California State University, Northridge

Performance Modeling of NodeJS and NoSQL Databases with AngularJS

A thesis submitted in partial fulfillment of the requirements
for the degree of Masters of Science
in Computer Science

by
Eric Ko Demauro

May, 2017

The thesis of Eric Demauro is approved by:

Professor George Wang	Date
Professor Li Liu	Date
Professor Adam Kaplan, Chair	Date

California State University, Northridge

Table of Contents

List of Figures

Abstract

Performance Modeling of NodeJS and NoSQL Databases
With AngularJS

By

Eric Ko Demauro

Master of Science in Computer Science

JavaScript as a server-side language coupled with the use of NoSQL databases has seen an emergence in recent years with the creation of Node.js and NoSQL databases, such as MongoDB. The purpose of this thesis is to act as a case study to determine the performance of Node.js as a web server along with a NoSQL database. Furthermore, it will seek to determine what benefits, if any, there are to using the AngularJS framework with Node.js to create a single-page application over a traditional web application. We herein describe the process of porting over an existing web-based MongoDB administration interface written in Node.js and Express to a complete MEAN stack implementation while following best practices, the design, execution, and results of benchmark tests. In particular, we investigate the ability of Node.js as a web server to deliver html content as opposed to JSON-formatted content. Furthermore, we evaluate the performance impact of AngularJS. Analysis of the results showed that Node.js performed better when returning JSON-formatted data than HTML. Further, the AngularJS implementation consumed far more memory and CPU resources than the pure Node.js implementation.

Introduction

To truly understand the methodology employed in the thesis, the context in which the study took place must first be discussed. PhpMyAdmin is a software tool written in PHP with the express purpose of administering MySQL and MariaDB databases over the internet [2]. It is included by default in the popular LAMP stack packages and its variations—WAMP, XAMPP, and MAMP [3]. PhpMyAdmin allows users to perform all common MySQL tasks, such as browsing and querying databases.

NoSQL databases have been gaining in popularity with companies, such as Facebook, utilizing them [5]. One of the most popular NoSQL databases is MongoDB. MongoDB is a schema-free, document-based database. For the purposes of this study, we decided it would be best to port an existing web-based MongoDB admin interface (Mongo-Express) [1] written in Node.js and Express to a complete MEAN stack implementation. This project was chosen due to the similarities to the analogous gold-standard for MySQL database management in the LAMP stack: PhpMyAdmin [2].

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine, allowing for execution of JavaScript code on the server side. Node.js first emerged in 2009 and was invented by Ryan Dahl [20]. Node.js brings an event-driven, non-blocking model to server-side processing [4]. Furthermore, since Node.js is event-driven, it is single-threaded, leading to a "Continuation-passing style" of code. Continuation-passing style is a style of programming in which results are passed into callback functions rather than simply being returned [39]. Since then, Node.js has been adopted by numerous companies, such as LinkedIn, eBay, and Microsoft [5]. Furthermore, it is one of the most starred repositories in Github.com, occupying the 8th position as of February 2016 [6]. Node.js has experience a tremendous amount of

community support in the form of libraries made available via the Node Package Manager—also known as NPM [26].

AngularJS is a client-side MVC JavaScript framework developed by Google, Inc. According to AngularJS's documentation, AngularJS is "what HTML would have been, had it been designed for applications [8]." AngularJS's main features include two-way data binding, custom directives, and dependency injection. AngularJS's two-way data-binding allows for synchronization of data between the view and model layers. This provides developers with a "single source of truth" for the application's state [21]. Custom directives are used in AngularJS for manipulating the Document Object Model. In other words, they are markers on DOM elements which attach special behavior. Dependency Injection is a software design pattern that handles how components obtain their dependencies; its purpose is to remove tightly-coupled code, and instead, to make components reusable. As of February 2016, AngularJS occupies the 4th spot in Github's list of most starred repositories and the 6th spot in Github's list of most forked repositories [6]. The list of companies using AngularJS includes companies such as Apple, Lyft, and PayPal [9].

With the advent of Web 2.0, applications required processing a large amount of data. It is often cited that at least 90% of all of the world's data has been generated in the past couple of years [40]. Applications developed by companies such as Facebook, Amazon, and SalesForce necessitated the ability to process "big data" and to provide data in real-time, which relational databases are unable to accomplish at scale [10]. NoSQL databases provide the high level of performance required by these

applications. Typically, they do so by forgoing ACID (atomicity, consistency, isolation, durability) constraints found in relational databases [14]. One of the motivations for moving from relational databases to NoSQL databases include the desire for simpler horizontal scaling—a noted pain point for relational databases. Horizontal scaling is the process of adding more machines to a pool of machines, whereas vertical scaling is the process of adding more resources to an individual machine [41]. To answer these scaling difficulties, companies have had to turn to NoSQL databases. For example, Facebook originally developed a NoSQL database now known as Apache Cassandra. It was created to solve Facebook's problem of handling Facebook's "inbox search problem [11]." Cassandra is capable of serving over 100 million users continuosly [15]. Facebook is not the only company embracing NoSQL databases; companies such as Craigslist, Verizon, and Adobe have adopted MongoDB [12]. Craigslist migrated over to MongoDB since they found the information they kept was not highly relational and would be better served in a NoSQL database [13].

Figure 1: NoSQL and RDBMS comparison

Figure 1 shows a comparison between how MySQL and MongoDB would store the same two entries. MongoDB is a NoSQL database in which the data

structure is not fixed. Data is stored in key-value pairs as documents similar to JSON [16]. Documents make up collections which are analogous to tables. Each database can have multiple collections and each collection can have multiple documents. Due to the MongoDB being schema-free, documents can evolve to have different attributes. In other words, a single document in a collection may only have 2 attributes, whereas another document in the same collection may have 20 attributes [17]. Mongo-Express is a web-based MongoDB admin GUI; it is analogous to phpMyAdmin, but for MongoDB instead of MySQL. Similar to phpMyAdmin, it provides a range of functionality for administering MongoDB databases, such as querying databases [1].

While the MongoDB installation package does not include a GUI interface, MongoDB does provide a list of third-party GUI interfaces, such as the aforementioned Mongo-Express [18]. Of the currently available third-party tools, adminMongo is the most similar to Mongo-Express. Both interfaces are implemented using NodeJS and ExpressJS. However, Mongo-Express supports more features and has over 600 more commits and 1000 more stars than adminMongo. Mongo-Express was first created back in April 2012, whereas adminMongo was initially created in January 2016, and is expected to undergo further revision and enhancement.

Goals and hypothesis

The goals of this thesis project is to measure the performance and viability of Node.js as a web server when coupled with a NoSQL database. Further, we will analyze how Node.js performs when working with HTML versus JSON data. We hope to accomplish this by creating a set of benchmark tests to simulate common interactions with Node.js and MongoDB. We will perform typical CRUD operations on the database under varying scenarios and varying workloads using Apache Benchmark. We have established testing criteria based on the relevant information provided by Apache Benchmark. We also seek to analyze the performance benefits, if any, of AngularJS.

We have formed the following hypotheses. First, we believe that Node.js will offer better performance when returning JSON-formatted data than when returning HTML data. Second, we also believe that using AngularJS and creating a single-page application will speed up the entire application. We believe this to be the case because of the nature of a single-page application. With a single-page application, the majority of the work has been off-loaded to the client-side with HTTP requests being predominantly AJAX requests. However, we also anticipate that the initial loading of the application will be noticeably slower due to the additional JavaScript assets.

