COMP 4971C - Independent work
Spring 2015

Investigation into Full-Stack JavaScript as a web server

Final Report

Completed by: Melkar Muallem

Supervised by: Dr. David Rossiter

Table of Contents

INTRODUCTION

Motivation

There is a niche in the market for social networks that target specific professional segments, especially in the arts community. This project started with the initial goal of building such platform to facilitate business networking between the different parties involved in theater production. This product has a high opportunity of success due to the lack of any service of this sort which forces these stakeholders to conduct a lengthy process to find and contact other parties, or be forced to hire a third party agent that handles it.

The nature of the project soon changed to become an experimentation with the current trending tools in web development. At the center of the investigation is Node.js, a JavaScript platform for building network applications. Using front-end JavaScript libraries, and a JSON-based database puts JavaScript on all parts of the stack, increasing efficiency, scalability, speed, and resource re-usability. This stack is dubbed full-stack JavaScript.

Throughout the development, I explored various solutions to build my full-stack JavaScript application, and I modified it to solve the problems I faced. The sole motivation was to learn and test different open source APIs and to improve my personal skills in web development. Because of this shift in the focus of the project. The report will focus more on the process of development than the final product.

Deliverables

A web application that is fast and scalable and built in a single-page application format. which provides the users with the ability to conduct basic social network activities such as search for other users, chatting, posting updates, and creating connections.

A full-stack JavaScript application with a server built with Node.js using the Express.js web framework, and a secure Database that holds all user data built with MongoDB, and a responsive, single-page front-end developed using React. The server is hosted on Heroku and the database on Mongolab thus the users can access the application through the Herokuapp live website without the need for any further deployment.

FEATURES

This website allows the users to conduct basic social networking activities such as creating and browsing profiles, contacting other users, and adding them to their personal networks.

- Single-page Application

  As seen in the picture bellow, the application follows the single-page pattern, under which no reloading occurs. Each component is rendered separately during runtime; making the experience for the user more fluid.

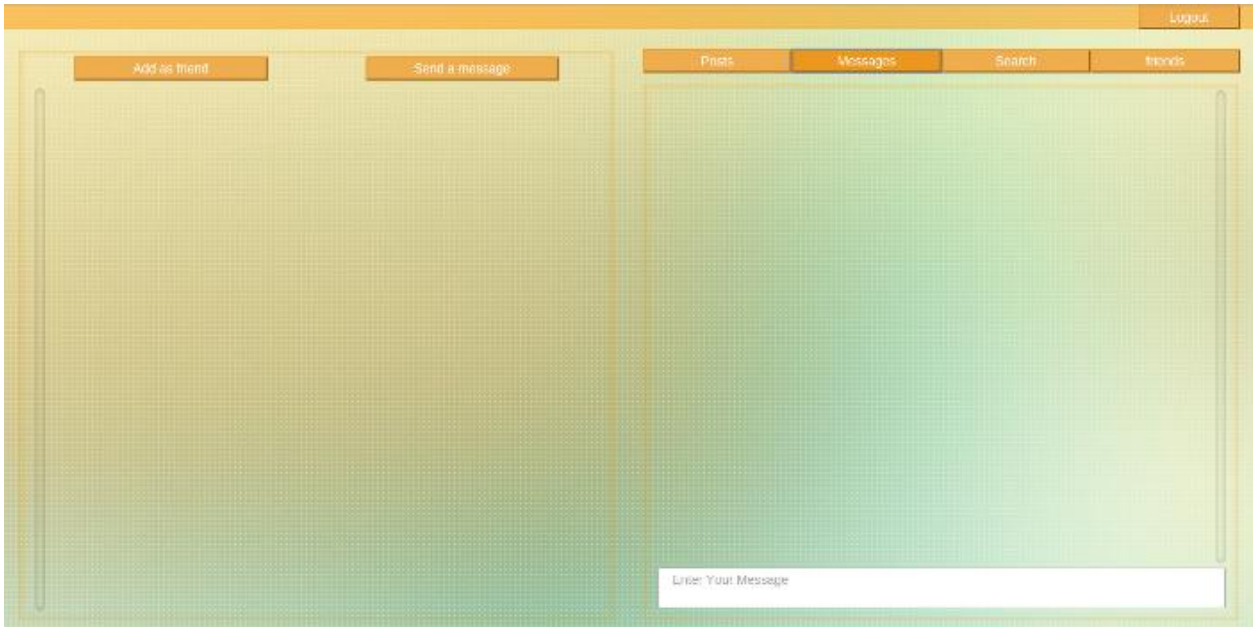


Figure 1. Main view of the application

- Specialized profiles

  These will be displayed on the left section.

  To better accommodate for the different parties in the targeted audience, each user account can have one profile associated with it which can be one the following: Individual, Team, or Organization. Each profile contains general information about the user such as name, location, picture, but differs from detailed information in the other profile types. For example, a team can have a field for number of members which is unnecessary for individual accounts.

  This is more representative of the users' information and self, and can be further improved to contain all possible information. The separation also facilitates the searching as users can locate targets of interest in a more natural fashion.

  In the future it would be very easy to create new profiles for the site or down the hierarchy.

- Posting

  A user is able to post status updates in this section that can be viewed publicly by other users who are viewing this person's profile. To do this, the posts component will display the posts of the user whose profile is currently being viewed -in the left section.

