TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Software Science

Anton Dorofejev 143061IAPB

DEVELOPMENT OF A FREELANCE MARKETPLACE SERVER-SIDE USING EXPRESS.JS AND MONGODB

Bachelor's thesis

Supervisor: Veronika Repnau
Co-Supervisor: Gunnar Piho
Docent

Tallinn 2017

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Anton Dorofejev 143061IAPB

TÖÖPORTAALI SERVERI OSA ARENDAMINE KASUTADES EXPRESS.JS JA MONGODB

Bakalaureusetöö

Juhendaja: Veronika Repnau
Kaasjuhendaja: Gunnar Piho
Dotsent

Tallinn 2017

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Anton Dorofejev

22.05.2017

Abstract

Development of a Freelance Marketplace Server-side Using Express.js and MongoDB

A freelance marketplace currently named Million Mothers is a project that is directed at helping unemployed mothers carry out various tasks and be able to look after their children at the same time. This project therefore requires such a platform to be built as a web application.

Web applications are usually built in a client-server model. The client-side of this project is considered to be already provided by default. Therefore, a server-side part of the application has to be built. Express.js and MongoDB are the pre given instruments that have to be used and the application has to be built around those. The server-side, though, can be constructed in many ways and it ultimately has a lot of obligations to fulfill.

The goal of this work is to analyse the pre given instruments, choose an architecture for building the server-side, determine the process of developing and test the resulting project.

This thesis is written in English and is 43 pages long, including 6 chapters, 1 figure and 2 tables.

Annotatsioon

Tööportaali serveri osa arendamine kasutades Express.js ja MongoDB

Tööportaal, mis on hetkel nimetatud Million mothers, on niisugune projekt, mis on suunatud sellele, et anda töötu emadele võimalust täita erinevaid ülesandeid ning hoolitseda oma laste eest samal ajal. See platvorm on siis loodud veebirakenduse kujul.

Veebirakendused on tavaliselt arendatud kasutades klient-serveri mudelit. Peetakse, et kliendi osa on etteantud. Serveri osa on siis see, mis peab olema loodud lisaks kliendi osale. Express.js ja MongoDB on need instrumendid, mida on vaja kasutada serveri osa loomisel. Kuigi on vaja mainida, et seda on võimalik arendada kasutades palju erinevaid viise ning on olemas ka palju aspekte, millele see peab vastama.

Selle töö eesmärgiks on analüüsida nõutud instrumente, valida arhitektuuri serveri osa arendamiseks, määrata arenduse protsessi ehk käigu ning testida lõpuks saadud projekti.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 43 leheküljel, 6 peatükki, 1 joonist, 2 tabelit.

List of abbreviations and terms

AJAX	Asynchronous JavaScript And XML [1]
BSON	Binary JSON [2]
Callback	A special function, which may be called back [1]
Company	A user, whose primary goal is to provide jobs and hire talents for executing those.
Express.js	"a relatively small framework that sits on top of Node.js's web server functionality to simplify its APIs and add helpful new features." [1]
JSON	JavaScript Object Notation, "a simple representation of data" [2]
MongoDB	"a powerful, flexible, and scalable general-purpose database." [2]
Mongoose	Node.js module simplifying connection to MongoDB [3]
Multilayered architecture	An architecture, which sees the application consist of multiple layers [4]
Node.js	"a JavaScript platform—a way to run JavaScript" [1]
NoSQL	A common term for non-relational databases [5]
Promise	"The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value" [6]
Talent	A user, whose primary goal is to apply for jobs and execute those that have been entrusted to them.
TLS	Transport Layer Security [7]

Table of contents

List of figures

List of tables

1 Introduction

There is a number of various freelance marketplaces already present in the world wide web. However, a lot of those are directed at a wide range of possible clients. The reviewed project, currently named Million Mothers, is targeted at helping helping mothers, especially those with young children, at finding a finding a job, which unfortunately turns out to be quite difficult nowadays. Since there is a certain interest for this market coming from various companies, it would be wise to create a respective platform.

The system analysis is reviewed in a Bachelor's thesis "Global Skill-based Hiring Platform Development Analysis" by Ksenia Shlyapnikova. This thesis' purpose, though, is to review the development process of the given project according to the requirements provided by system analysis.

1.1 Subject background

As the result of this Bachelor's thesis, a web application conforming to the pre given analysis is developed. Express.js and MongoDB are technologies that are used in combination quite frequently, though even in this case it is important to choose the connecting driver as well as other instruments that would benefit the resulting application the most. Afterwards, the database design and the whole application structure are to be reviewed and then developed.

1.2 The task

The resulting work reviews instruments that are used for developing the project as well as describes various aspects of those, such as security, error handling, structuring, which are essential for any web application.

Server-side scripting is created for serving the client-side part of the project that is to be provided in addition to combining it with the database.

1.3 Methodology

Express.js, which is based on Node.js and is "the de facto standard Node.js web application framework" [8], is used for developing server-side scripting. Being an unopinionated and minimal framework, it calls for additional modules to be used depending on the task's requirements.

MongoDB is used as a database for holding all of the information due to its' support for user-defined JavaScript functions as well as BSON being the main storing format.

1.4 Thesis review

Chapter 2 describes application architecture and structure, technologies used, security risks as well as error handling.

Chapter 3 reviews database design, modules used with Node.js, accessible web addresses.

Chapter 4 describes tasks that have been completed and results testing.

2 Analysis

This chapter reviews the architecture chosen for developing the project, MongoDB with Mongoose as an instrument connecting the database with Express.js, as well as security, error handling and structuring aspects.

2.1 Multilayered architecture

When talking about a client-server application, one of the most important steps is to choose a correct software architecture. A multilayered architecture was ultimately chosen due to the fact that the client-side of the application had been already done.