Background

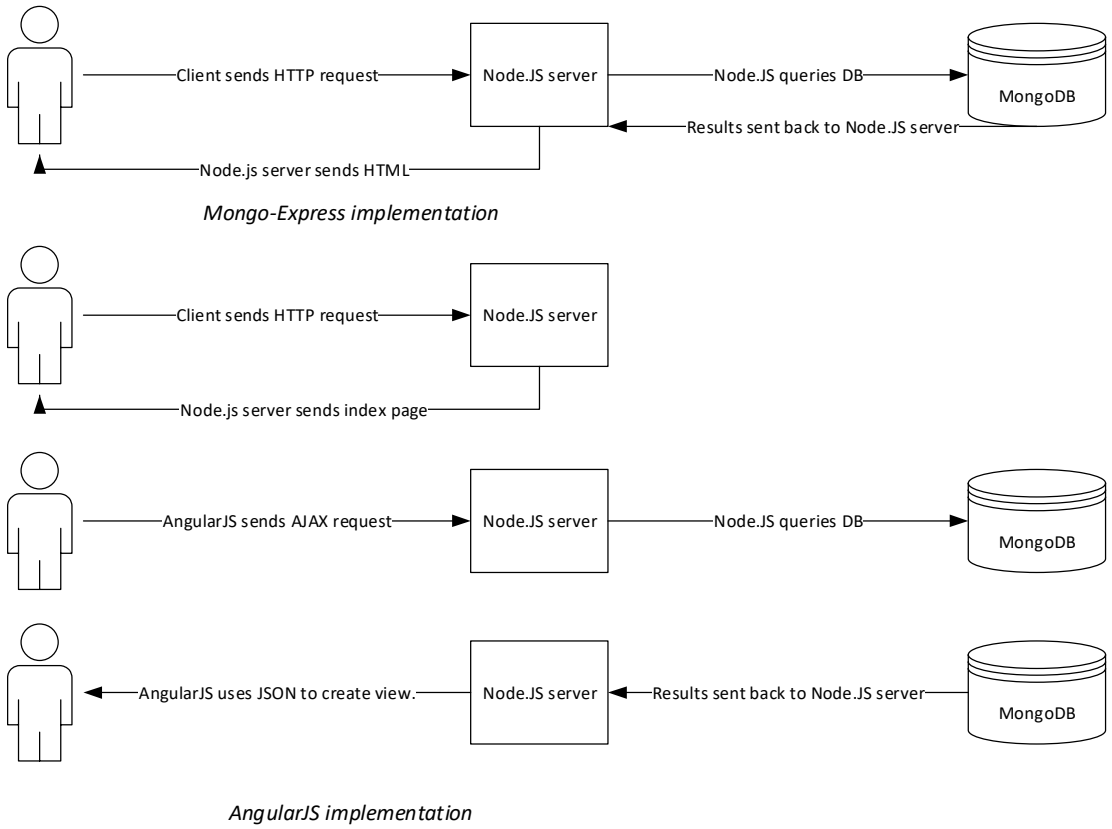

Figure 2: Mongo-Express and AngularJS implementations

The current implementation of the application (Mongo-Express) acts as a traditional web application, as shown by Figure 2. In the traditional web application, a client sends an HTTP request to the Node.js server which in turn contacts the MongoDB database. The results of the database query are then sent back to the Node.js web server and returned to the client as HTML. With the AngularJS implementation, the application transitions into a single-page application. Further, AngularJS differs from other popular JavaScript libraries, such as jQuery, in that is it an entire framework for building single-page applications. With AngularJS, a client will still send an initial HTTP request. However, the client only receives HTML in the initial request along with assets for

AngularJS. All of the future requests are sent via AJAX and all responses return JSON-formatted data instead of HTML.

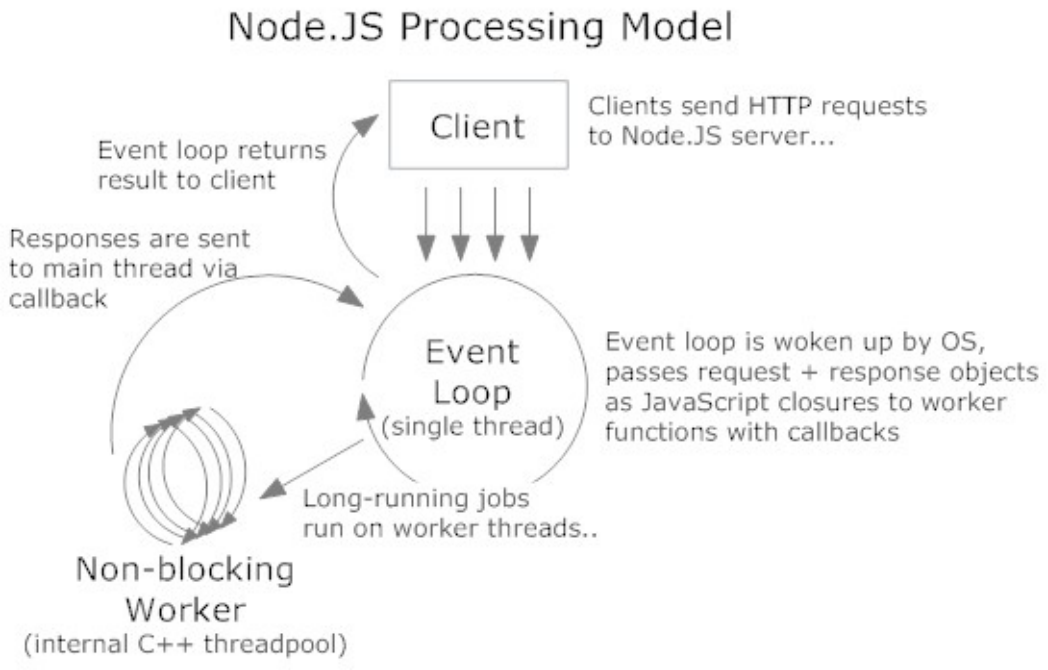

Figure 3: Node.js processing model

Figure 3 illustrates how Node.js processes requests. Since Node.js is built from Google's V8 JavaScript engine, it shares the single-threaded, event-driven core. Although, the management of the event loop is single-threaded, Node.js works with the libuv I/O library written in C++. The libuv performs the I/O by creating a pool of worker threads, allowing Node.js to continue processing requests [38]. A highly detailed explanation of Node.js's architecture is beyond the scope of this thesis. However, we will describe a high-level overview.

When a request is made, an event is placed along with a callback to be called once the operation is completed. If an I/O operation must be completed, Node.js spawns a worker thread using libuv, and once the operation is complete, the response is sent back to the end of the event loop. Node.js is known as being "run-to-completion," which

means that any request taken off the event queue will be not be interrupted and will finish executing before another event is taken from the event loop. For this reason, synchronous code is highly discouraged. Synchronous code can result in blocking of the event loop, and as such, Node.js employs a "continuous-passing style" using callbacks.

A highly detailed explanation of AngularJS's architecture is beyond the scope of this thesis. However, we will describe a conceptual overview of AngularJS. First we must begin with a discussion of the primary components of an AngularJS application. AngularJS applications chiefly consist of directives. A directive is a "behavior which should be triggered when specific HTML constructs are encountered during the compilation process [27]." AngularJS provides a service known as the compiler service which traverses the DOM in order to collect all directives, creates a scope object, combines the directives with the scope object, and produces a live view of the page. From this scope object, AngularJS provides a "single source of truth [21]." Since AngularJS is an MVC framework, AngularJS also provides controllers and services to facilitate the separation of concerns found in MVC frameworks. From these components, AngularJS is able to provide two-way data-binding and the usage of dependency injection, which is a software design pattern governing how components obtain dependencies.

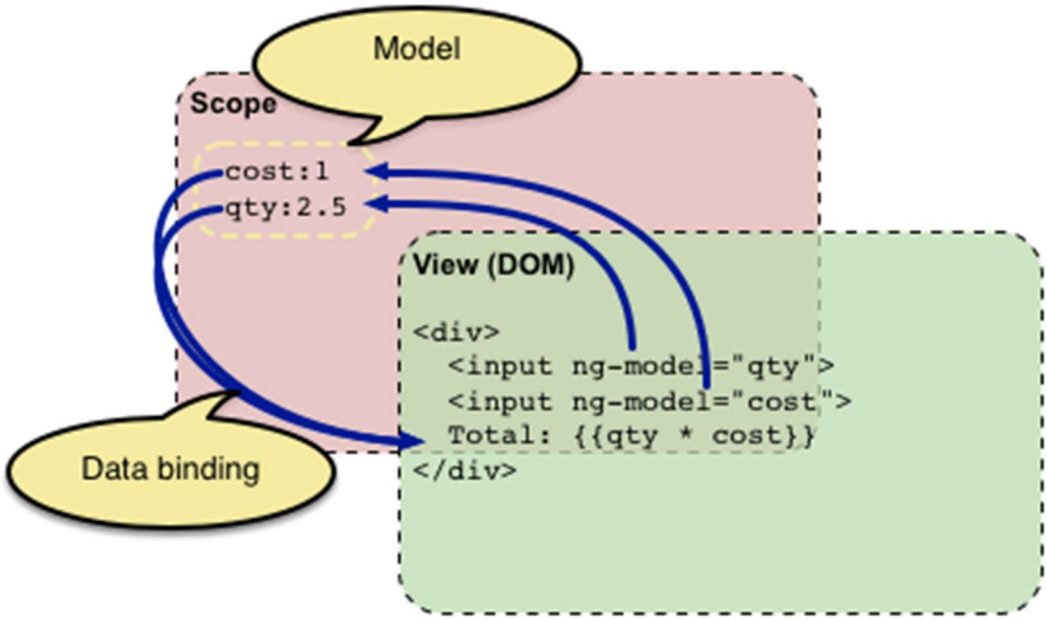

Figure 4: AngularJS data-binding example

In Figure 4, we are presented with HTML marked up with AngularJS directives—the ng-model attributes on the input elements and the content within the double-curly braces. When AngularJS begins, the compiler service runs, creating the scope object and the binding between the scope object and the view. In this example, two key-value pairs are created on the scope: cost and quantity. These variables are created from the ng-model attributes on the input elements and they are used to store and update the values from the input fields into the variable and vice versa. From this, we are provided two-way data-binding. In other words, altering the contents of either input field will result in the value of the respective variable changing and altering the value of the variable will result in the respective input field changing as well.

