

Designing and developing a web application for Meeting Automation



Project part- II (CSE 4292)

A Project Paper

**Submitted for the partial fulfillment of the requirements for the degree of B.sc
Engineering in Computer Science & Engineering**

Submitted By:

Md. Bepul Hossain (1710976146)

Md. Ebrahim Ali (1710376116)

Rakib Sheikh (1710576142)

Supervised By:

Md. Omar Faruqe

Assistant Professor, CSE, RU

**Department of Computer Science and Engineering
University of Rajshahi**

Acknowledgement

At first, I thank to almighty Allah for his kind blessing.

Then, I would like to express my gratitude and respect to my project supervisor Md. Omar Faruque, Assistant Professor, Department of Computer Science and Engineering, University of Rajshahi.

I am grateful for his inspiring encouragement, regular guidance, realistic appreciation and valuable suggestions throughout my project work.

Obliged thanks are also extended to all other teachers of Department of Computer Science and Engineering, University of Rajshahi, for there support and help during the project work.

Finally, I am thankful to my friends for their action co-operation during the whole project work.

Abstract

Meeting Automation is an online Web based application. This app can be used by any organization to call for official meetings. Using this app admin can create committee, send invitation email to a head of committee with head credentials. Head can login, add member, create meeting, send invitation to committee members with meeting details.

After meeting, Head can add resolutions which is not editable, send resolutions.

All meetings information are stored for future purpose, in future head can search with just a keyword of resolution.

This application is developed by Node.js, React, Bootstrap, HTML5, CSS and MongoDB.

Contents

1. Introduction	5
2. Background to the problem.....	6
3. Study and usage of MongoDB	6
3.1 The emergence of NoSQL.....	6
3.2 Basic information about MongoDB	8
3.3 Features of MongoDB	9
3.3.1 The MongoDB document model	9
3.3.2 MongoDB scaling and replication	11
3.3.3 Indexing	15
3.4 MongoDB performance	15
3.5 Usage of MongoDB in the service	17
3.5.1 Reasoning behind the use of MongoDB.....	17
4. Study and usage of Node.js.....	18
4.1 Basic information about Node.js	18
4.2 Features of Node.js	19
4.2.1 Compatibility with JavaScript.....	19
4.2.2 The Node.js event loop	20
4.2.3 Node Package Manager	24
4.3 Node.js performance.....	26
4.4 Usage of Node.js in the service	29
4.4.1 Reasoning behind the use of Node.js	29
5. Study and usage of Express.js	30
5.1 Basic information about Express.js.....	30
6. Study and usage of React.js	31
6.1 Basic information about React.js.....	31
6.2 Features of React.js	31
6.2.1 JSX	31
6.2.2 Virtual DOM.....	34
6.2.3 One-way data binding.....	35
6.3 React.js performance	36
6.4 Usage of React.js in the service.....	37
6.4.1 Reasoning behind the use of React.js.....	37
6.4.2 The front-end application.....	38
7. System Design	40

7. 1 UML Design	40
7.2 Types of UML Diagrams	40
8. System Implementation.....	43
9. Conclusion	51
10. References	51

1. Introduction

It is true that arranging a meeting is very painful. Every meeting has lots of resolutions and lots of agendas. If any time needs specific meeting resolutions or meeting agendas. Then members face difficulty to find them. To solve this problem, we can use this

online app called 'Meeting Automation'. In this application, there are three types of users (a) Admin (b) Head (c) general members. Every user has a specific rule, such as the admin can create a meeting and can select a Head of this meeting. Admin also can show all the meeting names that he created. The Head can select members for this meeting, create resolutions, and can send auto emails to every member.

Meeting automation can do wonders for any business in terms of organization. But there's more to it than meets the eye! In order to help us explore this exciting technology.

In order to make a website, we used MERN (MongoDB, Express.js framework, ReactJS library, NodeJS platform) is one of the powerful stacks that can help us to develop a meeting automation web application. Also, used Bootstrap SASS module for design purposes. Redux for state management.

2. Background to the problem

In the traditional way, when need to create a meeting, the admin selects a head. Then head select members and notify them manually. This is a very time-consuming process. When members need to know what is the resolution of a specific meeting, they can face difficulty. As well sometimes they do not find a hard copy.

3. Study and usage of MongoDB

3.1 The emergence of NoSQL

Ever since the inception of the SQL (Structured Query Language) database in 1980 by IBM based on the relational database model invented by Edgar Codd in 1973, relational databases, also known as Relational Database Management Systems (RDBMS) have

been the predominant method of storing digital data. Historically the most popular databases have been Microsoft SQL Server, Oracle Database, MySQL and IBM DB2. The relational model has proven itself a sturdy underpin to data storage through decades of active use. However, with the rapid growth of the amount of digital data in recent years, demands have increased both for the types of data to be stored as well as the frequency of operations required by database software.

The rigidity of the relational database model mainly stems from two of its core attributes – the necessity of a predefined schema in SQL form, and vertical scaling. In SQL data is represented as rows in an Excel-like tables where each unique attribute of the schema is represented by a column in the table. The attributes are limited to SQL's predetermined set of basic data types such as integer, string, and date, which causes issues when the data that requires storing is irregular or dynamic. Examples of data not covered by SQL's data types are arrays, nested items, and custom objects. Vertical scaling (scaling up) refers to the characterization of the growth in resources of single-node computer systems, where the resources of the system are increased through the addition of CPU power or memory to the single computer running the system. This type of scaling is made inconvenient by the scaling itself requiring downtime, as well as the potential difficulty in coming by heavier, more powerful and therefore also more expensive computers.

One solution that arose to solve this lack of flexibility is NoSQL, also known as “Not Only SQL”. The term was coined by Carlo Strozzi in 1998 for his RDBMS, Strozzi NOSQL. The NoSQL model offers more fluidity by not requiring a distinct schema, and by scaling horizontally. Horizontal scaling refers to the opposite of vertical scaling, where the resources of a system consisting of multiple separate hardware units connected to a single logical cluster are increased by adding additional nodes to the cluster. Because each node in the cluster is running a copy of the software, additional nodes can be added without an interruption to the system's operation.

NoSQL databases are commonly divided into four categories:

- Document databases
- Key-value databases

- Column databases
- Graph databases

All the different NoSQL database types have their own strengths and weaknesses, and none are a strict replacement of the SQL database. NoSQL databases tend to have faster performance for read and write operations by not requiring joins to relate data and by dropping much of the overhead found in SQL operations. However, this boost often requires the sacrifice of certain data integrity features offered by SQL. SQL databases invariably hold true to ACID (Atomicity, Consistency, Isolation, Durability) guarantees, whereas NoSQL tend to only offer BASE (Basically Available, Soft state, Eventually

consistent) guarantees. MongoDB, however, offers ACID guarantees on a single document level with multi-document ACIDity scheduled for summer 2018.

3.2 Basic information about MongoDB

MongoDB is an open-source NoSQL database under the document database category [1], created in 2007 in the U.S by Dwight Merriman, Eliot Horowitz and Kevin Ryan. Mongo operates under the GPL (GNU Public License) and is written in C++. As the most popular non-relational database at present by a significant margin, MongoDB functions as the de facto representative championing the non-relational logic. MongoDB was developed in a New York based organization named 10gen and was initially intended to be a PAAS (Platform As A Service), but the project morphed over time into an open source server. 10gen was later renamed into MongoDB Inc, and the first version was released in February 2009. Since its release MongoDB has attained remarkable popularity, especially taking into consideration its age compared to the longer standing database options.

3.3 Features of MongoDB

3.3.1 The MongoDB document model

In MongoDB data is stored in a binary JSON format, commonly referred to as BSON. One entry in the database is consists of a single JSON object called a document. The database contains multiple groupings of documents called collections.

JSON, or JavaScript Object Notation is a format commonly used with browsers, which makes MongoDB an excellent fit for storing data for web applications, since the same JSON data can be transported between the browser and the database without having to be converted into another format between them, as is the case with SQL. Although these conversions are made easier by ORM tools, some extra operations are nonetheless required from the software.

Listing 1.1 in Kyle Banker’s 2012 book “MongoDB in Action” displays the difference between how MongoDB and MySQL store the same example data of an entry on a social media site[2]. The SQL style of storing the entry divides the types of data that constitute the entry into 5 different schemas which are connected to the single entity through the unique identifiers (primary keys), whereas in MongoDB the same data is stored in a single JSON-object, also referred to as a “document” containing all the inner components in further nested objects as attributes of the main container object.

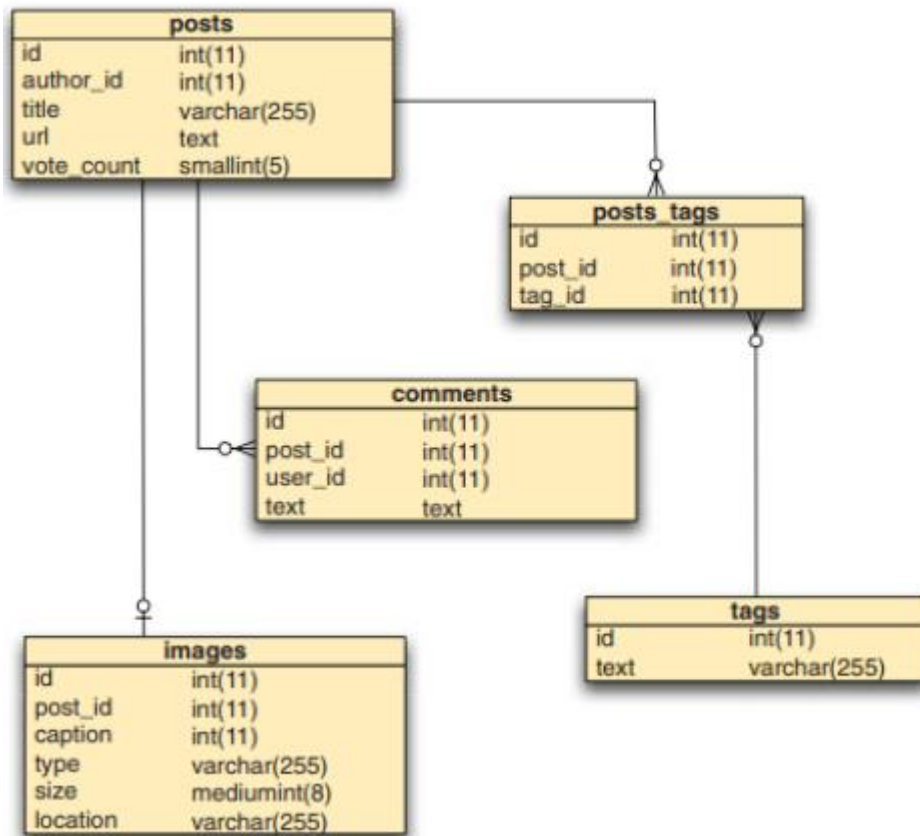


Figure 1. A likely MySQL representation of an entry on a social media site

```

{ _id: ObjectID('4bd9e8e17cefd644108961bb'),
  title: 'Adventures in Databases',
  url: 'http://example.com/databases.txt',
  author: 'msmith',
  vote_count: 20,
  tags: ['databases', 'mongodb', 'indexing'],
  image: {
    url: 'http://example.com/db.jpg',
    caption: '',
    type: 'jpg',
    size: 75381,
    data: "Binary"
  },
  comments: [
    { user: 'bjones',
      text: 'Interesting article!'
    },
    { user: 'blogger',
      text: 'Another related article is at http://example.com/db/db.txt'
    }
  ]
}

```

Annotations:

- 1 id field is primary key
- 2 Tags stored as array of strings
- 3 Attribute points to another document
- 4 Comments stored as array of comment objects

Figure 2. A likely MongoDB representation of an entry on a social media site

As can be observed in figures 1 and 2, the document model allows for the entire object to remain in a single data instance, which noticeably reduces complexity.

3.3.2 MongoDB scaling and replication

As is the case with many other NoSQL databases, MongoDB scales horizontally as opposed to vertically. In vertically scaling databases all the data resides on a single machine and improving the speed of the database involves upgrading the CPU and RAM resources of the machine. In horizontally scaling databases multiple hardware units running a copy of the software are connected to the same logical unit or cluster, where the responsibility for handling the inputs is divided among the nodes constituting it. Horizontal scaling is generally considered preferable, since adding new nodes doesn't require downtime as with vertical scaling, and because more demanding databases require stronger and heavier machines, which are more difficult to come by.

In MongoDB horizontal scaling is done via sharding, which is a mechanism where the initial database process is divided into additional processes that, depending on the sharding strategy, maintain either a subset of the data or a copy of the entire data set called a “replica set”. These shards are accessed through a query router called mongos which receives the input and connects the query to the shard containing the correct data, or, in a case where the shards are deployed as replica sets, to a random shard. The sharded cluster requires at least one config server that holds all metadata reflecting the state and organization for all data and components of the cluster.

A single replica set consists of 2-50 bearing nodes (mongod processes) where one node is the primary node and the rest are secondary nodes. Data can be read from any of the data-bearing nodes as the read operation requires no changes to the data. Write operations, however, are only received by the primary node and recorded to an operation log, which the secondary nodes keep themselves up to date with this log so that within milliseconds of the operation they reflect identical data with the primary node. It is also possible to attach an arbiter node which holds no replica of the data but allows for an uneven number of nodes to speed up the process of resolving a new primary node in a case where the current primary node becomes unavailable due to a crash, power outage or some other unexpected event.