A multilayered architecture is perhaps more commonly known as a multitier architecture. The difference between a tier and a layer is arguable because some consider these as the same thing, while others tend to view a tier as a physical entity in contrast to layer being a logical one. Since application and database reside on the same server during the early stages of development, it would most likely be confusing to name the planned architecture a 3-tier one, therefore a clearer name is used instead. [9]

Multilayered architecture is a client-server architecture that includes . The number of layers could vary. One of the examples could be a 2-layered architecture, in which presentation layer or client communicates directly with data management layer or the database server. A more popular solution, though, is using a 3-layered architecture, where an additional application or domain logic layer is used between presentation and data storage layers. Other popular variations are a 4-layered architecture, where a business logic layer is implemented in addition to the application layer, and even a 6-layered one, which includes a client layer, presentation logic layer, a database, data access layer, business logic layer and a web server itself. The bigger number of layers leads to greater modularity, while complicating the development in the process. An incorrectly developed layer could lead to bigger security risks, which means that the number of used layers has to stay reasonable. [4] [10] [11]

Considering the fact that presentation layer was originally planned to be implemented by another side as well as the fact that Node.js and MongoDB were the required technologies

for developing the given web application, a 3-layered architecture was chosen as the one suiting the needs the most.

2.2 MongoDB

For this project, the database, which is MongoDB in accordance with the requirements, is a part of the data layer.

The database management system for the given task is MongoDB, a document-oriented database program, which therefore makes it a so-called NoSQL database. NoSQL databases are ones that are not modeled in tabular relations found in relational databases and use other means of storing information. This means various advantages, such as better performance in certain situations as well as a simpler horizontal scaling of machines, though does lead to drawbacks either. The NoSQL characteristic of a database is important to note because it first of all, requires a different modelling approach than with a relational database, and secondly, leads to different security risks. Perhaps the greatest example of that is SQL injection, a code injection attack often used against relational databases. [5]

A document-oriented database has data stored in so-called documents, a type of key-value pairs. Documents could be implemented in various ways, such as by using XML, JSON or BSON. BSON or Binary JSON is a format for representing the stored data precisely in a MongoDB database. It originated from JSON and is quite similar to it, having only slight differences. This is exactly the reason why MongoDB is quite often used along with Node.js, Express.js and sometimes even AngularJS, thus making a full MEAN stack, in which every component supports JavaScript and the whole client-server application could be written in it as a result. [5] [2]

Just as with relational databases, MongoDB allows the developer to use multiple databases. The difference starts within. Database includes collections, which in return consist of documents. A collection and a document may roughly correspond to a table and a row of data. Document includes various fields that are similar to attributes in relational databases. However, it is important to note that the only field required by MongoDB by default is an _id field. [2]

MongoDB is considered to be a simple database at the beginning, though it is known for certain more complex features as well. [2]

MongoDB is, for instance, well known for sharding – a process of handling data growth via horizontal scaling, when the number of physical nodes is increased to spread all of the data between those. It is used so that the capacity of original node would not have to be increased and could stay the same. [2]

Another important MongoDB feature is replica set. By default, there is only one mongod process of MongoDB running, leaving the whole database inaccessible in case of a crash. Replica sets allow holding the same data within separate instance of mongod process, serving as a backup service. [2]

Resorting to sharding or replica sets will most likely be necessary during the later stages, once the database starts holding a lot of data and it has to be accessible at all times. At the moment of development stage there are no such needs.

2.2.1 Transparent huge pages

There is a MongoDB warning during mongod process start about a Linux operating system feature called transparent huge pages being set to "always", and it is recommended to change this value to "never". Transparent huge pages are used to reduce overhead occurring during sequential memory access traditionally performed by relational databases. MongoDB, though, does not rely on sequential memory access as much, so it is of no benefit. Moreover, using huge pages might lead to more frequent disk access if not all of the data can fit into memory. [12]

While data size during development stage is hardly that great, it is definitely going to grow with time, thus it would be better to disable huge pages right away. A solution is provided by MongoDB developers themselves, which requires an executable init.d script creation and setting it to start on system boot. If the server is either a RedHat or a CentOS distribution of Linux, it will most likely be necessary to override dynamic kernel tuning tools tuned or ktune as well. [12]

2.2.2 MongoDB Security

The foremost important aspect of security to consider is the fact that MongoDB has to be set up on a server that is not publicly addressable. The server thus has to include firewall protection along with being accessible only within private networks. [2]

Another option is to disallow JavaScript scripting that is allowed by default within MongoDB and is known for being a potential security loophole. However, sometimes such scripting might be important for the application to work properly. The best choice is to consider this measure at the end of development stage. [2]

2.3 Mongoose

Since Node.js and MongoDB are a pair of instruments used quite often together, there already is a solution for combining these: a mongodb driver provided by MongoDB developers and obtainable via npm. Thus, mongodb driver could be considered as a standard approach. [3]

However, such an approach usually requires certain precautions with various factors like opening or closing a connection. It still somewhat follows the unopinionated design because different developers might prefer using different designs within their projects. In a way, while mongodb driver does handle the database usage well, it still has a room for simplifying this process. [3]

This is exactly where a module called Mongoose steps in. Mongoose's purpose is to allow a developer to model data using schemas. A schema is where all the keys' information is specified, such as data type, possible default value. The majority of MongoDB data types are related to those used in JavaScript, but in case a data type inherent to MongoDB is needed, it is possible to reference it using a mongoose module. Mongoose is then chosen in place of MongoDB driver because of simplicity it provides during the early stages of development. [1]

2.4 Express.js

As it was already mentioned, Express.js is running with the help of Node.js. Node.js is a platform for running JavaScript, which often happens outside of a web browser, where

JavaScript has been used since its' creation. Node.js is famous for asynchronous I/O, which is mentioned below. On top of that, it allows running a web server in a relatively simple manner, which is used by Express.js. [1]

Express.js therefore adds certain features, like middleware and routing, which makes it a web framework. There are not many of those features and it is possible to create a web application without resorting to Express.js. However, these are quite helpful and even essential for developing a web application, so rejecting Express will most likely make the developer create the same features by himself despite the fact that they have already been made before him. Thus, Express.js is a rather effective web framework. [1]

2.5 Express.js Security

Security is one of the most important aspects of any application that has access to user's personal information. The threats mentioned will be either those that are directly related to application and data layers or the ones, which could be at the very least partly prevented on the server side.