- Messaging
  - Messages between users are stored in a thread format; i.e. it is stored as a continuous conversation between two parties.
  - A user is be able to message any other member of the network by clicking on the Message User button in the profile section.
  - When the message tab is first open, it displays the latest messages the user received.
  - Clicking on one expands that message into the full conversation between the two users.
  - If both users are viewing the same conversation at the same time, they can have real-time chat.

- Search
  - Through the search function, users are be able to search for other members of the website.
  - The search query looks for any user whose name is matched with the query.
  - There should be several filters for the search. Most importantly, occupation and location.
  - The result should be a short summary of the users' profiles, and when one is clicked, their full profile would be displayed in the profile section.

- Friends
  - This tab displays the list of friends the user is in connection with (friends).
  - The result should be a short summary of the users' profiles, and when one is clicked, their full profile would be displayed in the profile section.

Some other features of the website include:

- Keeping user sessions using secure cookies without the implementation of SSL.
- A stateless server, where all states are stored either in the front-end or database-stored cookies and none is stored on the server.

Before dwelling into the details about the implementation of these feature, I will first explain the technologies used in the next section, after which I will explain how I programmed the problems above, and other challenges that I faced.

While the working concept for the website did not change much since its inception, the tool set I used is continuously changing. Nevertheless, I decided from the begging that I want to try out Node.js which I had no idea about prior to this project. I have heard a great deal of positive feedback about Node.js which made me very curious, and because I have had experience in JavaScript before, I thought this was appropriate. From that point forward, I learnt a great deal about JavaScript, Node.js and its modules, and various open source libraries which I will now discuss more intensively.

HYPOTHESIS

JavaScript

JavaScript originally named Mocha is a “multi-paradigm language, supporting object-oriented, imperative, and functional programming styles.”[1] JavaScript started on web browsers as language to dynamically interact with the user and control the components of the page. JavaScript’s language architecture is quite unique from other languages which is especially handy when developing web application that rely on non-blocking operation.

JavaScript architecture combines object-oriented, functional and scripting languages paradigms. Syntax wise, JavaScript is based on C’s structured layout, with subroutines, block structures and loops. While it is object-oriented, JavaScript does not have classes, but instead rely on object prototypes for inheritance. This means that the methods and fields of the objects’ prototypes may be dynamically changed for future construction of these objects. Functions in JavaScript are also objects, and can be sent as arguments to other functions as the case in functional languages; functions can also have methods and properties of their own. JavaScript also provides dynamic typing as well as static typing. Variables could be defined with a specific type such as Number of String, or casted dynamically by initializing it as a var. Moreover, JavaScript, like scripting languages, allows the use of variadic functions (number of arguments is not defined), and supports Regular expressions such as in Perl. Finally, JavaScript’s support for associative arrays is the basis for the construction of JSON data formats. Using a single language throughout the stack enables the reuse of resources such as JSON objects which can be manipulated the same way by any part of the stack.

There were several attempts aiming to place JavaScript on the server side ever since the mid-1990s. But, none of these solutions were as successful as Node.js which is pioneering a new way of server programming focused mainly around asynchronous operations. There are three main reasons that made JavaScript the language of choice of Node.js.

1. Google’s V8 is an open-sourced high-performance JavaScript execution engine built for Chrome. It complies JavaScript to native machine code instead of interpreting it in real time. V8 and other JavaScript runtime engines also provide a concurrency model using a message queue and an event-loop which allows JavaScript to stack operations and their callbacks and execute the callback when the operation is done.
2. JavaScript is built around an event-driven interaction model because it depends on user actions, which makes asynchronous, non-blocking and callbacks natural, as they are event driven as well.
3. JavaScript is an interpreted language which makes it platform independent.
4. JSON (JavaScript Object Notation). JavaScript has been the language of choice to control the web for a long time; and since the old days, data had to be marshelled into JSON objects when sent to the web. Because of the high dependence on JSON objects, a new

1 http://en.wikipedia.org/wiki/JavaScript

kind of JSON like databases were created, enabling the easy exchange of data between back-end and front-end.

Node.JS

Node JS is a JavaScript platform for building fast, scalable, network applications built on Google’s V8 Engine. Node is single threaded and built around the paradigm of none-blocking IO. With Node.js each incoming request by the user is handled by one single thread in opposition to the multi-threaded techniques used by PHP to scale the operations. Each request handled by this thread is coupled with a callback function that is called upon completion of the task. This is possible due to the fundamental support of JavaScript for events, Asynchronous operations, and callbacks; and Node.js puts JavaScript on the server side.

Node has several advantages, some of which are:

- RESTful API. As Node can build an HTTP server out of the box, it can communicate with all other components through HTTP methods for CRUD operations based on the RESTful paradigm.

- Single Threaded. Since it is not blocking I/O operations, Node can handle all user requests using a single thread, instead of allocation of new thread for each request, which has a large memory footprint. Nonetheless, Node does use a thread pool at the kernel level to guarantee that the operations are being executed asynchronously without blocking the event-loop. This is necessary because the kernel does not support all operations asynchronously.