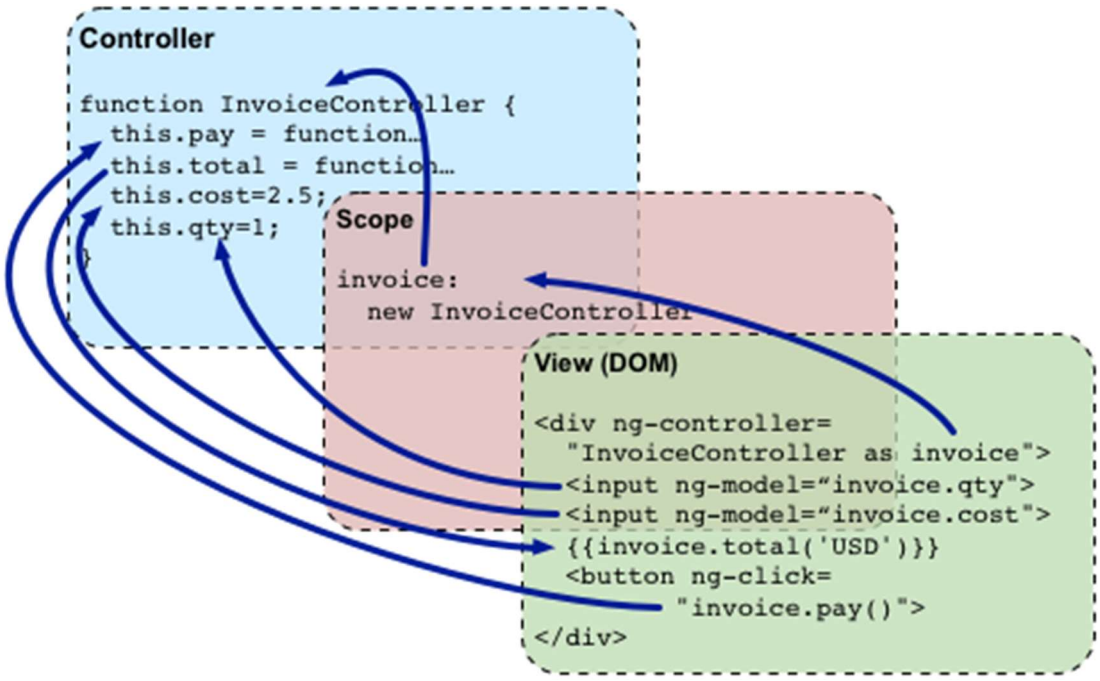

Figure 5: AngularJS controller example

In Figure 5, the same ng-model directives are present along with a new ng-click directive. The purpose of this example is to highlight the usage of a controller object as AngularJS is ultimately an MVC framework. The scope object is still present as well. However, the scope now contains an instance of the controller object which contains the variables from the previous example. Instead of storing the variables directly on the scope object, the variables are contained within the controller instance. AngularJS uses the ng-controller directive to determine which controller instance applies to that portion of the HTML and to provide the live two-way data-binding.

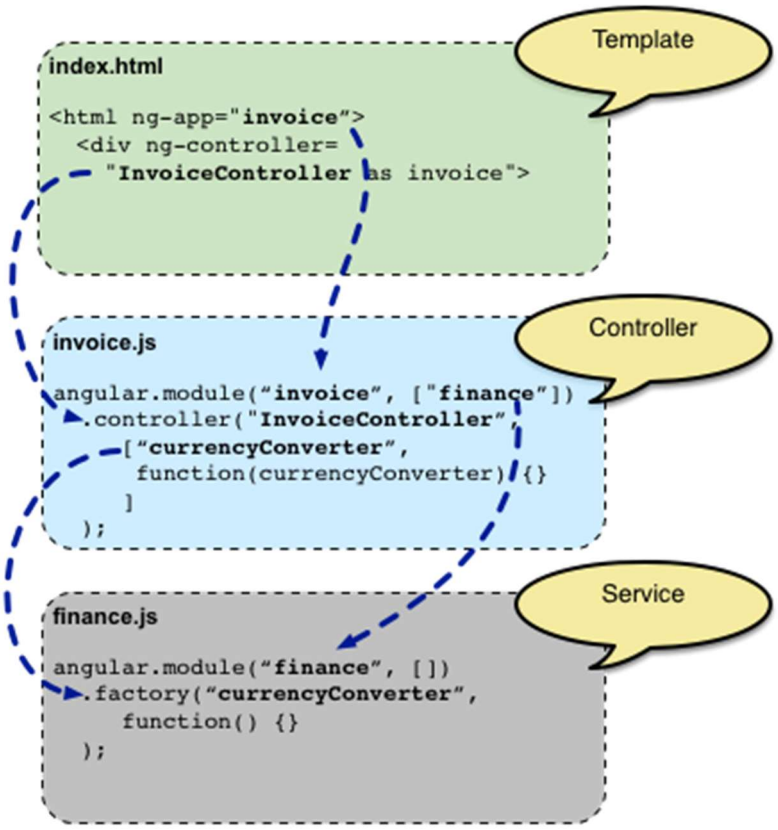

Figure 6: AngularJS service example

In Figure 6, we see a further separation of concerns along with dependency injection. In the controller, both the finance module and the currency converter are declared as a dependencies and will be injected into the invoice module once AngularJS is bootstraps. The finance module houses the currency converter service. The definitions of both the controller and the service has been omitted from Figure 6, however. Typically, all of the business logic will be placed in the service and injected into controllers that require them.

A discussion of the internal architecture of MongoDB is beyond the scope of this thesis, but we will provide a high-level overview of MongoDB. Among the multiple different categorizations of NoSQL databases, MongoDB is categorized as a document database. In this type of NoSQL database, documents are typically use a JSON-formatted

structure. MongoDB stores its data in binary JSON format. In MongoDB, each document is viewed as a single object. Documents contain one or more fields which could potentially store another object. Due to this nesting of objects, records are not spread across different columns and tables with associations. As a result, this simplifies data access without the need for complex operations. Furthermore, in MongoDB, schemas are dynamic. In other words, each document can contain different fields and the structure of a document is not set in stone. While most NoSQL databases do not provide any sort of support for ACID transactions, MongoDB provides limited support for ACID transactions. MongoDB provides ACID transactions only at the document level, but does not support them at any other level [28]. To accomplish this, MongoDB uses write-ahead logging to an on-disk journal.

Methodology

We devised benchmark tests using Apache Bench running against a Node.js server returning both HTML and JSON-formatted data. The Node.js server is hosted on a physical computer running CentOS 6.7 with an eight-core CPU and 16GB RAM. This machine is hosted by the Computer Science department at California State University, Northridge. Node.js, MongoDB, and Apache Benchmark are all installed on the aforementioned machine. The reason we installed all of the programs on the same machine were to remove latency as a factor in our benchmark tests.

We chose to use Apache Benchmark because previous studies in web benchmarking successfully utilized Apache Bench [23]. We arrived at our current concurrency and total request parameters by reaching the threshold at which Node.js would crash and dialing our parameters back to ensure stability. The Node.js server, when running on a single process, was incapable of consistently handling a more than one hundred concurrent requests. As such, we settled on running the benchmarks using one hundred concurrent requests when performing read operations. While Node.js running on a single process could serve one hundred concurrent requests performing read operations, it could not reliably serve more than twenty-five concurrent requests when performing write operations. Thus, for write operations, we settled on performing the tests with 25 concurrent requests and 5000 total requests. Read operations consisted of both reading a single document and reading an entire collection of documents from MongoDB. The collections that were read consisted of 1000 documents. Document read operations consisted of reading a document with two values: an id and a string. For both read benchmarks, we completed 5000 total requests with a concurrency level of 100

using a varying degree of processes, namely one, two, three, four, and eight processes. Write operations consisted of writing a single document with two key-value pairs to MongoDB. The write tests also consisted of five-thousand total requests for the same varying degree of processes. Each test for each number of processes was completed five times to remove any invalid data, e.g., the single-process read operation test was ran five times. Further, the average of these five test was taken. This proved useful as occasionally we would obtain some data that, if included, would skew the overall data set. All of the benchmark tests were ran against both the Mongo-Express and AngularJS implementations for consistency.