2.5.1 TLS

Various applications may have different approaches to which pages could be visited by an anonimous user. The given application should be accessible to such a user only via an index page, where the user is given an opportunity to either register or log in in case they have already registered beforehand. [7]

User authentication should always be implemented using HTTPS or HTTP over TLS because a simple HTTP request without any encryption can be easily read by a 3rd party. An express application uses standard HTTP by default, but there are both tls and https modules already available with Node.js. [7]

During production stage the certificate used with TLS has definitely got to be valid and confirmed. During development stage, though, a simple certificate created locally is sufficient for testing purposes.

2.5.2 Helmet - HTTP Headers

HTTP Headers are an important part of a secure web application due to the fact that these set a lot of operating parameters. For instance, it becomes possible to disable client-side caching or force the user to resort to using HTTPS connection. [7]

A popular Node.js module called Helmet includes 9 other modules that help server set the headers correctly in order for the application to be as secure as possible. These include, as follows: [7]

- Content-Security-Policy header against cross-site scripting and cross-site injections
- X-Powered-By header disabling. This is the most important header to consider because it allows an attacker to detect applications running an instance of Express.js
- Public Key Pinning headers against man-in-the-middle-attacks
- Strict-Transport-Security header enforcing secure connection
- X-Download-Options header set for Internet Explorer of version 8 or newer
- Cache-Control set to NoCache and Pragma headers against client-side caching
- X-Content-Type-Options header set to NoSniff against MIME-type sniffing
- X-Frame-Options header to protect from clickjacking
- X-XSS-Protection header applied to allow cross-site scripting filters

It must be noted that some of these headers are already enforced by modern browsers by default, in which cases setting a header does not impact application security as a whole. However, it is better to use these nevertheless as user could be using any type of browser he wishes to. [7]

2.5.3 Cookies used for keeping a session

To implement authentication, a module called express-session is used. In contrast to a module cookie-session, express-session stores only session identifier, while all the

session data is stored on the server. This is particularly useful in case the application requires user's personal information to be stored in a session at some point in future. [7]

The things to remember while using express-session is, firstly, to set an original name to a session. Otherwise, an attacker may exploit the application knowing session's name. Secondly, the default session storage is in-memory, which is well suited during debugging stage but leads to memory leaks during production one. Therefore, a different type of storage has to be used after debugging is complete. Connect-mongo is a popular solution for such purposes and is considered to be used at the moment of this thesis' writing. [7]

2.5.4 Using secure modules

Due to modular design of Node.js applications it is necessary to be aware of vulnerabilities among all the used modules. There is a number of instruments that use their own databases, which include a list of modules with certain vulnerabilities. [7]

One of such instruments is a package called nsp that checks dependencies listed in package.json file of the project. In case there are no known vulnerabilities, nsp's response is "(+) No known vulnerabilities found" [7]

2.6 Error handling

Since application layer serves as a connection between presentation layer and data layer, it is especially necessary to prepare for errors to be emerging during operations with a database and responding to a client's request.

Errors with database operations might appear for various reasons, the simplest one being a connection failure. In such cases a callback function usually returns a non-null first argument describing the error. Reading a corresponding error message is useful during development stage for faster debugging but is critically dangerous in production for it could help an attacker find a new exploit. Thus, the best solution is to send an empty response with a general 500 HTTP status code, which tells a user that the problem is not related to their actions. [13]

Errors associated with presentational layer are especially common and thus have to be controlled in the majority of the client requests. When it comes to simple GET messages that request a full web page, there are usually no parameters passed along with the request.

In case a parameter is used, it needs to be sanitized before being applied to query the database. The same approach is to be used if certain web addresses require identifiers to be passed as part of an address, such as a job's identifier to get a requested job's web page.

If a requested address does not exist, a web page with 404 HTTP status code should be served. In order to further conceal the web framework used it is a good idea to create a custom page reporting such an error. [13]

However, the majority of the POST requests do pass along some parameters. Sanitizing those parameters is a top-priority. Despite the fact that MongoDB is not prone to SQL Injection attacks, it is still necessary to sanitize the passed parameters against certain patterns like a dollar sign at the beginning of a string. Express-mongo-sanitize module is used for these purposes. If a required parameter is missing, it could be defined as an empty string, zero or any other similar value as long as it would not lead to unexpected actions. [14]

The main approach to handling errors in Express.js, which is recommended by developers themselves, is to apply a special function with 4 arguments to the express instance. That function is quite similar to those used for routing, which include request, response and next middleware arguments. The 4th argument is the error itself, which has to be placed as the first one in a typical Node.js fashion because handling that error should be a top-priority. This error handling function should be applied after all of the routing instances and the 404 Not found error handling function. [13]

The Express.js developers have provided an example of a possible error handler for both development and production stages. The difference is that the user should not get a trace of the error during the production stage. This example is currently being used as it fulfills the needs of error handling within the application. [13]

2.7 Application structure

Since Express.js is stated to be an unopinionated framework, it literally gives the developer a wide choice in terms of what kind of an application they are going to create. Of course, such cases as routing or usage of middleware are an irreplaceable core of Express.js, but those are pretty much the only remaining parts of such kind in the 4th version of the framework. While the 3rd version had some middleware like

express.bodyParser or express.session, the 4th one discarded those in favor of stand-alone modules, thus giving the developer more freedom. [15]