Figures 3 and 4 from the official MongoDB documentation visualize the basic architecture of the MongoDB sharded database. Figure 3 displays a single replica set contained in a shard, and figure 4 the full setup of a sharded MongoDB database.

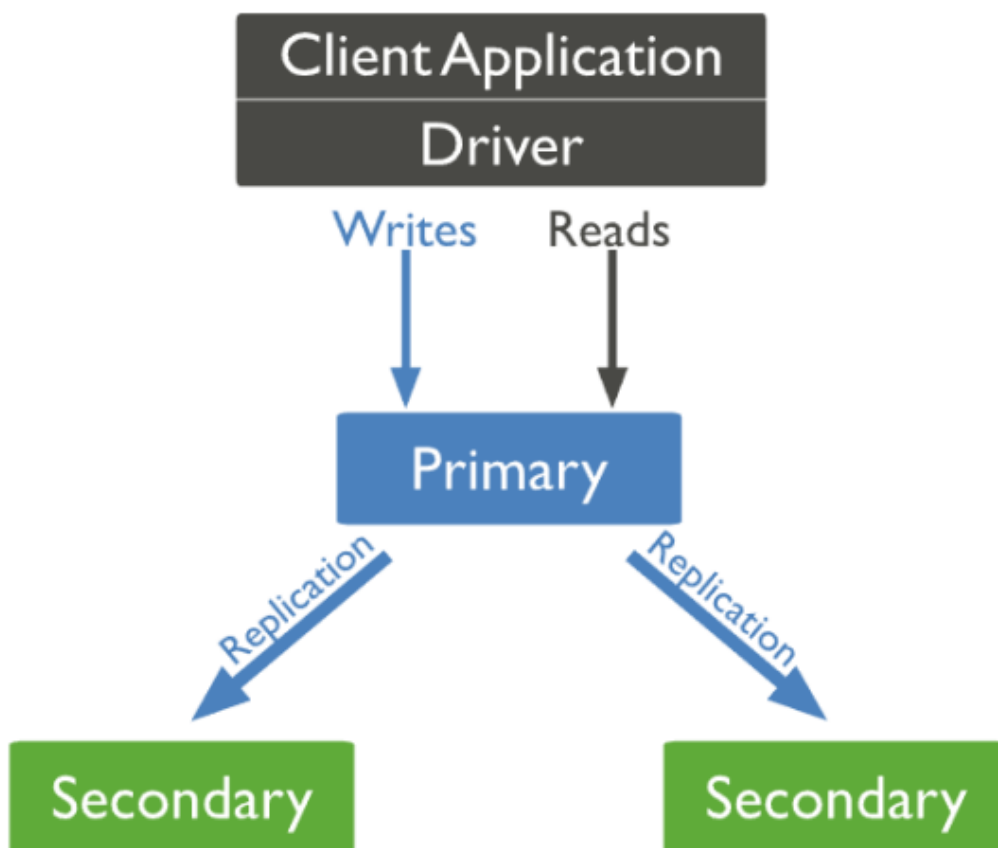


Figure 3. A graphical depiction of a single replica set with no arbiter node

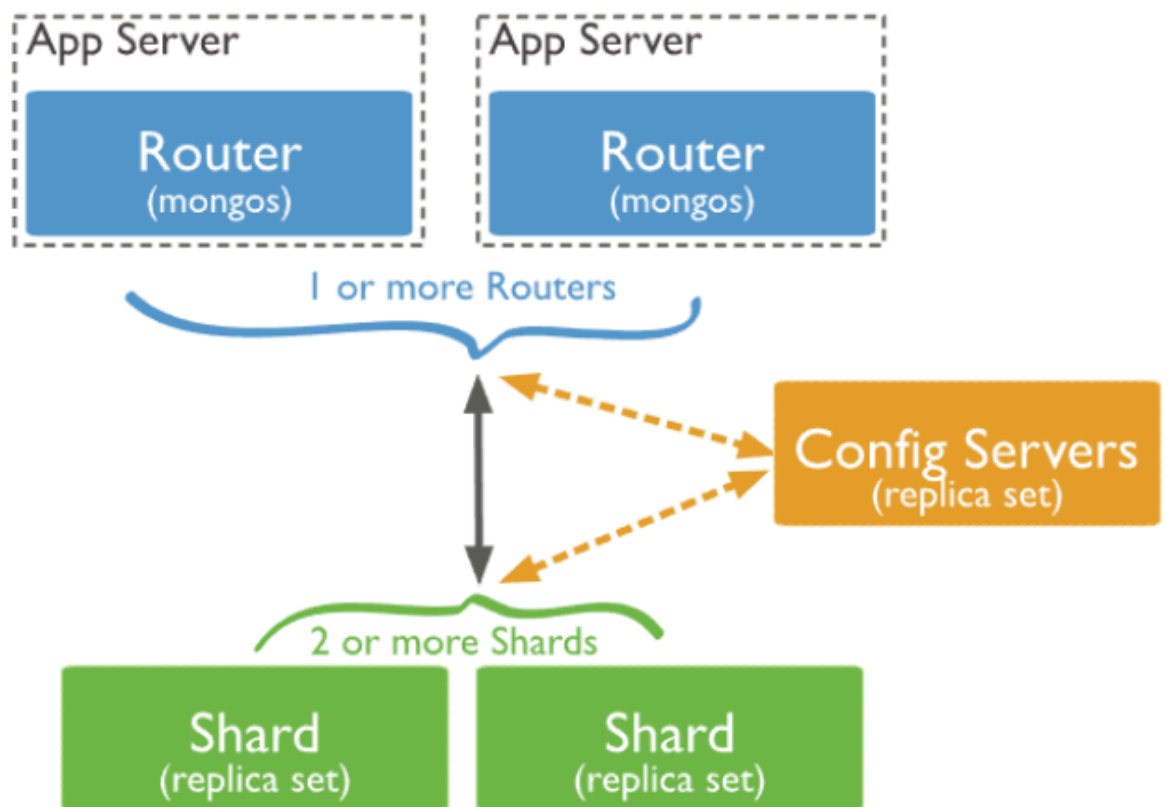


Figure 4. A graphical depiction of a sharded MongoDB database

In addition to easy horizontal scaling, sharding also provides a layer of security against unexpected crashes, as the loss of a single database server doesn't affect the functionality of the others. Sharding can also be useful in scenarios where the same data can be hosted in different data centers around the world, speeding up operations locally and enabling geo-specific shards where some or all the data contained in the shard is only relevant to the location of the data center containing it.

In MongoDB sharding requires no additional setup on the application level, as the internal management of the data and its updates after the initial configuration is done internally and interaction with the entire sharded cluster happens the same way as interaction with a single node.

3.3.3 Indexing

MongoDB supports a variety of indexing strategies to help speed up queries, namely single field, compound, multikey, geospatial and hashed indexes[3]. These indexes are defined at the collection level, and all collections have a default un-droppable index for the “_id” -field. In MongoDB all indexes are stored in B-trees (not to be mistaken for binary tree), which is a tree data structure optimized for reading and writing large blocks of data.

In a standard single field index, the documents are sorted in either ascending or descending order based on the field in question as they are in SQL databases. The other index types, however are rarely supported by traditional SQL databases. In compound indexes multiple fields form a single index entry, which means that instead of having to traverse a separate index tree per each field to find the disk location of the document and then calculate the answer, the fields are combined, which in most cases limits the queried area. Multikey indexes are used on fields containing an array, creating a separate index entry for each of the values in the array, allowing queries based on the nested array value without having to break the document. Geospatial indexes are created either as 2d or 2dsphere or geoHaystack indexes each of which are optimized for the type of geometry that is used by the query. Text indexes are optimized for finding string values in the collection, supporting some useful tools like language-based stemming and recognition of stop words. Currently MongoDB text indexes do not support fuzzy text search. Hashed indexes are offered for the purpose of hashed based sharding to help partition fields across the sharded cluster.

3.4 MongoDB performance

For the purposes of performance evaluation MongoDB is compared with MySQL, which is the respective open-source leader sporting the relational logic. Between the two either one can be favored in terms of performance depending on the scenario, but for the most common operations MongoDB tends to have a small advantage in performance which can add up to a significant boost for larger applications.

The differences in performance stem purely from the design features of the databases. The reason for this is that both MongoDB and MySQL are under the GNU General Public License, meaning that if one produced an update that made raw I/O code faster, the other would be able to copy the same logic into their software. In the scenario illustrated in figures 1 and 2, fetching all the data associated with the social media post using the relational model would require 4 SQL JOIN-operations between the different tables. In MongoDB since the data is contained in a single object, accessing it only requires a single lookup action from the computer. This phenomenon along with the reduced overhead of the operations themselves causes MongoDB to have better performance in most scenarios.

In chapter 4 of a dissertation conducted in the University of Edinbrough by Christoforos Hadjigeorgiou (MSc) titled “RDBMS vs NoSQL: Performance and Scaling comparison” benchmark tests are conducted to demonstrate the speeds of different operations between MySQL and MongoDB[4].

Tables 1 and 2 graph results of these benchmark tests concerning the speeds of some of the most common basic operations required of databases, namely INSERT, DELETE and SELECT.

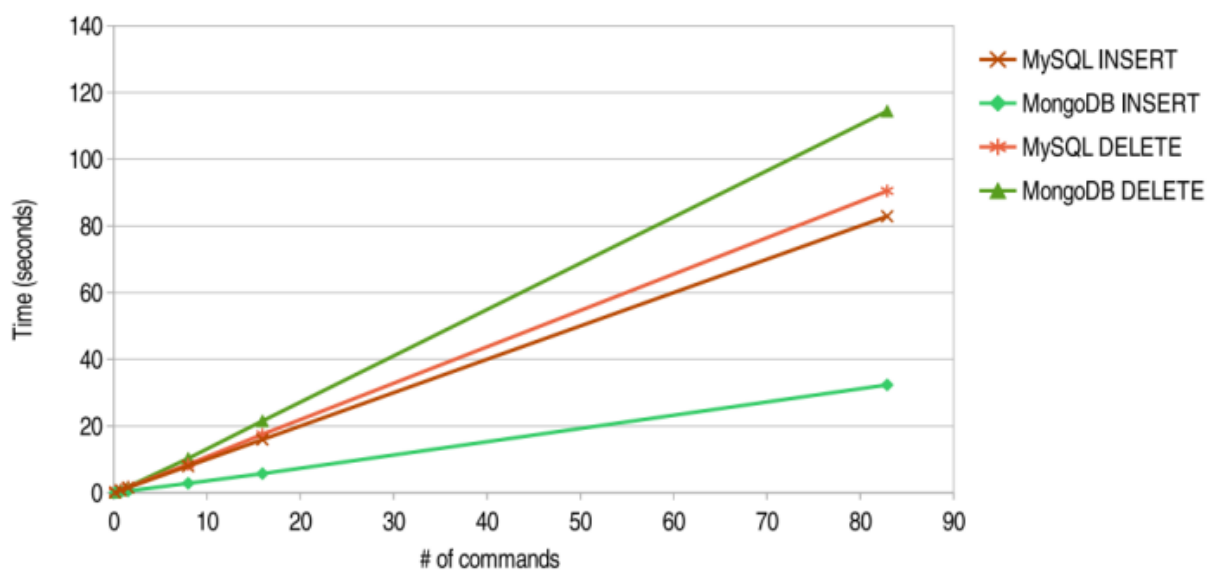


Table1: Graph depicting the speeds of MySQL and MongoDB performing INSERT and DELETE operations. Lower time is better.

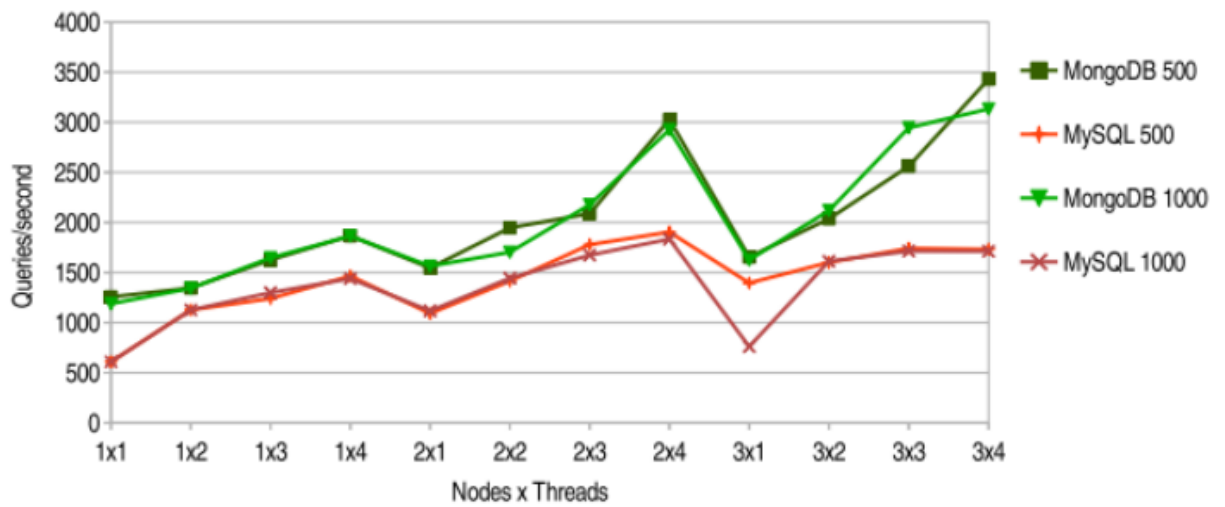


Table 2: Graph depicting the speeds of MySQL and MongoDB performing a SELECT INNER JOIN query, the number next to the name on the right side of the graph is the number of times the query was repeated. Higher queries per second is better.

