

Markus Keinänen

Creation of a web service using the MERN stack

Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Programme

Thesis

21 August 2018

| | |
|---|---|
| Author Title | Markus Keinänen Creation of a web service using the MERN stack |
| Number of Pages Date | 52 10.9.2018 |
| Degree | Bachelor of Engineering |
| Degree Programme | Information Technology |
| Professional Major | Software Engineering |
| Instructors | Janne Salonen |
| <p>In recent years web development has seen large changes in every layer of the web service stack. The emergence of ECMAScript2015 and the rapid development of NoSQL-databases have given rise to a new paradigm of “JavaScript everywhere” which is now rapidly spreading in popularity.</p> <p>The MERN stack, short for MongoDB, Express.js, React.js and Node.js is one option for a technology stack that uses JavaScript in every layer of the web service – the database storing data in BSON (Binary JavaScript Object Notation) format, the back-end using the JavaScript-based Node.js as the server process and Express.js as an additional JavaScript framework for additional server features, and the front-end using React.js to forge JavaScript-generated client UI components.</p> <p>The problem introduced at the beginning of the thesis is the difficulty of accessing and finding relevant information from a set of documents related to construction consulting. The solution proposed is a web service created using the MERN stack where the documents are accessible and searchable through a browser-based viewer application. The thesis explores each component of the MERN stack in depth regarding their features and performance and explains how they were used in the creation of the service.</p> <p>A web service that meets the requirements is successfully created and deployed using the MongoDB Atlas and Heroku cloud platforms.</p> | |
| Keywords | MongoDB, Express, Node, React, JavaScript |

| | |
|--|---|
| Tekijä Otsikko | Markus Keinänen Creation of a web service using the MERN stack |
| Sivumäärä Aika | 52 10.9.2018 |
| Tutkinto | Insinööri (AMK) |
| Koulutusohjelma | Tieto- ja viestintäteknikka |
| Suuntautumisvaihtoehto | Ohjelmistokehitys |
| Ohjaaja | Janne Salonen |
| <p>Viime vuosina verkkopalveluiden kehityksessä on tapahtunut laajoja muutoksia verkkopi- non jokaisella tasolla. ECMAScript2015:n ilmaantuminen ja NoSQL-tietokantojen nopea kehitys ovat mahdollistaneet uuden ”JavaScriptiä kaikkialla”-ajatusmallin, jonka suosio le- viää nopeaa vauhtia.</p> <p>MERN-pino, joka on lyhenne nimistä MongoDB, Express.js, React ja Node.js on teknolo- giapino, joka käyttää JavaScriptiä verkkopalvelun jokaisella tasolla – tietokannassa säilyt- tämällä dataa BSON (Binary JavaScript Object Notation) -formaattissa, palvelinpäässä käyttämällä JavaScript-pohjaista Node.js:ää palvelinprosessina sekä Express.js-viiteke- hystä palvelimen lisätoimintojen rakentamiseen, ja käyttäjäpäässä UI-komponenttien luo- miseen React.js JavaScript-viitekehityksen avulla.</p> <p>Tutkimusprojektin alussa esitellään ratkaistava ongelma, joka on tiettyjen rakennuskonsul- tointiin liittyvien dokumenttien saatavuus ja niissä sijaitsevan olennaisen tiedon löytyvyys. Ratkaisuksi esitetään MERN-pinolla kehitetty verkkopalvelu, jossa dokumentit ovat saata- villa ja niissä sijaitseva tieto etsittävässä selainpohjaisen katseluohjelman avulla. Tutkimus- projektissa syvennytään MERN-pinon jokaiseen rakennosaan niiden ominaisuuksien ja suorituskyvyn osalta ja selitetään, miten pinon teknologioita käytettiin palvelun luomisessa.</p> <p>Vaatimukset täyttävä verkkopalvelu luodaan onnistuneesti ja otetaan käyttöön MongoDB Atlas ja Heroku -pilvipalveluiden avulla.</p> | |
| Keywords | MongoDB, Express, Node, React, JavaScript |

Contents

List of Abbreviations

| | | |
|-------|---|----|
| 1 | Introduction | 1 |
| 2 | Background to the problem | 2 |
| 3 | Study and usage of MongoDB | 3 |
| 3.1 | The emergence of NoSQL | 3 |
| 3.2 | Basic information about MongoDB | 4 |
| 3.3 | Features of MongoDB | 5 |
| 3.3.1 | The MongoDB document model | 5 |
| 3.3.2 | MongoDB scaling and replication | 7 |
| 3.3.3 | Indexing | 9 |
| 3.4 | MongoDB performance | 10 |
| 3.5 | Usage of MongoDB in the service | 12 |
| 3.5.1 | Reasoning behind the use of MongoDB | 12 |
| 3.5.2 | Storing the application data in MongoDB Atlas | 13 |
| 4 | Study and usage of Node.js | 14 |
| 4.1 | Basic information about Node.js | 14 |
| 4.2 | Features of Node.js | 15 |
| 4.2.1 | Compatibility with JavaScript | 15 |
| 4.2.2 | The Node.js event loop | 16 |
| 4.2.3 | Node Package Manager | 20 |
| 4.3 | Node.js performance | 21 |
| 4.4 | Usage of Node.js in the service | 24 |
| 4.4.1 | Reasoning behind the use of Node.js | 24 |
| 4.4.2 | Use of Node.js on the server side | 24 |
| 4.4.3 | Use of Node.js on the client side | 26 |

| | | |
|---------|--------------------------------------|----|
| 5 | Study and usage of Express.js | 27 |
| 5.1 | Basic information about Express.js | 27 |
| 5.2 | Usage of Express.js in the service | 28 |
| 5.2.1 | Serving static content | 28 |
| 5.2.2 | REST API Routes | 29 |
| 5.2.2.1 | Content delivery routes | 29 |
| 5.2.2.2 | The search engine | 30 |
| 5.2.2.3 | Authentication | 32 |
| 6 | Study and usage of React.js | 34 |
| 6.1 | Basic information about React.js | 34 |
| 6.2 | Features of React.js | 34 |
| 6.2.1 | JSX | 34 |
| 6.2.2 | Virtual DOM | 36 |
| 6.2.3 | One-way data binding | 37 |
| 6.3 | React.js performance | 39 |
| 6.4 | Usage of React.js in the service | 40 |
| 6.4.1 | Reasoning behind the use of React.js | 42 |
| 6.4.2 | The front-end application | 42 |
| 7 | Deployment of the service | 44 |
| 8 | Summary | 45 |
| | References | 46 |

List of Abbreviations

| | |
|------------|---|
| RDBMS | Relational database management system. Software for maintaining, querying, and updating data and metadata in a relational model database. |
| SQL | Structured Query Language. The language used for programming and managing data in relational databases (RDBMS). |
| NoSQL | “Non SQL” or “Not Only SQL”, a database type categorization for databases that are not based on Structured Query language |
| ECMAScript | <p>A trademarked scripting-language specification formed by Ecma International to standardize JavaScript.</p> |
| HTTP | The Hypertext Transfer Protocol. An application protocol for distributed, collaborative hypermedia information systems used for data communication in the World Wide Web. |
| API | Application programming interface. A set of subroutine definitions, communication protocols and tools for building software. |
| CSS | Cascading Style Sheets. A style sheet language used for describing the presentation of a document written in a markup language. |
| ORM | Object relational mapping. A programming technique for converting data between incompatible type systems using object-oriented programming languages. |
| CPU | Central processing unit. The electronic circuitry within a computer that carries out the instructions of a computer program. |
| RAM | Random access memory. A form of computer data storage that stores data and machine code currently being used by the computer. |

| | |
|-------|---|
| JRE | The Java Runtime Environment. A set of software tools required for running Java applications consisting of the Java Virtual Machine (JVM), platform core classes and supporting libraries. |
| NGINX | A lightweight open source web server created by Igor Sysoev often used as a load balancer or a reverse proxy. |
| SSL | Secure Sockets Layer. A cryptographic protocol designed for providing communications security over a computer network. Often actually refers to Transport Layer Security (TLS), which is an updated version of SSL. |
| DNS | Domain Name System. A hierarchal decentralized naming system for resources connected to the Internet or a private network that assigns domain names to participating entities. |
| REST | Representational State Transfer. An architectural style used in creation of web services where the requesting systems can access and manipulate web resources through a uniform and predefined set of stateless operations. |
| HTML | Hypertext Markup Language. The standard markup language for creating web pages and web applications. |

1 Introduction

With the recent progress of ECMAScript 2015, NoSQL databases and Node.js, as well as the increasing resources and capabilities of cloud service providers, web development has become practical and abstracted enough to enable even sophisticated full-stack web solutions to be created with significantly lesser amounts of effort and work hours than what was required by the previous popular technologies.

This thesis explores the process of creating a service that allows users to browse and search for documents related to the terms and conditions in a set of documents relevant to the Finnish construction consulting industry using modern web development technologies, including descriptions of the technologies used and the reasoning behind their selection. The goal of the thesis is to produce a solution that provides users with all the desired functionalities, and to research and judge the fidelity of the chosen technologies for their respective purposes.

The project is limited both in scope and available resources. Because of this, suitable technologies are limited to open-source software. The technology stack used for the project in this study consists of MongoDB for the data storage layer, Node and Express for the service layer with Json Web Token for API authentication, React for the frontend client and CSS with Bootstrap for the client styling. The reasoning behind these choices is expounded on a per-technology basis in later chapters. Due to the limited scope, certain aspects of the service such as payment software and parsing scripts are not covered in the thesis. The thesis will only explore the 4 core technologies comprising the MERN stack, namely MongoDB, Express, React and Node, as well as certain smaller libraries immediately attached to them in the final service.

The thesis includes 8 sections in total, starting with an explanation of the problem that the service constructed in the thesis seeks to solve, then conducting a technical investigation of each of the 4 core technologies including their practical application in the service, and ending with the conclusion explaining details about the final product and the creation process.

The topic of the thesis was chosen both out of personal interest and because the service in question was already in the works in an extracurricular context, which allowed for a combination of the two projects.