However, there are still some conventions present to help a new developer with getting a ready-to-go application structure. The most widely known structure is created by using an Express application generator. Aside from the app.js and package.json files as well as node_modules directory as a part of standard Node.js application, the main directory contains public, routes and views directories. [16]

Looking at the views directory will most likely lead to questions regarding a Model-View-Controller pattern, which is arguably the most common pattern used with web frameworks. Such an approach is definitely possible with Express.js because routes may be used as controllers and models are an integral part of Mongoose. Structuring the application in such way is not a goal during the development process, though, and stepping aside from it at any point is possible if certain benefits are to follow. [17]

Public directory's contents are be available to any unauthenticated user unless specified otherwise, which is why it has to be used especially carefully. This directory basically becomes root within the web address. For instance, a file named styles.css that could be found in this directory will be available at http://localhost/styles.css in case the server is running on localhost. For this reason all of the static files are placed in the public directory. Though It has to be noted that any HTML or template files have to be placed in the views directory or otherwise the same file could become available via both routing and public directory, which is at the very least a mistake with application structure or at the worst case a possible security loophole. [1]

Routes directory is used for placing application routers. Routers define, which web addresses or endpoints will be responded with which actions. Routers also define a request method. A function handling such a request takes at least 2 arguments: a request and a response. Request holds all the requested information while response is passed to send some kind of a response back to the client. Sending this response is therefore a developer's responsibility as omitting this part will most likely lead to request timeout. There is also an opportunity to include a 3rd parameter, usually named next, which is a callback to the next middleware function. This callback function is necessary if there is a common error handling function placed after all the routers. In case some kind of an error

is thrown, it can be passed via next to the error handling function. Sending a response is not needed then because only the last function in the middleware chain is responsible for that. [1]

Creating a separate directory for different routers is not obligatory. All of those could very well be defined in the main application file app.js. However, developing a resulting file of such size would become rather problematic after awhile, which is why it was chosen to keep routers separately. The routers are further separated according to the Handlebars files in views directory with a one-to-one relationship. Some of the views could be requested via various methods, such as the list of jobs is accessible using both GET and POST, though separating such handlers would be pointless due to the fact that these share the same dependencies. In addition to these routers, a response directory is also a part of routes directory. This one contains routers responsible for various dynamic requests, such as AJAX. The separation is performed in terms of logical differences between requests as well as dependencies involved. For example, requests for uploading and removing user's photos or logos are all handled in one file for the reasons specified. [1]

Views directory contains template files for rendering, which are called by routers. The template system used in this project is Handlebars. This system was chosen because of it's simplicity and support of standard HTML code. While another template system, named Jade, is arguably more popular, it's advantages and purposes are not the ones needed in this project. Jade allows the developer to rewrite HTML code into a more simple and readable format, which is not beneficial in this case since static web pages have already been written beforehand. Jade does support standard HTML, but it was not originally designed for that. Moreover, if Jade is going to be selected as a template system in the future, migrating from Handlebars to it is most likely going to be a less complicated process than vice versa. [1]

In addition, some modules might also require separate directories. Among the used modules, Mongoose model files are usually placed in models directory, Handlebars helper functions could be placed in helpers directory, various commonly used functions in functions directory and testing should be separated into yet another directory.

The models directory usually contains only those files associated with the database. Since module.exports could allows access to a schema, every collection's model is present in a separate file. [3]

Handlebars helper is a function that lets the developer extend Handlebars potential using JavaScript. It is strictly optional, a real application could very well be developed without resorting to it, but it does simplify template usage. There are already some standard Handlebars built-in helpers by default, such as "if", "unless", "with" etc. Therefore, creating a new helper does not complicate or slow down rendering too much. [18]

There are are at least 2 functions within the project that need to be exported into a separate directory, namely those that check if the client is authenticated and if they are not. Functions directory is used specifically for this case.

Testing might require a different amount of files depending on what kind of testing is to be performed. This part should be decided during development process.

The resulting application structure is as follows:

```
./
    app.js
    package.json
    functions/
        auth_condition.js
        <other possible commonly used functions>
    helpers/
        helper.js
    models/
        jobs_model.js
        <other model files>
    node_modules/
        async
        <other node modules>
    public/
        image
        css
        js
        <other static files>
    routes/
        index.js
        <other route files>
        response/
            apply_for_task.js
            <other dynamic response route files>
    test/
        test.js
    views/
        index.handlebars
        <other handlebars files>
```

Figure 1 Application structure

2.8 Callbacks and Promises

One of the most significant advantages of using Node.js as a server-side run-time environment is it's event-driven architecture that is well known for asynchronous I/O. Of course Node.js supports synchronous, blocking I/O as well, but it is rather disadvantageous in comparison to asynchronous one because a Node.js application runs on a single thread without letting separate tasks run in parallel, which decreases application's speed drastically. This is especially important for web applications that have to handle many requests at the same time and is exactly the reason why these applications usually work asynchronously. [1]

Knowing that it is important to arrange function calls asynchronously as well. This is quite simple with JavaScript because it supports higher-order functions. If a function

comes out as another function's argument, it is usually called a callback function because it can be called back. The callback function is either defined along with the called function anonymously or defined as a normal function elsewhere and it's name is then passed as an argument. It has to be remembered that this callback function is going to start performing once it is called in the called function's body. The majority of callback functions' definitions for this project occur as a result of MongoDB operations, like save, update, find. Defining one of such functions is not problematic, but if their number is bigger, code readability changes for the worse. [19]

A phenomenon, which sees a lot of anonymous callback functions being defined one inside another, is called "callback hell". When scripting in such style, javaScript code itself starts growing horizontally, making it more difficult to maintain. One of the ways to prevent that is to use promises. A promise is an object that holds a state of pending, fulfilled or rejected. It is going to be pending before being called. After that it is either going to become fulfilled in case operation was successful or rejected if it was not. Promise's then() method is made for handling a fulfilled and optionally a rejected promise and catch() is for rejected one only. Though reading through promises might be trivial at the beginning, they are considered to be a better option than standard anonymous functions. [6]

3 Experiment

The goal of the experiment is to develop an efficient database design for the given task in accordance with functional analysis, in addition to constructing the application layer serving as a base of logical operations.

