate
- fullchain.pem: cert.pem and chain.pem combined
- privkey.pem: Your certificate's private key

These files should be relocated to another location, ideally, to the in /etc/letsencrypt/archive folder. They should be linked to the /etc/letsencrypt/live/cungdanxua.com folder. And to be extra careful with the application, another command can be utilized:

```
sudo openssl dhparam -out /etc/ssl/certs/dhparam.pem 2048
```

The command is a 2048-bit Diffie-Hellman group encryption, which takes a few minutes to complete the process.

After creating the encryption files, they should also be integrated with NGINX. Again, the previous NGINX file is opened and commented out the original 80 port block in the NGINX configuration.

```
#       listen 80 default_server;
#       listen [::]:80 default_server ipv6only=on;
```

Since secured ports of application are run in the 443 port, so one can base on that and set up a new configuration:

```
# SSL configuration

listen 443 ssl default_server;

listen [::]:443 ssl default_server;

ssl_certificate /etc/letsencrypt/live/cungdanxua.com/fullchain.pem;

ssl_certificate_key /etc/letsencrypt/live/cungdanxua.com/privkey.pem;



ssl_protocols TLSv1 TLSv1.1 TLSv1.2;

ssl_prefer_server_ciphers on;

ssl_dhparam /etc/ssl/certs/dhparam.pem;

ssl_ciphers 'ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-DSS-AES128-GCM-SHA256:kEDH+AESGCM:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-DSS-AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-DSS-AES256-SHA:DHE-RSA-AES256-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-
```

```
SHA256:AES256-SHA256:AES128-SHA:AES256-SHA:AES:CAMELLIA:DES-CBC3-SHA:!aNULL:!eNULL:!EX-PORT:!DES:!RC4:!MD5:!PSK:!aECDH:!EDH-DSS-DES-CBC3-SHA:!EDH-RSA-DES-CBC3-SHA:!KRB5-DES-CBC3-SHA';

ssl_session_timeout 1d;

ssl_session_cache shared:SSL:50m;

ssl_stapling on;

ssl_stapling_verify on;

add_header Strict-Transport-Security max-age=15768000;
```

After the configurable block is updated, it should be wise to test the syntax error with the command line: sudo nginx –t and to restart with the NGINX service after checking successfully.

FIGURE 4. Secured appliction with Let's Encrypt

3.1.3 PM2

PM2 is a “production process manager for Node.js applications with a built-in load balancer” (12.). The module helps the application with the availability by supporting a mechanism to reload it to eliminate downtime and to facilitate common system admin tasks (12.).

Availability is an important aspect in web application and it should always be seriously considered in the software development and maintenance. Nevertheless, it is only logical to have a backup plan for such scenario when the application has failed to operate. One simple strategy for that is to restart the application when it encounters critical errors. Fortunately, PM2 provides with a simple API to implement this strategy (12.).

Apart from that problem, PM2 also supports the accessible means to control the processes with NodeJS applications. In the scope of the application, it is compulsory to make a configured JSON file to define the service. Therefore, PM2 can have a constant state of operation.

```
{
  "apps": [{
    "name"       : "application",
    "script"     : "./build/index.js",
    "args"       : " --production 80",
    "watch"      : true,
    "env": {
      "NODE_ENV": "development"
    },
    "env_production" : {
      "NODE_ENV": "production"
    }
  }]
}
```

3.2 BACKEND ENVIRONMENT SETUP

Local development environment and remote servers are usually different in many aspects from the operating systems to the version and the compatibility of the installed softwares. Therefore, it is quite time consuming to setup the web application in the production environment once the local application has been developed and tested. To optimize the time and effort to develop and maintain the software, it is wise to create a sandbox for the remote instance to synchronize as much as one can to the local development process.

First and foremost, it is established in the beginning that the application will be developed via a defined list of restrictions in a remote server:

- Operating System: Ubuntu 14.04 LTS (Long term support).
- Programming Language: NodeJS 6.0.5
- Database: MongoDB and Redis
- Nginx support

The fundamental restrictions in both a remote and a local machine has to be matched in order to minimize the differences and unexpected issues that crash the application when deploying to the remote machine.

Already demonstrated before in this report, Vagrant and Virtual Box are responsible for creating and maintaining the application state and hosting the application locally.

3.2.1 Vagrant and Virtual Box

In the scope of development, Vagrant and Virtual Box are locked to version 1.8.1 and 5.0.16 respectively.

The instruction and development of Vagrant and Virtual Box in the official websites can be found in the online official documentation (https://www.vagrantup.com/docs and https://www.virtualbox.org/wiki/Documentation). However, to speed up the development process and remove any unnecessary hassles, it has been agreed to use the online tool to create a development environment with puphpet.com.

Puphpet.com is an online GUI tool which supports multiple configuration options for Vagrant, it eradicates the redundant time for a newcomer to dig in and study the Vagrant development environment.

3.2.2 Project folder structural setup

After the local environment initialization, the project can be safely started with the development process.

In the NodeJS application, NPM command utilities can be used to manage the application and packages. The application as a big module for NPM can be considered. The application uses the command `npm start` to start the procedure automatically.

Expectedly after the process, the application got a `package.json` file, which contains the required information that indicates the state of the application which is used to maintain the synchronization between its local and remote environments. The

information stored in the package.json file can be used later in the remote server to recreate the installed modules that have been used in the local environment. Thus, it is much easier for a deployment than transferring the entire local development project to the remote environment. To further detail, the thesis will have separate sections for the backend development and frontend development study later in this report.

3.2.3 Setup version control

In order to maintain and have a flexible ability to keep in check the application, the project will use version control softwares: Git and Github.

Git's installation and setup is followed strictly in the official guideline installment on their website (https://git-scm.com/doc).

In this project, the code is also updated regularly in the Github website for the code management. It will act as a cloud and it will be integrated to the remote server easily. There are two basic repositories for the projects:

- Vagrant Box: https://github.com/thomasbui93/thesis-vagrant
- Main project: https://github.com/thomasbui93/thesis-blog

In the first repository, the code is basically simple and does not need regular update and maintenance. It will not be divided into multiple branches, at least in the scope of this project.

In the second repository, the application is developed actively, therefore, it is only logical to create three development branches:

- master: The production version of the application. It will be used in the remote instance.
- frontend: All of the frontend development will concur in this branch, ReactJS components and SCSS styles will happen here.
- backend: All of the backend development will be deployed in this branch.

3.3 FRONTEND ENVIRONMENT SETUP

The Frontend development has evolved rapidly in the past five years, from a very basic DOM manipulation, such as jQuery and Prototype to a component based library ReactJS, AngularJS, VueJS. Additionally, many new frontend workflows have been demonstrated to the technology world how elegant it can be.

In the scope of this thesis, the application will establish the ideal workflow and folder structure for the SCSS and ReactJS application.

3.3.1 Webpack

In many frontend applications, JavaScript and CSS are included in HTML files as multiple separate files. This approach has a huge drawback because when a user uses a bad internet connection, the web page is unavailable until connections are ready and loaded. Therefore, it is quite bad for the user experience.

Nowadays, it is much more versatile than that.

One popular solution is RequireJS, which considers every file in its system as a module and uses dependencies to decide which is the actual order for loading the important files for a web page. RequireJS is originally used in backend (NodeJS). But it is very flexible. The same concept is applicable to the frontend, too.

Apart from a module loader workflow, there are many workflows that have followed different approaching, such as bundling (Webpack and Rollup).

In this project, the project will follow the methodology and workflow which is recommended by Webpack.

Webpack simplifies the workflow by constructing a dependency graph of the application and bundling the workphases in the right order. Webpack can be configured to optimize the code and it also runs a development server that reloads the code without a page refreshing (13.).

The core of the application frontend development workflow resides in the file webpack.config.js. This file is in the root folder of this application. Webpack will reply to the configuration of this file to define and shape this project into a working

application. Generally speaking, webpack.config.js is a NodeJS module which outputs the configuration object, which contain necessary parameter for configuring the frontend application.