2 Background to the problem

The problem that this project in this seeks to solve has to do with the accessibility of important documents in the field of construction consulting. The most commonly used documents, namely YSE (Rakennusurakan Yleiset Sopimusehdot, 1998), KSE (Konsulttitoiminnan yleiset sopimusehdot, 2013) and JYSE (Julkisten hankintojen yleiset sopimusehdot, 2014) are currently only available on the internet as unregulated PDF files. These documents are used by various actors in the construction industry and deal either with strict laws regarding construction or non-governmental sets of rules produced by 3rd parties that have solidified as industry conventions used in contracts.

For a consultant to find information contained in these documents, they would need to first seek and download one of these publicly hosted PDF files, and then scroll through a daunting body of text to find portions that might be associated with the desired topic. This is an inconvenience especially in cases where information needs to be found quickly or where the name and location of the desired content is unknown. Examples of this might be a mobile user at a construction site, a boardroom meeting, or an argument regarding a term that requires knowledge of the context to understand. The information contained in these documents has never been standardized in a digital form beyond text inside of a PDF-file.

The information in question would become significantly easier to access and browse through the provision of the following features:

- All the documents concentrated in a single endpoint, accessible through a variety of digital devices with a user-friendly UI
- Document components hyperlinked to enable quick navigation through an index
- A search engine for all the documents to help find relevant portions of the text in various ways

In order to fulfill the requirements, the service in question would have to be accessible through the web. A web application can be constructed in multiple different ways, but in a traditional web application there are 3 layers – the web layer, the service layer, and the repository layer. The web layer is the entry point of the application displayed to the user and handles the interactions between the user and the application. The service layer provides the internal logic of the service including communication with the repository layer along with a public interface (API) exposed to the web layer and accessible through

the transport protocol (most commonly a HTTP request). The repository layer is responsible for the data storage, which is to say, communication with the database.

To solve the listed problems, an appropriate solution would be a web-based application that stores the desired documents in a database and allows for a browser-based client to fetch and query them.

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 [1], 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 [2]. 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 [3], demands have increased both for the types of data to be stored as well as the frequency of operations required by database software [4].

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 [5], 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 [5]. 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 [6]. This type of scaling is made inconvenient by the scaling itself requiring downtime [7], 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 [8]. 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 [6][7].

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 [9]. 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 [10]. MongoDB, however, offers ACID guarantees on a single-document level with multi-document ACIDity scheduled for summer 2018 [11].

3.2 Basic information about MongoDB

MongoDB is an open-source NoSQL database under the document database category, created in 2007 in the U.S by Dwight Merriman, Eliot Horowitz and Kevin Ryan [12]. 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 [13], 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 [14]. 10gen was later renamed into MongoDB Inc, and the first version was released in February 2009 [14] [15]. 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 consists of a single JSON object called a document. The database contains multiple groupings of documents called collections [16].

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" [17] displays the difference between how MongoDB and MySQL store the same example data of an entry on a social media site. 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.

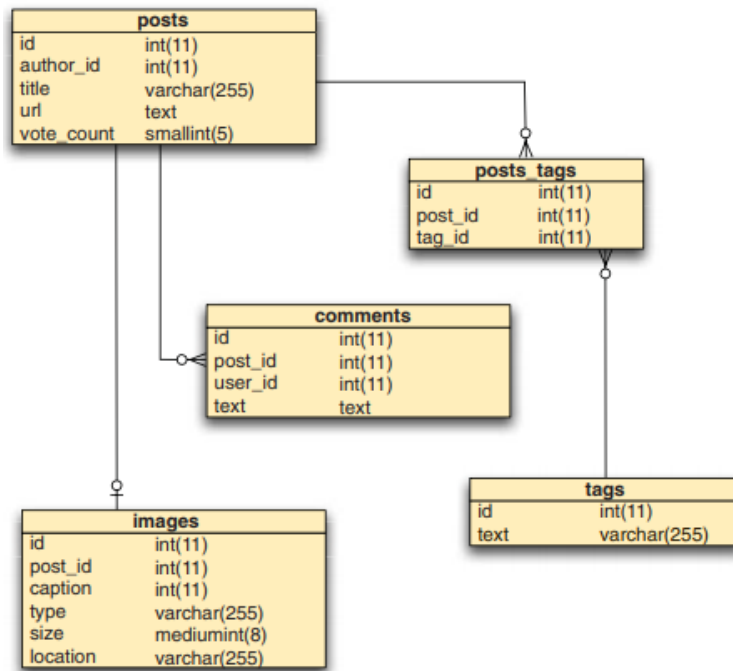


Figure 1. A likely MySQL representation of an entry on a social media site

```

{ _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'
    }
  ]
}

```

← **id field is primary key**

1 **Tags stored as array of strings**

2 **Attribute points to another document**

3 **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 [16]. 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 [17].

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 [18].

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 [18].

Figures 3 and 4 from the official MongoDB documentation [18] 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.

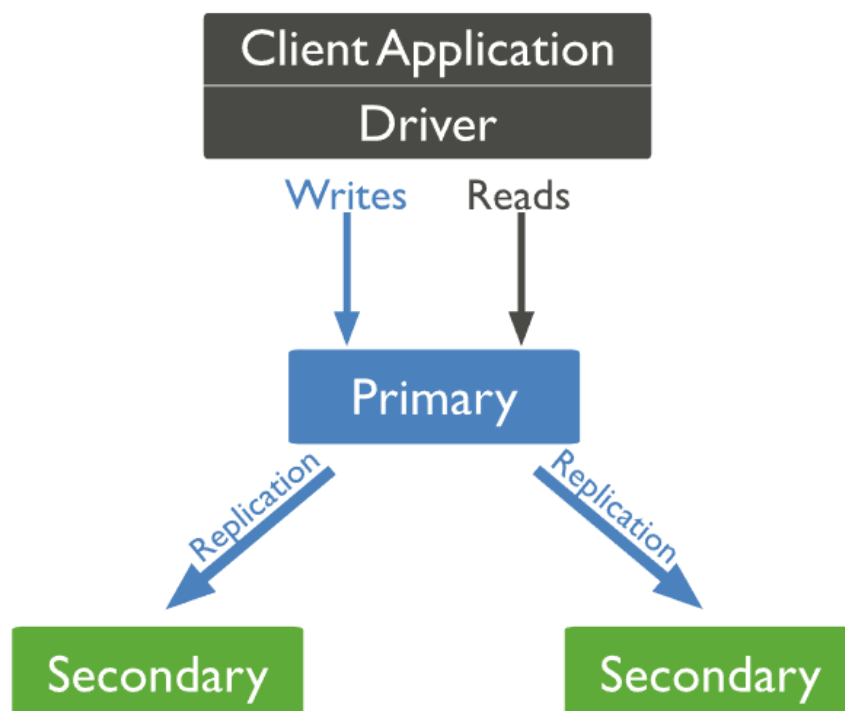


Figure 3. A graphical depiction of a single replica set with no arbiter node

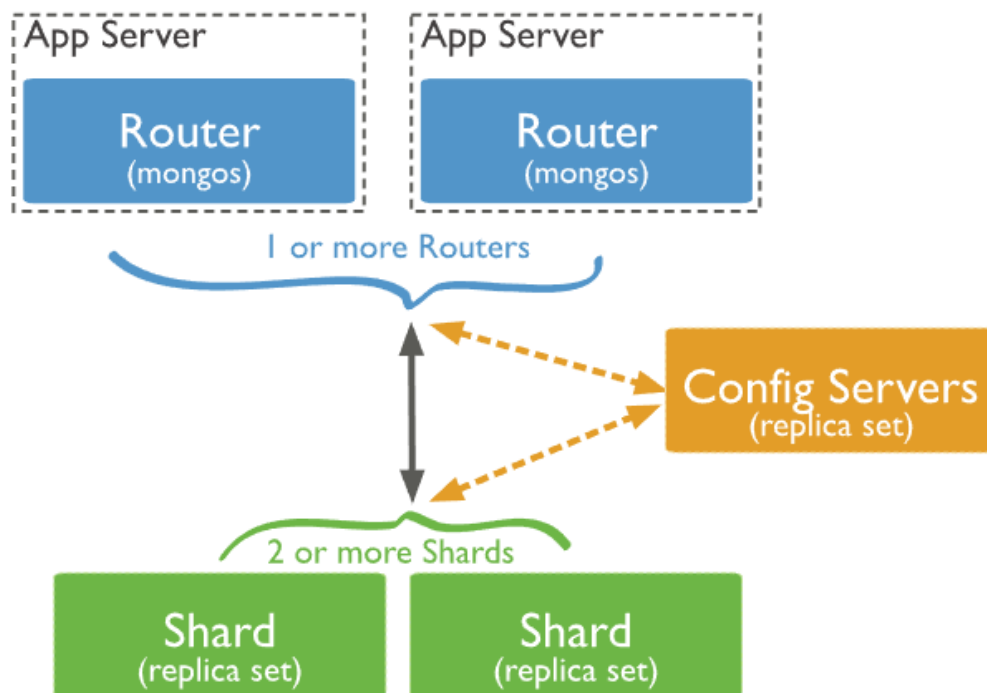


Figure 4. A graphical depiction of a sharded MongoDB database

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 [18]

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 [18].

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. 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 [19].

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 [19].

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 [20]. 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” [21] benchmark tests are conducted to demonstrate the speeds of different operations between MySQL and MongoDB.

Tables 1 and 2 [21] 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.