3.1 Database physical design

Considering the fact that MongoDB is capable of holding a lot of data in one collection, it would be logical to limit the number of used collections to as few as possible. Aside from the classifiers, which each needs it's own collection, there are 2 collections remaining: a user and a job, where a user's collection contains either company or talent data. Due to the fact that different documents in a MongoDB collection may have different fields and the fact that a company and a talent share a lot of common fields, it is fairly possible to hold both companies' and talents' data in one collection.

Originally, the database was designed to have all the classifiers' documents in a form used in relation databases, where a row is represented by a unique code, a unique name and an optional description. Such a design is possible in MongoDB as well by specifying an _id field explicitly while inserting new data. Documents from other collections then simply reference necessary classifiers via code corresponding to that found in the _id field. A design of this kind prevents data redundancy and therefore makes managing the database less complicated.

However, MongoDB as a NoSQL database is well known for allowing higher levels of denormalization, as it is capable of storing all of the database data within one collection. This would lead to data redundancy, as some of the data would be held in multiple collections. This would ultimately improve speed of reading operations while slowing down writing, updating and deleting because the affected data would have to be checked in multiple collections. Since reading operations occur more frequently than the other ones, a higher level of denormalization is considered to be a better choice, in addition to the fact that this design is more common with MongoDB. The classifiers are then used for reference before changing data, but not during reading. [2]

The list of classifiers' collections is as follows:

- Job category
- Job industry
- Job type
- Expertise level
- Location
- Flexibility
- Skill
- Job status
- Timezone

All of the classifiers therefore do not have the _id field overridden, leaving only a required, unique name field and a not required description field, both being of String data type. Administrator's document currently has username and password field for logging in, both of which are required String fields. Password is encrypted within application layer, which is why it becomes a simple String field in MongoDB.

User's and Job's collections are, on the other hand, more complex, amassing a bigger amount of fields.

Field name	Field data type	Field description
email	String	User's email, required, unique
is_company	Boolean	Defines if user is a company, required
first_name	String	User's first name, required
last_name	String	User's last name, required
confirmed	Boolean	Defines if user confirmed their profile via email, required
join_date	Date	User's date of joining, required

last_sign_in	Date	User's last appearance, required
location	String	User's location, required
language	String	User's language, required
ranking	Number	User's ranking, required
categories	[String]	List of user's categories, required
skills	[String]	List of user's skills, required
address	String	User's address
telephone	String	User's telephone number
website	String	User's website or web page
is_active	Boolean	Defines if user is active
profile_public	Boolean	Defines if user's profile is public
video_id	String	User's video id
story	String	User's story
skype	String	User's skype id
instagram	String	User's instagram id
timezone	String	User's timezone, required
photo	String	User's photo
name	String	Company name
field_of_activity	String	Company's field of activity
offered_jobs_limit	Number	Max number of jobs to be offered by company
fav_talents	[Mongoose.ObjectId]	Company's favourite talents
logo	String	Company logo
expertise_level	String	Talent's expertise level
applied_jobs_limit	Number	Max number of jobs to be applied to by talent
applied_jobs	List of objects	List of jobs applied to by talent
favourite_jobs	[Mongoose.ObjectId]	Talent's favourite jobs
favourite_companies	[Mongoose.ObjectId]	Talent's favourite companies
education	[Object]	Talent's education
working_experience	[Object]	Talent's working experience

resume	String	Talent's resume

Table 1 User's collection

Field name	Field data type	Field description
title	String	Job's title, required
description	String	Job's description, required
job_type	String	Job type, required
budget	String	Job budget, required
salary	String	Job salary, required
salary_type	String	Flexibility, required
category	String	Job category, required
skills	[String]	Job skills, required
bonus_skills	[String]	Job bonus skills
time_offset	String	Job timezone, required
expertise_level	String	Job expertise level, required
company_id	Mongoose.ObjectId	Company posted, required
company_name	String	Company name, required
location	String	Job location, required
contract_duration	String	Contract duration, required
other_requirements	String	Other requirements
questions	[Object]	Job questions
status	String	Job status, active by default
date_created	Date	Job creation date, required
date_ends	Date	Job closing date, required
popularity	Number	Job popularity, zero by default
keywords	[String]	Job keywords
photo	String	Job photo
logo	String	Job logo

Table 2 Job's collection

3.2 Instruments used for developing the application layer

Aside from Express.js itself and Mongoose, the application requires the following modules:

- async, additional asynchronous functions. Using async extends the effect of asynchronous functions across the Node.js application, though it is not always necessary. It offers many interesting features, such as filtering, parallel tasks, performing tasks in series. In this project, the main use of async is an asynchronous loop, in which every new iteration is performed as a separate asynchronous function. [20]

- bcryptjs, encryption of passwords to save in a database. MongoDB does not provide encryption of data by default. It is possible to use a slightly upgraded version of MongoDB to use encryption, but a more simple way would be to use bcryptjs that encrypts data by itself. [21]

- bluebird, a promise library. As it was already mentioned, using promises allows to prevent callback hell. Furthermore, it is recommended by Mongoose developers to override their standard mpromise that is not supported anymore. [22]

- body-parser, handling various client requests. It is possible to choose various formats parsers. JSON and URL-encoded parsers are used in this project. [23]

- express-handlebars, template system used with Express.js. Express.js does not have any template systems by default, which is why one of those has to be chosen. [24]