Beside the handy configuration, Webpack also provides with an ecosystem of plugins that provide many utilities to frontend development. In this project in particular, the project uses:

- Autoprefixer: Auto prefix for a aCSS stylesheet, fallback for an old browser and cross browser issue.
- babel-core, babel-eslint, babel-loader, babel-preset-es2015, babel-preset-react: Babel ecosystem for ES6 fallback
- CSS-loader, sass-loader, style-loader: Packing the CSS style into one file.
- cssnano: Minify CSS.
- eslint, eslint-loader, eslint-plugin-react: eslint for this project.
- extract-text-webpack-plugin: support extracting inline CSS into a separate file.
- html-webpack-plugin: injects JS and CSS into a target HTML file.
- postcss-loader: Postcss supports for this project, Postcss is the experimented CSS style for CSS4, which can be a fallback to a widely used CSS3 version.

3.3.2 Utilization of modern technology with SCSS and ES6

As already demonstrated in the earlier chapters, SCSS and ES6 are used officially in this project development. Despite the fact that both technologies are not widely supported in browsers however, with Webpack, both can be used normally and later they will be transformed into the average and widely supported features in browsers (45.).

Babel will be used to transform an ES6 JavaScript, JSX syntax into a normal JavaScript syntax, so that a normal browser can support the application.

The command line below will monitor for file changes in the frontend application and compile the code:

```
webpack --watch --progress --profile --colors --optimize-occurence-order --optimize-dedupe
```

3.3.3 Basic folder structure:

Without considering further the business logic of our application, the frontend part of this project can be configured.

In the frontend, all of the static files including SCSS, JavaScript, JSX and images will be hold under an 'src' folder. This folder is the center of the frontend development and the code in this folder will be compiled into a working application by Webpack. There are two folders in the src folder, one is scss and other is app. They hold responsibilities for the style and JavaScript logic, respectively.

With SCSS development, the application will centralize all SCSS files into one file `src/scss/main.scss`.

- This file will be the placeholder for all other SCSS files. The content and logic of other SCSS files will be imported into this file.
- Each files should therefore serve for a specific purpose, except for those holding general rules. There should never be a file holding many style rule for different components.
- Variables should follow the lisp-case, in which every logical word should be separated by a hyphen.

On the JavaScript side, since this application will implement the Flux architecture, therefore the basic skeleton of the project can also be set from the beginning.

A Flux application, generally can have four types of classes: actions, components, store, containers. Therefore, the application should have four folders specializing and associating with their functionalities, respectively. The project also will have a folder which holds the utilities function and settings variables that are used globally across the application.

And in this case, JavaScript will have a focal point in the src/app/index.js file, which holds the general logic of this application.

4 BACKEND DEVELOPMENT

4.1 INITIAL SETUP AND STRUCTURE

The backend of the application will be written entirely in the NodeJS programming language. However, developing and maintaining the application from the scratch would be rather complicated. Thus, the application will be built upon a framework called ExpressJS and the entire architecture of this application will resolve around the recommended structures for ExpressJS. Also, as mentioned in the earlier chapter, the backend will also be written entirely in EcmaScript 2016 and later it will be compiled to compatible code in the backward compatible JavaScript.

4.1.1 Babel compiler setup

To start the backend development process, one shall start with the Babel compiler integration setup in package.json. It should be noted that Babel only belongs to development process and once the server operates, the code will be interpreted as EcmaScript 2015. Therefore, all modules belonging to the Babel integration will assign to the devDependencies of the package.json file.

As Babel also involves the frontend development workflow, therefore, the application installs all of the Babel plugins for the frontend: babel-core, babel-eslint, babel-loader, babel-preset-es2015, babel-preset-react, node-babel.

Once they have been installed in the local development, the developer shall create the script for transforming all of the development backend script to the production script and also for creating script for running the application in the development and production mode. Respectively, they are aliased as:

- transform: the command converts the EcmaScript 6 code to compatible code for NodeJS.
- production: the startup backend application for production, mostly used in the remote server instance.
- start: startup the backend service for the development process.

TheNPM (Node Package Manager) software allows a user to operate a custom and complicated command via their aliases in the package.json file by simply writing: npm run $_alias_$, where $_alias_$ represents the alias in the package.json file.

4.1.2 ExpressJS application structure

As mentioned earlier in the report, all of the backend development code is located inside the folder 'server' and the hub of the application will centralize around the file server/index.js. ExpressJS is an elegant backend JavaScript framework, which is very flexible and agile enough to let a user to design their own architecture of application. However, in the scope of this project, the application will follow the recommended guideline of ExpressJS author to construct the application.

In the ExpressJS application, in particular, all of its routing is registered and configured to be designed in index.js. However, traditionally, the controller for the routing will be categorized in a specific folder, ideally separated by its routed name. In this project, the controller for routing is established in a folder 'server/routes'. To be more specific, 'server/routes' not only contains the controller and models but also the middleware in the application, which will be demonstrated in detail later in the report.

In term of persisting a database in the application, MongoDB has already been demonstrated and appointed as the official database for the project. To get MongoDB integrated with the ExpressJS, there are many options with all their pros and cons. After researching and testing with the actual scenario implementation, it was decided that MongooseJS (http://mongoosejs.com) would be best suited for the application as it offers many layers of coupling and abstractions to MongoDB for developing the ExpressJS project. With MongooseJS, in this application, each model corresponding to controllers will be located in the same folder as that controller, thus it will be easier for the code management.

One of the significant parts and one of the most useful feature of NodeJS is an event system. It will be deeply explored and used in the project. The events configuration and development will be held under the folder 'server/events', where it will be categorized depending on what purposes it serve. Excluding the database,

every application requires a customized configuration and this will be consistent throughout the application. Without any exception, this project also holds its own customized variables in a folder named ‘server/config’. In this folder, many variables are established in a way that later they can be used via importing and so that the code would be much cleaner due to the prevention of repeating unnamed constants. In this application, there are many repeated used variables:

- Database credential and configuration objects, including both Redis and MongoDB
- Authentication token
- Authentication configuration settings.
- Static files settings.

4.2 ROUTING

After setting up the environment, it is logical to setup and create the fundamental logic for this application: routes.

Routing refers to determining how an application responds at a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on) (34.). In a web application, routing is the fundamental entry point for the user to access the application. A normal user can access the application via entering the URLs to their web browser so that the server can send responses with HTML to display in their devices. Additionally, routing also plays very important roles in the internet service. In this application, routes are registered in the file: routes/index.js, complete code of the file will be included in the appendix 1.

4.2.1 Response

Routing in this application, especially the backend routing protocol will only response with the JSON object with the exception of the index URL, as it will serve the most basic frontend static assets.

JSON (JavaScript Object Notation) is "a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate" (14.).

Compared to normal HTTP responses such as a full page response, which renders all HTML and RSS feeds, JSON is the most lightweight response among them all. Firstly, JSON is originated from the purpose of sending and receiving data for software development, JSON has a slight advantage compared to XML as it is lighter and easier to manipulate between frontend and backend sides, also it fits right into this system because it is derived from JavaScript and it can simply parse by the engine in both the frontend and backend. The HTML full rendering is quite heavy for this web application and it would be a block to the performance of the web product, since data can be transferred and requested by AJAX. Therefore, a full rendering HTML would not fit into the context in the application, in which the frontend will be designed in the Single Page Application and there will be mainly the JSON response. Despite, JSON is a lightweight datatype, however, recently, there is a new datatype , MessagePack (http://msgpack.org) in development and it has already been installed in many applications. It is the fastest and lightest datatype in a web application. However, MessagePack does not belong to the HTTP response field because it requires decoding and encoding in both end. JSON does not claim to be the best in the HTTP response, because every technology has its own advantages and disadvantages, but in the scope of this application, JSON is a perfect fit for developing the technological stack.

4.2.2 Access control

In order to maintain the authenticity of a software product and especially a network application, the application requires a proper authentication and an access control mean to prevent illegal attempts to temper with the user content. The Access control is a common concept not only in information technology but also in other profession as well. Generally, the access control (AC) is a selective restriction of access to a place or a resource. The act of accessing may mean consuming, entering, or using. A permission to access a resource is called authorization (15.).