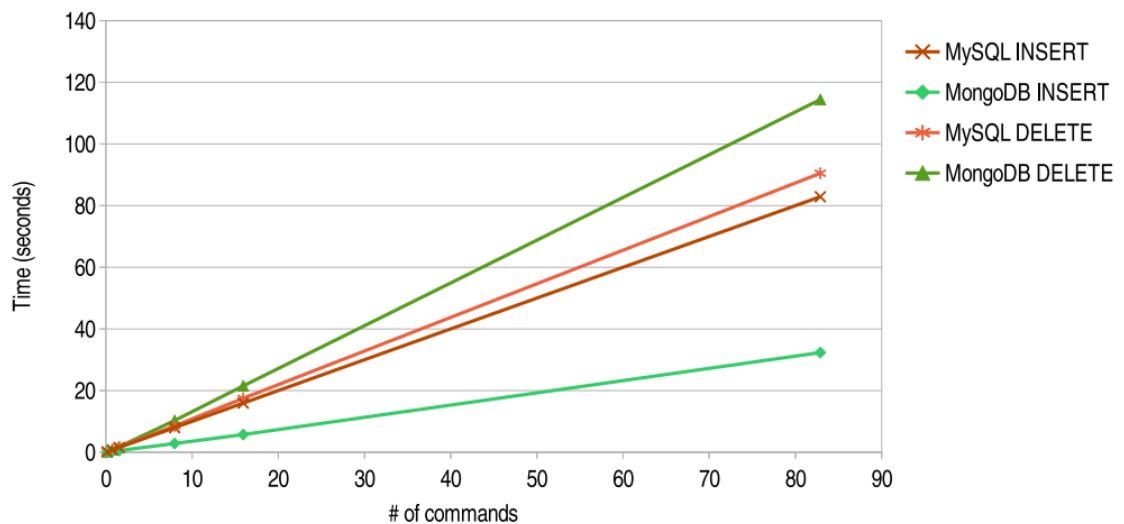


Table 1. Graph depicting the speeds of MySQL and MongoDB performing INSERT and DELETE operations. Lower time is better.

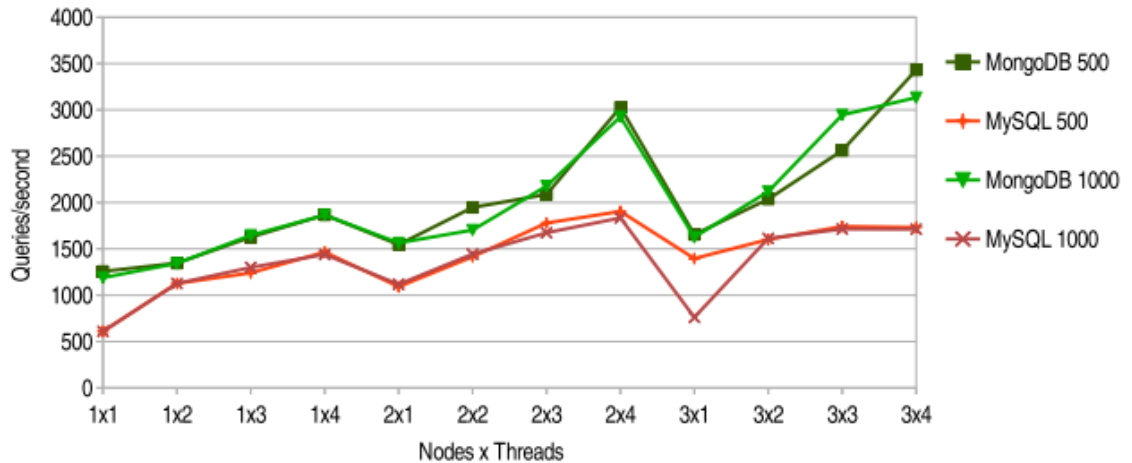


Table 2. Graph depicting the speeds of MySQL and MongoDB performing a SELECT INNER JOIN query, the number next to the name on the right side of the graph is the number of times the query was repeated. Higher queries per second is better.

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

The application in question would contain the following unique types of data:

- User
- Glossary item
- YSE 1998, KSE 2013 or JYSE 2014 clause

When formatting this data into the database, the user and clause types would fit neatly into an SQL structure by virtue of having no nested elements. However, when considering the clauses and glossary items, a nested structure would enable the entire item to fit into one object in MongoDB but would, depending on design, require either 2 or 3 SQL tables to persist the data.



In order to fetch a single clause, the SQL form would also require 2 or 3 JOIN operations for a selection. After this the data would need to be meshed, converted into JSON and then transported to the front-end client for consumption. The direct JSON format is therefore evidently simpler for this type of data.

Considering the faster query speed of MongoDB over MySQL, MongoDB's document storage is the better choice for the application. The main advantage that MySQL has over MongoDB is its default transactionality, but since the actions performed using the database are almost entirely limited to selection operations of one or more documents and no feature involving transactions is required for the application, there is little reason to choose MySQL over MongoDB in this instance.

3.5.2 Storing the application data in MongoDB Atlas

MongoDB Inc., in addition to maintaining the software itself, offers a database hosting service called Atlas. Atlas provides a free tier package with 0\$/hour upkeep with 512MB of storage and a maximum of 500 collections. When creating a new database, Atlas M0 is deployed by default with 3 replica sets and out-of-the-box SSL encryption along with a management console that enables an easy set up and maintenance of the database [22]. Figure 7 below displays the Atlas management console of the cluster used for storing data in the project.

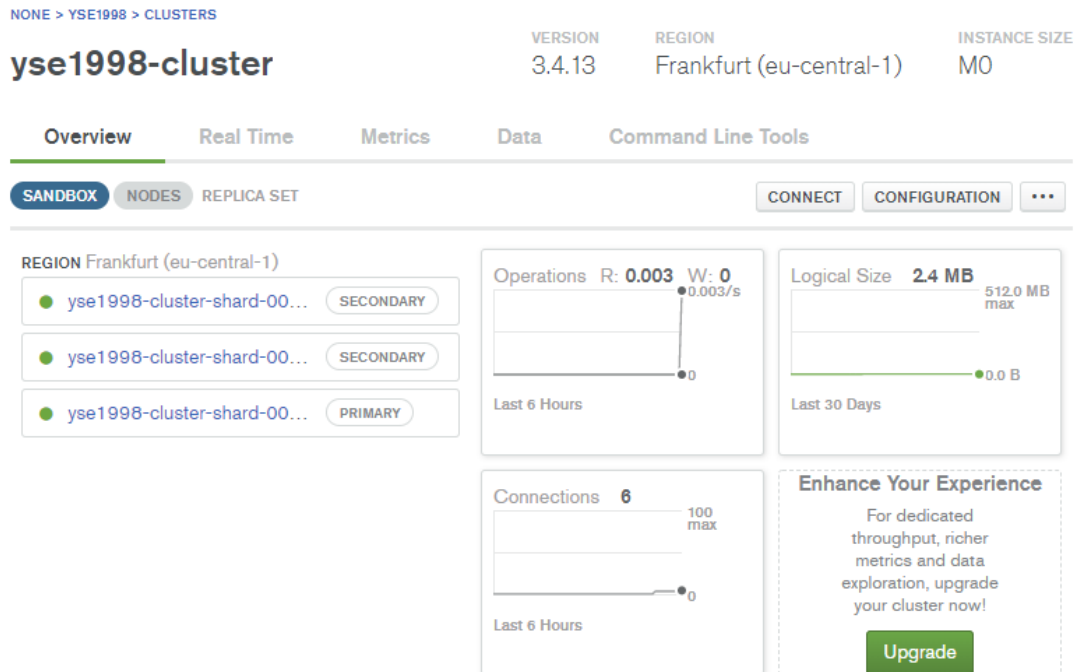


Figure 5. A screenshot of the MongoDB Atlas management console running the cluster that contains the data used in the application

The data from YSE 1998, KSE 2013 and JYSE 2014 from the public PDFs were converted to the JSON format using a personal custom-made Java-based parser program. The documents were then inserted to an Atlas cluster running in Frankfurt using MongoDB Compass, which is a program provided by MongoDB for easier interaction with the database.

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 [23]. 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 [23]. 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 an NGINX-like base library of modules related to the most common requirements such as file management, HTTP, SSL, DNS, compression and cryptography [24]. The basic modules are aimed at making the Node.js process into a functional web server [23] 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) [25] 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. [26] and is used in nearly 0,4% of all websites compared to 0,3% last year [27].

Node.js was created in 2009 by Ryan Dahl [28], 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 EventMachine and Twisted [28], 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 [30].

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

As can be observed from Figure 8, the HTTP request is fired away, and the future result of the request is tied to the onreadystatechange 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.

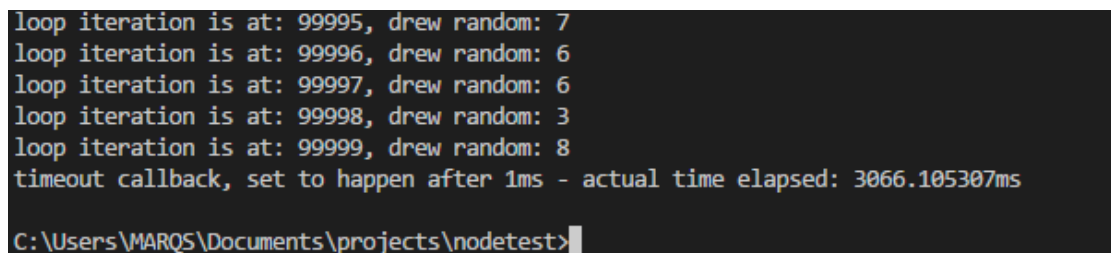
The Node.js process is a loop that performs polling and blocking calls to the system kernel on a constant basis while active. The operating system kernel in turn notifies Node.js when an operation is complete, after which its associated callback function is added to the poll queue for eventual execution. 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 7. 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 8. The end of the console output of the script in figure 9

As can be seen from the output of the script in figure 10, 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 [23] [31] (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 [33].

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 FileSystem, DNS and crypto) [31]. 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 [34].

Figure 11 below from a Node.js foundation article [31] 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.