As can be seen based on tables 1 and 2, MongoDB tends to have a slight advantage in performance over MySQL in these scenarios. It is however important to note that these benchmark tests only measure the performance speed within the confines of the database itself. When considering full scale web applications, additional time might be saved by not having to convert the database's output into another format for the browser.

3.5 Usage of MongoDB in the service

3.5.1 Reasoning behind the use of MongoDB

MongoDB is built on a scale-out architecture that has become popular with developers of all kinds for developing scalable applications with evolving data schemas.

As a document database, MongoDB makes it easy for developers to store structured or unstructured data. It uses a JSON-like format to store documents. This format directly maps to native objects in most modern programming languages, making it a natural choice for developers, as they don't need to think about normalizing data. MongoDB

can also handle high volume and can scale both vertically or horizontally to accommodate large data loads.

MongoDB was built for people building internet and business applications who need to evolve quickly and scale elegantly. Companies and development teams of all sizes use MongoDB for a wide variety of reasons.

- Document Model
- Deployment Options
- Get Started Quickly
- Fully Scalable
- Find Community

4. Study and usage of Node.js

4.1 Basic information about Node.js

Node.js is an open-source platform that utilizes Google Chrome's JavaScript runtime V8 Engine [5]. Node.js can be characterized as being to JavaScript what the JRE is to java. Node.js compiles and executes JavaScript code inside of a VM (Virtual Machine) and thereby enables JavaScript code to be run on the server side. This makes JavaScript a viable alternative to other server-side languages like PHP, ASP or Java, and most importantly enables the use of JavaScript every layer of the web stack – server, transportation format (JSON), and the client.

Node.js comes with a NGINX-like base library of modules related to the most common requirements such as file management, HTTP, SSL, DNS, compression and cryptography. The basic modules are aimed at making the Node.js process into a functional web server in only a few lines of code, but Node.js is also highly customizable through the Node Package Manager, also known as “npm”, which enables Node.js users to install packages from among over 600 000 (and growing) open-source blocks of code to suit their purposes.

Node.js has experienced steady yearly growth since 2009. As of 2017, Node.js has over 8 million active users and 257 million total downloads, enjoying a whopping 100% yearly growth rate. and is used in nearly 0,4% of all websites compared to 0,3% last year.

Node.js was created in 2009 by Ryan Dahl, written in C/C++ and created, according to Dahl, to “provide a purely evented, non-blocking infrastructure to script highly concurrent programs”. Node.js runs on a single thread and was partly inspired by Event Machine and Twisted, which are both earlier event-driven frameworks that use a single event loop to enable high concurrency in applications.

Due to its strong performance under high concurrency both in terms of speed and resource usage, Node.js has been adopted by many large companies and organizations such as Netflix, LinkedIn, Walmart, Trello, Uber, PayPal, Medium, eBay and NASA.

4.2 Features of Node.js

4.2.1 Compatibility with JavaScript

JavaScript, a language specifically designed to be asynchronous, was an obvious fit for this event-loop based architecture. The nature of JavaScript itself is event loop based, using callback functions for nearly all non-trivially time-consuming tasks. Take for example a JavaScript classic, the XMLHttpRequest request. Although XMLHttpRequest has been made obsolete by ES6 Fetch, it serves better to demonstrate the culture in which JavaScript operates.

```

var xhttp= new XMLHttpRequest();
xhttp.onreadystatechange = function(){
    if (this.readyState == 4 && this.status == 200){
        //post-request-action
    }
};
xhttp.open("GET", "filename", true);
xhttp.send();

```

Figure 5. A typical way of performing an HTTP request in pre-ES6 JavaScript

As can be observed from Figure 6, the HTTP request is fired away, and the future result of the request is tied to the on ready state change function. After firing the request, the engine has already proceeded to the next task on the list, and only returns to the callback once the response to the HTTP request has returned. In this way, a thousand requests could be sent out in an instant and would be processed in the order of their return to the event loop using the callback system.

Node.js allows this same logic to be applied to server-side architecture. It doesn't allow anything to lock up the I/O, but rather forces every time-consuming operation to pass through the event loop and to be tied into callback events.

4.2.2 The Node.js event loop

The event loop is what allows Node.js to take advantage of multi-threaded system kernels while still maintaining non-blocking I/O. What is meant by I/O in the context of Node.js is usually accessing external resources like disks or network resources, which are the most expensive due to the time they take to complete.

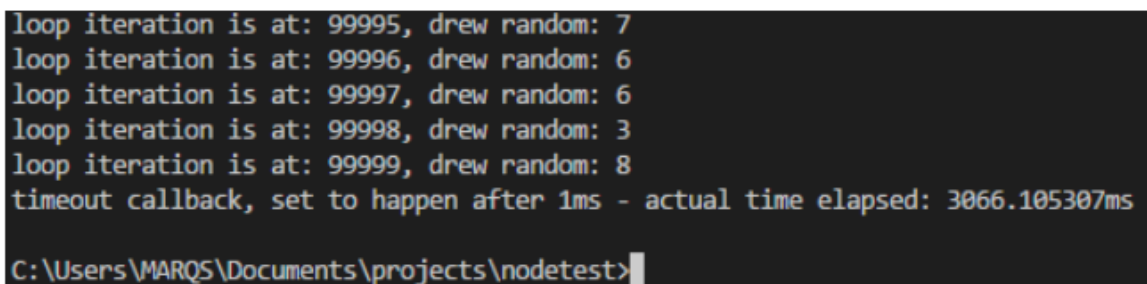
The Node.js process is a loop that performs polling and blocking calls to the system kernel on a constant basis while active. The operating system kernel in turn notifies Node.js when an operation is complete, after which its associated callback function is added to the poll queue for eventual execution. Node.js exits when it runs out of events

to process, but in the context of a web server where listening for new requests is in and of itself a series of events, the process can be conceptualized essentially as a closed while-loop, repeating its internal phases continually.

It's important to note that regular operations which are necessarily synchronous by nature and therefore cannot make use of a callback (such as variable declarations or simple calculations) are completed on the spot and in synchronous order upon script start. Take for example the following code snippet and its output of a node.js script that first sets a timeout at 1 millisecond and then draws 100000 random integers between 0 and 10:

```
const NS_PER_SEC = 1e9;
let startTime = process.hrtime();
setTimeout(function() {
    const diff = process.hrtime(startTime);
    let nanoseconds = diff[0] * NS_PER_SEC + diff[1];
    console.log("timeout callback, set to happen after 1ms "
        + "- actual time elapsed: " + (nanoseconds / 1000000) + "ms");
}, 1);
for (let i = 0; i < 100000; i++) {
    let r = Math.round(Math.random() * 10);
    console.log("loop iteration is at: " + i + ", drew random: " + r);
}
```

Figure 6. The full script of the example node.js script running the synchronous operations



```
loop iteration is at: 99995, drew random: 7
loop iteration is at: 99996, drew random: 6
loop iteration is at: 99997, drew random: 6
loop iteration is at: 99998, drew random: 3
loop iteration is at: 99999, drew random: 8
timeout callback, set to happen after 1ms - actual time elapsed: 3066.105307ms
C:\Users\MARQS\Documents\projects\nodetest>
```

Figure 7. The end of the console output of the script in figure 8

As can be seen from the output of the script in figure 9, the timeout is in fact set first and its callback is scheduled to happen after 1ms, but because the script encounters a loop where all the contents are necessarily synchronous operations (`Math.round`, `Math.random` and `console.log`), the event loop blocks until everything is done before proceeding to the callback queue. This example, however, is very contrived and not descriptive of any real scenario – the real benefit of the loop-based architecture is made clear when running programs that are heavy on operations that make use of callbacks, such as a web server

The Node.js event loop is internally implemented by a C library called libuv (short for Unicorn Velociraptor Library), which is a multi-platform open-source support library for base level asynchronous I/O management. Libuv is primarily designed for its use in Node.js and can take advantage of the native polling queue mechanisms of each operating system to achieve high performance levels and effective offloading of I/O operations to the system kernel whenever possible.

Node.js leverages the v8 Engine to compile the user script into native code to exercise the libuv loop, which in turn maintains an internal worker thread pool for execution of operations which are necessarily blocking (such as File System, DNS and crypto). According to one of the key figures behind libuv, Bert Belder (IBM), a common misconception is that the userland JavaScript code and the libuv event loop are run on separate threads, but in fact the libuv loop (including its internal thread pool) and the user-defined JavaScript program are incorporated under the one master thread that is active whenever code is run by Node.js.

Figure 10 below from a Node.js foundation article displays the cycle of operations that the event loop repeats continuously while active. If the loop has no events to complete, it starts blocking at the poll phase until new events show up.



Figure 8. The order of phases in a single cycle of the Node.js event loop

Each of the phases in the cycle pictured in figure 11 has its own callback queue specific to that phase, which is processed in order of entry (first in, first out). When the event loop enters a phase, it tries to exhaust its queue of stacked callbacks and moves to the next phase.

The first phase of the Node.js event loop called “timers” executes callbacks set by JavaScript’s `setTimeout()` and `setInterval()` functions. A callback is tied in to the timer and the related operation is run in the timer phase as soon as possible after the scheduled time has passed. The estimated time is however only a threshold and the real point of execution can be delayed by system operations or callback executions in other phases.

The pending callbacks phase executes I/O callbacks of errors and certain system operations that were deferred from previous operations to the next loop iteration, such as TCP socket connection error `ECONNREFUSED`. Pending callbacks is however only a backup execution phase for certain exceptions - normal I/O callbacks are executed in

the poll phase. Idle/prepare phase is only used for libuv's internal operations and is therefore not covered.

The poll phase is the primary phase in the event loop. The poll phase first calculates how long it should block for, retrieves new I/O events into the loop for the allocated time and then processes the stacked events in its queue. Almost all callbacks are executed in the poll phase, with the exception of timers, `setImmediate()` callbacks and closing callbacks. The poll phase will attempt to execute the queued-up callbacks in synchronous order until they either run out or until the system-dependent hard time limit is reached and the execution is forced to be cut short. If there are no callbacks queued up when entering the poll phase, the loop will either wait in the poll phase for new ones to be added to the queue and begin execution immediately, or, if there are callbacks scheduled by `setImmediate()`, continue to the check phase right away.

The check phase is dedicated to executing `setImmediate()` operations. Commands set with the `setImmediate()` function are guaranteed to execute as the first thing when the current polling phase ends, that is to say, after its current callback queue has been exhausted.

Close callbacks phase is a cleanup phase, used for shutting down sockets and emptying expired data. Close callbacks are triggered for example when sockets or handles are closed or destroyed abruptly. This phase is necessary to not have unnecessary elements floating around and eating up resources.

4.2.3 Node Package Manager

The Node Package Manager or NPM is currently the world's largest software registry,[6] boasting over 600 000 packages of code and 3 billion downloads per week. NPM consists of the website used for discovering new packages, the actual software registry as well as the CLI used to perform NPM-related commands. The packages themselves are simply open-source blocks of code created for specific purposes in JavaScript-based projects. Because NPM is open source, there are often multiple potential packages created for the same purpose. NPM sorts packages by popularity,

ensuring that over time the more reliable and bug-free packages are filtered to the top. The NPM website also provides several metrics by which users can judge the quality and reliability of the package, such as download statistics, open issues, versions, dependencies (other packages required by the package in question), dependents (other packages that require the package in question), collaborators, as well a creator-provided introduction to the package potentially with a small demo example and a link to the actual source code of the package.

NPM comes with Node.js upon installation and is accessed from the command line. Installing packages can be done globally or locally through the `npm install` command. The code block itself is then fetched from the NPM registry via HTTP and saved into a `node_modules` folder, which is created if it doesn't exist. The name of the package on the website always corresponds to its name in the registry, making it directly accessible using the install command "`npm install`".

The management of npm packages is practically always done through a file called `package.json`, which Node.js checks for during NPM-related operations. `Package.json` lists the names and versions of the packages that the project requires to run along with other build-related data and commands in the form of a single JSON object. NPM's integration with `package.json` is comparable to Maven or Gradle in the Java world, enabling the dependency information to be moved as pointers to an external repository rather than having to move the libraries themselves as a part of the project. The simplicity of the single-file solution is one reason for the ease with which Node.js-based projects can be developed and deployed. NPM also warns about outdated packages and alerts users about new NPM versions as well as newly discovered security risks in the packages installed by the user and offers direct and automatic repairs for the packages that have been found to contain potential risks.

After packages have been installed via NPM as modules, they can easily be imported into the user's code using Node's `require()` function, which concatenates parts from multiple different JavaScript files into one during compile time.

4.3 Node.js performance

The performance of Node.js in comparison to its alternatives has much to do with the types of operations performed during the benchmark tests [7]. Although Node.js is newer than many of its alternatives, it is not an across-the-board superior replacement for other systems, but rather one solution that can be faster or slower depending on the nature of the program. The strong suit of Node.js tends to be highly concurrent applications with frequent database visits, which is why it has been adopted by many companies with significantly large user bases, some of which were mentioned in section 4.1.

For the purposes of the performance analysis, Benchmark tests were selected where Node.js is compared with Java, which has been overall the most popular programming language for over 2 decades now. One benchmark test was conducted by PayPal before their decision to transition from Java to Node.js for their service in 2013. PayPal had two teams build the same functionality, one with a Java-based build using Spring and one with a Node.js-based build using Express.js. The results of the tests are displayed below in table 3.