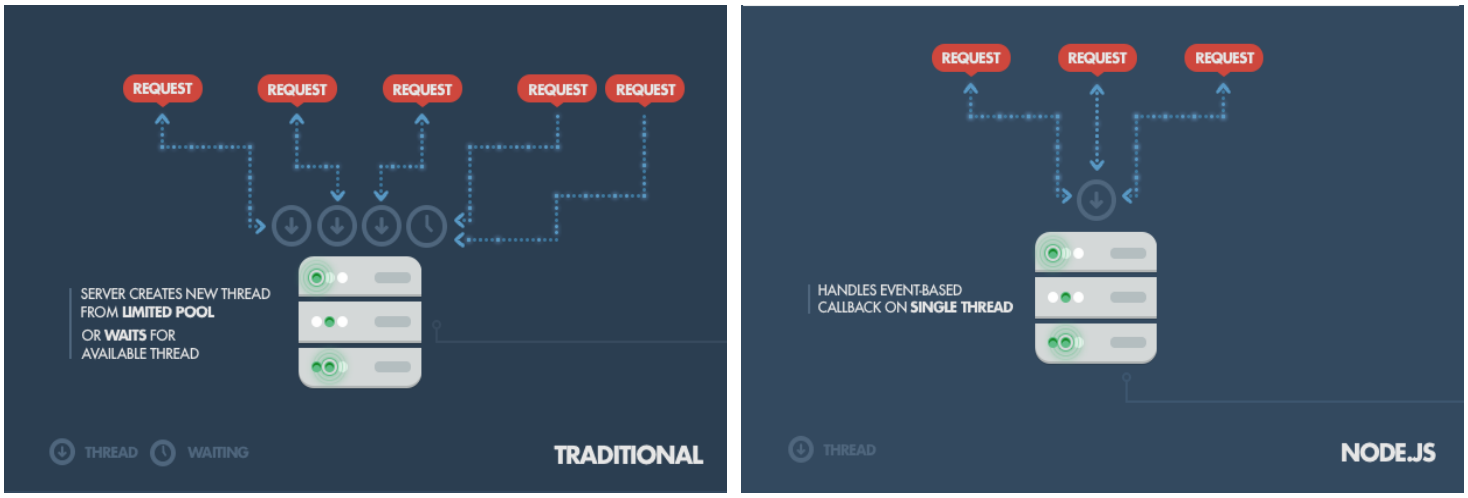


Figure 2. Node.js single-thread model vs traditional servers *http://pt.slideshare.net/GustavoCorra/node-js-javascript-*

- NPM: The Node Package Manager is based on JavaScript's npm. A built-in module that supports package management, it can be used to easily download and install modules for a Node application. Moreover, Node already has many packages and libraries developed to work on top of it; all of which confine to the asynchronous nature of Node.js.

- None-blocking I/O. Because Node uses an Event-loop to allow asynchronous operations and callback functions, it is capable to setting the requesting function on the side till the task is completed and meanwhile handling other tasks.

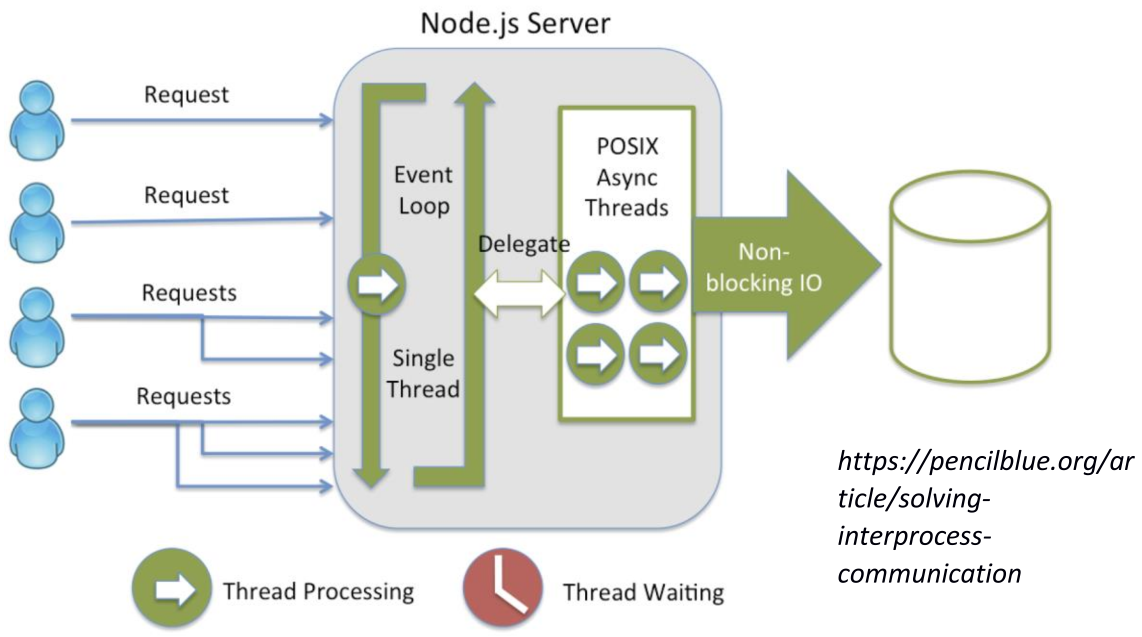


Figure 3. Node.js concurrency model and event-loop

- There is also an advantage for using JavaScript on the server because data structures and logic may be shared by both front-end and back-end which increases the synergy of the developers' team as resource may be shared. Furthermore, using a unified data structure across the stack makes transfer easier and removes the overhead of data conversion and casting.

- Finally, Node is supported by a great community that is very vibrant and dedicated to constantly improve Node.js itself or develop new modules for it.