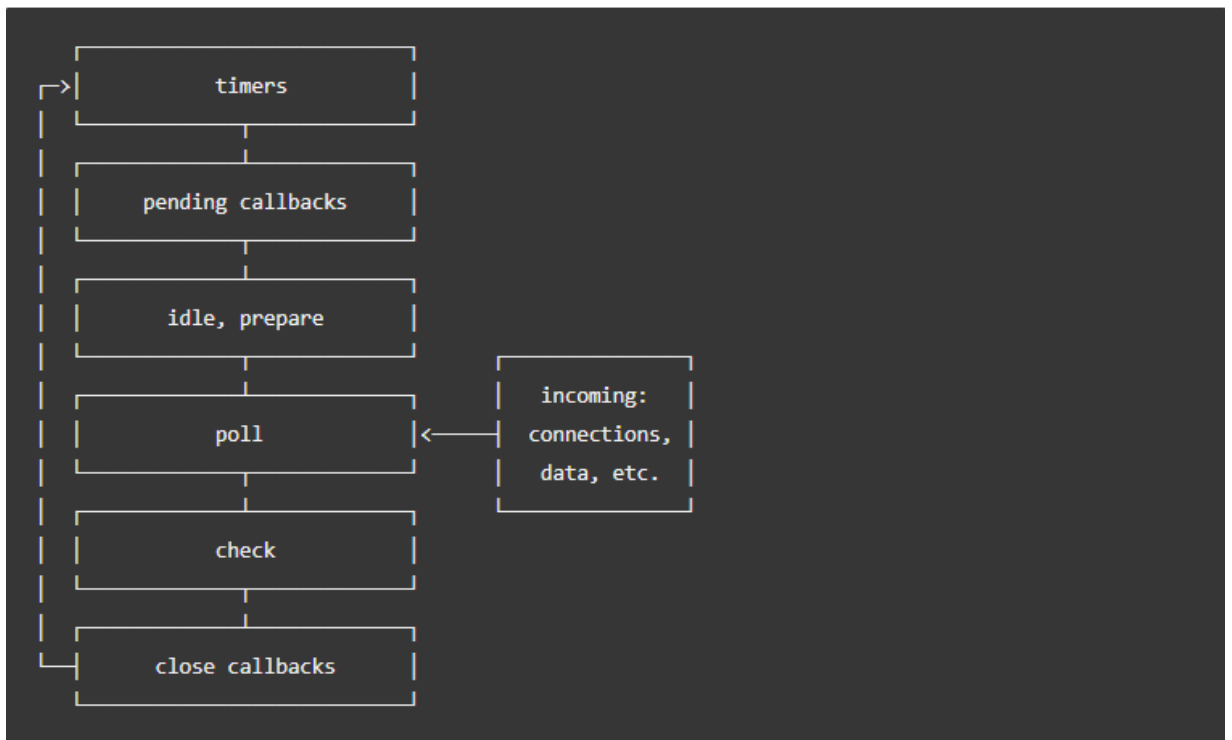


Figure 9. The order of phases in a single cycle of the Node.js event loop

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. [31]

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 [31].

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 [31].

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 [31].

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 [31].

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 [31].

4.2.3 Node Package Manager

The Node Package Manager or NPM is currently the world's largest software registry, 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 [35].

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 <package name>`" [35].

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 [35].

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 [24].

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. 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 [36]. 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 [37]. The results of the tests are displayed below in table 3.

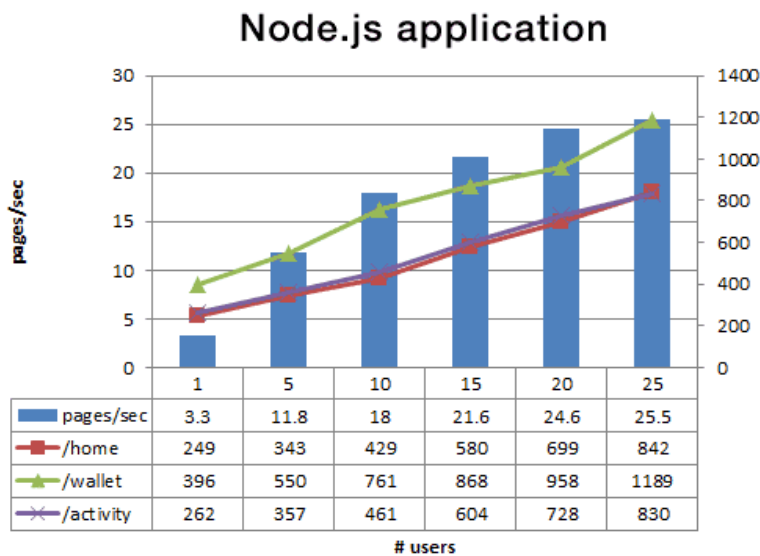
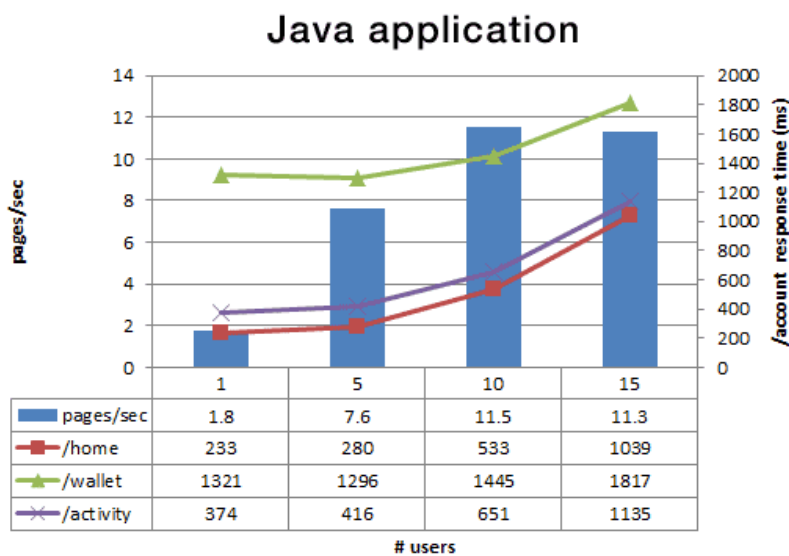


Table 3. Graphs depicting the PayPal Node.js vs Java benchmark test results [37]

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 [37].

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) [38]. 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.

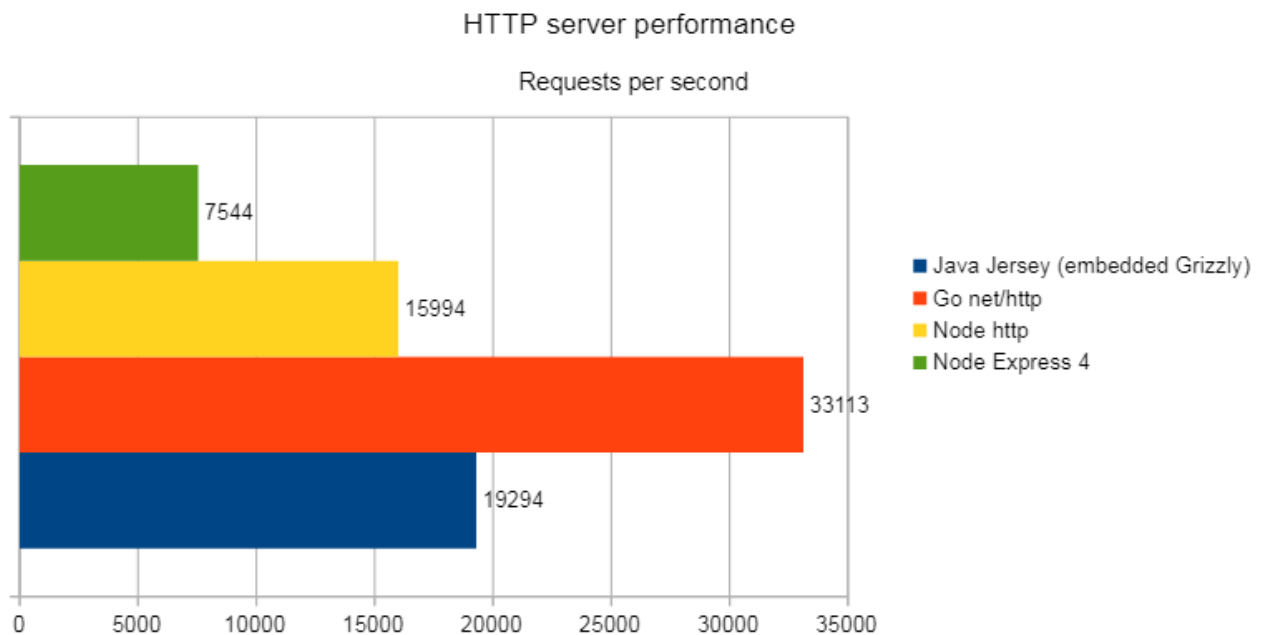


Table 4. Results of the lower concurrency web server benchmark test [38]

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 pre-compiled [41] as opposed to JavaScript which is dynamically typed [39] and continuously dynamically recompiled during runtime by the V8 engine for optimization purposes which requires extra heuristics that slow down the overall processing [40].

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 expect- edly 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 com- bination 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.

4.4.2 Use of Node.js on the server side

In the service Node.js was employed as a web server process using Express.js, which is a framework that implements common web server functionalities using Node.js core features. The web server in the application serves both the static browser resources and the JSON content. Express.js is covered in its own analysis later, which is why the web server's more specific features contained in the API routes (although technically imple- mented through Node.js) are not covered in this section.

In addition to providing the web server process, NPM was used through package.json for importing dependencies used in the various API routes, such as Bcrypt for encryption and hashing functionality, Snowball for Finnish word stemming in the search engine, JsonWebToken for authentication and NodeMailer for sending notification emails from the server. Package.json also specifies the scripts that are run after deployment to the cloud.