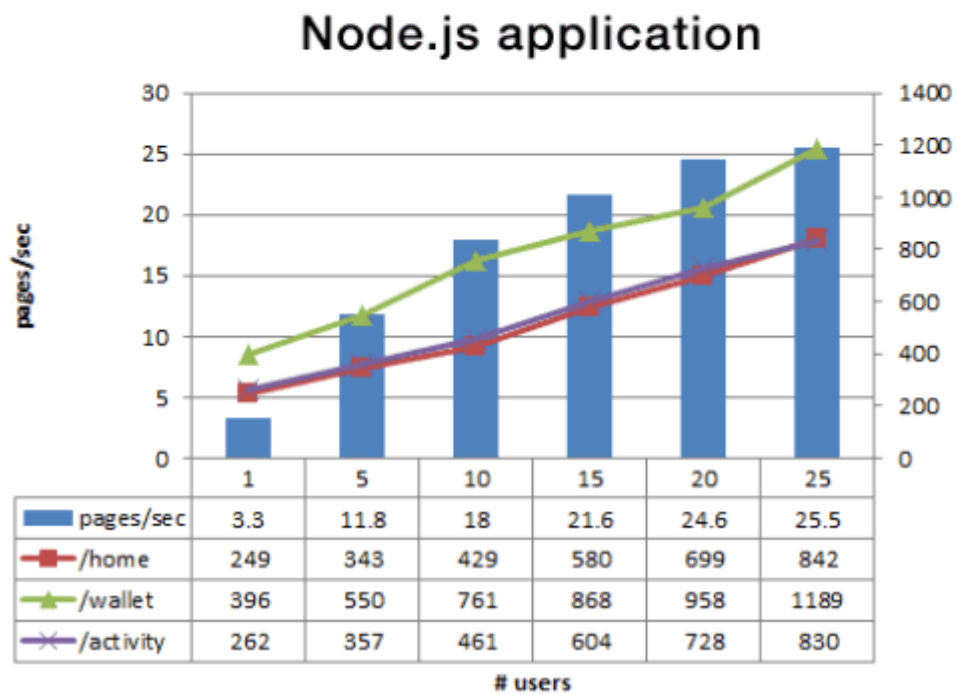
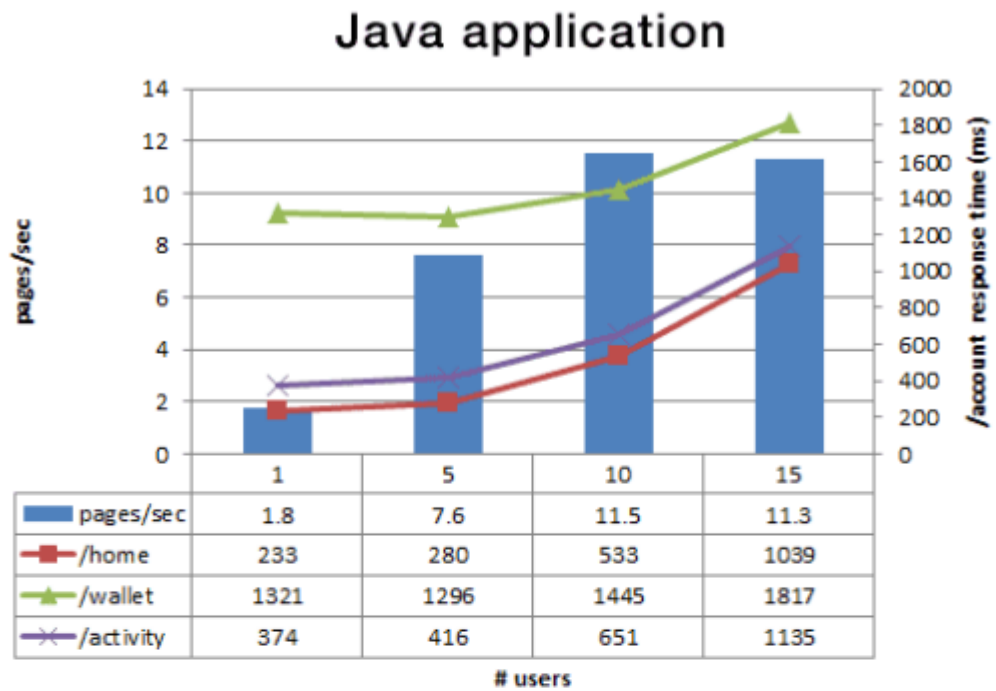


Table 3. Graphs depicting the PayPal Node.js vs Java benchmark test results

As per the tests in Table 3, Node.js consistently provided lower response times, with the gap getting progressively larger as the concurrency grew. According to PayPal, the

Node.js solution resulted in a 35% decrease in average response time, and an average of 200ms reduction in page serving time.

It is important to note that these sorts of clear-cut and one-sided results are only possible when starting with high levels of concurrency. In another benchmark test conducted by Ferdinand Mütsch in 2017 with only 32 concurrent clients demonstrated that Java has a small but noticeable edge over Node.js when dealing in this range (which is much more realistic for small to medium-scale applications). In this benchmark test, Java was employed in a web server using Jersey with embedded Grizzly, and Node.js with both a plain http package and standard Express 4 (which will be covered later in this thesis). The content of the test included performing 100,000 API requests to the server. The results of the tests are displayed below.

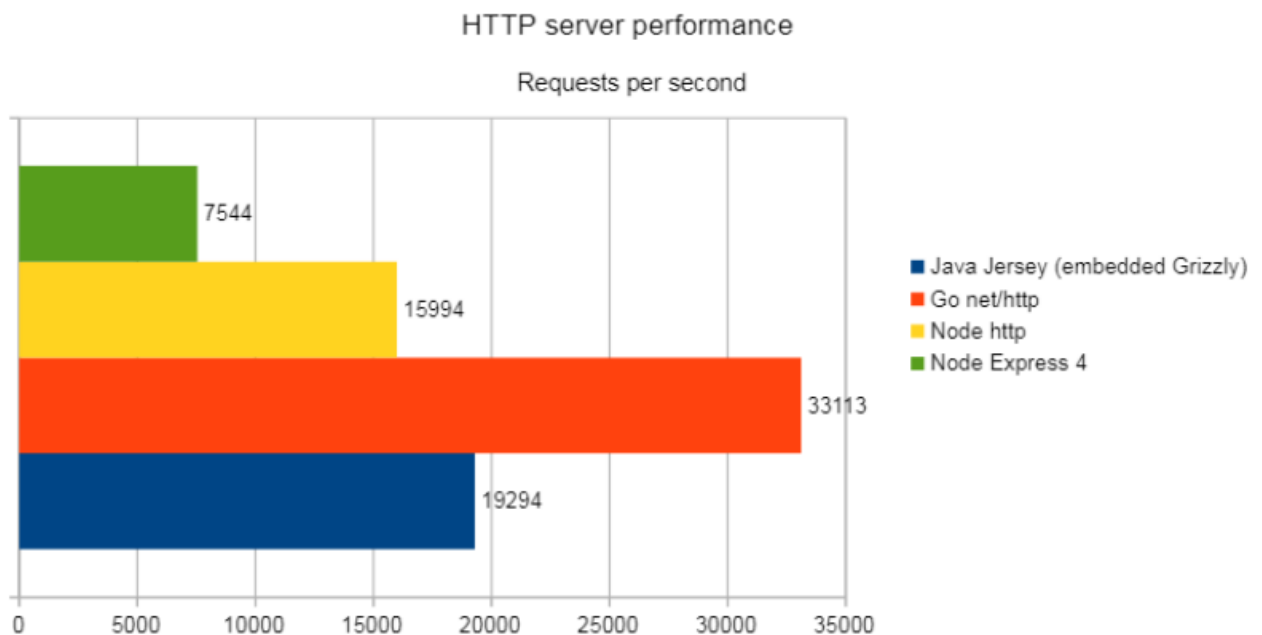


Table 4. Results of the lower concurrency web server benchmark test

As can be seen from the test, the Java-based server does perform better in lower concurrency situations. However, because benchmark tests rarely test for a highly specific situation and in a real-life scenario the differences might be boosted one way or the other through use of frameworks and different types of authentication neither can be said to be strictly faster than the other outside of the general trend of high

concurrency favoring Node.js. Some reasons for Java's faster base level performance are the superior performance of the Java Virtual Machine compared to the V8 runtime engine, which in turn has to do with internal design mechanics and the fact that Java is statically typed and precompiled as opposed to JavaScript which is dynamically typed and continuously dynamically recompiled during runtime by the V8 engine for optimization purposes which requires extra heuristics that slow down the overall processing.

4.4 Usage of Node.js in the service

4.4.1 Reasoning behind the use of Node.js

The obvious first alternative for Node.js in the open source, world would have been Java (which also allows for the use of the most recent MongoDB drivers) in combination with a web server like Grizzly, Jetty or Apache. The application in question would not expectedly have thousands of concurrent users, which would make a Java-based solution more suitable in terms of performance. One important thing to consider, however, is the better compatibility and integration of Node.js with MongoDB, which is observed both in the ability to handle JSON data from MongoDB directly without conversions as well as the nature of the MongoDB connector, which receives commands in JavaScript. This in combination with the easier set-up and development due to the project structure and lower compile time of Node.js in comparison to Java makes it a more appropriate technology to use on the server side in this instance. In addition, the use of Node.js allows for the overall project to be written entirely in one language (JavaScript), adding an extra degree of unity and congruency to it.

5. Study and usage of Express.js

5.1 Basic information about Express.js

Express.js is an unopinionated open source, web server framework written purely in JavaScript, designed specifically to be used with Node.js[8]. Due to its reliability and widespread use Express has become the de facto standard server software for Node.js. Express in relation to Node.js can be considered analogous to what Ruby on Rails is to Ruby. Express is optimized for performance and is minimal by nature, placing only a thin layer on top of Node.js base level web features. Due to its lightweight design and standardized status, Express acts as the foundation for several other server-side JavaScript frameworks that incorporate it such as Feathers, KeystoneJS, Kraken and Sails, which are more tailored to specific types of applications as opposed to Express, which only provides a robust implementation of more general server-side functionalities such as routing, HTTP caching and view templating.

Express.js was originally created and initially released in 2010 by TJ Holowaychuk, an open-source mass-contributor and guru. In June 2014, the ownership of the GitHub repository for Express.js was sold by Holowaychuk to StrongLoop, a Node.js-based startup company, which in turn was acquired by IBM in 2015. In 2016 the Express.js project was placed by IBM under the Node.js Foundation Incubator Program, which is a way for the Node.js Foundation to affect the future governance and development decisions of the framework in order to maintain its relevance and competitiveness.

In terms of popularity within the Node.js ecosystem Express is dominant. In addition to being the top Node.js related repository on GitHub, Express has been downloaded over 420 million times since its inclusion in the NPM registry in 2014 and is currently enjoying an average of 20 million downloads per month. Express is also used in production by several large companies such as Accenture, IBM, Fox Broadcasting and Uber.

6. Study and usage of React.js

6.1 Basic information about React.js

React.js is a JavaScript library maintained by Facebook designed for building user interfaces for web services[9]. React was originally created by a Facebook employee called Jordan Walke and was first used inside Facebook in 2011 when it was used in the design of the Facebook app's newsfeed . According to Walke, the framework was inspired by and started out as a JavaScript port of XHP, an augmentation of PHP also developed at Facebook aimed at creating reusable HTML components for browser applications. XHP, however, still suffered from being forced into many roundtrips to the server during application use in regular PHP fashion. React.js emerged as a solution to the problem, delivering the application components in an initial JavaScript bundle and then efficiently managing renders to the DOM, allowing for easily reusable and customizable HTML views.

In 2018 React is still the most popular front-end JavaScript framework, having held the title for several years. This popularity was achieved by React being found sturdy through widespread use. While front-end JavaScript frameworks are notorious for their volatility and sudden expiration, React has reached a level of reliability over time that hasn't yet been matched by other frameworks.

6.2 Features of React.js

6.2.1 JSX

JSX or JavaScript XML is an HTML-like extension to the JavaScript language syntax [10]. Although React doesn't require the use of JSX, it is a default companion to React as it provides clean separation between component and non-component code and makes use of developers' already existing HTML knowledge. When compiled, the JSX components are converted to regular JavaScript via the Babel.js transpiler.