Nevertheless, Node.js is not problem free and has flaws that sometimes hinder production level websites. The most confusing part with Node.js for a new comer, is to get adapted to the asynchronous programming. To assure that the single thread Node.js provides does not get delayed, it is necessary to guarantee that no blocking operations occur in the stack so that the thread does not get hanged which would deter the whole performance of the server. Getting used to the Async paradigm is not easy though, and callbacks tend to get nested and hard to trace back.

There is also the issue of scaling on different CPUs which Node is incapable of performing due to its single thread; it can only run on a single CPU and cannot be scaled on big server pools in contrast to distributed programming possible with other languages. At the same time this limits Node's capabilities. Node tend to perform very well with basic I/O operations which are blazingly fast with callbacks. But, since a single thread is used, all operations [callbacks], will be executed in the stack, on that single thread, which would harm the performance significantly if the task blocks for a relatively long time. This limit's Node capabilities as a full sever, and tends to do well only in simple websites like chatrooms and blogs.

Node.js has faced many problems due to its immaturity at this stage, especially when using modules which are not tested in big production means that there is big margin of error. This may cause in some cases memory leaks that would increase latency and CPU usage, and if it occurs while there are thousands of simultaneous requests, the server tends to crash and must be restarted.

There are hundreds of different programs, frameworks, plugins, and other software developed for Node.js. The main programs my stack will need is a database, a web server framework for Node, and a front-end engine. A commonly used full-stack JavaScript web application is the MEAN Stack (**M**ongoDB, **E**xpress.js, **A**ngular.js, **N**ode.js) which acted as my initial entry point to this project. In the final product, I am still using Node.js with Express.js on top, and MongoDB as my database, the only part that saw drastic changes was the front-end development framework. Of course, additional APIs were also used for other purposes as will be shown.

Express.js

Although Node is capable of independently act as a web server, there are frameworks designed to make it more powerful and efficient -the most popular being Express.js. It is fast, minimal, and has several applications built for it. Express was chosen for several reasons among them are:

- Minimalistic. Express makes it possible to build an HTTP server very easily by wrapping the backend code of Node.js, and it makes the building of a RESTful API very simple.
- Express supports middlewares which are functions called in a sequential order on a request or response. For example, a middleware I am using is body parser, which parses the body of incoming HTTP requests with a form submission. This allows me to access the parameters of the request directly. Middlewares can do anything, and they end with next(), which calls the next middleware.
- Routes are Express's way to handle incoming requests on a certain URL. Express after detecting the incoming requests directs it to a specific routing JavaScript file that handles all the logic associated with that URL. This is precisely helpful to keep the code lean and organized.
- Supports a wide range of available templates engines out of the box.
- Provides a range of extra helping features such as being able to redirect the user to another URL from within the server.
- A good community. Express is one of the original Node web frameworks, the community has developed very comprehensive tutorials, online examples, extensive documentation, and powerful tools for Express.

Express though, does not provide a proper MVC structure to the application, and models would have to be created using other libraries such as Mongoose which is being used in this project.

MongoDB

Additionally on the backend, there should be a database that stores user data. It would be preferable to user a JSON –NoSQL- based database to leverage the benefits of using JavaScript across the stack when the same objects stored in the DB can be processed by the server and the front-end without any additional conversion. I started my testing with a pioneer of NoSQL databases, and a member of the MEAN stack, MongoDB. Mongo is just the database software itself, and a driver is needed to connect node with the database instance and provide a layer for I/O with the database. Preferable also, there should be defined schemas for the database documents.

There are several advantages for using MongoDB such as:

- BSON (Binary JSON) storage types. JSON types used by JavaScript is very similar to xml in terms of its construction; instead of using a table, the whole collection (MongoDB's equivalent of an SQL table) can be written as a nested list of fields and values.
- On its own, MongoDB is schema less, which provides the users with an easier way to append objects into the database. It also certain documents within the collection may have different values from others. For example, a user can have as many number of phone numbers' fields in their document as they own without any need to change the other documents in the same collection.
- Indexing. This is a feature of MongoDB in which any field that is 'required' in a collection, if indexed, would be added to a sorted array with the values of this field in all documents. This way searching the collection can be made very quickly.
- Auto-sharding. In a production system with more than one server, MongoDB can split each collection across the different servers, thus enabling an easy way to scale horizontally.
- Location based data. MongoDB natively supports geo-spatial coordinates and data indexing accordingly.
- A strong, active community with many tutorials and examples available online.
- MongoLab. MongoLab is a cloud database hosting service for MongoDB providing up to 500MB of free storage and an online UI for monitoring and managing the database.

As mentioned, a driver should be used to connect the server to the Database. In this development, I tested two drivers. The first was the native MongoDB driver for Node, which provided a wrapper around Mongo's own commands. This though was not enough for me as I wanted to define my own schema in order to enjoy an MVC layout for my application, and also add an extra layer of security. For this reason, I quickly switched over to Mongoose an Object-Relational Mapper (ORM) driver for MongoDB.

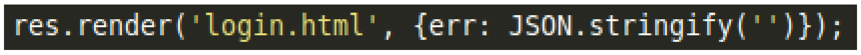
Mongoose enabled me to create schemas and models for my documents in the database, which I could use to verify data being inserted into the DB. The schema modeling includes "built-in type casting, validation, query building, business logic hooks and more, out of the box."[2] It also provides abstraction to top level Mongo operations such as creation of documents and collections, connecting to a database, and defining indexes, requires fields in a document, and more. Another important feature of Mongoose is population, which is similar to joining in relational databases.