As stated in the original idea and statement, the application would need a private zone for the user to edit and update their content and so that other public users would not be able to temper this application unless they would be authenticated. Therefore, in this simple situation, the application has two types of user, one is an admin with all the rights to the application including: view, update, remove and publish. Normal users have public access, and they can only view what has been updated and published. The restriction does not mean that the authentication method is ridiculously complicated. It means that the method should be simple but it would be impossible to fake the user's identity.

The methods of maintaining these rules will be handled below in the next two sections: middleware and token authentication.

4.2.3 Middleware:

In this application, middleware acts as a bridge between the user's request and application. Middleware processes the requests and provides a response in an efficient way.

ExpressJS is equipped with an implementation of middleware. It is simple and yet very useful and importantly very flexible to manipulate and develop. In ExpressJS, in particular, middleware is the functions that create a layer between requests and responded objects of the controllers. Generally speaking, ExpressJS is considered to be a giant middleware and it is composed of a series of middleware.

Middleware functions can perform the following tasks: (16.)

- Executing any code.
- Making changes to the request and the response objects.
- Ending the request-response cycle.
- Calling the next middleware function in the stack.

Middleware in this application will render a layer between the controller. The more abstract the middleware the lower priority it is considered to have and the one with the lower priority. It will be executed first. In ExpressJS, in every middleware,

it would be the best practice that it will end with the function next if it does not return a response object.

Middleware adapts properly into this system to enforce the authentication methods in the application. The scenario requires every HTTP request that crosses the admin panel to be authenticated. Therefore, the admin application domain which is under the POST/PUT/DELETE methods will require authentication. Every request matches the URL pattern ‘api/*’ with POST/PUT/DELETE will go through the middlewares to check if the request is legitimated.

The ExpressJS middleware syntax is quite flexible so that it will accept the URL patterns that would match the actual URL. Also it supports a regular expression as a parameter, thus a developer can simply put ‘api/*’, and it will match all needed URL.

However, the application cannot store the credential of the user in the frontend section and use it as a parameter to authenticate multiple times. To enforce that line of security the JSON web token must be used.

4.2.4 JSON Web Token

The JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure. It enables the claims to be digitally signed or integrity protected with a Message Authentication Code(MAC) and/or encrypted. (17.)

JWT has already been implemented in different programming languages (18.), which makes it accessible in multiple platforms and applications. What people praise is the simplicity and the security it has created for the web platform.

Authentication is probably the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token.

Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains (18.).

JWT only requires a user to sign in only one time and nothing is more irritating than a multiple sign in every time a user needs to do their work.

In the scope of this application, it would require a well-known module for JWT for NodeJS: jsonwebtoken (https://github.com/auth0/node-jsonwebtoken). And with this module, it is not complicated to implement the check function:

```
const URL = './server/routes/user/confidential.json';
const JWT = "./server/routes/user/JTW.json";
import {checkAuth} from './user/user.controller';

export default (req, res, next)=>{
    let token = req.body.token;
    if(token===undefined || token === null){
        token = req.query.token
    }
    if(req.method !=='GET'){
        checkAuth(token).then((response)=>{
            next()
        }, (error)=>{
            console.log(error);
            res.status(403).json({
                message: "not authorized"
            });
        });
    } else {
        next();
    }
}
```

4.2.5 CRUD application:

In the heart of our application, the controller provides many endpoints for the frontend to manipulate and develop. In this scope of application, it was chosen to implement the CRUD concept in the term of accessibility.

The term CRUD was likely first popularized by James Martin in his 1983 book Managing the Data-base Environment (19.), but later, CRUD is widely known and implemented across the web realm. CRUD is an acronym, which refers to four basic operations of a persistent storage: create, read, update and delete, respectively.

In this application, particularly, there are five actual end points:

- Creating an entry
- Reading all entry of a collection
- Reading one entry in a collection
- Updating one entry in a collection
- Deleting an entry

Each of these endpoints correspond to an actual controller for the scalability and code management. As sample of the CRUD concept will be demonstrated in the appendix 2.

4.2.6 Naming convention

Routing endpoints in this application will follow the CRUD fundamental implementation. Except the especial controller for the cache control and user management, the URL system should follow the same pattern:

- Group routing should be located and configured in the routes/index.js:

```
app.all('/api/*', checkAuth)
app.use('/api/posts', postController);
app.use('/api/categories', categoryController);
app.use('/api/images', imageController);
app.use('/authenticate', userController);
app.use('/', allController);
app.use('/api/cache', cacheController);
```

As put in the above snippet, the namespace and root controller are registered in file routes/index.js and exported as a module in ExpressJS for a later usage in the routing configuration in ExpressJS. For example, all of the URL prefixed as 'api/categories' are associated with the module categoryController and in the categoryController. There are methods to define the five basic endpoints operation.

And for the record, as the backend mostly serves as a service point, it can use the prefix 'api' to all none index access in this application.

- Creating an entry: POST/PUT method to 'api/root_path', where root_path is the URL fragment which is defined in index.js file.
- Reading all entry: GET method to 'api/root_path', where root_path is the URL fragment which is defined in index.js file.

- Reading one entry: GET method to ‘api/root_path/:id’, where root_path is the URL fragment which is defined in index.js file and id is the proper identification needed for the root_path to differentiate from each entry.
- Updating one entry: PUT method to ‘api/root_path/:id’, where root_path is the URL fragment which is defined in index.js file and id is the proper identification needed for the root_path to differentiate from each entry.
- Deleting one entry: DELETE method to ‘api/root_path/:id’, where root_path is the URL fragment in which defined in index.js file and id is the proper identification needed for the root_path to differentiate from each entry.

4.3 DATABASE

4.3.1 Introduction

As introduced earlier in the report, MongoDB has many advantages that can be harnessed and put into good use in this project. However, in order to integrate MongoDB and NodeJS or ExpressJS, it requires a rather complex and unyielding attempt which does not fit into the scope of this project. MongoDB in this application will use MongooseJS as the integration party to have a proper abstraction to this application database.

4.3.2 Schema

MongoDB is a very flexible application and it does not enforce the document structure. However, it would be wise for the application to define a standard database structure for the project. Fortunately, MongoDB provided such abstraction and MongooseJS provides a Schema object that defines how the structure of a collection would be and this Schema can be altered in the future development.

A schema in MongooseJS provides many options and types for an associated collection, and every schema object will define each field configuration. For example:

```
let Category = new Schema({
  createAt: {
    type: Date,
```

```
        default: new Date()
    },
    title: {
        type: String,
        required: true,
        unique: true
    },
    description: {
        type: String,
        required: true,
        unique: true
    },
    url: {
        type: String,
        required: true,
        unique: true
    },
    color: {
        type: String,
        required: true,
        unique: true
    }
});
```

In the above snippet, the Category schema has five primary fields and every field has its own definition. Every redundant field of a data object is saved into the database which will be stripped down and which leaves only the defined field in the Schema object. Each of the field has its own configuration object. For example, it is a string data type and it has to be present in the model object and it should be unique among its fellow documents. Nevertheless, the unique option is not natively supported in MongooseJS, but it has a plugin that implements the desired feature.

And after defining Schema, the schema object should be assigned to one MongoDB collection for it to work correctly.

```
export default mongoose.model('Category', Category);
```

4.3.3 Schema Reference

In the database development, quite often the application requires some sort of relation restrictions between collections, this is not an exception in this application.

For example, there is a scenario that a blog post that belongs to a category. A blog post and a category are represented by two collections: Blog and Category. It is possible to assign a field of Blog schema to be a category. However, this is not an optimized solution and rather a hazardous implementation of database because the database has been unnecessarily repeated twice and every time, it requires an update in one category and the application has to edit the object twice.

A schema reference has been introduced as a native implementation for the relation restriction between collections.