- express-session, keeping a client session. One of the simplest session keeping instruments for Express.js. [25]

- helmet, providing a set of headers helping maintain the application more secure. [26]

- mocha, a test framework used for Node.js. Allows to perform various kinds of testing with help of other modules. It is a part of devDependencies within

package.json, which means that it is not going to be uploaded by a non-developer. [27]

- multer, handling files sent by client. Body-parser module does not handle multipart-bodies sent by client due to their complexity. Therefore, multer has been chosen as a module serving for this purpose. Multer lets certain actions to be performed before uploading, such as checking size, mimetype etc., and then creates a random filename for it while uploading. [28]

- passport, authentication middleware. Passport's advantage is that it allows creating various strategies for authenticating, such as performing a standard one with a username and a password or using a 3rd party authentication. [29]

- passport-local, authenticating client using username and password. A standard authenticating strategy that is already provided. [30]

- supertest, providing HTTP testing with a high level of abstraction. It is a part of devDependencies within package.json, which means that it is not going to be uploaded by a non-developer. [31]

3.3 List of accessible web pages

The only web page available to both companies and talents is the home page:

- Home page, available at "/". One of the two pages available to any unauthorized users. In case an unauthorized user tries to visit any other web page, they will be redirected to the home page. This page does not require any data to be taken from the database.

There are 2 pages available for administration needs:

- Administration Login page, available at "/admin". This page should be configured to be accessible only within private networks. Allows the administrator to log in the system.

- Administration operations' page, available at "/back-office". This page should be configured to be accessible only within private networks. Shows the logged in administrator users' data.

A list of web pages available for a company is as follows:

- Company's profile page, available at "/company_profile", a home page for an authorized company user, where they will be redirected in case of 404 Not Found error. Requires company collection's data for this company to be taken from the database.

- Company's jobs page, available at "/company_jobs", displaying jobs posted by this company. Requires company collection's data for this company, jobs associated with this company and talents applied to these jobs to be taken from the database.

- A single job description page, available at "/company_profile_job_offer_details", displaying all of the requsted job's data. The user will be redirected to the company's jobs page if a requested job does not exist. Requires company collection's data for this company and the requested job's data to be taken from the database.

- Company's favourite talents page, available at "/company_fav_talents", displaying a list of talents marked as favourite for this company. Requires company collection's data for this company and talents present in fav_talents field to be taken from the database.

- New job creation page, available at "/company_add_job", allowing the company to create a new job. If a job with a given job identifier exists, it's data will be copied, prefilling a new job's fields respectively. Requires company collection's data for this company, a job if an identifier is given and all required fields to be taken from the database

- Company video page, available at "/company_video', displaying company's introductory video or a lack of such. Requires company collection's data for this company to be taken from the database.

- Find talents page, available at "/talents", allowing to search for any talents by category, expertise level or flexibility. Requires category, expertise level and flexibility collections' data to be taken from the database.

- Talent search results page, available at "/talents_list", displaying a list of talents in accordance with the chosen parameters using lazy loading with 12 results shown per request. If none parameters were given, talents are queried without restrictions. Requires talents in addition to category, expertise level and flexibility collections so that company could make a new search on this page.

    - A single talent description page, available at "/talents_public_profile", displaying a chosen talent's profile information. Requires a talent collection's data for this talent to be taken from the database.

A list of web pages available for a talent is as follows:

- Talent's profile page, available at "/personal_profile", a home page for an authorized talent user, where they will be redirected in case of 404 Not Found error. Requires talent collection's data for this talent to be taken from the database.

- Talent's skills page, available at "/personal_skills", displaying talent's skills arranged by categories. Requires talent collection's data for this talent to be taken from the database.

- Talent's favourite jobs page, available at "/personal_fav_jobs", displaying a list of jobs marked as favourite for this talent. Requires talent collection's data for this talent and jobs present in fav_jobs field to be taken from the database.

- Talent's applied jobs page, available at "/personal_applied", displaying a list of jobs marked as applied for this talent. Requires talent collection's data for this talent and jobs present in applied_jobs field to be taken from the database.

- Talent's favourite companies page, available at "/talent_fav_companies", displaying a list of companies marked as favourite for this talent. Requires talent collection's data for this talent and companies present in fav_companies field to be taken from the database.

- Talent video page, available at "/talent_video', displaying talent’s introductory video or a lack of such. Requires talent collection's data for this talent to be taken from the database.

- Find jobs page, available at "/jobs", allowing to search for any jobs by category, expertise level or flexibility. Requires category, expertise level and flexibility collections' data to be taken from the database.

- Jobs search results page, available at "/jobs_list", displaying a list of jobs in accordance with the chosen parameters using lazy loading with 12 results shown per request. If none parameters were given, jobs are queried without restrictions. Requires jobs' data to be taken from the database in addition to category, expertise level and flexibility collections so that talent could make a new search on this page.

- A single job offer page, available at "/job_offer_details", displaying all of the requsted job's data and allowing to mark this job as favourite or applied to. Requires the requested job's data and the company associated with it to be taken from the database.

By the time of writing this thesis the talent's part has not been analysed completely and there are still some web pages to be designed. Therefore, this list could be changed in the future.

Naturally, there are also web pages dealing with a pricing model, which defines user's method of payment. As its' ultimate format is still being discussed, the pricing model is not reviewed in this thesis.

3.4 List of required AJAX responses

In addition to standard web page responses, the application also contains certain web addresses for AJAX requests in order to allow the client-side to load some of the data dynamically. Such requests come using a POST method and the format used in both requesting and responding is JSON due to the fact that both client-side and server-side are written in JavaScript and thus do not need to parse this format.

The following web addresses could be divided intro 3 categories: those, available to either a company, a talent or both.