REACT

Finally I had to create my front-end, and to make full use stack it would be best to use a JavaScript library for generating UI components. The MEAN stack uses Angular.js and thus I wanted to start with it. But, I did not test Angular, instead I chose to try ReactJS a similar library by Facebook that is increasingly gaining popularity since its release. React was used to create the UI seen in the first figure.

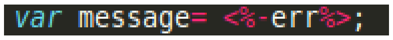
In comparison to Angular, which is a very complex framework, React is quite minimal because it is just plain JavaScript, whereas Angular leverages the benefits of templates. In my short experience with templates, I faced a situation in which template engines were extremely useful, but the React engine suffered greatly.

In this scenario, I try to pass an Express response to the front-end loaded with a value calculated on the server side.



```
res.render('login.html', {err: JSON.stringify('')});
```

The res here is the Express response object that is sent to the client. It is rendering login.html which is built by React, but rendered by a template engine called Ejs, and passes a JSON object as the error message. Ejs then enables me to access the err object from within the html file like this:



```
var message= <%-err%>;
```

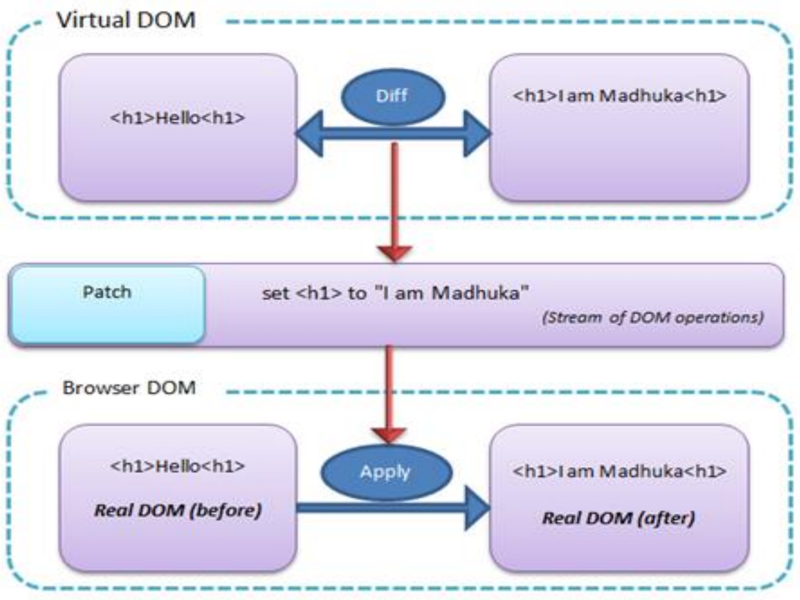
This is impossible with React by itself because it is translated to pure JS, which means I need to retrieve this data via Ajax. So why not just use a templating engine? The problem is when I wanted to create a single-page application as seen in figure 1 where no html files are rendered, thus I am relying purely on React to manipulate the page, and I must use a different engine called Browserify in order to accomplish that and it does not provide this functionality.

React itself is quite a unique library in its field. It has an object-oriented approach and builds UI by breaking them into classes and instances, and each components inherits from super

2 http://mongoosejs.com/

classes similar to other OO languages. It also relies on methodologies used by game engines to update the UI in real-time.

React has its own virtual DOM loaded with the current state of the browser DOM. After an interaction, or an update to a component in React, a Diff operation between new state of the object and the one in the virtual DOM is called -similar to that of Git, which checks the shortest path to apply this change, and then patches the browser DOM. This is extremely important because native DOM operations are very slow, while the JavaScript implementation of the DOM used by React is superiorly faster, and this operation should be as efficient as possible.



http://madhukaudantha.blogspot.hk/2015/04/reactjs-and-virtual-dom.html

Figure 4. React diff and update process

React is very responsive because of this, and its syntax allows for great abstraction in the structure of the code, and enables for easier debugging because decoupling is easy. Also, since React allows for a convenient way to control the state completely on the client-side, it can act as a complete MVC model on its own, where the view is compared to the virtual DOM -model- by the controller that is React within functions like ShouldComponentRender.

As mentioned I am also using a library called Browserfiy which makes it possible to require (import) in my html views. This means that my different React Components can be in their respectful JS files, and I just import them when needed.

In addition I am also using a library called Gulp that automates commands for compiling and deploying (or virtually any set of console commands) of my service.

The beautiful thing is that Browserify, Gulp, and any other library I use are all JavaScript based. Which makes the compatibility work beautifully, and the learning curve not steep.

IMPLEMENTATION

- Single-page

  Using React and Browserify, I split each part of the main-page into a single React class -components- in its own file. These components are Profile, Post, Chat, Search, and Friends; all of which are imported in a main component class. Since React allows for easy management of state variables, I am able to easily send down data from the main component to the others. Nevertheless, sideways data communication is harder, for this reason I used Socket.io; by taking advantage of the fact that a single socket -each user has their own socket- can broadcast messages to itself, I can send events and data between single components.

  Since my main application component can manipulate the page at well, and change between the rendered components, the page does not need to reload and no other pages need to be downloaded. All actions with the server happens through Ajax mostly via Socket.io.

- User-sessions

  To keep a user logged in I am using express sessions saved in my remote database. This allows me to keep secure sessions without the need to encrypt it as they are not stored on the client-side. Express associates a single IP with a session document stored in the database. When the IP disconnects or the cookie timeouts the session is deleted from the database.