Figure 12 below shows the full dependencies section of the package.json used in the root server folder of the server side application. After deploying the service, these dependencies are installed via NPM on the virtual instance running it.

```
"dependencies": {
  "bcryptjs": "^2.4.3",
  "body-parser": "^1.18.3",
  "braintree": "^2.5.0",
  "compression": "^1.7.2",
  "cookie-parser": "~1.4.3",
  "cors": "^2.8.4",
  "debug": "~2.6.3",
  "express": "^4.16.3",
  "express-minify": "^1.0.0",
  "express-sslify": "^1.2.0",
  "helmet": "^3.12.0",
  "json-format": "^1.0.1",
  "jsonwebtoken": "^8.1.1",
  "mongoose": "^4.13.7",
  "morgan": "^1.9.0",
  "nodemailer": "^4.6.3",
  "paypal-rest-sdk": "^1.8.1",
  "readline": "^1.3.0",
  "snowball": "^0.3.1",
  "url": "^0.11.0"
}
```

Figure 10. The dependencies specified in package.json in the root of the server-side application

Node.js was also used on the server side to provide database connection using the Mongoose framework. Mongoose is akin to Express in the sense that all the required functionality could be programmed directly on top of the underlying Node MongoDB driver [42], but Mongoose implements all the required functions in a much more competent way than what could reasonably be expected by creating new implementations for the same purposes from scratch, in addition to being more reliable by virtue of having been tested

on a large scale by other previous users of the framework. As is the case with Express, Mongoose is an unopinionated framework, meaning that the provided functions are abstract enough to leave the application-specific implementation in the hands of the user by implementing only the functionality that would otherwise be repeated by the user in every project where Node.js and MongoDB are used, most likely in an inferior way.

In the service Mongoose was used to create one connection instance which handles thread pooling automatically. The connection instance was then used to create named mongoose references to the database collections called models [42]. Figure 13 below displays one such model, in this case of the collection containing YSE 1998 clauses.

```
const db = require('../db.js');
const pykalaSchema = db.Schema({
  nimi: String,
  numero: Number,
  momentit: [String],
  contents: String
}, { strict: false });
const pykala = db.model("pykala", pykalaSchema, "yse");
module.exports = pykala;
```

Figure 11. The Mongoose model object created towards the YSE 1998 collection

In this instance the Schema that is used directly mirrors the JSON documents of the YSE 1998 clauses in figure 8, but properties of the document do not need to be required or even listed in the schema to be retrieved or saved into the database through Mongoose [42]. The Mongoose schema acts as the blueprint for potential new document objects that are instances of the model and provides a more customizable virtual interface of the real documents, but its properties, also referred to as “virtuals”, are not persisted in the database in any way by creating a new schema [42]. Although the schema could be customized, in this case the documents are retrieved and used as they are. The variable “db” is an import of the mongoose database connection instance. After its creation, the object returned by the mongoose model() function can be queried by the user code. Multiple models using different schemas can be created of the same collection [42].

4.4.3 Use of Node.js on the client side

Although the primary purpose of Node.js is to be used as a server, it is also commonly used as a platform for constructing front-end applications. In the service another Node.js

project is nested within the root server project folder. The reason for this is mainly to provide separation in the overall project structure between client and server-related dependencies and scripts, to enable the use of separate Docker files, and to enable a separate temporary server to be run during development that only serves the client-side application without the API, which speeds up the development process as the two can use different tool sets and be debugged independently. This is especially helpful for front-end applications bootstrapped with Facebook's Create-React-App boilerplate, which provides a built-in development server that enables Hot Reloading [43]. Hot reloading is a way of refreshing only the file that was changed without losing the state of the overall app and having to rebuild it entirely as would be done in a production setting.

A nested Node app for the client also enables the two to be easily deployed onto 2 different servers in cases where it is desirable that the 2 applications will not both crash if one of them encounters a fatal error. Although separate deployment was not used in this service, it is a useful feature that could potentially be used in the future if the service scales to such a degree that separation becomes worthwhile.

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. Due to its reliability and widespread use Express has become the de facto standard server software for Node.js [45]. 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 [44], 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 [44].

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 [48], a Node.js-based startup company, which in turn was acquired by IBM in 2015 [49]. 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 [47].

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 [50]. Express is also used in production by several large companies such as Accenture, IBM, Fox Broadcasting and Uber [51].

5.2 Usage of Express.js in the service

5.2.1 Serving static content

Serving the front-end application was done using a built-in Express middleware function called `static()`, which generates a public route in `[domain root]/static` that allows HTTP access to the specified resources [52]. After the static files are made public, the `index.html` file along with its references to the generated static file routes is served upon all non-API requests that hit the server. The listing below displays the generation and serving of static content.

```
//serve static
if (process.env.NODE_ENV == 'production') {
  app.use(express.static('client/build'));
  app.get('*', (req, res) => {
    res.sendFile(path.resolve(__dirname, 'client', 'build', 'index.html'));
  });
}
```

Figure 12. The code block that serves static files

By using Express.js these functionalities that require thousands of lines of code are from the developer point of view condensed into a small block of 6 lines simply by making use of the Express.js built-in modules. The variable “app” in figure 14 refers to the overall Express instance. `Express.use`, `express.static`, `express.get`, `send.sendFile` and `path.resolve` are all large functions that involve much more complexity and conditionality

internally, but because a nearly universally applicable standard implementation is possible for these operations, the relevant visible part in the root server file can be heavily minimized.

The declared routes are matched against in the order of their declaration. Because the named routes are declared before the static file route generation, the static files can be served from every address that does not first match something else by using `app.get('*')`, where the star character matches all valid route names.

5.2.2 REST API routes

5.2.2.1 Content delivery routes

In the server-side application, basic content fetching routes are provided for all the different types of data available (YSE 1998, KSE 2013, JYSE 2014). These routes do not involve functionality besides querying the mongoose model mentioned in section 4.6.2 of the Node.js section. Figure 15 below displays two API routes defined on the server, in this case providing the YSE 1998 clauses.

```

//ROUTE 1 - ALL CLAUSES
router.post('/yse', verifyjwt, function(req, res, next) {
  var query = pykala.find({}, 'nimi numero momentit').sort({numero: 1});
  query.exec(function(err, docs){
    if (err) throw err;
    res.json(docs);
  });
});
//ROUTE 2 - SINGLE CLAUSE
router.post('/yse/numero/', verifyjwt, function(req, res, next) {
  var numero = parseInt(req.body.num);
  //console.log(numero);
  var query = pykala.find({
    numero: numero
  }, 'nimi numero momentit').limit(1);
  query.exec(function(err, docs){
    if (err) throw err;
    res.json(docs);
  });
});
});

```

Figure 13. The section containing the two API routes regarding YSE 1998 clauses

Both routes contain a quick query using `mongoose.find()` to “pykala”, which is the YSE 1998 mongoose model. `Mongoose.find()` receives a JSON-object with the defined search parameters, which in this case are simply either empty (route 1) to receive all documents, or a number parameter (route 2) to identify a specific one. The result of the query is returned as a json response, meaning it contains the `application/json` content-type header and correctly formatted JSON. The same route and query structure was repeated for all other forms of data that are intended for read-only requests. Because the construction agreement clauses are immutable, they require no additional routes for deletion, adding or editing. In this way they could be considered static files, but because they are sizeable and rarely all used during a single site visit, it is more sensible to serve them on request instead of including them in the `index.html`.

5.2.2.2 The search engine

One API route at `[Domain-root]/hakukone` is provided via Express for searching documents within the service. The route returns a JSON-array of documents including document-specific match data regarding which words were found and how many times

they occurred in the document. Figure 16 below displays a partial JSON response from the server after the user has queried the YSE 1998 collection using the search engine with the term “mitä”. One child node of the “results” array represents data regarding one YSE 1998 clause.

```
"results":[
  {
    "nimi":"13 §\nSopimusasiakirjojen keskinäinen pätevyysjärjestys",
    "numero":13,
    "matchData":{
      "terms":{
        "mitä":2
      },
      "phraseFound":false,
      "forbiddenFound":false,
      "termTotalMatches":2,
      "phraseTotalMatches":0,
      "totalMatches":2
    },
    "matches":[
      "mitä"
    ]
  },
  {
    "nimi":"23 §\nMenettelytapamääräyksiä",
    "numero":23,
    "matchData":{
      "terms":{
        "mitä":1
      },
      "phraseFound":false,
      "forbiddenFound":false,
      "termTotalMatches":1,
      "phraseTotalMatches":0,
      "totalMatches":1
    },
    "matches":[
      "mitä"
    ]
  },
  {
    "nimi":"23 §\nMenettelytapamääräyksiä",
    "numero":23,
    "matchData":{
      "terms":{
        "mitä":1
      },
      "phraseFound":false,
      "forbiddenFound":false,
      "termTotalMatches":1,
      "phraseTotalMatches":0,
      "totalMatches":1
    },
    "matches":[
      "mitä"
    ]
  },
  {
    "nimi":"23 §\nMenettelytapamääräyksiä",
    "numero":23,
    "matchData":{
      "terms":{
        "mitä":1
      },
      "phraseFound":false,
      "forbiddenFound":false,
      "termTotalMatches":1,
      "phraseTotalMatches":0,
      "totalMatches":1
    },
    "matches":[
      "mitä"
    ]
  }
],
```

Figure 14. A part of the JSON response sent from the server when querying the YSE 1998 collection using the search engine with the term “mitä”

The search engine uses locally cached documents for faster search speeds and more customizability. The documents are first retrieved from the database in their entirety and then saved into local arrays. In the function that creates locally searchable copies of the documents they are assigned an extra property called “contents”, which compiles all the contents of all properties into a single searchable lowercase string value, which can then be used by the search engine.

Originally the service was intended to use MongoDB’s internal text search engine as it is generally preferable to offload searching to the database instead of reserving RAM on

the server, but the MongoDB text search quickly turned out to be inadequate due to the lack of partial match word search as well as match occurrence counting. These features are currently being developed for MongoDB text search, but for this version of the service the search engine had to be implemented outside of the database. The final decision to implement a self-created search engine was informed by first testing Lunr.js and Fuse.js, which are both JavaScript text search services that ultimately ended up being suboptimal for a variety of reasons.

The functionalities of the search engine are multiple word search and exact phrase search, along with options for selecting the target collection and the ordering of the search results. Surprisingly, a very high level of performance was reached with a simple brute-force letter comparison algorithm, with the result being returned near instantly, even with multiple search terms. This solution ended up being much faster than JavaScript's RegExp-based options, which would also prove lackluster in their inability to count the occurrences in a string.