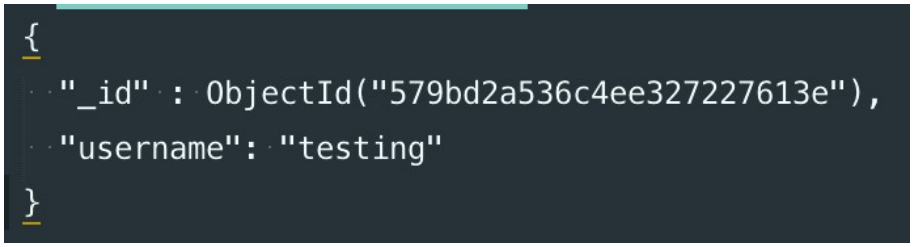

Figure 7: MongoDB document example

Figure 7 above illustrates what a single document in each of our tests consists of. As mentioned earlier, each document consisted of two key-value pairs with one pair holding a unique object id and the second pair holding a username field with a string value. For collection tests, the collection held 1000 documents with the structure as shown in Figure 7.

To handle the creation of multiple processes, we used a popular process manager for Node.js applications: PM2 [24]. We chose to use PM2 because of its popularity in the industry, with companies such as PayPal and Intuit using it with their production servers [37]. Furthermore, we found that it is readily available for installation using node package manager. PM2 made it simple to both create any number of processes and to

monitor the status of each process via the ‘pm2 monit’ command; it allowed us to monitor the memory usage and processor utilization for each process in real time.

In order to measure the impact of using AngularJS, we turned to Chrome DevTools to profile functions and for memory-usage analysis [25]. Chrome DevTools allows us to inspect the performance of our most-used functions and to determine how AngularJS performs during runtime. Further, it also allows us to see how network performance is impacted by using AngularJS. We determined that it would be best to have the Node.js server on a separate machine from the one running Chrome DevTools. The purpose of using a separate machine is to remove any inconsistences that may arise from having to share the resources on the same machine. Chrome version 54.0.2840.99 was used for testing since it was the most up-to-date version at the time. Furthermore, we ensured that all tests were ran in Incognito mode to remove any external variables.

As a reminder, the purpose of this methodology is to determine how Node.js performs when serving HTML data against how Node.js performs when serving JSON data. To summarize, the main thrust of this thesis is to compare and contrast Node.js returning both HTML data and JSON data on a single machine in hopes of identifying whether one data type exhibits a clear advantage in performance over the other on the same hardware. In other words, we investigate the strengths and weaknesses of Node.js with HTML and JSON data. In addition, we seek to explore the performance impacts of using AngularJS. We will discuss what benefits, if any, AngularJS provides and the performance impact of AngularJS. We have done this by following best practices in implementing the same application using AngularJS while minimizing the code differences between the two applications as much as possible.

AngularJS Implementation

For the process of porting the Mongo-Express application to a full MEAN-stack implementation with AngularJS, we decided to follow best practices and use the same tools found utilized by businesses. The following tools were used:

1. Gulp

Automated task runners are frequently used by businesses for application development. The two most popular task runners for JavaScript applications are Grunt and Gulp [29, 30]. Between these two, we chose to use Gulp mainly due to its popularity over Grunt. Despite being newer than Grunt, Gulp is more popular than Grunt. As of December 2016, Gulp has been starred 24,353 times on Github.com, whereas Grunt has been starred 11,185 times on Github.com. Furthermore, companies such as Netflix and Open Table use Gulp [31]. Gulp was used to lint all JavaScript files, enforce code style guidelines, compile ECMAScript 6 into ECMAScript 5, maintain and reload a web server, and to move files into their appropriate destinations.

2. Bower

For managing front-end assets, we chose to use Bower. Bower is a front-end package management application akin to NPM for Node.js. It was created by Twitter and originally released in 2012 as part of Twitter's open source effort [32]. Bower was used to maintain AngularJS along with all of the other AngularJS libraries used, such as Angular UI router.

3. Protractor

Protractor was used to run all end-to-end tests. Protractor is a testing framework created using Node.js specifically to test AngularJS applications [33]. Protractor uses Selenium

WebDriver for running a web server to automate tests against. Protractor was created by the AngularJS team to replace the now deprecated Angular Scenario Runner [34].

4. Karma

Karma was used run all non-end-to-end tests. Karma is a JavaScript test runner developed by Google [35]. Karma is a tool which spawns a web server and executes tests against the web server. Since Karma is a testing-framework agnostic, we chose to use Jasmine since that is the same syntax used in Protractor.

In order to minimize unintended effects, the porting process retained as much of the existing code as possible. The majority of the changes to the existing code resulting in returning JSON-formatted data to the client rather than the HTML data returned in Mongo-Express. Furthermore, the index page was altered to all of the AngularJS assets in addition to the original ones and an area to bootstrap the AngularJS application. All routing was shifted over to AngularJS using the Angular UI Router library, the preferred Angular routing library [36]. In addition, all of the Mongo-Express routes were converted to API routes to reflect this change. To ensure that the ported application worked as intended, an entire suite of tests was written using Protractor, Jasmine, and Karma, testing all of the features and expected behavior of the application.

Results

The following benchmarks consist of Node.js returning HTML data to the client. The benchmarks below were obtained using Apache Benchmark version 2.3 and Node.js version 5.6.0 running on CentOS version 6.7.

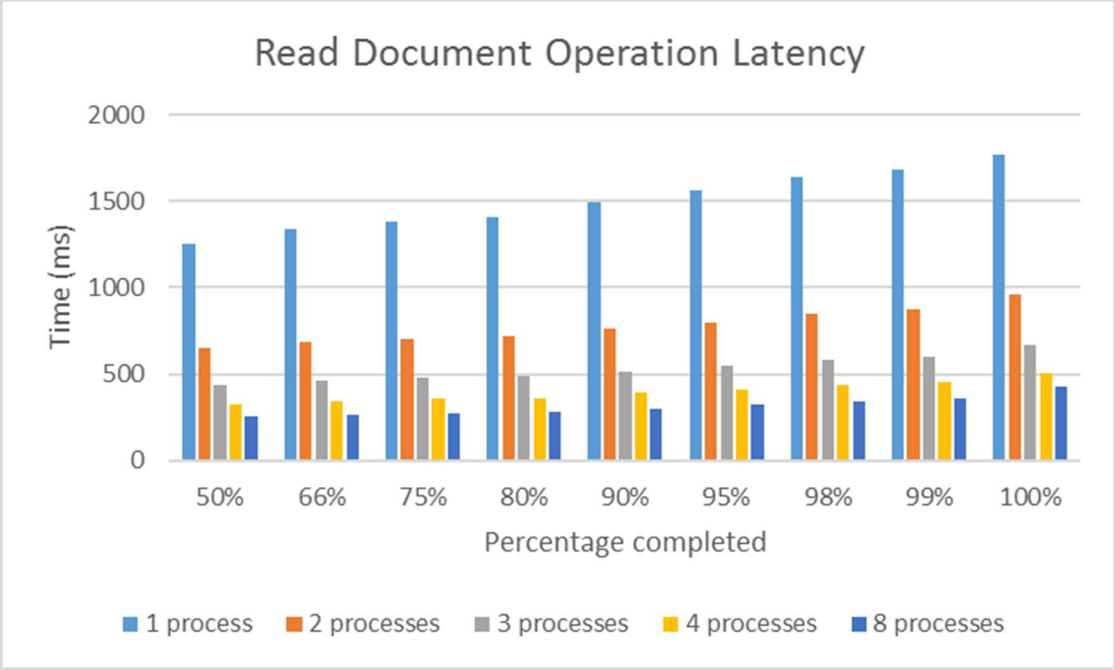

Figure 8: Latency of a document read operation returning HTML

In Figure 8, we test the latency of Node.js when performing a read operation on a single document. As stated earlier, these tests consisted of 5000 total requests with a concurrency level of 100. The read operation performed was a retrieval of a single document with two attributes, an id and a string, and returning HTML to the client. According to Apache Benchmark, the total document length of the response was 7312 bytes. The total percentage of requests completed for a varying number of processes is displayed along the x-axis, and the total amount of time taken in milliseconds is displayed along the y-axis. From this, we see that increasing the number of processes decreased the total amount of latency, with the greatest increase when going from a

single process to two processes. Based on our hypothesis, we expect the AngularJS implementation to run just as fast if not faster.