3.4.1 Company's web addresses

- /add-talent-to-favourites, adding the requested talent to company's favourite talents' list. Returns an empty message with 204 HTTP status code if successful.
- /remove-talent-from-favourites, removing the requested talent from company's favourite talents' list. Returns an empty message with 204 HTTP status code if successful.
- /save-job-answer, saving an answer for a requested question about a job. Returns an empty message with 204 HTTP status code if successful.
- /remove-job-answer, removing an answer from a requested question about a job. Returns an empty message with 204 HTTP status code if successful.
- /save-company-story, updating a company's brief description about itself. Returns an empty message with 204 HTTP status code if successful.
- /upload-company-logo, uploading company's new logo image. Returns an uploaded file's name on the server if successful.
- /remove-company-logo, removing company's logo image. Returns an empty message with 204 HTTP status code if successful.
- /upload-job-logo, uploading job's new logo image. Returns an uploaded file's name on the server if successful.

- /upload- job-photo, uploading job's new photo image. Returns an uploaded file's name on the server if successful.

- /remove- job-logo, removing job's logo image. Returns an empty message with 204 HTTP status code if successful.

- /remove- job-photo, removing job's logo image. Returns an empty message with 204 HTTP status code if successful.

- /add-job, adding a new job. Returns the _id field of a new job in case saving was successful.

3.4.2 Talent's web addresses

- /apply-for-task, adding a requested job to talent's list of applied jobs. Returns an empty message with 204 HTTP status code if successful.

- /disapply-from-task, removing a requested job from talent's list of applied jobs. Returns an empty message with 204 HTTP status code if successful.

- /add-job-to-favourites, adding a requested job to talent's list of favourite jobs. Returns an empty message with 204 HTTP status code if successful.

- /remove-job-from-favourites, removing a requested job from talent's list of favourite jobs. Returns an empty message with 204 HTTP status code if successful.

- /ask-question, saving a new question about a requested job. Returns an empty message with 204 HTTP status code if successful.

- /jobs_list_update, serving a new set of jobs that are not within jobs array if it is provided.

- /jobs_list_sort, serving a new set of jobs that are sorted according to the parameter provided.

3.4.3 Common user's web addresses

- /save-contacts, updating a list of company's contact information. Returns an empty message with 204 HTTP status code if successful.
- /toggle-profile-public, changing user's profile state from non-public to public. Returns an empty message with 204 HTTP status code if successful.
- /toggle-profile-non-public, changing user's profile state from public to non-public. Returns an empty message with 204 HTTP status code if successful.
- /toggle-active, changing user's state from inactive to active. Returns an empty message with 204 HTTP status code if successful.
- /toggle-inactive, changing user's state from active to inactive. Returns an empty message with 204 HTTP status code if successful.
- /upload-company-photo, uploading user's new photo image. Returns an uploaded file's name on the server if successful.
- /remove-company-photo, removing user's photo image. Returns an empty message with 204 HTTP status code if successful.
- /upload-video-link, setting user's YouTube video identifier to the requested one. Returns an empty message with 204 HTTP status code if successful.
- /remove-video-link, removing user's video identifier. Returns an empty message with 204 HTTP status code if successful.

As with the web pages, the AJAX responses are most likely going to be updated in the future.

4 Experiment results and analysis

By the time of writing this thesis, the bigger part of web pages mentioned above have been completed. Mainly, the administration and the majority of company parts with the exception of talent searching have been completed in addition to jobs' searching, listing and single job display. Web pages for displaying talent's favourite companies and jobs, applied jobs as well as video have not been designed, leaving talent's part development yet to be served. The index page also requires certain adjustments, leaving the use of TLS unnecessary for the time being. The completed pages are thus ready to be tested.

Mocha testing framework has been chosen as the primary testing component due to high popularity among Node.js projects and ease of use. [1]

Mocha provides a certain shell for testing, while Supertest package has been chosen as an assertion library for testing HTTP requests and responses. [1]

The method used for testing this application is to check if all the web pages respond with content type of "text/html" to verify that a web page is served indeed. If a web page responds with a 200 status code, then it means that there have not been any errors during database querying. An example of such a function is provided in Appendix 1. To assure that these responses are not automatic or random, it is also wise to try request an unexisting web address. The resulting status code should be 404 Not Found. All of the performed tests have concluded successful operations.

Dynamic requests, on the contrary, are not responded with "text/html" format. As mentioned above, the majority of these respond with an empty message in case an operation was successful. An example of such a function is provided in Appendix 2. Similarly to the web pages' testing, all of the AJAX tests have concluded successful operations.

Also, nsp, a module checking for threats among dependencies that was mentioned above, responds with "(+) No known vulnerabilities found". It can be then concluded that there the modules used in this project are safe.

5 Summary

The main goal of this work was to develop server-side part of a Freelance Marketplace web application by using Express.js and MongoDB and creating both application and data layers respectively.

The first chapter describes MongoDB and Express.js, mentioning security aspects for both of these, Mongoose module that helps connecting these with each other, error handling for Express.js as well as project structuring. As the result of this analysis, the conditions for developing the application and data layers as effectively as possible have been met.

The second chapter describes the experiment carried out, namely the database design and the Node.js modules that were chosen as well as the list of web addresses available to the client-side. Database was designed with a higer level of denozrmalization, which is a common approach among NoSQL databases. Express.js standard middleware and routing have been used for handling requests with all the other modules used for various supporting tasks.

The third chapter sees Mocha and Supertest modules being used for testing web addresses that are currently ready to be tested. Testing has highlighted that there are not any problems with the application. However, since these are web addresses that are tested, it is still possible that an incorrect page is served instead of another one or that some incorrect connections to the database take place. Therefore, while these testing methods are crucial for the application to be working correctly, it would ge a good idea to further extend these.

There are still tasks left to be carried out, which are mentioned above. On top of that, security and error handling aspects should be considered furthermore as there are many new threats appearing every day. There is also a 5^{th} version of Express.js currently in the alpha release stage, which should be considered in the future. Finally, a change from development to production stage should be prepared.