The search terms are first stemmed with Snowball.js, which is a multi-language word stemming library that shortens the given words to their "stem", which is the root word made more unspecific – for example, the word "tarkoituksenmukaisemmaksi" would be stemmed to "tarkoituksenmukais", which would cause the search engine to match for all words that start with the stem, for example "tarkoituksenmukaista", which broadens the results somewhat. Stemming is not used with the precise phrase search option.

5.2.2.3 Authentication

Several routes are dedicated to authenticating the users of the service. The authentication is done via a registered username-password combination, which allows the users to gain a JsonWebToken which in turn enables API access for a year.

Figure 17 below displays a full user document in the database with some details hidden for security reasons.

```
> {
  "_id": ObjectId("5b466b396e3e1700144b6550"),
  "devicelimit": 3,
  "email": "markusvk@metropolia.fi",
  "hash": "$2a$10$[REDACTED]I7PZ15z7vr9HQeu51tT2yn1yVKf[REDACTED]r1W0R1V6",
  "customer": Object {
    "id": "57833[REDACTED]"
  },
  "paymentMethod": Object {
    "token": "5ry[REDACTED]"
  },
  "subscription": Object {
    "id": "k7v[REDACTED]",
    "paymentMethodToken": "5r[REDACTED]",
    "status": "Pending",
    "billingPeriodEndDate": "2019-07-10",
    "billingPeriodStartDate": "2018-07-11"
  },
  "__v": 0
}
```

Figure 15. The document form in which user data is saved in the database

Upon registration a hash string like the one in figure 17 is generated via Bcrypt's one way hashing algorithm based on the user's password as a security measure to avoid saving passwords in plain text. By using a one-way hashing algorithm, the service doesn't need to handle the actual passwords at any other point than the HTTP request between client and server when registering and logging in, which in turn is protected via SSL encryption. This hash string is saved with the plain text e-mail address to connect the two during login. If the password matches against the saved hash, a new JsonWebToken is signed for a year using a hidden string code that is saved as a Node.js environment variable on the server. The received token is then saved onto the browser's local storage, which persists until being deleted by the user. In this way, the user does not need to be inconvenienced by having to log in every time they want to use the service. JWT is not the most secure method of authentication as the tokens can potentially be spoofed by a skilled attacker, but due to the lack of sensitive content within the service as well as the low-risk demographic of the users (persons who regularly deal with construction contracting), JWT is more than sufficient and can be signed for a very long period at once.

Multiple routes in the service are protected via JWT, meaning they do not give content to requests that do not have a valid token attached to the request body. This is done via Express router's middleware functionality, which is a way of making the request pass through separate functions that have access to the request and response objects, allowing different code blocks to be reused more efficiently. For example, in a scenario where 20 different HTTP routes all require a check for a valid JWT, instead of writing the JWT

validation logic into each route, the request can simply be deferred into a middleware function which in turn gives a response to the actual route that comes after.

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. React was originally created by a Facebook employee called Jordan Walke [53] and was first used inside Facebook in 2011 when it was used in the design of the Facebook app's newsfeed [54]. 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 [53]. 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 [55]. 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. 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 [56].

Figures 18 and 19 below from React's documentation [56] display the same component expressed first in JSX and then in normal React JavaScript.