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Tom Benzamin",
   "address_ids": [
   ObjectId("52ffc4a5d85242602e000000"),
   ObjectId("52ffc4a5d85242602e000001")
   ]
}
```

The above snippet demonstrates a document that has a reference field address_ids. In this example, the address_ids field is an array that contains two elements of ObjectId, which are used for identifying the document in MongoDB. The two object identifications are belong to another collection. The configuration allows the developer to save a database space and prevent meaningless and redundant information to be save. Moreover, MongoDB allows the developer to populate the objectId to be a full feature document.

4.4 EVENTS

Famously known as one of the most defining feature in NodeJS, the event system in NodeJS is very important and widely used among the web applications (45, p.58). And practically, it has proved its usability and flexibility in this application.

NodeJS has a dedicated module for its event system called 'events'. Only by importing the module to the application, the developer can use all of its perks from then on. However, it would be wise to create a single event instance and use the same instance across the application.

```
import { EventEmitter } from 'events';
const eventEmitterInstance = new EventEmitter();
export default eventEmitterInstance;
```

Events can be emitted from anywhere in the scope of the application. And if there is a listener to that event, it can be handled properly.

```
export default (eventEmitter)=>{
    "use strict";
    eventEmitter.on(VISITOR.ENTER, visitorEnter);
    eventEmitter.on(VISITOR.WANDER, visitorWander);
}
```

In the above snippet, for two events VISTER.ENTER and VISTOR.WANDER, both have associated functions which will be called whenever these events are emitted.

This feature adds to the NodeJS ecosystem another extra layer for the usability and the decoupling of NodeJS application. It allows a user to put almost any customization to the existing code and to extend the horizontal scope of application by extending the listener of the operation it wishes to alter.

In this application, events involved as much as the backend codes and all the events and their configuration reside under the folder /server/events and of course, it will be exported as one in the index file.

And due to the complexity of the naming convention, it is decided to use a constant to name a convention. Every micro system of events has its own configuration file to hold the constant of their own.

In this application, in fact, events represent in almost every controller and operation to a database. Every action in this application will emit an event that would be caught and handled properly. The sample code of the events system will be held under the appendix 2.

4.5 CACHING

In every application, especially in a web application, usability is the utmost important factor in the development and fine tuning. And statistically, a normal user would give up before the web page has finished loading if the response time is more than 3 seconds (37.). There are multiple segments in a web application that affect the speed of an application and there are many solutions to address that issue, but in this chapter, the thesis will discover the tactic of caching.

The Caching concept in a web application is relatively the same in many other other software application fields. It stores the data of the component in a hardware or software, which has a better response time. In the future, therefore the speed is faster and this improves the application usability (38.). However, in a web application, caching has many layers and the more thorough and meticulous they are, the more performance wise the application will get.

In the scope of this application, however, there are static content caching, and database caching, which are explained below.

4.5.1 Static content caching

Static contents in a web application are usually constant data, which has not usually being updated when the development circle has been closed. Therefore, it would be safe to assume that in the production mode, every static content should be compressed and cached to improve the web application performance.

Express has provided a simple solution to compress the static content, including HTML, CSS and JavaScript files (39.):

```
app.use(express.compress())
```

And also the caching functionality is included in the ExpressJS as a solution:

```
app.use(express.static(__dirname + '/public', { maxAge: oneDay }));
```

In the above line, the static content will be stored for one day whenever the browser requests the application any static content.

4.5.2 Database caching

Previously in the report, the thesis already introduced Redis, an exceptional in-memory database which is based on a key and a value. Redis has many implementations and integrations to a web application, but the standing application of Redis is implementing a caching mechanism. The caching strategy of this application will be based on Redis as an alternative database for a caching response.

In this application scope, the application would want a system that would cache all of the post and category queries and responses to Redis and later it could retrieve the data from Redis to utilize the speed of Redis rather than do the same operation in the MongoDB application. And to a further extend, the application would want the full control over the caching management system and a user interface that allows the admin user to operate in the caching system.

4.5.3 General strategy

Thus, to implement a system that saves the response MongoDB data into a Redis database is rather simple. However, to differentiate the response and identifying the data, in order to send a quick response to the user, it would require some configuration and setup.

In a common scenario, a user request for a blog post, a controller would send a query to a database and process the collection response to send the result to the user. In this case, it would be simple to store this collection query to the Redis database. In another scenario, in similar case, a request for category for all posts

from the category, the same solution would have to be saved to apply. Nevertheless, as Redis does not have a collection or table or any relatively concept resembling distinct types of data, this is where the real challenge is in the strategy.

The solution to the problem has been used in almost in the very beginning of the caching strategy, in the traditional caching in a file system: a cache tag. To differentiate from each entry and later retrieve from Redis, a system of naming tag should be put in place.

- Post tag: 'POST_TAG'+ 'POST_SLUG'
- Category tag: 'CATEGORY_TAG'+ 'CATGEGORY_SLUG'

The two systems will never overlap as long as the category prefix tag and post prefix tag are not the same. And amongst the post and category entry, it will be identified by the slug which is unique in every collection.

This approach is also rather simple for a controller to retrieve the Redis database for the application via the slug parameter.

4.5.4 Cache management

Similar to the core collection, the caching entry should also support the CRUD action for management. However, when implementing a different premise than a normal one, the collection and caching have a distinct approach.

a. Creating an entry:

When creating a cache tag and entry, the application utilizes the event system of NodeJS in this operation. In the post and category controller, before the response object is sent to the customer, the application attachs an event emitter which emits an event. Thus, in the event system, the application can listen to the event and save the query and query result to the Redis database depending on which controller they originated.

```
eventEmitterInstance.emit(CATEGORY.VISITED, posts, slug)
```

```
const cacheID = prefix.post + doc.slug;
delete doc._id;
redis.set(cacheID, JSON.stringify(doc));
```

a. Deleting an entry:

To remove a cache entry in Redis, the same strategy can be applied for this case.

```
redis.del(prefix.post + doc.slug);
```

b. Updating an entry:

To update a cache entry in Redis, the same strategy can be applied for this case.

```
redis.set(prefix.post + doc.slug, JSON.stringify(doc));
```

c. Removing all types:

This operation could be a little tricky. At first, all of the matching tags for a type of entry must be found and then they must be removed. Redis allows the user to scan for a key by a regular expression:

```
redisClient.keys(prefixTag + '*', (error, keys) => {
    if (error) return next(error);
    deleteKeys(keys).then(()=>{
        return res.status(201).json(true);
    }, (error)=>{
        return next(error);
    })

})
```

d. Cleaning Redis database:

In some scenario, it is necessary to clean all of the data in the cached database.

```
if (req.body.cacheType === cacheType.POST) {
    cleanCacheType(prefix.post, res, next);
} else if (req.body.cacheType === cacheType.CATEGORY) {
    cleanCacheType(prefix.category, res, next);
} else {
```

```
    redisClient.flushdb(function (error, response) {
        if (error) return next(error);
        res.status(201).json(response);
    });
}
```

5 FRONTEND DEVELOPMENT

In this chapter, the thesis will demonstrate the frontend development section in this application. Exceptionally, this application has been developed entirely in JavaScript and CSS, pure HTML will only be one file: src/index.html, which only presents as a placeholder.

5.1 ReactJS and Flux architecture

In this chapter, the thesis will demonstrate the practical solutions for a web application with ReactJS and Flux architecture.

5.1.1 Frontend Routing

In this application, routing is the most crucial part and makes no mistakes. Thefrontend section is not much different. For the context of this application, the project will use the library react-router for managing routes.

Similar to most of the modern routing framework and library, a react-router defines its URLs by a hash symbol (30.). Therefore, the URLs in a modern single page application always contain a hash symbol (#) as a prefix. However, modern browsers have allowed JavaScript to manipulate with the real URLs with the HTML5 History API and remove the redundant symbols in the URI (31.). But due to the browser compatibility, in this application, it will reply to this application in a hash URL.

For the best practice, all frontend routes will be defined in the file src/app/index.js. In this file, the whole frontend infrastructure and the site map is demonstrated. This application requires a nesting routing and a react router provides enough flexible API to be developed.