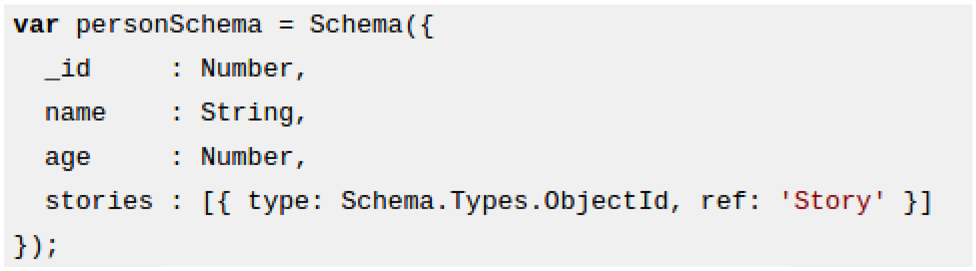
  This is also an extreme advantage as I store sensitive data in the session cookie such as the user's unique id from the user’s account in the database for quick data retrieval if needed. JavaScript does this very greatly using prototypes as I can add new fields over the session prototype at any given moment.

- State-less server

  As mentioned, the server keeps no state at all. The state of the user's session is either stored through React components or as part of the session object in the database.

- Specialized profiles

  The first level of defining these profiles comes at a schema level in the database. I have a single Schema that is universal for all user accounts regardless of the type of profile. This schema holds information such as name, username, email, and hashed password. But it also contains a reference to the user's profile. Originally I wanted to be able to use Mongoose's population to retrieve the users' profiles from their designated documents. Population in Mongoose copies data from one document to the other by storing the unique id of the first document in the latter.

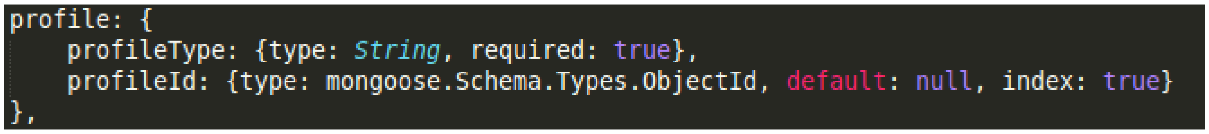


```
var personSchema = Schema({
  _id     : Number,
  name    : String,
  age     : Number,
  stories : [{ type: Schema.Types.ObjectId, ref: 'Story' }]
});
```

Figure 5. Mongoose Schema example

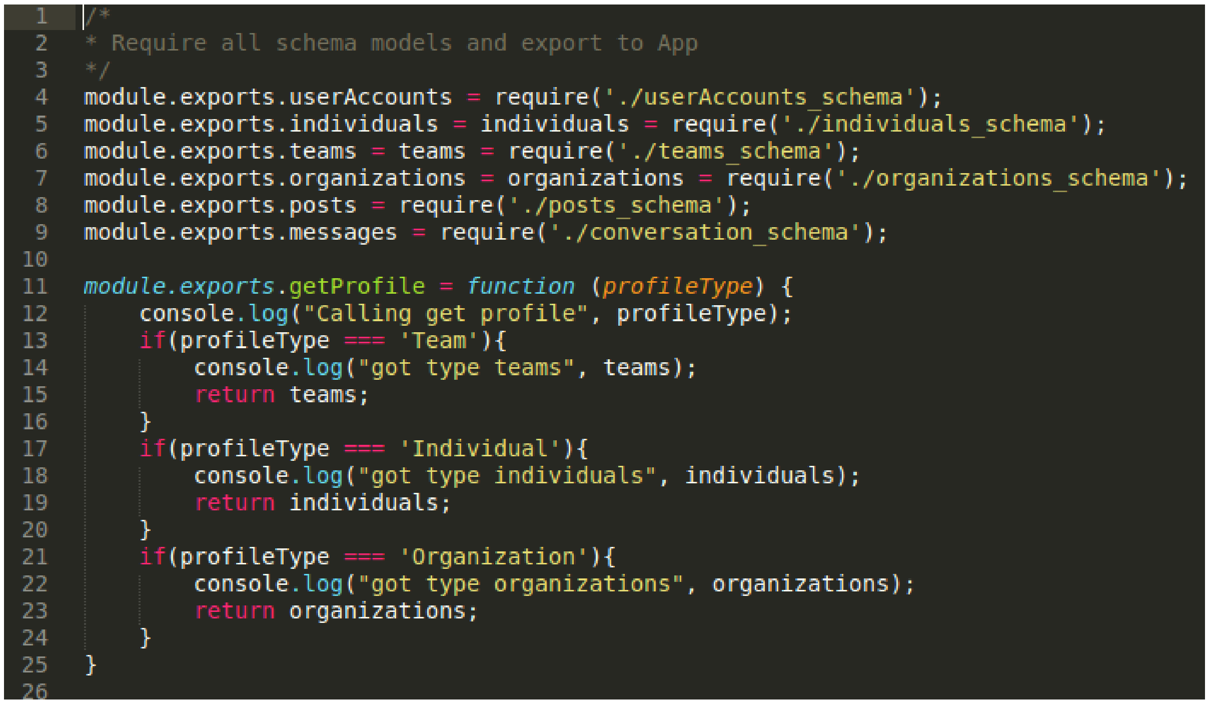
In this hypothetical example, the Person Schema has a reference to stories and is storing a unique id of a single document in the Story Schema. Using population, a query can retrieve the details of the person as well as the referenced story in a single operation. In my case this is impossible because I do not know which schema to refer to for population until runtime when the user signs up and chooses their profile type.