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

ReactDOM.render(
  <Hello toWhat="World" />,
  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 17. 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>
<li>Child 9</li>
<li>Child 10</li>
</ul>
```

Figure 18. 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 19. 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 in-memory tree model representation of the real DOM. 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 [56].

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 [56] 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 [56]. 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" [59] demonstrates this difference between one-way and two-way data binding using an input field example.

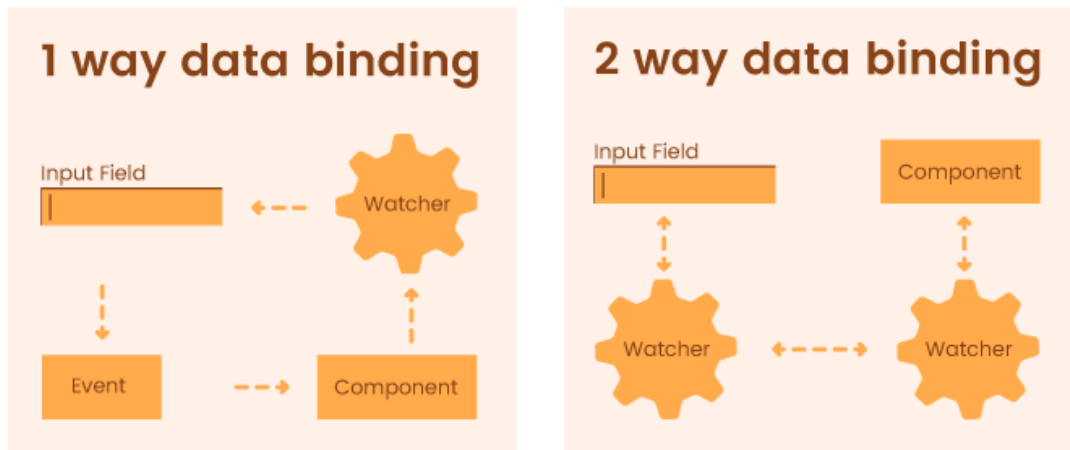


Figure 20. The difference between 1-way and 2-way data binding demonstrated using an input field [59]

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 top-level 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 [56]. 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 [56].

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 front-end 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) [58], 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 [57]. The graph below displays a composite summary of all the tests combined.

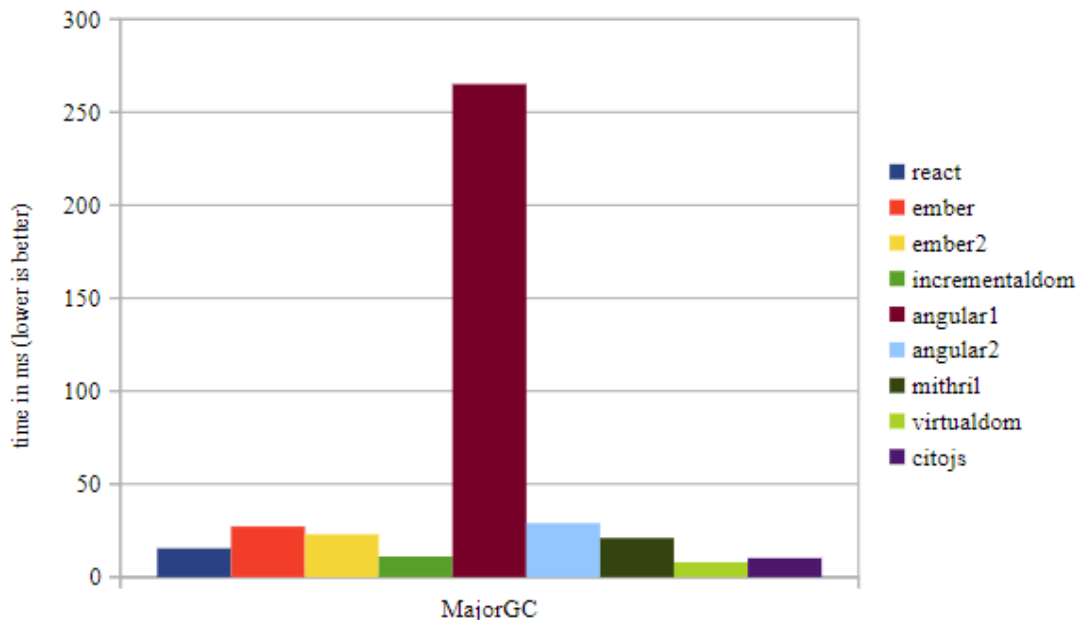


Figure 21. A composite of the Auth0 performance benchmark tests [57]

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 main components/views of the application are as follows:

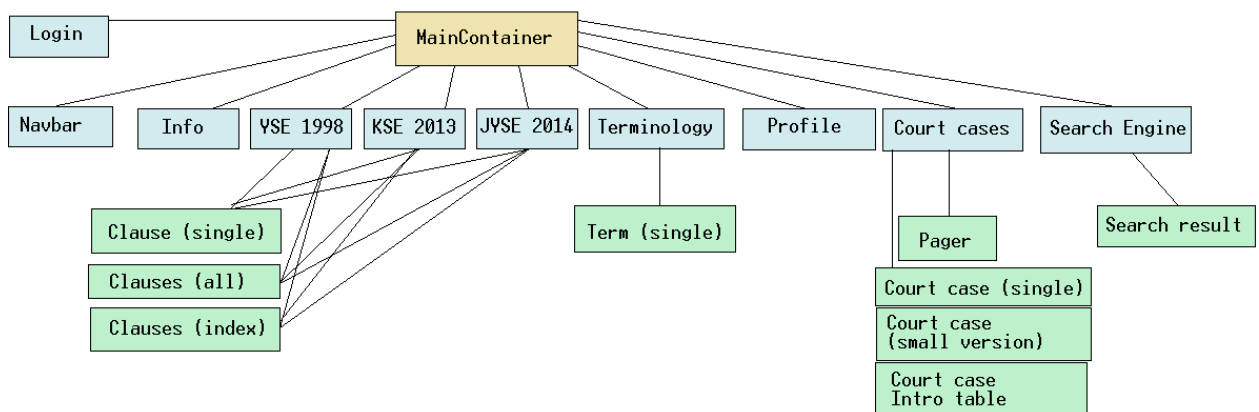


Figure 22. A tree diagram of the React components making up the front-end application

This diagram in figure 24 displays the views within the service, with the main components being colored light blue and the nested subcomponents in light green. 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 XXX 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 23. 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. Figure 26 below displays the index section of the YSE 1998 view which also includes the Navbar rendered in the main container.



Figure 24. A picture of the YSE 1998 section of the front-end application

Figure 27 below displays a section of the YSE 1998 document opened on a mobile device-sized screen.

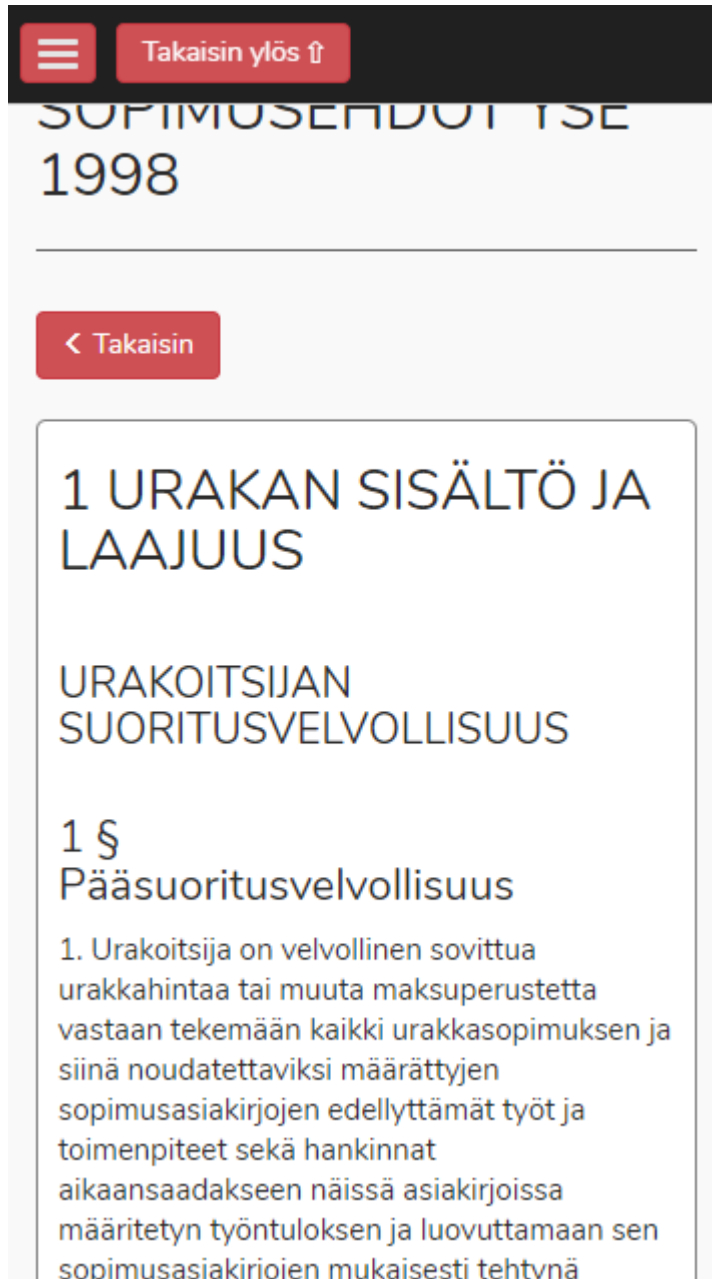


Figure 25. A picture of the opened YSE 1998 document in the front-end application

The front-end application is fully responsive and displays correctly on devices wider than 320 pixels. The customization based on screen size takes place on the main container's mount event and is subsequently recalculated upon JavaScript's window resize events.

7 Deployment of the service

The service was deployed onto an Amazon EC2 virtual linux instance using the Heroku cloud platform, which is a service that abstracts much of the standard server instance configuration away from the user. Heroku is a PaaS (Platform as a Service) as opposed to IaaS (Infrastructure as a Service), meaning it offers a fully configured instance using Heroku's preset settings in contrast to Amazon EC2, which offers a clean virtual instance with a fresh operating system without any configuration. Heroku is so heavily automated that it does not even allow for a direct SSH connection to the virtual instance, although this may change in the future due to the release of Heroku Exec Beta in 2017, which enables SSH-based terminal sessions through the Heroku CLI [60].

Because reliable infrastructure is difficult to set up correctly especially in regard to security concerns, a hosted platform tends to be the preferable option in cases where advanced customization is unnecessary. As far as the service in this project was concerned, the only criterion was safe and cheap availability via the internet, which made Heroku the optimal choice. In addition to providing a full-fledged environment with minimal set-up, Heroku also provides a developer dashboard with various features such as usage statistics, logging, environment variables, DNS settings and a marketplace for additional plugins that can be provisioned for projects.

The service runs on the Heroku platform in western Europe (in an unspecified location) as a basic node process. In addition to the default node scripts that are run upon deployment, Heroku allows additional scripts to be defined in a Procfile or in the project's root package.json. Because the API and client application are both served from the same Node/Express server, an extra command was included to run a build command from the create-react-app package, which transpiles and minifies the React application with all its JavaScript and CSS assets into a production-ready static folder, which in turn is easily used by Express' static asset route generating function from section 5.2.1. The size of the bundle of static assets upon initial page reload is ~250KB and the page loads DOM content in ~1,6 seconds on a relatively strong machine, which is reasonable performance. Figure 28 below displays the files that a user loads when loading up the page

with a fresh cache. Subsequent loads of the page become smaller depending on how much caching the client browser performs.

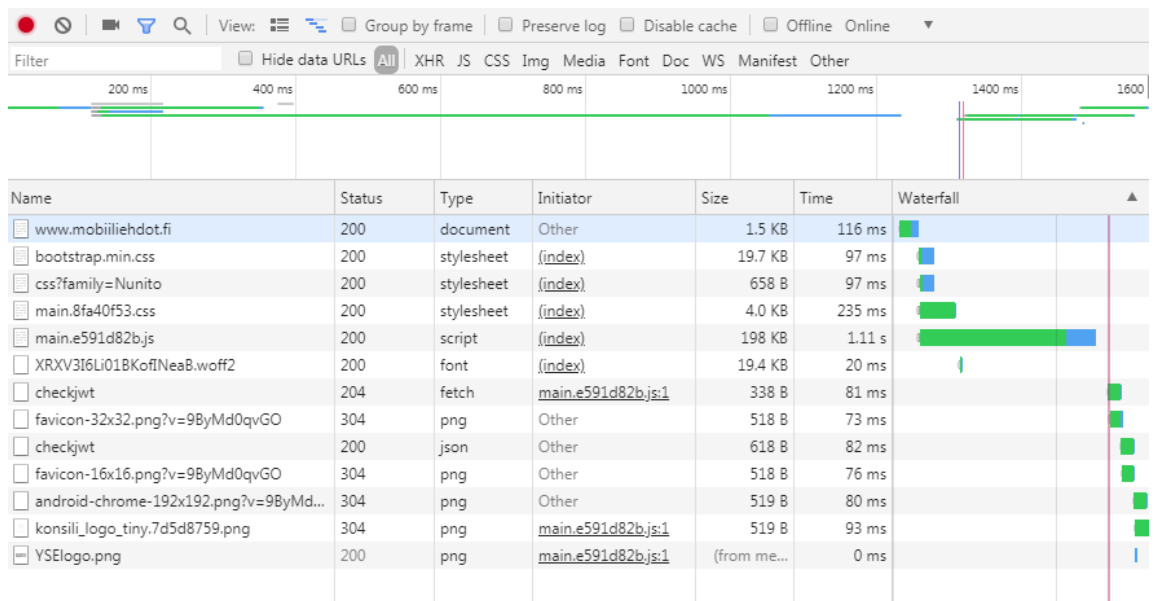


Figure 26. All the files transferred to the users upon initial load of the site, viewed from Chrome web development tools

8 Summary

The problem that the project explored in the thesis sought to resolve was the relative difficulty of accessing and finding relevant information in a set of documents associated with clauses in construction consulting, specifically YSE (Rakennusurakan Yleiset Sopimusedhdot) 1998, KSE (Konsulttitoiminnan yleiset sopimusedhdot) 2013 and JYSE (Julkisten hankintojen yleiset sopimusedhdot) 2014, which before the completion of the service were available only as semi-legal PDFs scattered around the internet or as blocks of text embedded in certain websites associated with construction consulting.

The thesis sought to solve these problems by creating a web service that stores the content in a database and fetches it when needed into a browser-based viewer application containing search functionalities that enable users to find those parts of the content that are relevant to their needs. The thesis also sought to study the core technologies used in the creation of the web application.

The process of creating the web service began with an analysis of the necessary core components that form the bulk of the application, namely MongoDB, Express.js, Node.js

and React.js - a combination of technologies is also known as the MERN stack. In addition to solving the problems laid out previously by creating a finished web service, the thesis also explored these 4 core technologies, including some of their internal mechanics and processes. Each section of the thesis where a core technology was explored included an analysis of the technology, details regarding why the technology was chosen as well as a breakdown of how the technology was used in the creation of the service. During analysis each part of the MERN stack was found to be appropriate on an individual basis for the requirements of the service. In addition to this, the MERN stack is highly congruent due to being able to use JavaScript as the programming language throughout all layers of the service, which also greatly aided the development process.

The final service called *Mobiliehdot* uses a MongoDB database hosted in Atlas, which is a cloud database service provided directly by MongoDB. The data ultimately placed in the database was persisted using a variety of custom programs that converted the data from a text file or an HTML node tree into JSON objects. The server architecture consists of a Node.js web server using the Express.js framework to serve both the client application and the API routes used by it. The client application was created using the React.js JavaScript framework and contains separate views for the different types of content involved as well as a search engine for finding parts where words or phrases are mentioned. The visual look of the client application was styled using bootstrap and regular CSS, and the project was deployed onto the Heroku cloud platform using the default Node.js buildpack.

The final product turned out to be successful in that it handily solves the problems initially laid out and offers a comfortable user experience. The implementation of the different solutions on a programming level produced issues during development due to lack of experience in all parts of the technology stack, but they were overcome through a process of learning and refactoring components several times. Future development is highly likely as more documents will likely be incorporated to the service, but due to the sturdy architecture of the service provided by the frameworks new types of content and client features can be easily added.