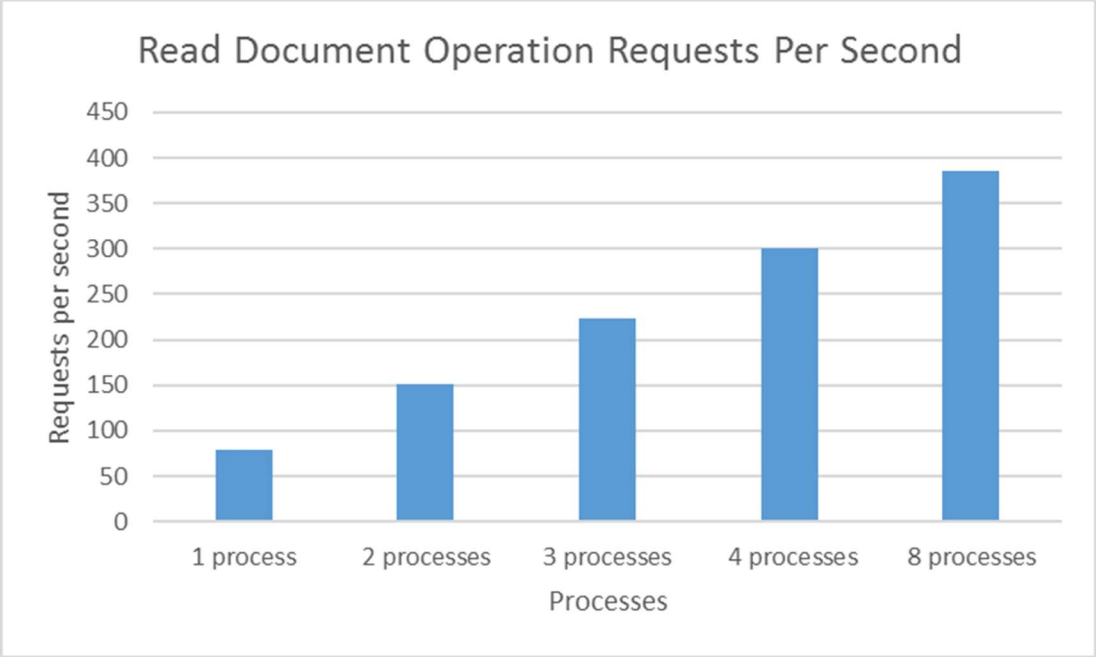

Figure 9: Requests per second of a document read operation returning HTML

Figure 9 reveals the number of requests that completed per second for different number of processes when reading the same single document from the MongoDB database and returning HTML to the client. These tests consisted of 5000 total requests with a concurrency level of 100. The number of Node.js processes is displayed on the x-axis, and the number of requests completed per second is shown on the y-axis. We see that increasing the number of processes increased the number of requests served with diminishing returns. Once again, we expect to see AngularJS perform better at every number of processes. Both tests above were run using a concurrency level of 100 with 5000 total requests.

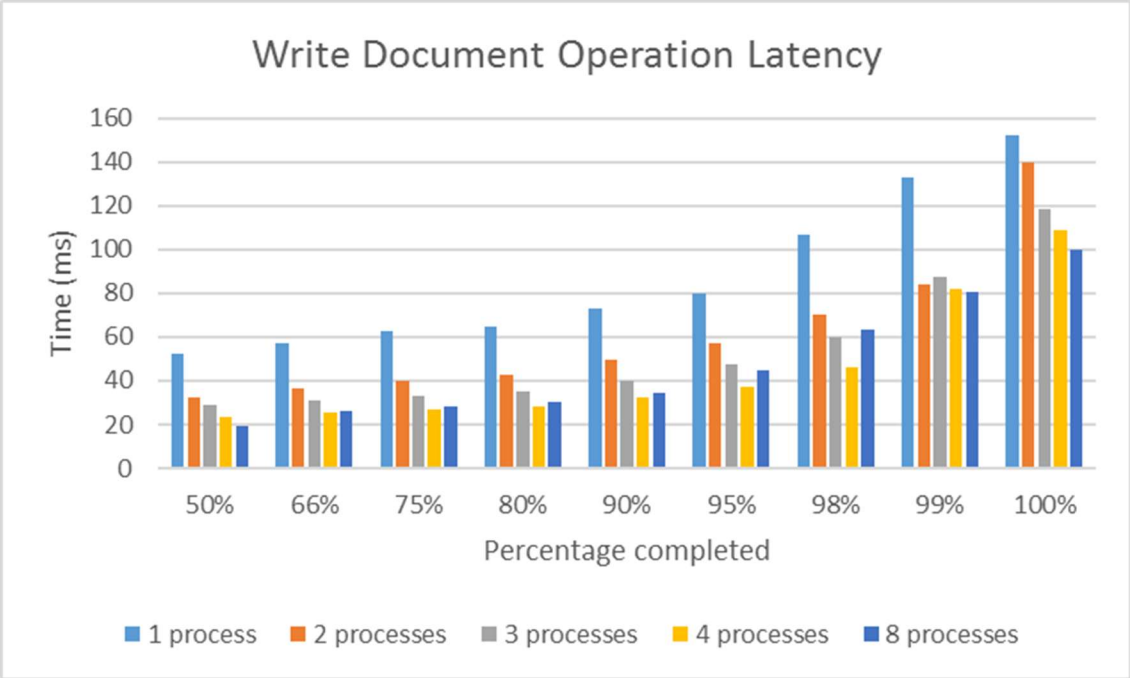

Figure 10: Latency of a write operation returning HTML

In Figure 10, we test the latency of Node.js when performing a write operation. The write operation performed was an insertion of a single document with two attributes, a unique id and a string, into the MongoDB database and returning HTML to the client. These tests consisted of 5000 total requests with a concurrency level of 25. According to Apache Benchmark, the total document length of the response was 35 bytes. The total percentage of requests completed for a varying number of processes is displayed along the x-axis, and the total amount of time taken in milliseconds is displayed along the y-axis. From this, we see that increasing the number of processes decreased the total amount of latency. Based on our hypothesis that using AngularJS will be faster, we expect to see AngularJS benchmarks to perform better at every number of processes.

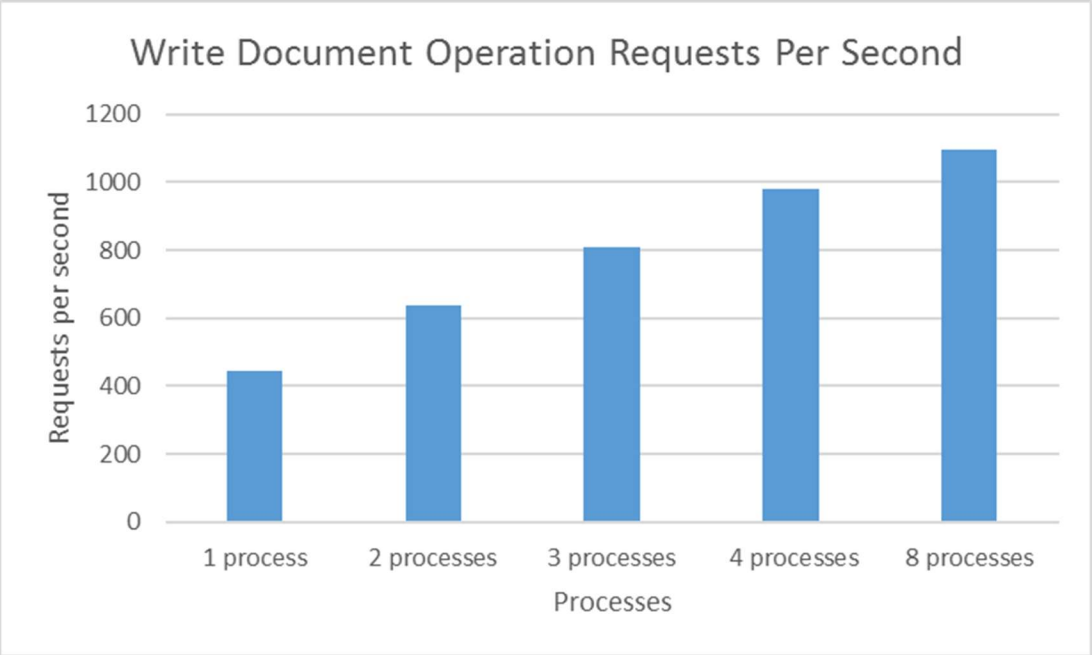

Figure 11: Requests per second of a write operation returning HTML

Figure 11 reveals the number of requests that completed per second for different number of processes when writing a single document to the MongoDB database and returning HTML. The number of Node.js processes is displayed on the x-axis, and the number of requests completed per second is shown on the y-axis. We see that increasing the number of processes increased the number of requests served with diminishing returns. Once again, we expect to see AngularJS perform better at every number of processes. The test above was run using a concurrency level of 25 with 5000 total requests.