```
<Route path="/" component={App}>
  <IndexRoute component={Home}/>
  <Route path="articles" component={Articles}/>
  <Route path="articles/:category" component={ArticlesCategory}/>
  <Route path="article/:slug" component={ArticleContainer}/>
  <Route path="dashboard" component={Dashboard}>
```

```
        <IndexRoute component={LoginContainer}/>
        <Route path="post" component={PostApp}>
          <IndexRoute component={PostManage}/>
          <Route path="new" component={CreatePost}/>
          <Route path="update/:id" component={UpdatePost}/>
        </Route>
        <Route path="media" component={ImageManage}/>
        <Route path="categories" component={CategoriesContainer}/>
        <Route path="cache" component={CacheManage}/>
      </Route>
      <Route path="about" component={About}/>
      <Route path="*" component={NotFound}/>
    </Route>
```

And the routes structure will be mounted into a DOM, which is registered in the src/index.js file:

```
const appHistory = useRouterHistory(createHashHistory)({ queryKey: false })
ReactDOM.render(
  <Router history={ appHistory }>
    { routes }
  </Router>
  , document.getElementById('content')
);
```

In a react-router, there are:

Route: a general ReactJS component that defines a path. It requires two parameters: a path, a string, which can use Regular Expression to generate URL, and a component, a ReactJS component that will mount it into the application once the route is requested.

IndexRoute: a more specific type of Route component. When dealing with a nesting or multiple routing scenario, it usually requires an index page to be the parent route to mount when an empty children path string is requested. In this case, if an IndexRoute component is presented, it will render IndexRoute.

And usually, most likely to prevent crashing the application while the user browses the application, the application will also define a NotFound component when a requested URL does not match any routes.

One other observation in this chapter would be that all of the components responsible for a defining path in a react router, are either a static component, in which the content is constant, or a container component, which will be explained in detail in the next chapter.

5.1.2 Components

Components are the key feature and the principle design philosophy of the ReactJS library. ReactJS intended to split complex and enormous into small, reusable components that can be developed without being interfered with each other (36.). And each of these pieces can scale and grow to a fit the application demand. Internally, ReactJS provides an abstract class ReactJS.Component. This class can be extended and implemented in the original one by putting the logic to a function render:

```
export class App extends Component{
  constructor(props){
    super(props);
  }
  render(){
    return (
      <div>
        <NavMenu/>
        <div className="page__wrapper">
          { this.props.children }
        </div>
        <Footer/>
      </div>
    )
  }
}
```

The output of the function render will mount the content into the DOM, which eventually shows to the end customers.

But the magical logic is not dwelled in the render function. The crucial part, however belongs to the cycle of the ReactJS component. ReactJS component class allows an extended class to override not only the render method but also other important methods. These methods compose an extraordinary life cycle for that component. In theory, ReactJS divides the methods group into three important types:

a. Mounting:

In this group of methods, ReactJS allows the developer to manipulate the process when the application is created and inserted to the DOM. These methods are called in the four events when the component is entered to the DOM.

- constructor: called before the component is created.
- render: called to insert the application to DOM.
- componentWillMount: called immediately before the component inserted into the DOM.
- componentDidMount: called immediately after the component rendered into the DOM.

b. Updating:

These methods are responsible for developing the state and updating functions in ReactJS DOM as they are presented at the key points of the update events that happened to the component.

- componentWillReceiveProps: called once the application is about to update their properties.
- shouldComponentUpdate: return Boolean, check if the component should be updated to prevent too much repainting the component.
- componentDidUpdate: operates when the component has updated.
- render: also updates the content.

c. Unmounting:

- componentWillUnmount is present when the component is about to removed.

The diversity of the API implemented by ReactJS provides a flexible ecosystem to dealing with a complex UI development. However, there are also two significant methods: setState and getState.

The ReactJS component has a very useful variable: state. The State represents the status of the component and every time the state changes, the application is updated. Hence, as the name suggested, the setState and getState provide the

means to update the component by altering the state of the component (35.). They do not take part in the life cycle of the component directly like the other methods, setState and getState are usually used to implicate the application events handling, as in these cases the property of the component does not change.

Theoretically, the component is not categorized into different types. However, in practice, it was decided to split the code into a readable structures and classify the component into three types.

a. Static component:

The output of this component will not change in any circumstances. They are usually deployed to a footer, a header or a static page such as not found page. In this application, the whole footer and header as well as the static page are all constructed by a static component.

b. Container Component:

These component is regularly associated to a Route for s react router and in fact, it is only used for Route in this application context. The Container component should not contain any logical functionality except authentication. It implements, however, a shelter destination to support a child component to render its business logic.

```
import ReactJS, { Component } from 'react';
import { NavMenu } from '../components/menu/NavMenu';
import { Footer } from '../components/menu/Footer';

export class App extends Component{
    constructor(props){
        super(props);
    }
    render(){
        return (
            <div>
                <NavMenu/>
                <div className="page__wrapper">
                    { this.props.children }
                </div>
                <Footer/>
            </div>
        )
    }
}
```

c. Normal Component:

Functional and specific components are categorized in this type.

5.1.3 Flux architecture

Established in the preceding chapter, Flux is an outstanding frontend architecture which is used in many huge web applications and in this application. It will implement the Flux structure for the blog platform with Alt.js

Alt.js is a small library which provides simple API to implement the Flux architecture (33.). To integrate Alt.js into this system, the application has to initialize an instance of Alt class:

```
import Alt from 'alt';
export default new Alt();
```

Ideally, every application will have one Alt instance and all the logic binding to this instance. Every stores and actions will resolve around this instance. Technically, to declare an object that contains an actions logic, there is a simple API to be used:

```
export default alt.createActions(ImageActions);
```

In the above snippet, while the ImageActions class defines a list of methods, these methods will be used in the components. Every action defined methods usually calls an event emitter:

```
requestList(){
    const actionDispatcher = this;
    this.dispatch();
    getImages().then((respond)=> {
        actionDispatcher.actions.receivedList(respond);
    }, (error)=>{
        actionDispatcher.actions.receivedError(error);
    })
}
```

Once, this.dispatch() is called, the actions have emitted an event for an associated store to manipulate.

The store, with the same premise with actions, defines an object and later exports it with a special method of the Alt.js library:

```
export default alt.createStore(ImageStore);
```

However, the store instance has to correspond with an action instance for it to make sense to the application. Usually, this procedure is established in the constructor method of store:

```
constructor() {
   this.bindActions(ImageActions);

   this.imageList = [];
   this.loading = false;
   this.error = null;
}
```

The store, on other hand, also defines the important states of a component. These state can be retrieved by call imageStore.getState(). The store e.g also contributes to the life cycle in a component