References

- 1 A relational model of data for large shared data banks, E. F. Codd, IBM Research Lab. Web document accessed 05.09.2018.
<https://dl.acm.org/citation.cfm?doi=362384.362685>
- 2 A Review of Different Database Types: Relational versus Non-Relational, Dataversity. Web document accessed 05.09.2018.
<http://www.dataversity.net/review-pros-cons-different-databases-relational-versus-non-relational>
- 3 Digital Data Storage is Undergoing Mind-Boggling Growth, Electronic Engineering Times. Web document accessed 05.09.2018.
https://www.eetimes.com/author.asp?section_id=36&doc_id=1330462
- 4 Modernizing Data Storage Archive Infrastructure, NEC. Web document accessed 05.09.2018.
<https://www.necam.com/Docs/?id=e93c05ba-ce09-47cb-aa54-68d440849be0>
- 5 MySQL documentation Chapter 11: Data Types. Web document accessed 05.09.2018.
<https://dev.mysql.com/doc/refman/8.0/en/data-types.html>
- 6 Scaling, Arcitura - WhatIsCloud. Web document accessed 05.09.2018.
http://whatiscloud.com/basic_concepts_and_terminology/scaling
- 7 “What Do We Mean by Database Scalability?”, Database Zone. Web document accessed 05.09.2018.
<https://dzone.com/articles/what-do-we-mean-by-database-scalability>
- 8 Mattson, Lith: Investigating storage solutions for large data, Chalmers University of Technology, 2010. Web document accessed 05.09.2018.
<http://publications.lib.chalmers.se/records/fulltext/123839.pdf>
- 9 SQL vs NoSQL, Computer Science presentation, Mohammed-Ali Khan, York University. Web document accessed 19.09.2018.
<http://www.cse.yorku.ca/~jarek/courses/6421/F12/presentations/NoSQLDatabases.pdf>
- 10 BASE analysis of NoSQL database, Deka Ganesh Chandra. Web document accessed 19.09.2018.
<https://www.sciencedirect.com/science/article/pii/S0167739X15001788>

- 11 MongoDB Drops ACID, MongoDB Inc. Web document accessed 19.09.2018.
<https://www.mongodb.com/blog/post/multi-document-transactions-in-mongodb>
- 12 MongoDB profile, Crunchbase. Web document accessed 19.09.2018.
<https://www.crunchbase.com/organization/mongodb-inc>
- 13 Database Engine Rankings, DB-Engines. Web document accessed 19.09.2018.
<https://db-engines.com/en/ranking>
- 14 “10gen embraces what it created, becomes MongoDB Inc”, Gigaom.com. Web document accessed 19.09.2018.
<https://gigaom.com/2013/08/27/10gen-embraces-what-it-created-becomes-mongodb-inc>
- 15 System Properties Comparison Couchbase vs MongoDB, DB-engines. Web document accessed 10.09.2018.
<https://db-engines.com/en/system/Couchbase%3BMongoDB>
- 16 MongoDB Architecture Guide, MongoDB Inc. Document downloaded 19.09.2018.
<https://www.mongodb.com/collateral/mongodb-architecture-guide>
- 17 MongoDB in Action, Kyle Banker, 2011.
- 18 Sharding, MongoDB Manual. Web document accessed 20.09.2018.
<https://docs.mongodb.com/manual/sharding>
- 19 Indexes, MongoDB Manual. Web document accessed 20.09.2018.
<https://docs.mongodb.com/manual/indexes>
- 20 Introducing NoSQL and MongoDB, InformIT. Web document accessed 20.09.2018.
<http://www.informit.com/articles/article.aspx?p=2247310&seqNum=5>
- 21 RDBMS vs NoSQL – performance and scaling comparison. Christoforos Hadjigeorgiou, The University of Edinburgh, 2013.
- 22 MongoDB Atlas FAQ, MongoDB Inc. Web document accessed 20.09.2018.
<https://www.mongodb.com/cloud/atlas/faq>
- 23 Cantelon, Harter, Holowaychuk, Rajlich. Node.js in Action, 2014

- 24 Node.js v8.12.0 Documentation. Web document accessed 20.09.2018.
<https://nodejs.org/dist/latest-v8.x/docs/api/index.html>
- 25 Npm introduction, NPM Inc. Web document accessed 26.09.2018.
<https://www.npmjs.com/>
- 26 Mikeal Rogers: Node.js Will Overtake Java Within a Year, TheNewStack, 2017. Web document accessed 26.09.2018.
<https://thenewstack.io/open-source-profile-mikeal-rogers-node-js>
- 27 Usage statistics and market share of Node.js for websites, W3Techs. Web document accessed 26.09.2018.
<https://w3techs.com/technologies/details/ws-nodejs/all/all>
- 28 Node.js introduction, Ryan Dahl, JSConf Berlin.
https://www.jsconf.eu/2009/video_nodejs_by_ryan_dahl.html
- 29 Original slides from Ryan Dahl's NodeJs intro talk, Aarti Parikh, SlideShare. Web document accessed 26.08.2018.
<https://www.slideshare.net/AartiParikh/original-slides-from-ryan-dahls-nodejs-intro-talk>
- 30 How are 10 global companies using node.js in production, To the New, web document accessed 26.09.2018.
<http://www.tothenew.com/blog/how-are-10-global-companies-using-node-js-in-production/>
- 31 The Node.js Event Loop, Timers and process.nextTick(). Node.js Inc, web document accessed 28.09.2018.
<https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick>
- 32 Node's Event Loop From the Inside Out by Sam Roberts, IBM.
<https://www.youtube.com/watch?v=P9csgxBgaZ8>
- 33 Libuv Design Overview, Libuv, web document accessed 26.09.2018.
<http://docs.libuv.org/en/v1.x/design.html>
- 34 Everything You Need to Know About Node.js Event Loop – Bert Belder, IBM.
<https://www.youtube.com/watch?v=PNa9OMajw9w>

- 35 NPM documentation. Web document accessed 10.10.2018.
<https://docs.npmjs.com>
- 36 Tiobe Index for October 2018. Web document accessed 10.10.2018.
<https://www.tiobe.com/tiobe-index/>
- 37 Node.js at PayPal, PayPal Engineering. Web document accessed 26.09.2018.
<https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/>
- 38 Http performance Java (Jersey) vs Go vs NodeJS, Ferdinand Mütsch. Web document accessed 26.09.2018.
<https://ferdinand-muetsch.de/http-performance-java-jersey-vs-go-vs-nodejs.html>
- 39 Status of Static Typing in ECMAScript, ECMAScript daily. Web document accessed 10.10.2018.
<https://ecmascript-daily.github.io/pages/status-of-static-typing-in-ecmascript>
- 40 How JavaScript works: inside the V8 engine + 5 tips on how to write optimized code, Alexander Zlatkov, SessionStack. Web document accessed 10.10.2018.
<https://blog.sessionstack.com/how-javascript-works-inside-the-v8-engine-5-tips-on-how-to-write-optimized-code-ac089e62b12e>
- 41 Understanding JIT compilation and optimizations, Oracle. Web document accessed 10.10.2018.
https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/underst_jit.html
- 42 MongooseJS documentation. Web document accessed 10.10.2018.
<https://mongoosejs.com/docs/>
- 43 Create-React-App Github documentation. Web document accessed 10.10.2018.
<https://github.com/facebook/create-react-app>
- 44 Express introduction, expressjs.com. Web document accessed 10.10.2018.
<https://expressjs.com>
- 45 Express.js in Action, Evan M. Hahn, 2016.
- 46 Interview with TJ Holowaychuk, My Open Source Journey. Web document accessed 10.10.2018.
<https://abdulhannanali.github.io/my-opensource-journey/interview-with-tj-holowaychuk/content>

- 47 Node.js Foundation to Add Express to its Incubator Program, Node.js Foundation. Web document accessed 10.10.2018.
<https://nodejs.org/en/blog/announcements/foundation-express-news>
- 48 TJ Holowaychuk Passes Sponsorship of Express to StrongLoop, StrongLoop. Web document accessed 10.10.2018.
<https://web.archive.org/web/20161011091052/https://strongloop.com/strong-blog/tj-holowaychuk-sponsorship-of-express>
- 49 IBM Acquires StrongLoop to Extend Enterprise Reach using IBM cloud, IBM. Web document accessed 10.10.2018.
<https://www-03.ibm.com/press/us/en/pressrelease/47577.wss>
- 50 Download statistic for package express, npm-stat. Web document accessed 10.10.2018.
<https://npm-stat.com/charts.html?package=express&from=2014-01-01&to=2018-12-08>
- 51 Companies using express in production, Express. Web document accessed 10.10.2018.
<https://expressjs.com/en/resources/companies-using-express.html>
- 52 Serving static files in Express, Express documentation. Web document accessed 10.10.2018.
<https://nodejs.org/en/blog/announcements/foundation-express-news>
- 53 The history of React.js on a timeline, RisingStack. Web document accessed 13.10.2018.
<https://blog.risingstack.com/the-history-of-react-js-on-a-timeline>
- 54 Pete Hunt, Texas JavaScript Conference 2015. Web document accessed 13.10.2018.
<https://www.youtube.com/watch?v=A0Kj49z6WdM>
- 55 Angular core vs angular vs react vs vue, npm trends. Web document accessed 13.10.2018.
<https://www.npmtrends.com/@angular/core-vs-angular-vs-react-vs-vue>
- 56 React.js documentation, React. Web document accessed 13.10.2018.
<https://reactjs.org/docs>
- 57 More Benchmarks: Virtual DOM vs Angular 1 & 2 vs Other, Auth0. Web document accessed 13.10.2018.
<https://auth0.com/blog/more-benchmarks-virtual-dom-vs-angular-12-vs-mithril-js-vs-the-rest>

- 58 Framework sizes minified, GitHub user Restuta. Web document accessed 13.10.2018.
<https://gist.github.com/Restuta/cda69e50a853aa64912d>
- 59 The Angular 2 vs React Contest Only Liveness Up, RubyGarage. Web document accessed 13.10.2018.
<https://rubygarage.org/blog/the-angular-2-vs-react-contest-only-liveness-up>
- 60 Introduction Exec (beta) – Connect to a dyno via SSH, Heroku. Web document accessed 13.10.2018.
<https://devcenter.heroku.com/changelog-items/1112>