References

[1] E. M. Hahn, Express in Action, 2016.

[2] K. Chodorow, MongoDB: The Definitive Guide, 2013.

[3] P. Shan, "Mongoose vs mongodb native driver – what to prefer?," 7 June 2015. [Online]. Available: http://voidcanvas.com/mongoose-vs-mongodb-native/. [Accessed 22 May 2017].

[4] M. Wright, "2-Tier vs. 3-Tier Application Architecture? Could the Winner be 2-Tier?," 14 May 2015. [Online]. Available: http://nitrosphere.com/2-tier-vs-3-tier-application-architecture-could-the-winner-be-2-tier-2/. [Accessed 21 May 2017].

[5] MongoDB, "NoSQL Databases Explained," [Online]. Available: https://www.mongodb.com/nosql-explained. [Accessed 21 May 2017].

[6] Mozilla Developer Network, "Promise," 19 May 2017. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. [Accessed 21 May 2017].

[7] Express, "Production Best Practices: Security," [Online]. Available: http://expressjs.com/en/advanced/best-practice-security.html. [Accessed 21 May 2017].

[8] P. Serby, "Case study: How & why to build a consumer app with Node.js," 7 January 2012. [Online]. Available: https://venturebeat.com/2012/01/07/building-consumer-apps-with-node/. [Accessed 20 May 2017].

[9] J. Meier, "Layers and Tiers," 5 September 2008. [Online]. Available: https://blogs.msdn.microsoft.com/jmeier/2008/09/05/layers-and-tiers/. [Accessed 21 May 2017].

[10] Microsoft Developer Network, "Tutorial 2: Creating a Business Logic Layer," June 2006. [Online]. Available: https://msdn.microsoft.com/en-us/library/aa581779.aspx. [Accessed 21 May 2017].

[11] N. Loorits, "Disainimustrid," February 2017. [Online]. Available: https://ained.ttu.ee/pluginfile.php/15760/mod_resource/content/1/3.%20Disainimustrid.pdf. [Accessed 21 May 2017].

[12] "Disable Transparent Huge Pages (THP)," [Online]. Available: https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/. [Accessed 20 May 2017].

[13] Express, "Error handling," [Online]. Available: http://expressjs.com/en/guide/error-handling.html. [Accessed 21 May 2017].

[14] Zanon, "NoSQL Injection in MongoDB," 17 July 2016. [Online]. Available: https://zanon.io/posts/nosql-injection-in-mongodb. [Accessed 21 May 2017].

[15] Express, "Moving to Express 4," [Online]. Available: http://expressjs.com/en/guide/migrating-4.html. [Accessed 22 May 2017].

[16] Express, "Express application generator," [Online]. Available: http://expressjs.com/en/starter/generator.html. [Accessed 22 May 2017].

[17] Ian, "Creating an MVC Express.js Application," 18 October 2014. [Online]. Available: http://www.9bitstudios.com/2014/10/creating-an-mvc-express-js-application/. [Accessed 22 May 2017].

[18] Handlebars, "handlebars," [Online]. Available: http://handlebarsjs.com/. [Accessed 22 May 2017].

[19] M. Vollmer, "Understanding callback functions in Javascript," 22 March 2011. [Online]. Available: http://recurial.com/programming/understanding-callback-functions-in-javascript/. [Accessed 21 May 2017].

[20] C. McMahon, "async," [Online]. Available: https://www.npmjs.com/package/async. [Accessed 22 May 2017].

[21] D. Wirtz, "bcryptjs," [Online]. Available: https://www.npmjs.com/package/bcryptjs. [Accessed 22 May 2017].

[22] P. Antonov, "bluebird," [Online]. Available: https://www.npmjs.com/package/bluebird. [Accessed 22 May 2017].

[23] Express, "body-parser," [Online]. Available: https://www.npmjs.com/package/body-parser. [Accessed 22 May 2017].

[24] E. Ferraiuolo, "express-handlebars," [Online]. Available: https://www.npmjs.com/package/express-handlebars. [Accessed 22 May 2017].

[25] Express, "express-session," [Online]. Available: https://www.npmjs.com/package/express-session. [Accessed 22 May 2017].

[26] Helmet, "helmet," [Online]. Available: https://www.npmjs.com/package/helmet. [Accessed 22 May 2017].

[27] Mocha, "mocha," [Online]. Available: https://www.npmjs.com/package/mocha. [Accessed 22 May 2017].

[28] Express, "multer," [Online]. Available: https://www.npmjs.com/package/multer. [Accessed 22 May 2017].

[29] J. Hanson, "passport," [Online]. Available: https://github.com/jaredhanson/passport. [Accessed 22 May 2017].

[30] J. Hanson, "passport-local," [Online]. Available: https://www.npmjs.com/package/passport-local. [Accessed 22 May 2017].

[31] "supertest," [Online]. Available: https://www.npmjs.com/package/supertest. [Accessed 22 May 2017].

Appendix 1 – Testing Web Address Response

Testing if a web page is being served using Supertest module.

```
var supertest = require('supertest');
var app = require('../app');

describe('HTML text response', function() {

    it('should return an HTML company_profile page', function(done) {
        supertest(app)
            .get('/company_profile)
            .set("Accept", "text/html")
            .expect("Content-Type", /text\/html/)
            .expect(200)
            .end(done);
    });

});
```

Appendix 2 – Testing AJAX Response

Testing if an AJAX response is being served using Supertest module.

```
var supertest = require('supertest');
var app = require('../app');

describe('HTML text response', function() {

    it('should return an HTML company_profile page', function(done) {
        supertest(app)
            .get('/company_profile)
            .set("Accept", "text/html")
            .expect("Content-Type", /text\/html/)
            .expect(200)
            .end(done);
    });

});
```