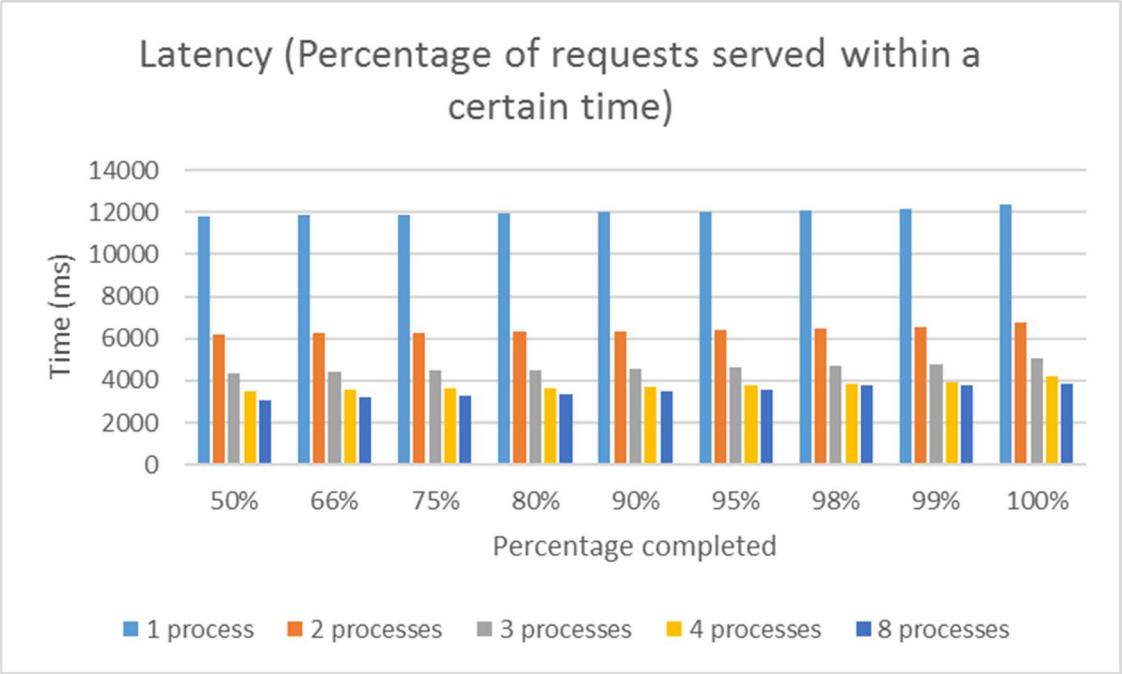

Figure 12: Latency of a collection read operation returning HTML

Figure 12 shows results from benchmarking read operations consisting of fetching a collection of 1000 documents and returning HTML to the client. This test was run using a concurrency level of 100 with 5000 total requests. According to Apache Benchmark, the total document length of the response was 783,799 bytes. The x-axis displays the percentage of requests completed by each varying number of processes, and the y-axis shows the time taken to complete the requests in milliseconds. Once again, we see that increasing the number of processes yielded noticeable performance improvements, with the greatest improvement going from one process to two processes.

Figure 13: Requests per second of a collection read operation returning HTML

Figure 13 shows the number of requests completed per second for each of the varying number of processes when fetching a collection of one thousand documents and returning HTML to the client. Along the x-axis, the number of processes is displayed, and along the y-axis the number of requests completed per second is displayed. We see that the greatest improvement comes from increasing one process to two processes with diminishing returns when adding even more processes. Ultimately, we see that write operations are significantly faster than read operations. This test was run using a concurrency level of 100 with 5000 total requests.

First, we discuss the results of the benchmark tests using the AngularJS implementation individually. Afterwards, we compare and contrast these results with the results from Mongo-Express benchmark tests. We will follow the same format as before by discussing the document read operation first, followed by the document write operation, and finally, the collection read operation.

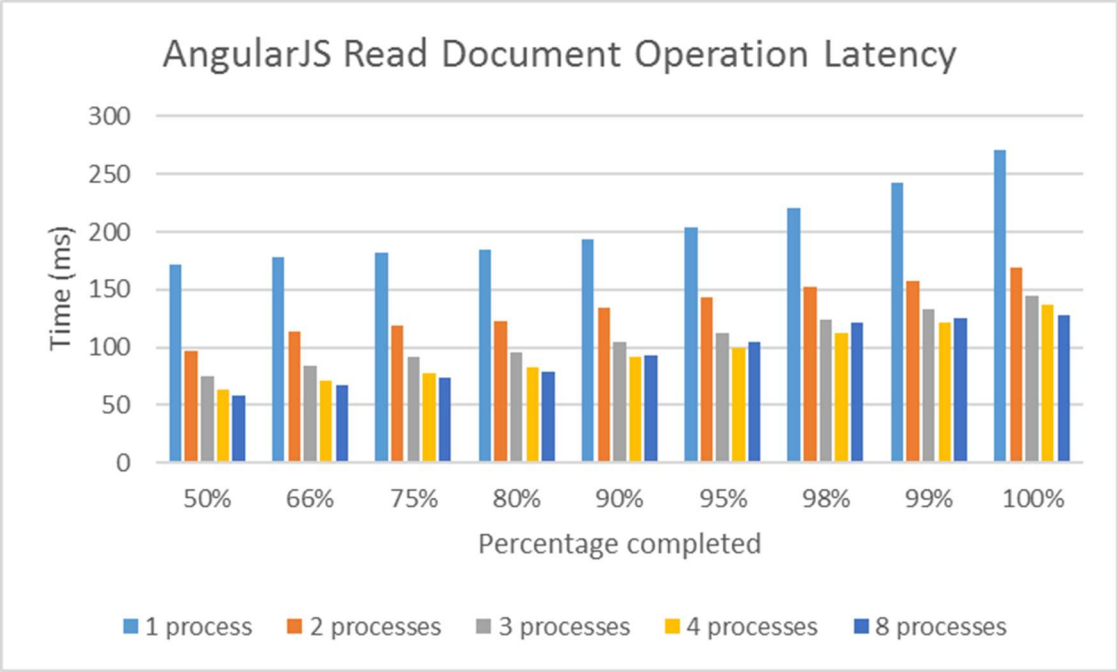

Figure 14: Latency of a document read operation returning JSON

Figure 14 shows results from benchmarking read operations consisting of fetching the same single document as the previous document read tests and returning JSON data to the client. Similar to the previous tests, this test was run using a concurrency level of 100 with 5000 total requests. According to Apache Benchmark, the total document length of the response was 452 bytes. The x-axis displays the percentage of requests completed by each varying number of processes, and the y-axis shows the time taken to complete the requests in milliseconds. As expected, we see that increasing the number of processes yielded noticeable performance improvements, with the greatest improvement going from one process to two processes.

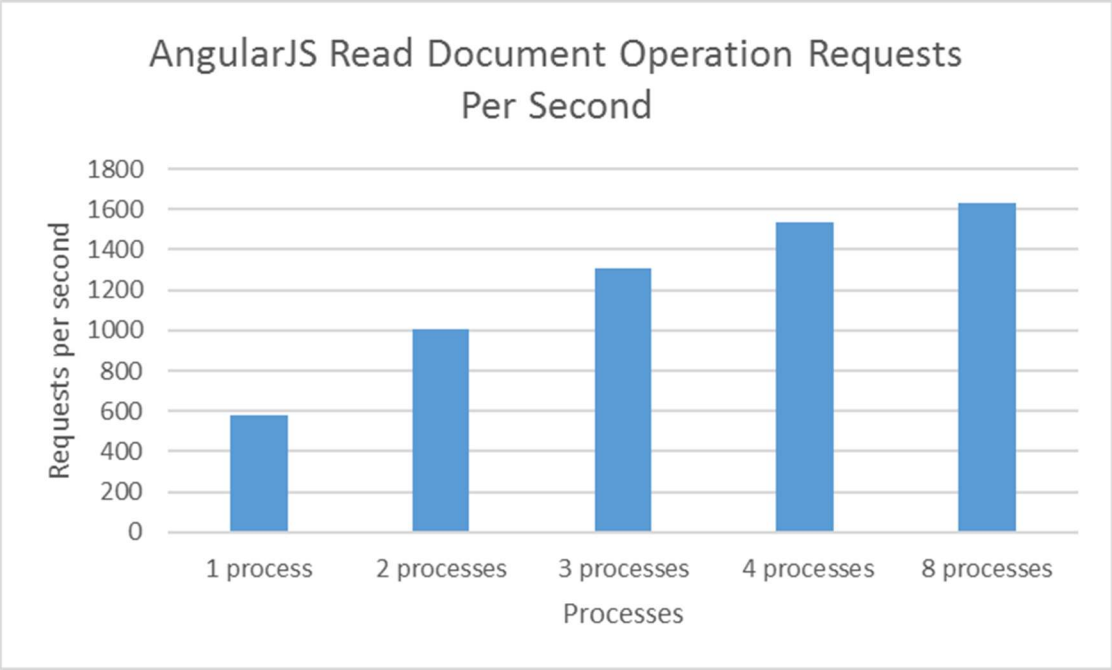

Figure 15: Requests per second of a document read operation returning JSON

Figure 15 shows the number of requests completed per second for each of the varying number of processes when fetching the same single document as the previous document read tests and returning JSON data to the client. This test was run using a concurrency level of 100 with 5000 total requests. Along the x-axis, the number of processes is displayed, and along the y-axis the number of requests completed per second is displayed. We see that the greatest improvement comes from increasing one process to two processes with diminishing returns when adding even more processes.