Figures 18 and 19 below from React's documentation display the same component expressed first in JSX and then in normal React JavaScript.

```
class Hello extends React.Component {  
  render() {  
    return <div>Hello</div>;  
  }  
}
```

```
ReactDOM.render(<Hello name="Ebrahim" />,  
document.getElementById("root"));
```

Figure 16. A react component written in JSX

```
class Hello extends React.Component {  
  render() {  
    return React.createElement("div", null, `Hello  
${this.props.toWhat}`);  
  }  
}
```

```
ReactDOM.render(  
  React.createElement(Hello, { toWhat: "World" },  
null),  
  document.getElementById("root")  
);
```

Figure 9. The react component from Listing 10 written without JSX

In this way JSX is simply a more HTML-like way of using React's functions. Although JSX on a surface level appears very similar to regular HTML, it still compiles to JavaScript and therefore allows the benefits of script code such as loops and other

calculations, making the components even more compact and dynamic. The parts where regular JavaScript is used in JSX are demarcated by curly brackets. Take for example a scenario where the desired HTML view is a list with 10 child elements. In a traditional web application the list would be written in pure HTML as per Figure 20.

```
<ul>
  <li>Child 1</li>
  <li>Child 2</li>
  <li>Child 3</li>
  <li>Child 4</li>
  <li>Child 5</li>
  <li>Child 6</li>
  <li>Child 7</li>
  <li>Child 8</li>
</ul>
```

Figure 10. A traditional HTML list with 6 child elements

If the list were to get too long, another somewhat newer way would be generating the elements via JQuery or vanilla JavaScript from another file or from a `<script>` tag and injecting it directly to the DOM at some point. In JSX, the HTML and JavaScript syntaxes are combined into a compact block in a single file. The HTML list from listing 12 could be expressed in JSX as per Figure 21 below.

```
render() {
  let arr = [];
  for (let i = 1; i <= 6; i++) {
    arr.push(i);
  }
  return (<ul>
    {arr.map(n => <li>Child {n}</li>)}
  </ul>);
}
```

Figure 11. The HTML list from listing 12 expressed in JSX form inside a React component's render method

In the example above the child elements are generated inside the HTML-like list element itself. The JSX syntax in this case appears more difficult, but when complex components are repeated or when component-specific calculation, customization or conditionality is required dynamically, the JSX syntax becomes very helpful.

6.2.2 Virtual DOM

The way React.js achieves quick and efficient DOM manipulation is by setting up an inmemory tree model representation of the real DOM[11]. The virtual DOM can be conceptualized as a blueprint for the actual HTML content on the page. When changes are made to the virtual DOM, React runs a quick diffing algorithm to find out which parts of the blueprint have been changed and handles the actual HTML changes to the real DOM. The diffing algorithm is optimized to do the least amount of DOM manipulation required to keep the React components up to date.

When using React, a new virtual DOM subtree is constructed out of each component that calls `render()`. Transforming one of these tree structures into another with brute force has at a bare minimum an $O(n^3)$ order of complexity, but according to React.js documentation the internal diffing algorithm reaches a decrease in complexity down to $O(n)$ based on two assumptions which are valid for almost all practical use cases:

1. Two elements of different types will produce different trees
2. The developer can hint at which child elements may be stable across different renders with a key prop

The performance of the React diffing algorithm is investigated more closely in the performance section.

6.2.3 One-way data binding

One-way data binding is a feature of React design where the data flows unidirectionally from higher to lower level components. In two-way data binding changes to the view cause changes to the component's data and vice versa. For example, an input field might be tied to the component data in such a way that when a key is pressed, the data is mutated to reflect the changes. Respectively you might have the input field value be changed without a key press to match changes set from elsewhere. Examples of popular front-end JavaScript frameworks that use two-way data binding are Vue.js and Angular.js. Neither design is strictly better or necessary, but one-way data binding is a good way of ensuring that the data flow architecture remains clear and the most up to date application data is only available from a single source of truth. Figure 22 below by Sviatoslav A in his article for RubyGarage “The Angular 2 vs React Contest Only Livens Up” demonstrates this difference between one-way and two-way data binding using an input field example.

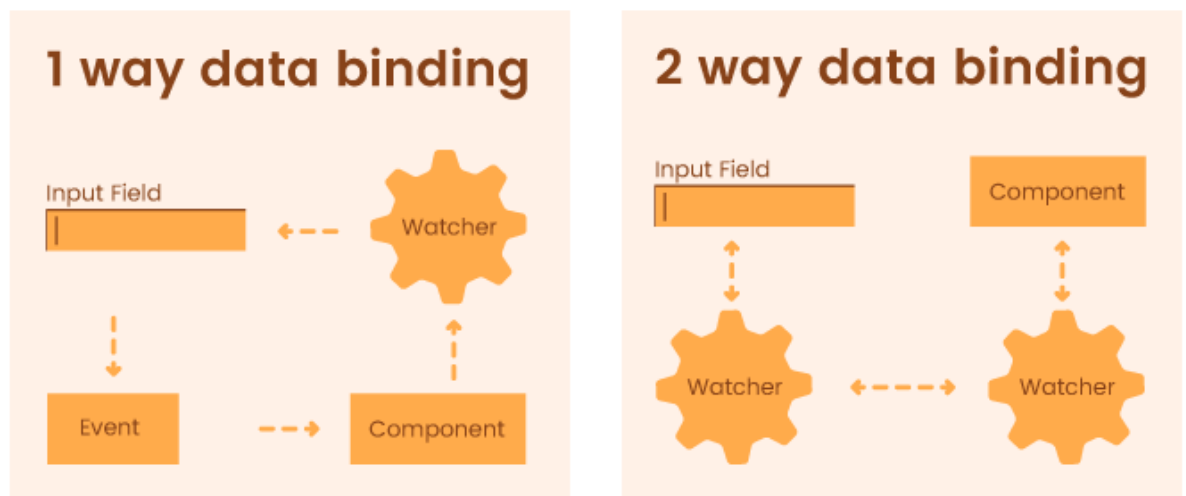


Figure 12. The difference between 1-way and 2-way data binding demonstrated using an input field

The way one-way data binding is handled in React is by placing all the application's components into a nested tree/subtree model where each component is either the toplevel container itself or contained within some other component. Each component has its own state in the form of a simple JSON-object that contains its specific up to

date attributes. Whenever a component's state is changed, React runs a series of lifecycle methods including `render()` which causes DOM changes. In this way, direct DOM manipulation is completely abstracted away from the developer. The state of a component can only be mutated through a private `setState()` command, meaning the different components cannot change each other's states unless they have been given a function containing `setState()` as a prop, which is a piece of data passed from one component to another that can contain virtually anything, including a reference to a function that is only available to the other component.

What makes this data flow unidirectional is that props can only be given by higher level components to lower level ones, meaning if a component 3 levels below the top-level container should cause a change or read data from higher up, that action needs to first be passed as a prop through all the components between them. In this way, a strict data flow hierarchy is maintained in the application where the immediate parent of the component always has the relevant information instead of there being a web of cross-references.

6.3 React.js performance

When considering the performance of React or any other framework it is important to remember that they are all ultimately just ways of writing JavaScript – the main purpose of these frameworks is more so to provide a scalable structure for UI development than to achieve the highest levels of performance optimization for any given scenario. Any front-end application could be stripped down to its base level DOM manipulation for maximum performance, but the result would likely be poorly structured. Because all the frontend JavaScript frameworks ultimately make use of the same base level JavaScript API, the differences in performance boil down to how much extra code the framework surrounds those tools with.

React's two-copy virtual DOM performs surprisingly well even though it sports more code at 31.8kb than some of the more lightweight libraries like Inferno (20kb) or Preact (4kb), but it is evidently not the fastest among the current most popular frameworks.

React's strengths largely lie outside of performance. One reliable benchmark test of the current popular front-end JavaScript frameworks was created by auth0.com, where many of the modern frameworks were made to perform series of tens of thousands of different DOM actions. The graph below displays a composite summary of all the tests combined.

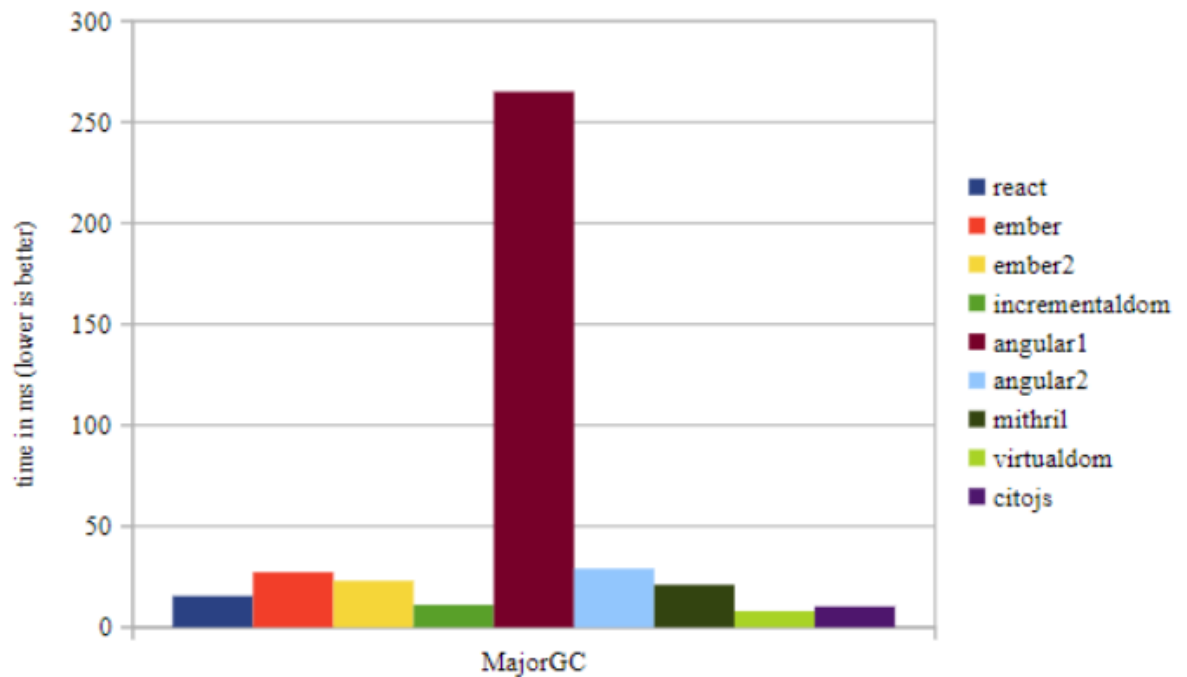


Figure 13. A composite of the Auth0 performance benchmark tests.

Although Angular 1 with its known performance issues makes the graph somewhat lopsided, it can be observed that React.js is in the middle of the pack when it comes to performance, which is very reasonable considering its other benefits.

6.4 Usage of React.js in the service

6.4.1 Reasoning behind the use of React.js

Because the front-end application part of the service is straightforward and lacks functionalities unique enough that one framework would be significantly favored on a

performance basis, the choice came down largely to personal preference. React.js does have certain advantages compared to other front-end frameworks, such as a relatively quick learning curve, backing and plans for future development from a large organization (Facebook) as well as strong documentation that made it a feasible framework to learn and use in the service. The main points that made React.js the framework of choice in the project were the ease of HTML-like developing in JSX, as well as the current and projected popularity of the framework.

6.4.2 The front-end application

The front-end application part of the service was developed entirely in React.js. The application contains a set of main components with nested subcomponents.

The client UI requires a lot of calculation to ensure bugless transitions between views without excessive repetition. All the main view components in the application that require data from the API use a similar pattern of ensuring that the API is only hit once through use of React lifecycle methods. When a new component is opened from Navbar in the MainContainer, the `componentWillMount()` method of the component is called, and if the content is not yet fetched, it is requested from the API via ES6 `fetch()`. After this, the subcomponents are generated in the MainContainer and cached in its state. This approach makes the content instantly available on subsequent loads. Listing below displays this pattern in the context of a single agreement clause, where `updateStorage()` is a function passed as a prop from `mainContainer.js` that generates the document elements.

```

componentWillMount() {
  window.scrollTo(0,0);
  if (this.props.luvut === undefined) {
    this.props.updateStorage(this.props.documentType);
  }
}

componentWillUpdate(nextProps) {
  if (nextProps.luvut === undefined) {
    this.props.updateStorage(this.props.documentType);
  }
}

render() {
  if (this.props.luvut === undefined) {
    return <div className="loader"></div>;
  } else if (this.props.luvut === null) {
    return <Redirect to="/notfound" />;
  }
}

```

Figure 14. The caching pattern used in every component that fetches data from the API

As per the listing above, the loading spinner is rendered by default to indicate a fresh state. If the application was much bigger, this design would be an unviable choice due to memory reservation and it would be necessary to have all document elements be recreated when remounting, but with the size of the current content the application runs smoothly even on weaker mobile devices.

Because the application is by nature a static content viewer, the content also has the option to have certain words highlighted. The application stays up to date on the currently opened view, and if the user opens a document from a search result, the text content is passed through a highlighter function displayed in attachment 1. The application was originally intended to use a highlighter package, but because they were either too heavy or lacked the capability for multi-word phrase recognition, a custom set of functions had to be implemented.

Much of the work in designing the front end was visual design. Most of the feel of the application is determined not by its performance or its internal design (since they are not relevant to the user unless done wrongly) but by its aesthetic, which ultimately deals purely with the design of elements in HTML and CSS. The role served by React.js involves the generation of these elements and correct behavior of the application during user interaction but React has no opinion on the visual look of the application. The final product makes use of Bootstrap CSS themes and personal opinion in the form of custom CSS rules to achieve a sufficient level of intuitiveness and aesthetic appeal.

7. System Design

7.1 UML Design

Design is the first step in the development phase for an engineered product or system. Design is the place where quality is fostered in software development. Design is the product or system. Software design serves as the foundation for all software engineers and software maintenance steps that follow. Without design we risk building an unstable design that will fail when small changes are made, one that may be difficult to test and one whose quantity cannot be accessed until late in the software engineering process. Taking software requirements specification document of analysis phase as input to the design phase we have drawn Unified Modeling Language (UML) diagrams. UML depends on the visual modeling of the system. Visual modeling is complexity better when it is displayed to us visually as opposed to written textually. By producing visual models of a system, we can show how system works on several levels. We can model and the interactions between the users and the system.