```
_onChange(){
   this.setState(this.getCacheState());
}
componentDidMount() {

   CacheStore.listen(this._onChange);
}
componentWillUnmount() {
   CacheStore.unlisten(this._onChange);
}
```

With Alt.js, it is not necessary to bind the actions and store methods to each other. Magically, when every method is defined in the action, it will be handled by another method in the store which has the same name and a prefix ‘on’.

In this application, there are five entities:

- Post: holds logic and data of blog post
- Category: associates with a category handling
- Image: manages the image application
- Cache: caches control
- Authentication or Credential: manages the authenticated issue

Each of these entities have their own stores and actions.

5.1.4 Markdown Editor

Apart from a regular implementation of category, post or image uploading and handling, the most crucial component and system in this application is the text editor, which follows the markdown syntax to convert the Markdown syntax to a readable HTML.

Markdown is a syntax language that converts text to HTML with ease. It is written and formatted closely to HTML itself (29.). Markdown was chosen for various reasons in this project:

- Markdown is simple and quite easy to learn.
- Markdown compared to normal HTML and other type of markup syntax, usually (WYSIWYG editor) is more light-weight (29.). Hence, Markdown simplifies and reduces the data storage with MongoDB.
- Markdown can be transform to meaningful HTML and text with ease (29.).

The Markdown Editor component in this application is located in the file src/app/dashboard/MdEditor.js.

i. Logical algorithm:

- Editable Content:

Markdown Editor is heavily relied on the component editbleContent.

Traditionally, whenever enter a long and customized text is entered to the browser, the textarea element is used to handle this feature. However, textarea does not provide a good user interface, hence the editor requires a different API to implement the UI.

HTML5 has provided a good impression to develop such API: contentEditable. contentEditable is an attribute in the div component. When set to true, contentEditable allows the user to edit the inner content without turning on the developer mode in browsers (32.).

For further customization, in the application, the system also implements the ContentEditable to handle an update from a parent scope. The ContentEditable component will update every time the user keyboard is called and it will transform the text into the html markup.

- Markdown customization

Implementing the normal user interface is not the ultimate goal. In the scope of this application, the system also implements a programming language detection and a syntax highlight for the scheme.

Natively, markdown allows the user to enter a block of code of a chosen language. However, to go an extra mile, the application added a script to detect a language and to add a color scheme to code block.

ii. User Interface:

Essentially, Markdown editor has two modes, one is a normal markdown markup and the other is a preview mode, where the user can preview an output of what they have been typing so far. The user interface of the editor also includes a small button for the user to switch the panel between modes.

Additionally, the editor also allows the user to assign the post to a specific category. A list category is dynamically rendered in the frontend. The post associated with the category will also be presented in the category management.

The Markdown editor also supports the editing mode with a same procedure with the creation of a post.

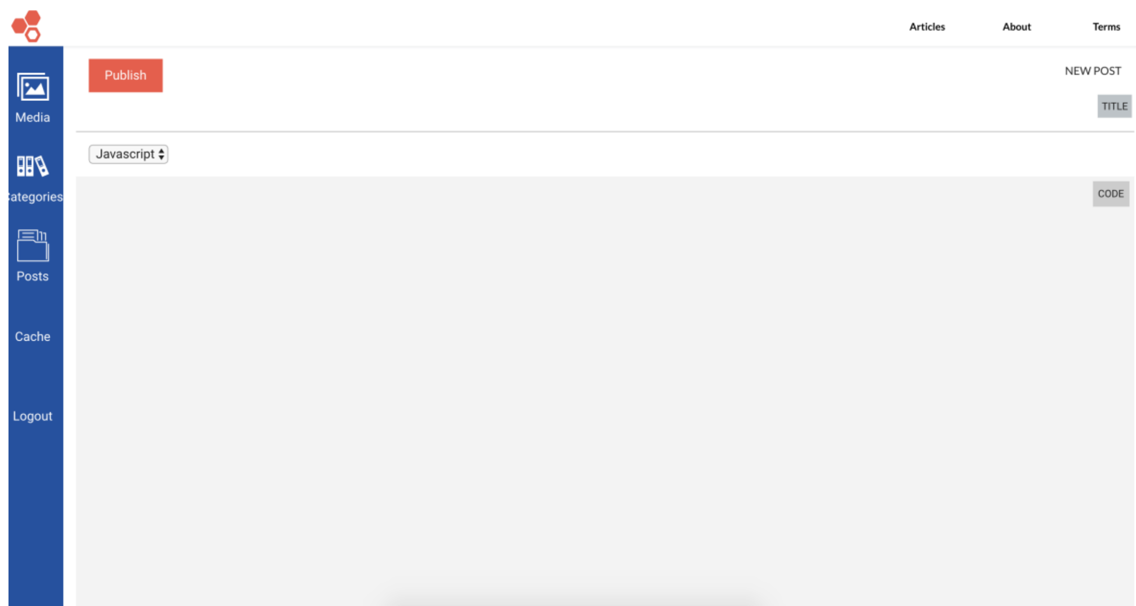

FIGURE 5. Markdow editor

5.2 Style Implementation

ReactJS and the Flux architecture have a fantastic implement in the functionality and usability. However, the good user experience does not implicate only JavaScript, it also requires a charming semantic design. In this chapter, the thesis will demonstrate the theory and practical implementation of CSS development in the project. It will have the practical application of SCSS and SMACSS.

5.2.1 Application structure

In this application focal points of SCSS are concurred in the file src/scss/main.scss. In this file, every major files and generic focal points will register and present themselves for their ruling to show in the frontend.

Following SMACSS architecture, there are four main folders:

- Vendor: the third party modules files, in this case, it contains highlight.scss(for mark up the code detection functionality).
- Base: it is utilized to establish the generic rules to future usage. This case contains base.scss, color.scss, settings.scss and mixins.scss.

- Modules: Every crucial files are located here. This folder also has a focus point as main.scss and it is divided into smaller purpose-oriented modules: news, posts, dashboard, etc.
- Layout: This folder comprises of the grid configurations. It should be noted this application built the whole grid from the scratch.

Leaving other folders to a later examination, in this section, the base configuration will be properly introduced.

a. Color.scss

Deserved its own spot in the application, color.scss define the color setting for entire project.

```
/*------------------------------------*\
  color settings
\*------------------------------------*/
$color-primary:#e45f4d;
$color-secondary: #191919;

$color-safe: #2ecc71;
$color-danger: #e74c3c;
$color-warning: #f1c40f;

$color-dashboard: #26519e;
$banner-background: #bdc3c7;
```

Structured for the functionality of the color and implication of their name, color variables are grouped into three groups: theme specific colors, status specific colors and uncategorized colors. By default, the primary color and the secondary color will be utilized for most of the base component: button, paragraph, text, menu, footer. Meanwhile, status specific colors are usually supported by a variety of notifications, banner and status bars.

b. Settings.scss

Generalized as the one file to hold all the base variables from the entire project, settings.scss also imports variables from all of siblings which have the same functionality. But in this case, there is only one file color.scss

c. Mixins.scss:

Normally, mixins would require a specific folder for managing the mixins system for a better outcome. However, due to the simplicity of the application, it is settled for a single file. In this file, every usable functions and mixins are declared in their generic form to support all of the scenario in the components.

d. Base.scss

Base.scss defines the most generic element rules, such as fonts and generic text settings.

5.2.2 Layout

Traditionally, a grid will be defined by regular bootstrap components. However, as deemed redundant to a simple project, the application would adopt the new API of CSS to develop our customized grid: flexbox.

Flexbox consists of flexible and useful features, provided by new web browsers. It alters the old and out of dated rules for width and height supporting a much clearer and extensive set of rules for defining the width and height of elements in the layout (27.). And due to this reason, many developers and CSS framework have already migrated to use Flexbox to implement their own grid. Even Bootstrap 4 in the alpha release of its newest version is building the grid system based on Flexbox (28.).

In this application, it is redundant to implement the complex grid to solve the layout problems. An abstract layout development is extended to solve a specific layout issue. The Flexbox grid would require a container and inside container components. They have the direct child elements. The Container component is the most important in the flexbox grid. It defines how the child render in a basic and generic container:

```
position: relative;
width: 100%;
height: auto;
padding: 0.5em;
display: flex;
flex-wrap: wrap;
```

```
margin: 0 auto 0 auto;
align-items: stretch;
```

Essentially, declared as a flexbox component, the container component also requires another rule: flex-wrap: wrap. This rule allows thgrid to contain itself in the given width and it does not grow the size bigger than that.

To extend our application to be much more specific, the grid also supports the vertical align and horizontal aligns.