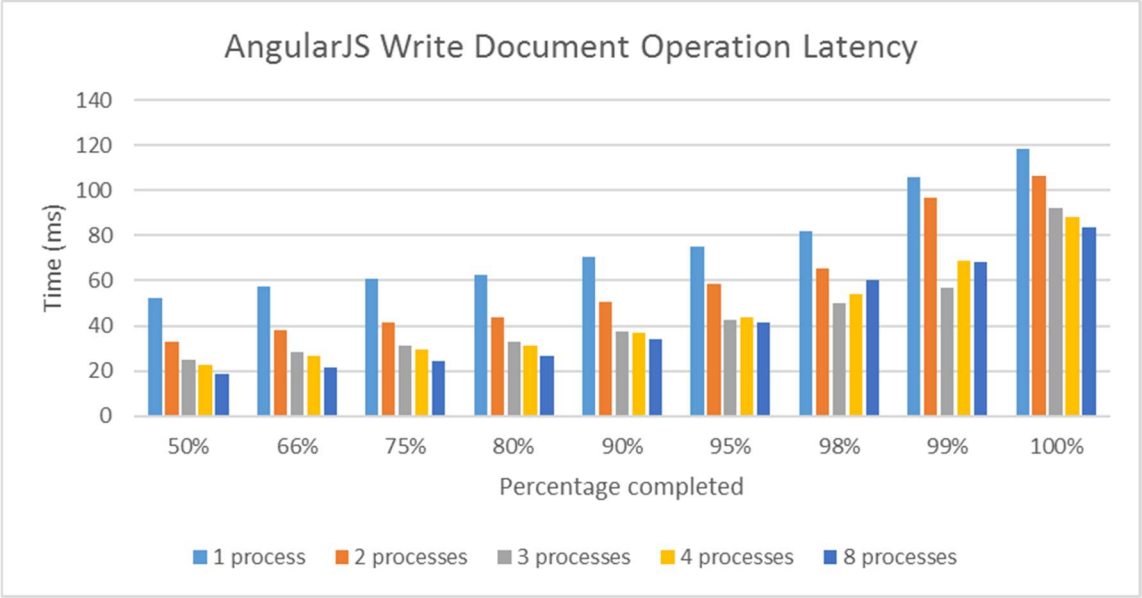

Figure 16: Latency of a write operation returning JSON

Figure 16 shows results from benchmarking write operations consisting of writing a single document as the previous write tests and returning JSON data to the client. This test was run using a concurrency level of 25 with 5000 total requests. According to Apache Benchmark, the total document length of the response was 29 bytes. The x-axis displays the percentage of requests completed by each varying number of processes, and the y-axis shows the time taken to complete the requests in milliseconds. As expected, we see that increasing the number of processes yielded noticeable performance improvements.

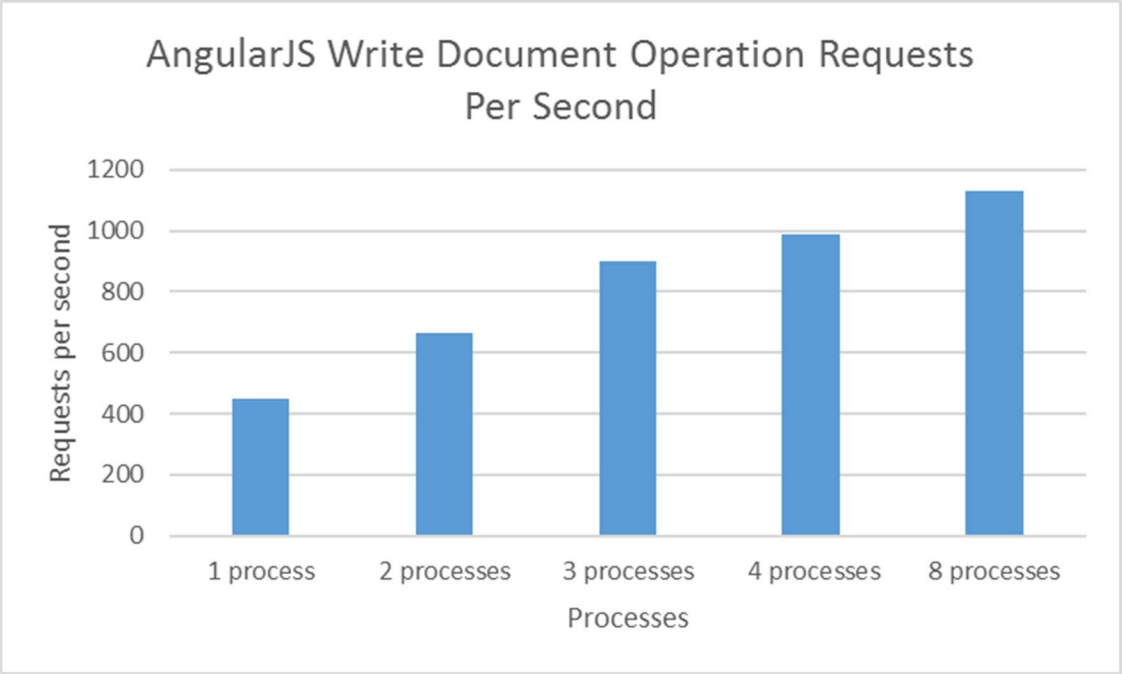

Figure 17: Requests per second of a write operation returning JSON

Figure 17 shows the number of requests completed per second for each of the varying number of processes when writing the same single document as the previous write tests and returning JSON data to the client. This test was run using a concurrency level of 25 with 5000 total requests. Along the x-axis, the number of processes is displayed, and along the y-axis the number of requests completed per second is displayed.

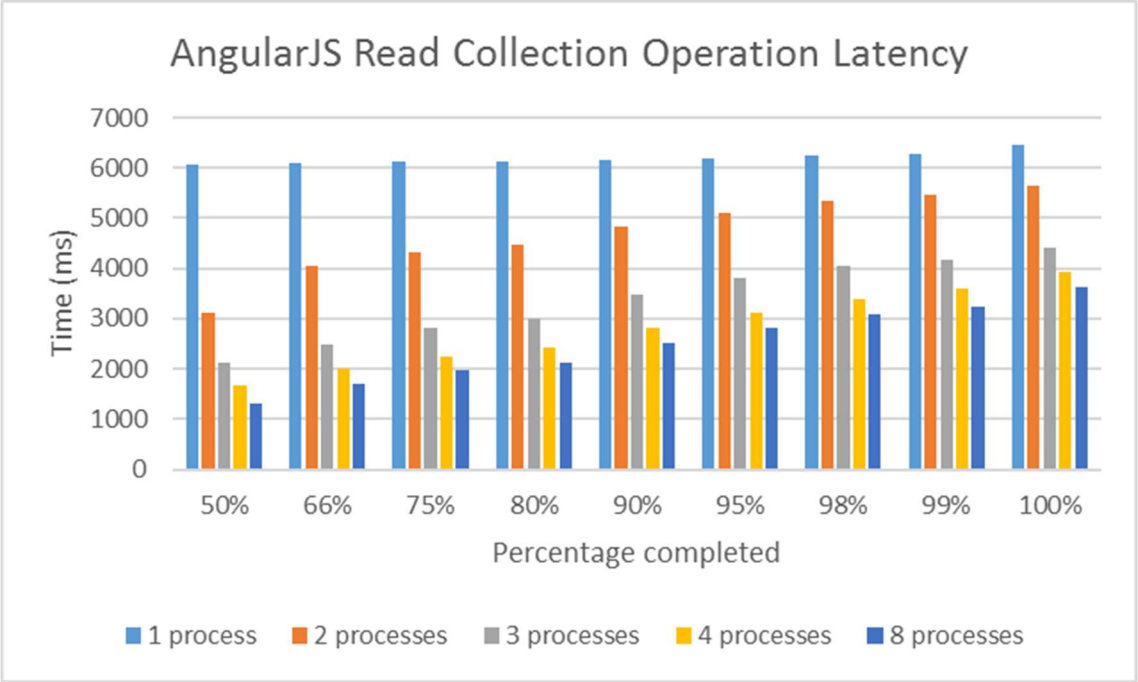

Figure 18: Latency of a collection read operation returning JSON

Figure 18 shows results from benchmarking read operations consisting of fetching the same collection of documents as the previous document read tests and returning JSON data to the client. This test was run using a concurrency level of 100 with 5000 total requests. According to Apache Benchmark, the total document length of the response was 157434 bytes. The x-axis displays the percentage of requests completed by each varying number of processes, and the y-axis shows the time taken to complete the requests in milliseconds. As expected, we see that increasing the number of processes yielded noticeable performance improvements. However, it appears that the greatest improvement was found when going from two processes to three processes.