7.2 Types of UML Diagrams

Each UML diagram is designed to let developers and customers view a software system from a different perspective in varying degrees of abstraction.

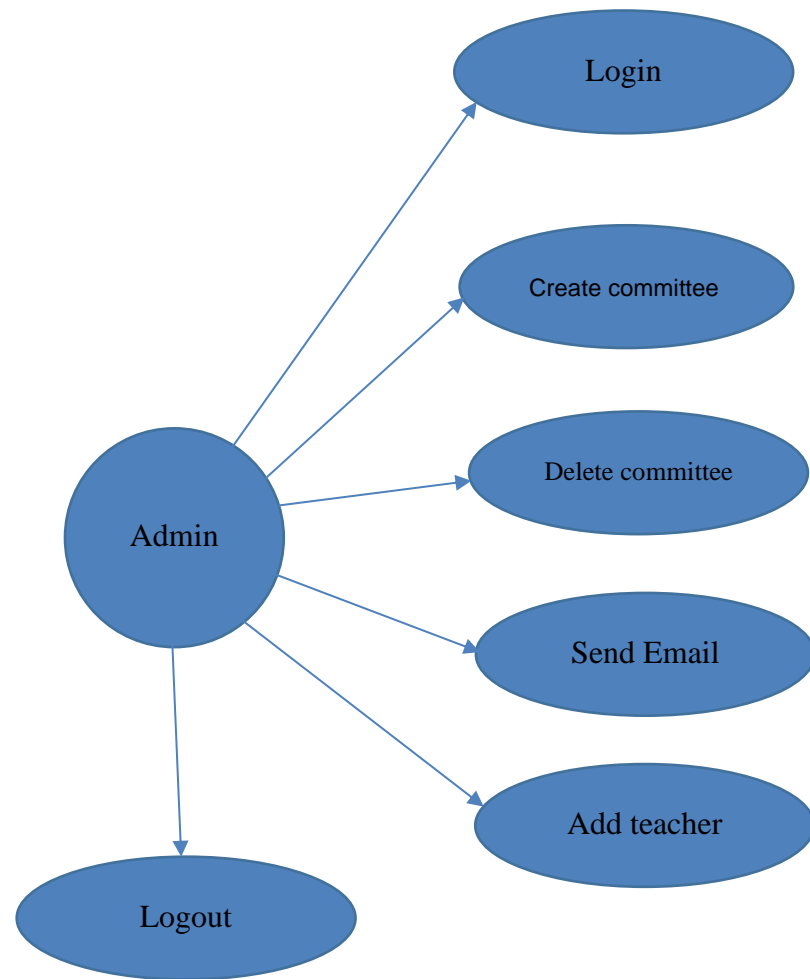


Figure 15. Use-case diagram for admin.

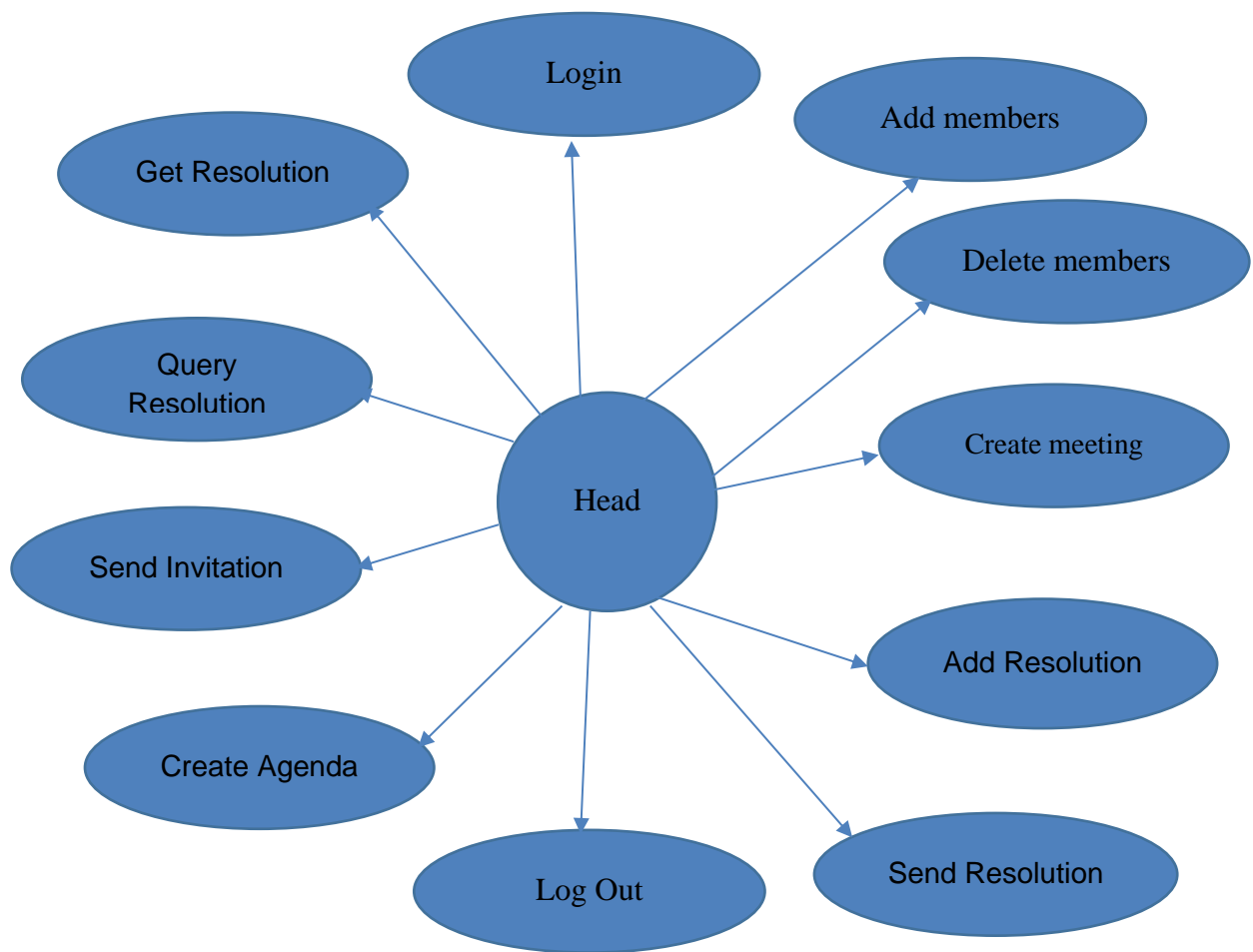


Figure 16. Use-case diagram for Head.

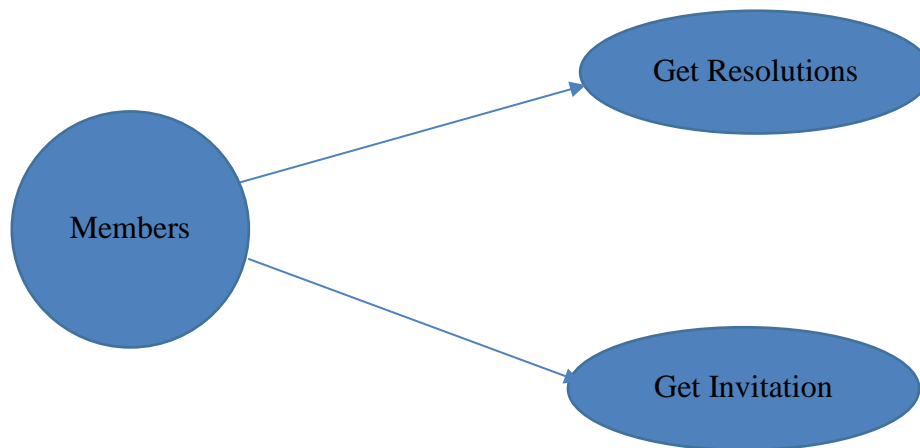


Figure 17. Use-case diagram for member.

8. System Implementation

This application deals with three modules

1. Administration Module
2. Head Module
3. Members Module

Administration Module:

1. Admin can login as a super user
2. Admin can create committee with a Head.
3. Admin send invitation with Head credentials
4. Admin can delete a committee
5. Admin can add new User
6. Admin can delete a User

Head Module:

1. Head can login.
2. Head can create meeting.

3. Head can add Member.
4. Head can delete Member
5. Head can send invitation to meeting members with meeting details
6. Head can add resolutions
7. Head can send resolutions to all members
8. Head can search with resolutions

Member Module:

1. Member receives meeting invitation from Head
2. Member receives meeting resolutions.

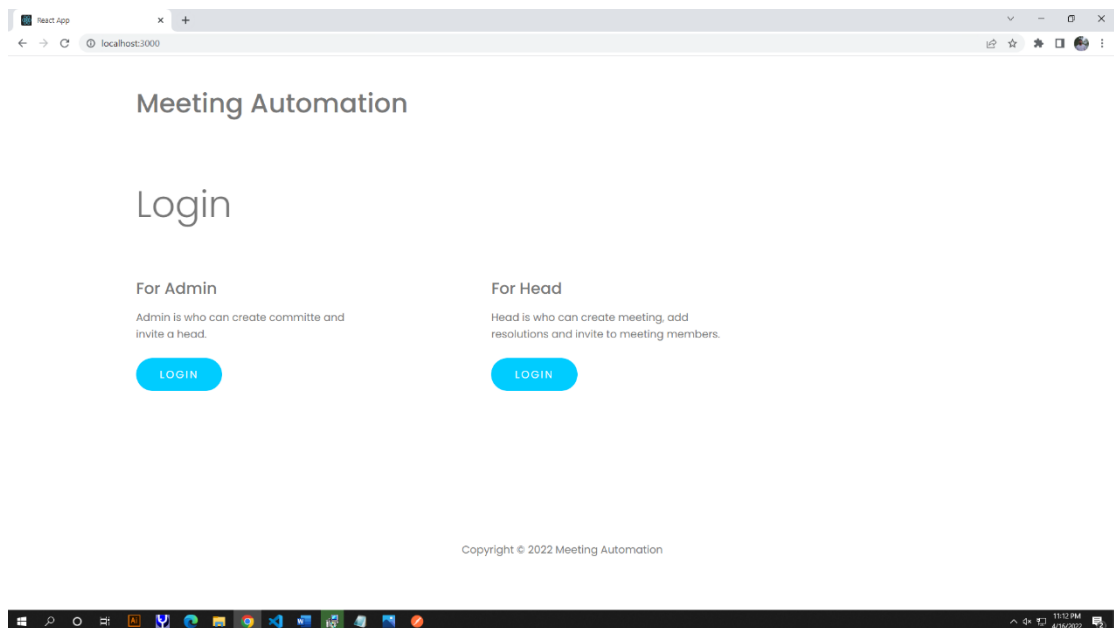


Figure 18. Home page.

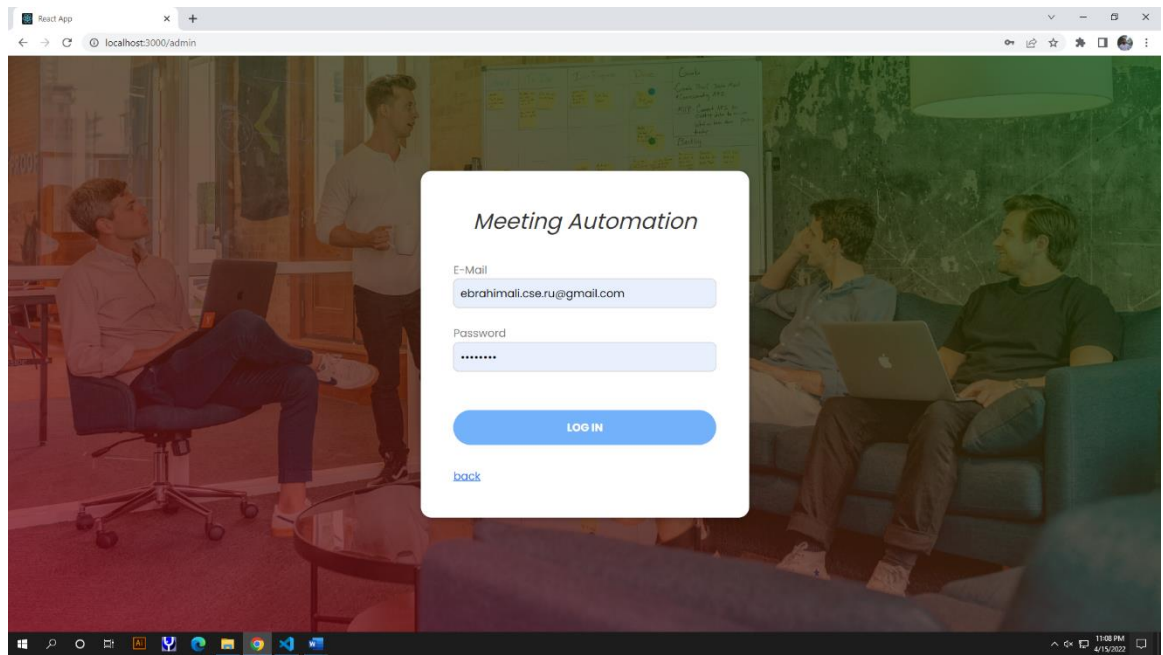


Figure 19. Admin login page.