Due to this, I must store an additional variable holding the user's profile's schema type and a field holding the unique Id.



```
profile: {
    profileType: {type: String, required: true},
    profileId: {type: mongoose.Schema.Types.ObjectId, default: null, index: true}
},
```

Furthermore, to imitate population, whenever I am querying the database for user accounts, I need to also retrieve their profiles. To do this, I must know which collection to search. I created an index file for the Schemas with static methods to achieve this. The file looks like this:



```
/*
* Require all schema models and export to App
*/
module.exports.userAccounts = require('./userAccounts_schema');
module.exports.individuals = individuals = require('./individuals_schema');
module.exports.teams = teams = require('./teams_schema');
module.exports.organizations = organizations = require('./organizations_schema');
module.exports.posts = require('./posts_schema');
module.exports.messages = require('./conversation_schema');

module.exports.getProfile = function (profileType) {
    console.log("Calling get profile", profileType);
    if(profileType === 'Team'){
        console.log("got type teams", teams);
        return teams;
    }
    if(profileType === 'Individual'){
        console.log("got type individuals", individuals);
        return individuals;
    }
    if(profileType === 'Organization'){
        console.log("got type organizations", organizations);
        return organizations;
    }
}
```

This way I can reference directly to the desired collection through Mongoose models and can retrieve the unique profile faster. This is not the ideal solution though and is much slower than using population, thus I am still exploring the possibilities of filling the 'ref' field and referencing to a collection in runtime.

- Posting

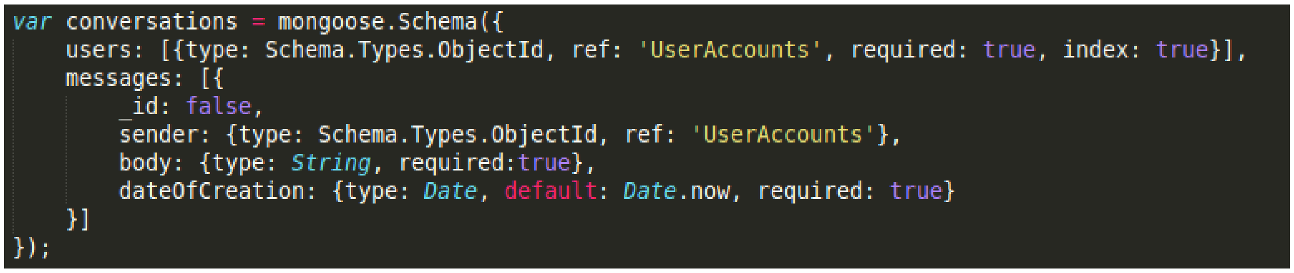
  Currently all users have the same posting parameters and their posts are public to all members of the website. The posts are being stored in their own schemas with userId, body, and date of creation. Since the user Id is indexed as well, Mongo keeps the field in assorted array makeing querying and retrieval very fast. Since I am storing the user's Id in my session, I can easily create a new post document with the UserId as the creator. When viewing a user's post page, I lookup all posts where the creator's Id matches the user's Id whose profile is currently being displayed. This means, the posts displayed are always those of the profile being displayed in profile area on the main-view. Since React is good at keeping state and securely, I always have the unique Id stored on the client side and I send it to the server whenever such operation is needed.

  The aim is to improve the posting capabilities by allowing user to add attachments or hashtags and other profile-type related setting; such as exposure.

- Messaging

  There are two important aspects for the messaging feature, most importantly, the inbox should be designed in a thread formate. That is, each two way communication between two users is saved as a conversation. Secondly, if two users are chatting simultaneously with each other, the data should be updated in real-time similar to a chat application.

  On the schema level, I saved each message as 'blog post' and messages -comments- are saved as an array. Furthermore, each comment has a body, sender, and date of sending, which allows for the presentation of the data in a conversation. Furthermore, each conversation has an array of users that are involved, this allows for group conversation and not just two way.



```
var conversations = mongoose.Schema({
    users: [{type: Schema.Types.ObjectId, ref: 'UserAccounts', required: true, index: true}],
    messages: [{
        _id: false,
        sender: {type: Schema.Types.ObjectId, ref: 'UserAccounts'},
        body: {type: String, required:true},
        dateOfCreation: {type: Date, default: Date.now, required: true}
    }]
});
```

With Mongoose's rich querying, I can search for all the documents of conversation which the user is part of, and then sort them in descending order of date of creation of the individual messages in the messages array. With this, I can retrieve only the latest 'n' number of conversations the user had, and display the latest message in each conversation.

Once a single conversation is selected, I retrieve the latest 'n' messages within that conversation and display them in descending date. At the same time, using socket.io I create a socket room, a group in which a socket can belong to and can broadcast to the members inside, with that conversation's unique database Id as room Id. When one user sends a message, the message is broadcasted to the room, if the other party happens to be viewing the same conversation, and thus registered in the same socket room, then the messages is captured by their client side socket and the message is updated [on their client-side] without the need to call the DB again to update their chat view. Of course, the data is being written to the database simultaneously. This way, I am able to accomplish a real-time chatting experience, without having to worry about peer to peer connection.

- Search

  Through the search, users are able to find other members of the website and view their profile, message them, or add them as a friend. Currently the search is limited to a basic regex expression that tries to match the query input to any part of the user account's name. In the future, the search will be modified with extra filters such as location, profile type, and others.