```
.flex-row{
  position: relative;
  width: 100%;
  height: auto;
  padding: 0.5em;
  display: flex;
  flex-flow: row wrap;
  margin: 0 auto 0 auto;
  align-items: flex-start;
  flex-direction: row;
}
.flex-column{
  position: relative;
  width: 100%;
  height: auto;
  padding: 0.5em;
  display: flex;
  flex-flow: row wrap;
  margin: 0 auto 0 auto;
  align-items: center;
  flex-direction: column;
  justify-content: center;
}
```

Based on the setup grid skeleton, any rule can be demonstrated in this application. For example, upon the horizontal grid and the support of media query, two columns and three columns grids are developed:

```
.col-1-2{
  width: 100%;
  @media (min-width: $grid-desktop-breakpoint - 1){
    width: 50%;
  }
}
```

and

```
.col-1-3{
  width: 100%;
  margin-bottom: 2em;
```

```
  @media (min-width: $grid-tablet-breakpoint){
    width: 50%;
  }
  @media (min-width: $grid-desktop-breakpoint - 1){
    width: 33.33%;
  }
  &.md{
    @media (max-width: $grid-desktop-breakpoint - 1){
      width: 100%;
    }
  }
}
.col-2-3{
  width: 100%;
  @media (min-width: $grid-tablet-breakpoint - 1){
    width: 100%;
  }
  @media (min-width: $grid-desktop-breakpoint - 1){
    width: 66.57%;
  }
}
```

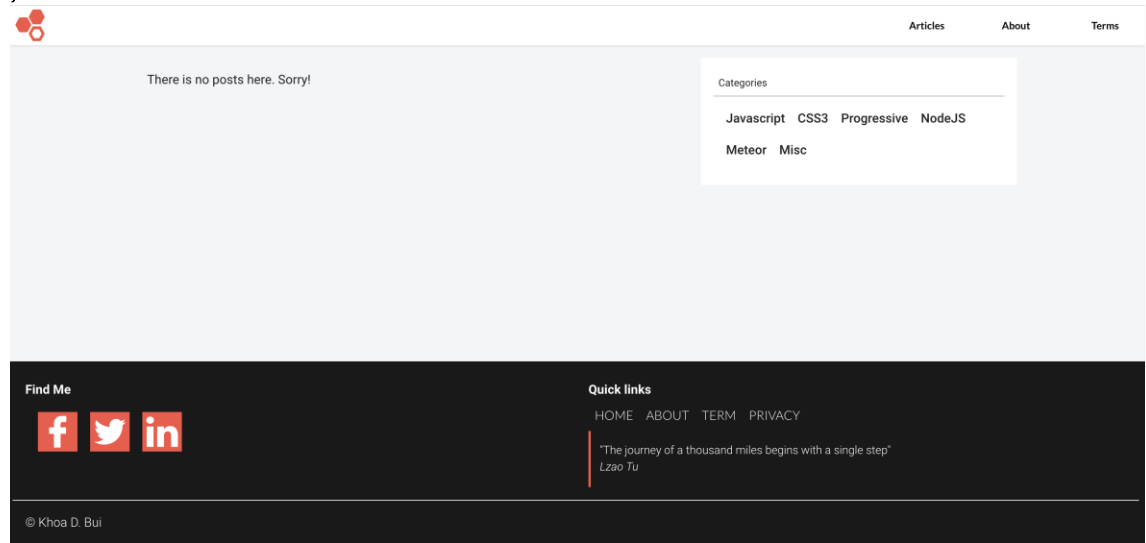

FIGURE 6. Category page layout

5.2.3 Browsers support

One of the most important aspect when it comes to the frontend development is the browser support. Whether the application is sustainable across the browsers or is it crashed in some of the browsers (23.). There is no exception in this application. There are many flaws in this project as the application experiences some advanced and experimental features in the web application with CSS.

To implement a cross browser development, this application requires CSS to have prefixes automatically to match specific rules to all browser syntax. Luckily,

the Webpack configuration already has a plugin that helped to prefix all syntax to support the two latest versions in every browser (24.). Therefore, it was a relieve for the browser support of this application.

6 CONCLUSION

JavaScript is a magnificent programming language, suitable for the web application and domain driven development. The simplicity and flexibility has been demonstrated through time not only in this project but in many other open source applications. Additionally, the community of developers experienced with JavaScript is huge and still growing as it sees the potential of the said languages (24.).

In this application, the thesis also established an extensive frontend workflow to eliminate the hassles in developing a single page application. Everything that runs in the scope of workflow is automated. This leaves developers a lot of room to develop the application logic and resolve the business.

The main aim of this project was to implement and explore the possibilities and the API of JavaScript eco-system, but the application is a little bit simple than a practical application. Nevertheless, due to flexibility and generic, abstract approach, there are possibilities to develop the application to be a full scale system.

REFERENCES

1. InternetLiveStats.com. Number of Internet User. 2017. Date of retrieval: 02.04.2017
   http://www.internetlivestats.com/internet-users
2. Automattic Inc. Introducing the New WordPress.co. 2017. Date of retrieval: 02.04.2017
   https://developer.wordpress.com/calypso
3. Wikipedia. MongoDB – Wikipedia. 2017. Date of retrieval: 02.04.2017
   https://en.wikipedia.org/wiki/MongoDB
4. Wikipedia. NoSQL – Wikipedia. 2017. Date of retrieval: 02.04.2017
   https://en.wikipedia.org/wiki/NoSQL
5. Mongo, Inc. MongoDB Documentation. 2017. Date of retrieval: 02.04.2017
   https://docs.mongodb.com
6. RedisLab. FAQ – Redis. 2017. Date of retrieval: 05.03.2017.
   https://redis.io/topics/faq
7. Wikipedia. Redis. 2017. Date of retrieval: 05.03.2017.
   https://en.wikipedia.org/wiki/Redis
8. NGINX Inc. Welcome to NGINX Wiki! | NGINX, 2017. Date of retrieval: 05.03.2017.
   https://www.nginx.com/resources/wiki/
9. Wikipedia. ReactJS (JavaScript Library) – Wikipedia, 2017. Date of retrieval: 02.04.2017
   https://en.wikipedia.org/wiki/ReactJS_(JavaScript_library)
10. Facebook, Inc. Flux | Application Architecture for Building User Interfaces, 2017. Date of retrieval: 02.04.2017
    https://facebook.github.io/flux
11. Linux Foundation. Let’s Encrypt – Free SSL/TLS Certificates. Date of retrieval 05.03.2017
    https://letsencrypt.org/
12. Strzelewicz, Alexandre (2017). Unitech/pm2: Production process manager for Node.js apps with a built-in load balancer. Date of retrieval 05.03.2017
    https://github.com/Unitech/pm2

13. Koppers, T., Ewald, J., Larkin, S., Vepsäläinen, J.; Kluskens, K. (2017). Get Started. Date of retrieval: 05.03.2017
https://webpack.js.org/guides/get-started/
14. JSON, 2017. Date of retrieval: 05.03.2017
http://www.json.org/
15. Wikipedia. Access Control, 2017. Date of retrieval: 05.03.2017
https://en.wikipedia.org/wiki/Access_control
16. Strongloop, IBM. Using Express Middleware, 2017. Date of retrieval:
http://expressjs.com/en/guide/using-middleware.html
17. Jones, M. B., Bradley, J., Sakimura, N. 2015. RFC 7519 - JSON Web Token (JWT). Date of retrieval: 05.03.2017
https://tools.ietf.org/html/rfc7519
18. Auth0, Inc. JSON Web Token Introduction - jwt.io. 2017. Date of retrieval: 05.03.2017
https://jwt.io/introduction/
19. Wikipedia. Create, read, update and delete – Wikipedia. 2017. Date of retrieval: 05.03.2017
https://en.wikipedia.org/wiki/Create,_read,_update_and_delete
20. Snook, Jonathan. 2017. Book - Scalable and Modular Architecture for CSS. Date of retrieval. 02.04.2017
https://smacss.com/book/categorizing
21. Wikipedia. LAMP (software bundling). Date of retrieval: 06.04.2017.
https://en.wikipedia.org/wiki/LAMP_(software_bundle)
22. TheServerSide, Your Enterprise Java Community. How big data and distributed systems solve traditional scalability problems. Date of retrieval: 06.04.2017.
http://www.theserverside.com/feature/How-big-data-and-distributed-systems-solve-traditional-scalability-problems
23. Ochin, Jugnu Gaur. 2017. Cross Browser Incompatibility: Reasons and Solutions. Date of retrieval: 09.04.2017.
http://www.airccse.org/journal/ijsea/papers/0711ijsea05.pdf
24. BuiltWith® Pty Ltd, Twitter Bootstrap Usage Statistics. Date of retrieval: 09.04.2017.

https://trends.builtwith.com/docinfo/Twitter-Bootstrap

25. BuiltWith® Pty Ltd, JavaScript Usage Statistics. Date of retrieveal: 09.04.2017.
https://trends.builtwith.com/javascript
26. Sitnik, Andrey. 2017. Autoprefix. Date of retrieval: 09.04.2017. https://github.com/postcss/autoprefixer
27. W3 School.com. CSS3 Flexible Box. Date of retrieval: 09.04.2017. https://www.w3schools.com/css/css3_flexbox.asp
28. Twitter Inc. Bootstrap 4 Alpha. Date of retrieval: 09.04.2017. http://blog.getbootstrap.com/2015/08/19/bootstrap-4-alpha
29. Wikipedia. Markdown. Date of retrieval: 09.04.2017.
https://en.wikipedia.org/wiki/Markdown
30. Wikipedia. Single Page Application. Date of retrieval: 09.04.2017. https://en.wikipedia.org/wiki/Single-page_application#Browser_history
31. Tennison, Jeni. 2017. Hash URIs. Date of retrieval: 09.04.2017. https://www.w3.org/blog/2011/05/hash-uris
32. Mozilla. Contenteditable. Date of retrieval: 09.04.2017.
https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attrib-utes/contenteditable
33. Perez, Josh. 2017. Alt - Flux - Managing your state. Date of retrieval: 09.04.2017.
http://alt.js.org
34. Wikipedia. Routing. Date of retrieval: 09.04.2017.
https://en.wikipedia.org/wiki/Routing
35. Facebook, Inc. State and Life Cycle – React. Date of retrieval: 09.04.2017. https://facebook.github.io/react/docs/state-and-lifecycle.html
36. Facebook, Inc. Components and Props – React. Date of retrieval: 09.04.2017.
https://facebook.github.io/react/docs/components-and-props.html
37. Kissmetrics. Speed is A Killer. Date of retrieval: 09.04.2017. https://blog.kissmetrics.com/speed-is-a-killer
38. Wikipedia. Cache (computing). Date of retrieval: 09.04.2017. https://en.wikipedia.org/wiki/Cache_(computing)

39. StrongLoop, IBM. Production best practices: performance and reliability. Date of retrieval: 09.04.2017. https://expressjs.com/en/advanced/best-practice-performance.html
40. SolidIT consulting & software development. DB-Engines Ranking - Trend Popularity. Date of retrieveal: 10.04.2017. https://db-engines.com/en/ranking_trend
41. TheServerSide, Your Enterprise Java Community. Growing the demand for NoSQL technology, and non-relational solutions. Date of retrieval: 10.04.2017. http://www.theserverside.com/feature/Growing-the-demand-for-NoSQL-technology-and-non-relational-solutions
42. Minnick, Chris. 2017. The Real Benefits of the Virtual DOM in React.js. Date of retrieval: 10.04.2017. https://www.accelebrate.com/blog/the-real-benefits-of-the-virtual-dom-in-react-js
43. Q-Success Software Quality Management Consulting. Market share trends for content management systems for websites. Date of retrieval: 10.04.2017. https://w3techs.com/technologies/history_overview/content_management
44. Nolan, Ashley. 2016. The State of Front-End Tooling 2016 – Results. Date of retrieval: 10.04.2017. https://ashleynolan.co.uk/blog/frontend-tooling-survey-2016-results
45. Zaytsev, Juriy. 2017. ECMAScript 6 compatibility table. Date of retrieval: 10.04.2017. https://kangax.github.io/compat-table/es6
46. Chaffey, Dave. 2017. Global social media research summary 2017. Date of retrieval: 10.04.2017. http://www.smartinsights.com/social-media-marketing/social-media-strategy/new-global-social-media-research
47. Cogneau, Alexander. 2017. Node.js Events and EventEmitter. Date of retrieval: 10.04.2017. https://www.sitepoint.com/nodejs-events-and-eventemitter

APPENDIX 1

1. Complete code of routes/index.js for defining all routes:

```
import postController from './posts';
import categoryController from './categories';
import imageController from './images'
import userController from './user';
import checkAuth from './auth';
import allController from './all';
import cacheController from './cache';

export default (app)=>{
    app.all('/api/*', checkAuth)
    app.use('/api/posts', postController);
    app.use('/api/categories', categoryController);
    app.use('/api/images', imageController);
    app.use('/authenticate', userController);
    app.use('/', allController);
    app.use('/api/cache', cacheController);
}
```

2. CRUD controller:

```
import Post from './post.model';
import Category from '../categories/category.model';
import {prefix, POST, CATEGORY} from '../../events/config';

import redisClient from '../../config/redis';
import eventEmitterInstance from '../../events/events';

const index = (req, res, next)=> {

    Post.find({})
        .populate('category')
        .exec((error, posts)=> {
            if (error) {
                return next(error);
            }
            if (!posts) {
                return next(error);
            }
            res.json(posts);
        })
}

const showBriefs = (req, res, next) => {
    Post.find({})
        .limit(5)
```

```
        .sort({createdAt: -1})
        .select({content: 0, keywords: 0})
        .populate('category')
        .exec((error, posts)=> {
            if (error) {
                return next(error);
            }
            if (!posts) {
                return next(error);
            }
            res.json(posts);
        });
}

const update = (req, res, next)=> {
    let id = req.params.postId;

    Post.findById(id)
        .populate('category')
        .exec((error, post)=> {
            if (error) return next(error);
            if (!post) return next(error);

            post.update(req.body, {runValidators: true, context: 'query'}, (error,
pos)=> {
                if (error) return next(error);
                res.json(pos);
            });
        });
}

const create = (req, res, next)=> {
    let newPost = new Post(req.body);

    newPost.save((error, post)=> {
        if (error) return next(error);

        res.status(201).json(post);
    });
}

const remove = (req, res, next)=> {
    let id = req.params.postId;

    Post.findById(id)
        .exec((error, post)=> {
            if (error) return next(error);
```

```
            if (!post) return next(error);

            post.remove((error)=> {
                if (error) return next(error);

                res.status(204).end();
            });
        });
    }

    const getOne = (req, res, next)=> {
        const id = req.params.postId;
        redisClient.get(prefix.post + id, (error, postJSON)=> {
            if (postJSON) {
                console.log('from cache post')
                return res.json(JSON.parse(postJSON));
            } else {
                Post.findOne({slug: id})
                    .populate('category')
                    .exec((error, post)=> {
                        if (error) {
                            return next(error);
                        } else if (post === null) {
                            return res.status(404).json({
                                msg: '404'
                            });
                        } else {
                            eventEmitterInstance.emit(POST.VISITED, post);
                            res.json(post);
                        }
                    });
            }
        });

    }

    const getCategoryPost = (req, res, next)=> {
        const slug = req.params.categorySlug;

        redisClient.get(prefix.category + slug, (error, postList)=> {
            if (postList) {
                console.log('from cache category')
                return res.json(JSON.parse(postList));
            }else {
                Category.findOne({url: slug}).exec((error, category)=> {
                    if (error) {
                        return next(error);
```

```
                }
                if (category === null) {
                    res.status(404).json({
                        msg: '404'
                    });
                } else {
                    Post.find({category: category._id})
                        .populate('category')
                        .exec((error, posts)=> {
                            if (error) {
                                return next(error);
                            }
                            if (!posts) {
                                return next(error);
                            }
                            eventEmitterInstance.emit(CATEGORY.VISITED, posts, slug)
                            res.json(posts);
                        })
                }

            });
        }
    });
}

export {index, showBriefs, update, create, remove, getOne, getCategoryPost} ;
```

3. Events System:

```
import {CATEGORY, prefix} from '../config';
import redis from '../../config/redis';

const onNewCategory = (doc)=>{
    "use strict";

}

const onUpdateCategory= (doc)=>{
    "use strict";

}

const onCategoryRemove = (doc)=>{
    "use strict";
}

const onVisitCategory = (doc, slug)=>{
    "use strict";
    const cacheID = prefix.category + slug;
    console.log(cacheID);
    redis.set(cacheID, JSON.stringify(doc));
}
```

```
export default (eventEmitter)=>{
    "use strict";

    eventEmitter.on(CATEGORY.NEW, onNewCategory);
    eventEmitter.on(CATEGORY.UPDATE, onUpdateCategory);
    eventEmitter.on(CATEGORY.REMOVE, onCategoryRemove);
    eventEmitter.on(CATEGORY.VISITED, onVisitCategory);
}
```