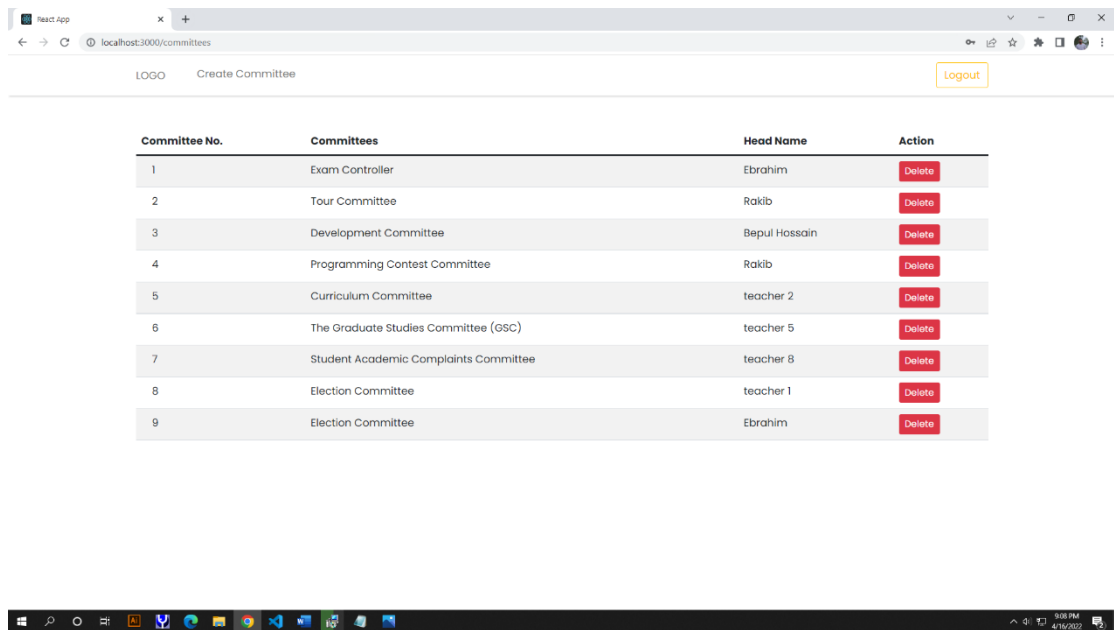


Figure 20. All committees list page.

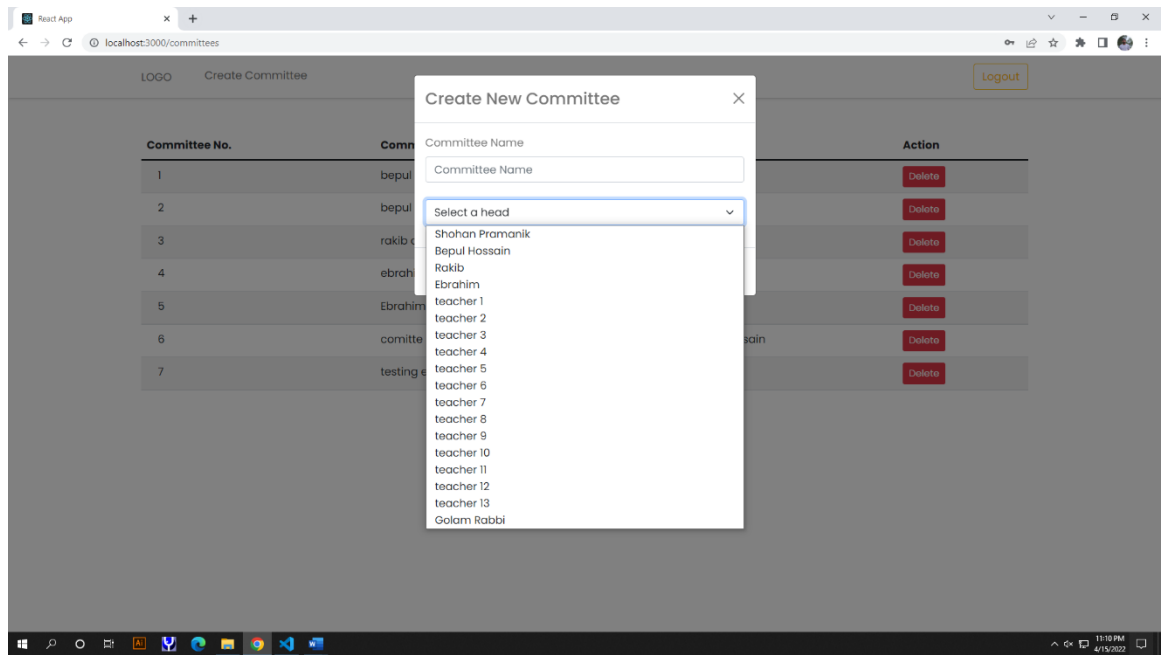


Figure 21. Create committee page.

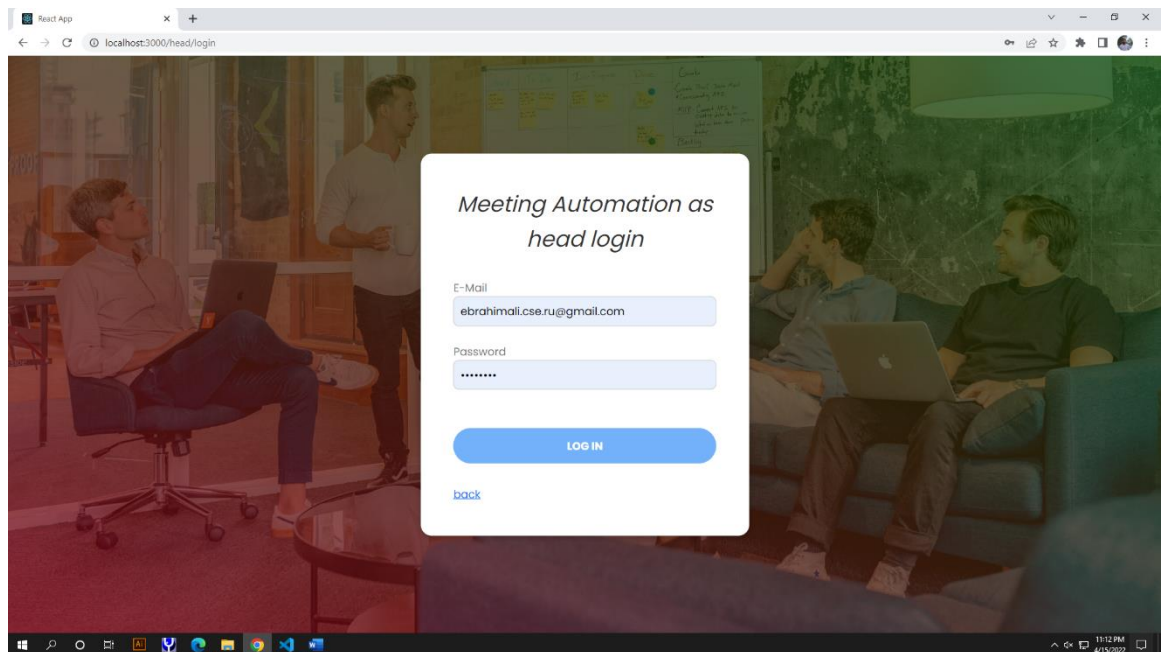


Figure 22. Head login page.

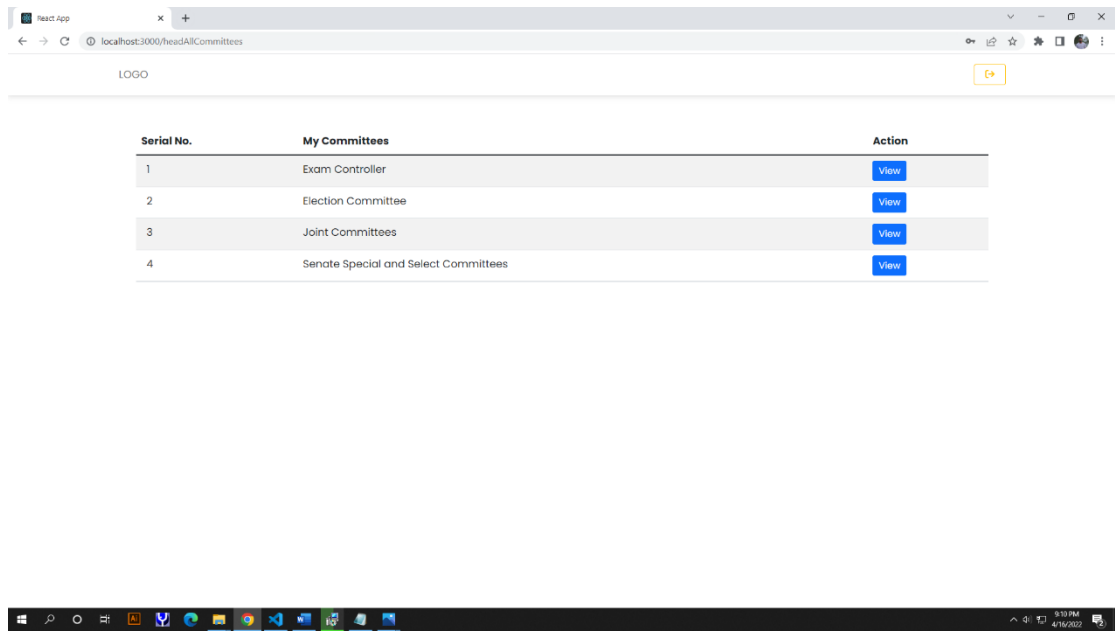


Figure 23. Head's committee page.

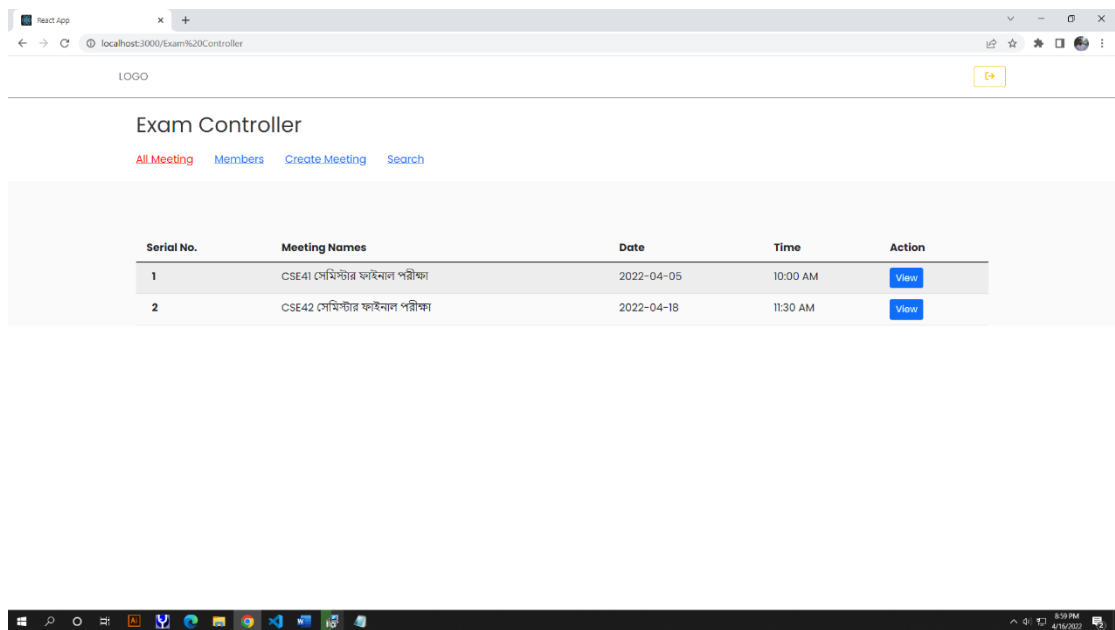


Figure 24. All meeting page.

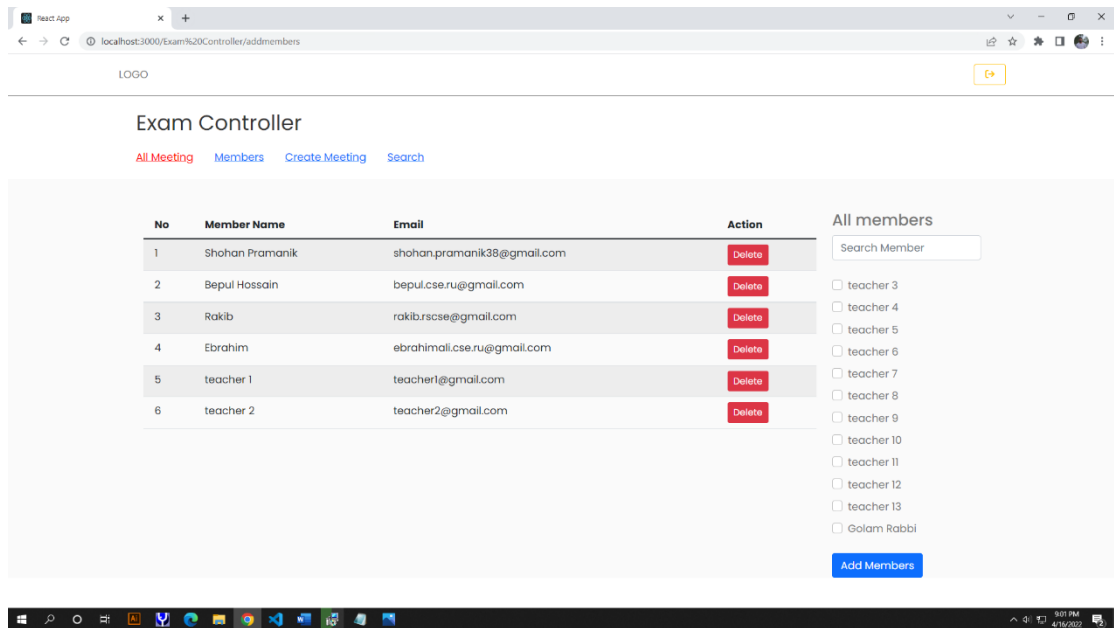


Figure 25. Add member page. where head can add a member.