- Friends

  This tab would display the list of friends the user has in alphabetical order. In the upcoming updates, a search function should be added for the friends list to quickly browse through.

For both searching and friends tab, I wanted to display the results in a concise way; only showing the user's name, picture, and profile type. But, when clicked, the profile view on the left section should be updated to the complete profile of the chosen user, similarly the post tab should now display the new user's posts. To achieve this in an efficient way, I decided to sacrifice performance levels of these queries to assure that this data is available on demand.

What I am doing right now is obtaining the full data of each profile through my initial search queries, and sending that over to the front-end. That means, I only require one IO access to the DB instead of accessing the DB on demand each time a new user is chosen from the results. This increases the time of the initial query, but makes updating the profile views seamless. Also, thanks to the small size of JSON object, I can store the whole list of profile results in an array on the front-end without sacrificing performance capabilities.

POSSIBLE IMPROVEMENTS

A more complete UI.

For the UI, I have to implement some extra pages like info, and complete the design of the main page. For this I will need to spend a lot of time working with React to improve my skills and build a more fluid webpage.

Location-based database storage.

A full use of MongoDB's geolocation should be implemented to save the location of users to improve the searching function.

Improved search function.

Currently the search function is only searching for people according to name, thus there is a need to add extra filters to improve the search such as location and profession.

Improved database queries.

At the moment there is no implementation of paging due to the small amount of data tested. But it will be needed when the size grows as to keep the database operation time efficient.

This also means a newer, more efficient schema especially assigning profile references at run-time to improve query population and improve overall IO with the DB.

Testing.

I should conduct vigorous testing to the environment, and install debugging tools for better testing.

CONCLUSION

Node.js is a JavaScript platform for building network applications. It is built around the concept of using asynchronous operations and none-blocking IO. It is fundamentally different from competitors such as PHP, which uses a multi-thread paradigm to scale its operations, since it uses a single thread to handle all incoming requested to the server, and relies on its asynchronous model to manage concurrency. Another great opportunity Node.js provides is the ability to use JavaScript at both ends of the application; making resource usability between developers more applicable.

Overall the experience was very fulfilling and I managed to grasp the important parts of Node.js and full-stack JavaScript applications. I gained extensive knowledge in JavaScript and asynchronous programming with Node.js. I found that once adapted to the asynchronous paradigm the development with Node.js becomes very simple and the benefits of JavaScript over the full stack definitely improved and simplified the experience. Most importantly, I found myself capable of solving many of the big challenges using the different libraries available for Node.js and with the help of the community.

I also grew highly fond of NoSQL databases and ReactJS, both of which introduced new concepts and methodologies that work superbly with the JavaScript stack. Moreover, I appreciated the opportunity to try the different new libraries built for Node.js that tackles many aspects of the development. Nonetheless, I also understand now the reasons behind the slow adaptation of Node.js in its current premature state for intensive applications. While Node.js does IO very efficiently, it lacks the ability of heavy processing because of its single thread functionality which limits Node.js possible applications.

Full-stack JavaScript development tool set is a great way for developing basic http application that relies mostly on message sending and basic IO. It is also a great tool for teaching because it relies on a single language, has great resources, and many great tools, all of which are completely free and open-sourced. Furthermore, JavaScript is a great language that is considered less difficult than other alternatives such as Java or Scala.

In conclusion, I greatly recommend Node.js as a gateway to learning web development and getting engaged in the open-source community, and as a tool for rapid developing of http applications. In addition, the cloud has provided platforms such as Heroku and MongoLab for deploying and hosting servers and databases without any extra work on the programmer who can now focus more on the development. The amount of tools and passion the community has put into the node ecosystem and web applications in general makes development straightforward and abstracts the complex deployment difficulties.

BIBLIOGRAPHY

1. Crockford, Douglas. JavaScript: The World's Most Misunderstood Programming Language. http://www.crockford.com/javascript/javascript.html. Retrieved May 29th, 2015.

2. Init.js: A Guide to the Why and How of Full-Stack JavaScript. http://www.toptal.com/javascript/guide-to-full-stack-javascript-initjs. Retrieved May 29th, 2015.

3. An Introduction To Full-Stack JavaScript. http://www.smashingmagazine.com/2013/11/21/introduction-to-full-stack-javascript. Retrieved May 29th, 2015.

4. The MEAN Stack: MongoDB, ExpressJS, AngularJS and Node.js. http://blog.mongodb.org/post/49262866911/the-mean-stack-mongodb-expressjs-angularjs-and. Retrieved May 29th, 2015.

5. Learning React.js: Getting Started and Concepts .https://scotch.io/tutorials/learning-react-getting-started-and-concepts. Retrieved May 29th, 2015.

6. Understanding the Node.js event loop. https://nodesource.com/blog/understanding-the-nodejs-event-loop. Retrieved May 29th, 2015.

7. Comparing Node.js vs PHP performance. http://www.hostingadvice.com/blog/comparing-node-js-vs-php-performance/. Retrieved May 29th, 2015.

8. Understanding Express. http://evanhahn.com/understanding-express/. Retrieved May 29th, 2015.

9. Object Modeling in Node.js with Mongoose .https://devcenter.heroku.com/articles/nodejs-mongoose. Retrieved May 29th, 2015.

10. You're Doing Node.js Wrong! Avoid Synchronous Code. http://www.nodewiz.biz/your-doing-node-js-wrong-avoid-synchronous-code/. Retrieved May 29th, 2015.