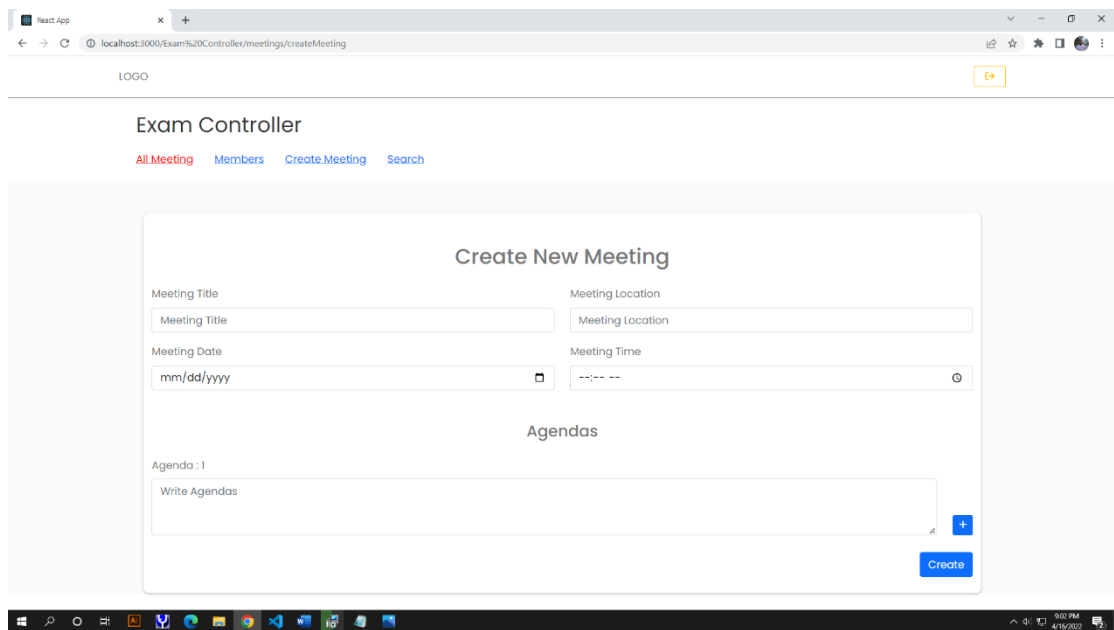


Figure 26. Created meeting page. Where head can create a meeting.

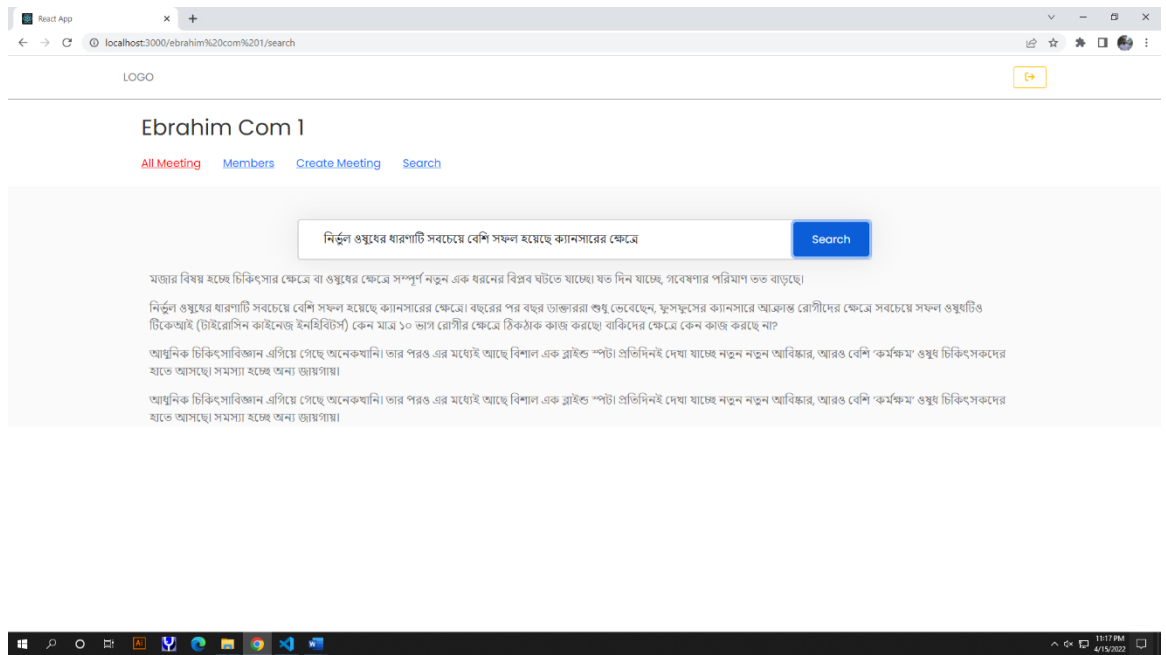


Figure 27. Search page. Where member search the resolutions.

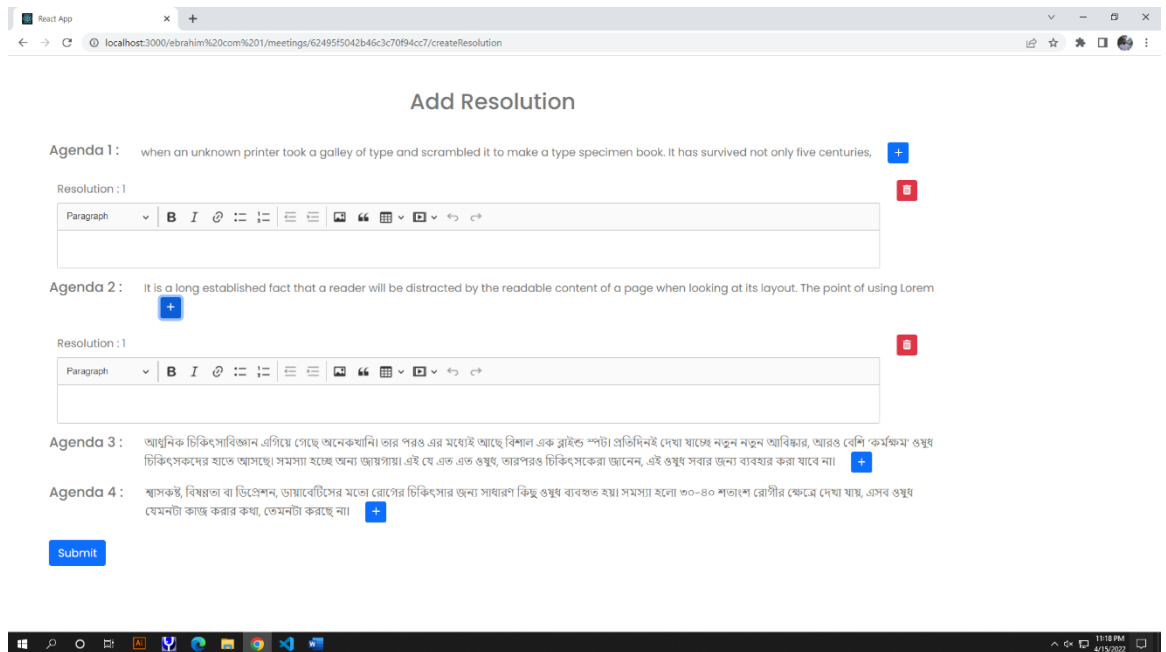


Figure 28. Resolution page. Where head can add resolutions.

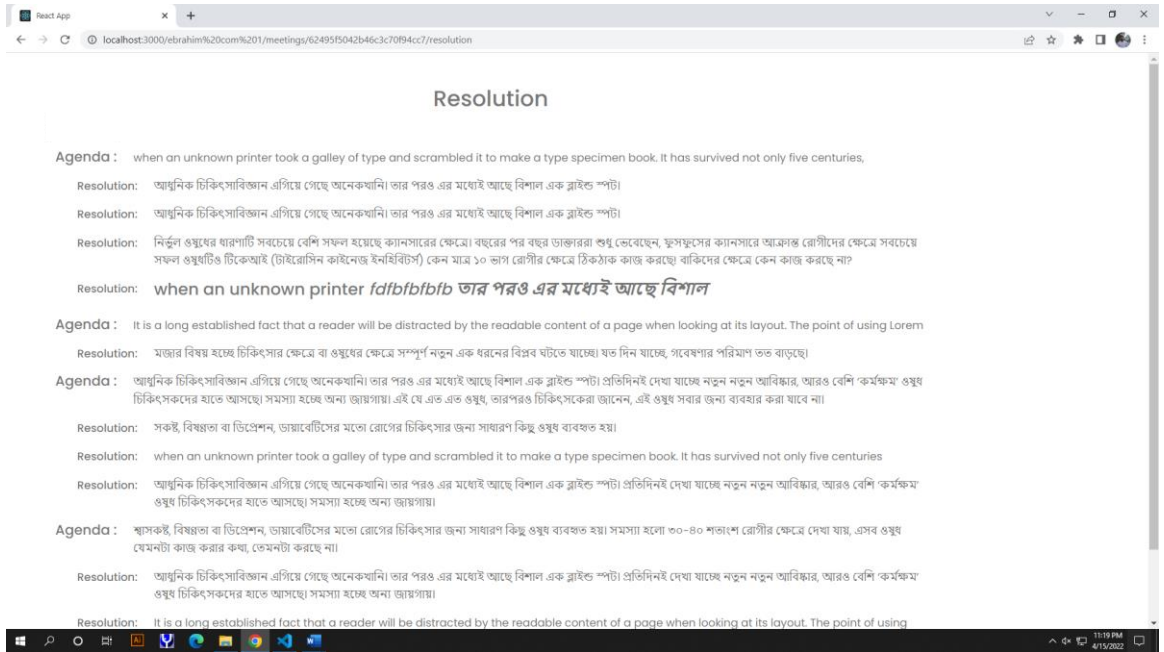


Figure 29. All resolutions show page.

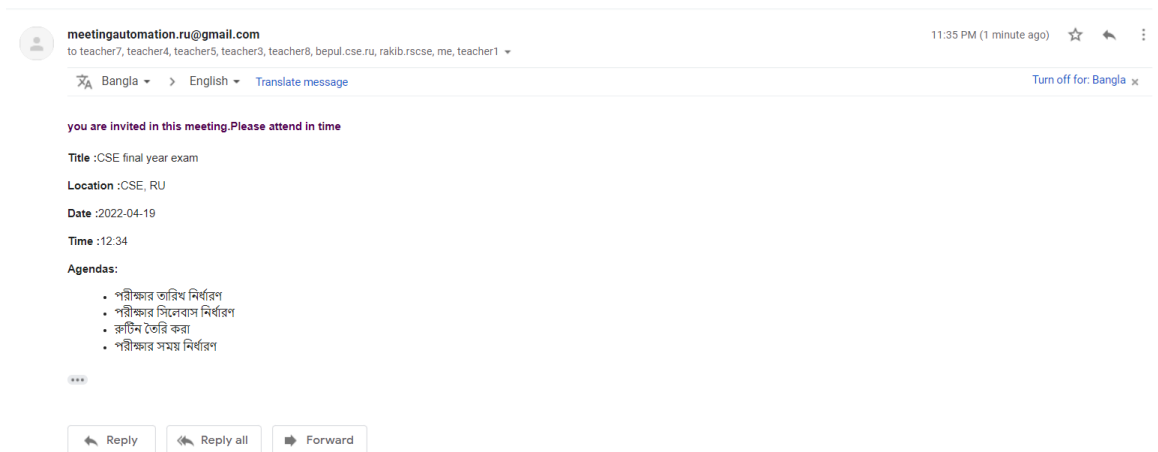


Figure 30. Invitation mail to member.



Figure 31. Head invitation mail go to from admin email.

9. Conclusion

It is web based application in which our main goal is to reduce office papers and store the documents of any meeting in live. In future, we can see the past resolution of meeting, which often we forget. It will be more informative and user friendly. We develop this project with React which use Virtual DOM. So it will be very faster. Will also use MongoDB which is no sql so, searching will be faster.

10. References

- [1] <https://www.mongodb.com/why-use-mongodb>
- [2] Kyle Banker, Peter Bakkum, Shaun Verch, Douglas Garrett, Tim Hawkins, "MongoDB In Action", second edition, Manning
- [3] <https://www.mongodb.com/docs/manual/core/index-compound/>
- [4] Christoforos Hadjigeorgiou, "RDBMS vs NoSQL: Performance and Scaling Comparison", The University of Edinburgh, August 23, 2013
- [5] <https://nodejs.dev/learn/the-v8-javascript-engine>
- [6] <https://www.npmjs.com/>
- [7] <https://www.toptal.com/back-end/server-side-io-performance-node-php-java-go>
- [8] <https://expressjs.com/>
- [9] <https://reactjs.org/>
- [10] <https://reactjs.org/docs/introducing-jsx.html>
- [11] <https://reactjs.org/docs/faq-internals.html>