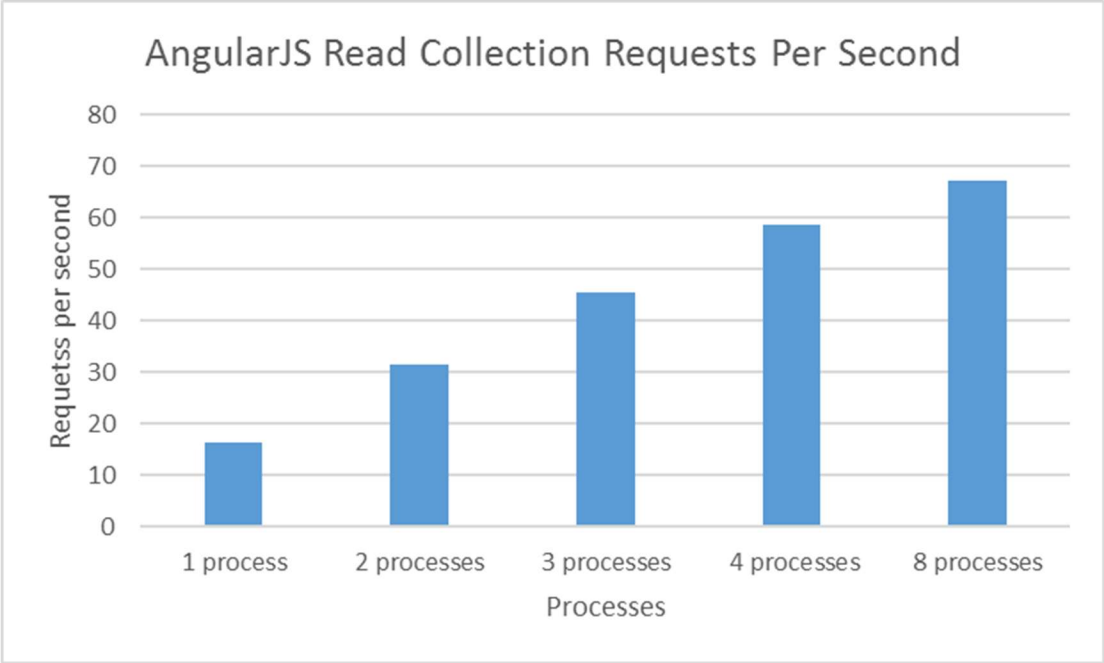

Figure 19: Requests per second of a collection read operation returning JSON

Figure 19 shows the number of requests completed per second for each of the varying number of processes when fetching the same collection of documents as the previous document read tests and returning JSON data to the client. This test was run using a concurrency level of 100 with 5000 total requests. Along the x-axis, the number of processes is displayed, and along the y-axis the number of requests completed per second is displayed. We see that the greatest improvement comes from increasing one process to two processes, whereas the greatest reduction in latency comes from increasing two processes to three processes.

Comparison Results

In this section, we are going to compare and contrast the results of the Mongo-Express benchmark tests with the results of the AngularJS benchmark tests.

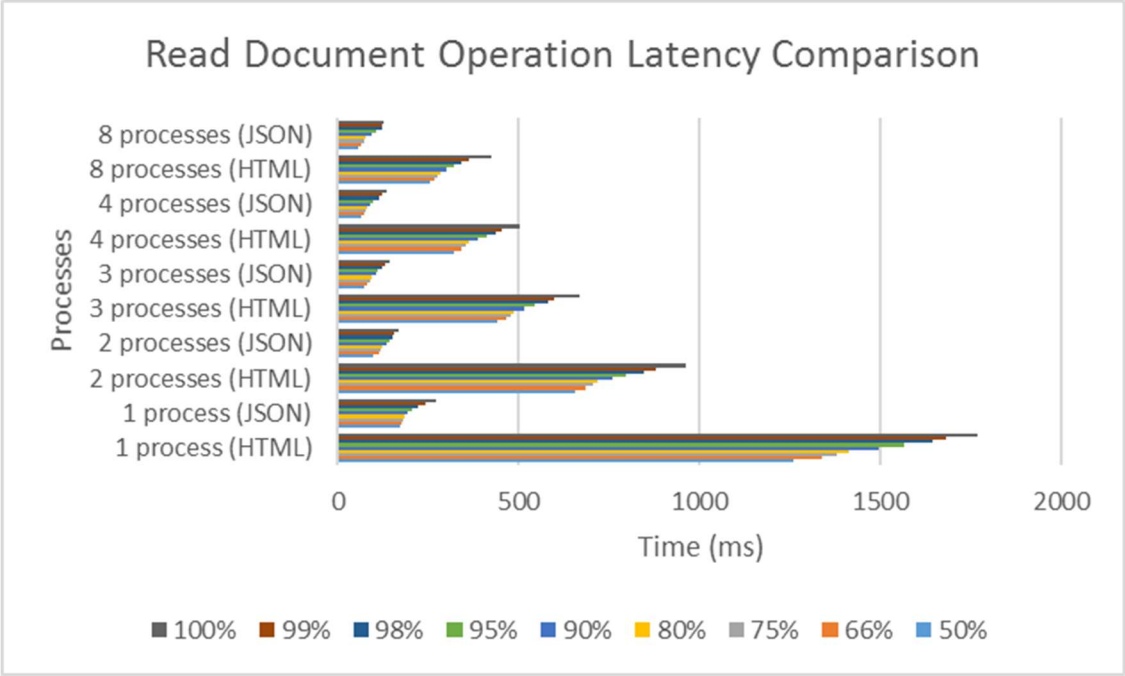

Figure 20: Latency comparison of a document read operation

Figure 20 highlights the differences between Node.js serving HTML content and Node.js serving JSON content when performing a document read operation. This test was run using a concurrency level of 100 with 5000 total requests. We see that for every process level, Node.js performs significantly better when returning JSON data over HTML data.

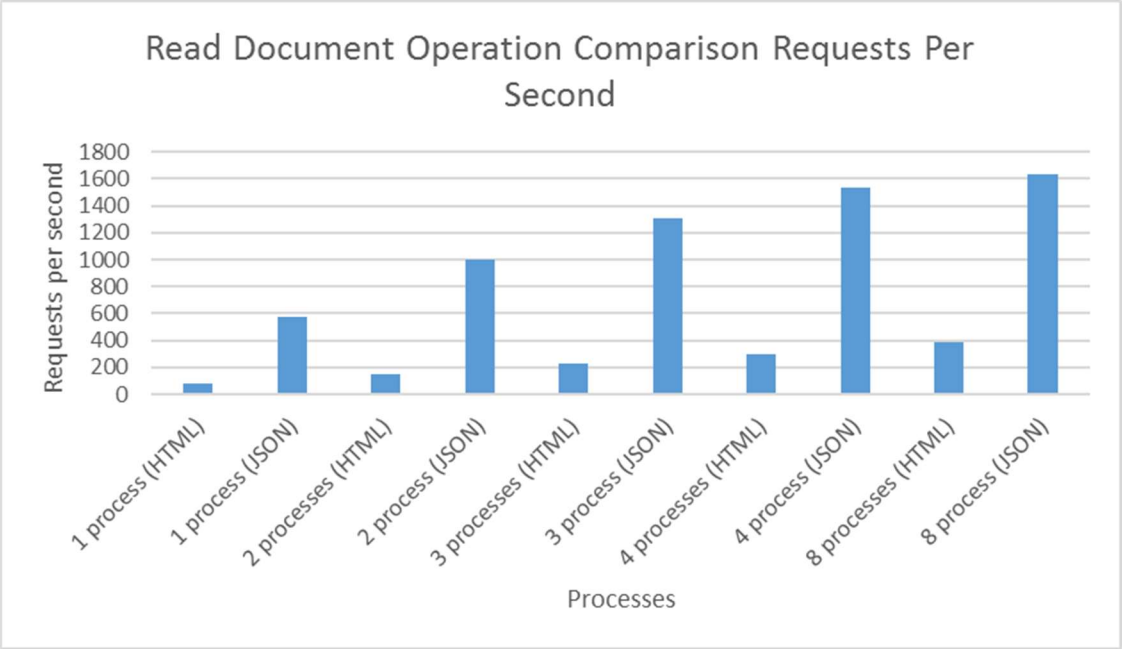

Figure 21: Requests per second comparison of a document read operation

From Figure 21 we can see that when Node.js reads a single document from the MongoDB database, Node.js has higher throughput when the returned data is formatted in JSON than when it is formatted as HTML. This is exactly what we would expect after looking at the latency information for document read operations. This test was run using a concurrency level of 100 with 5000 total requests.

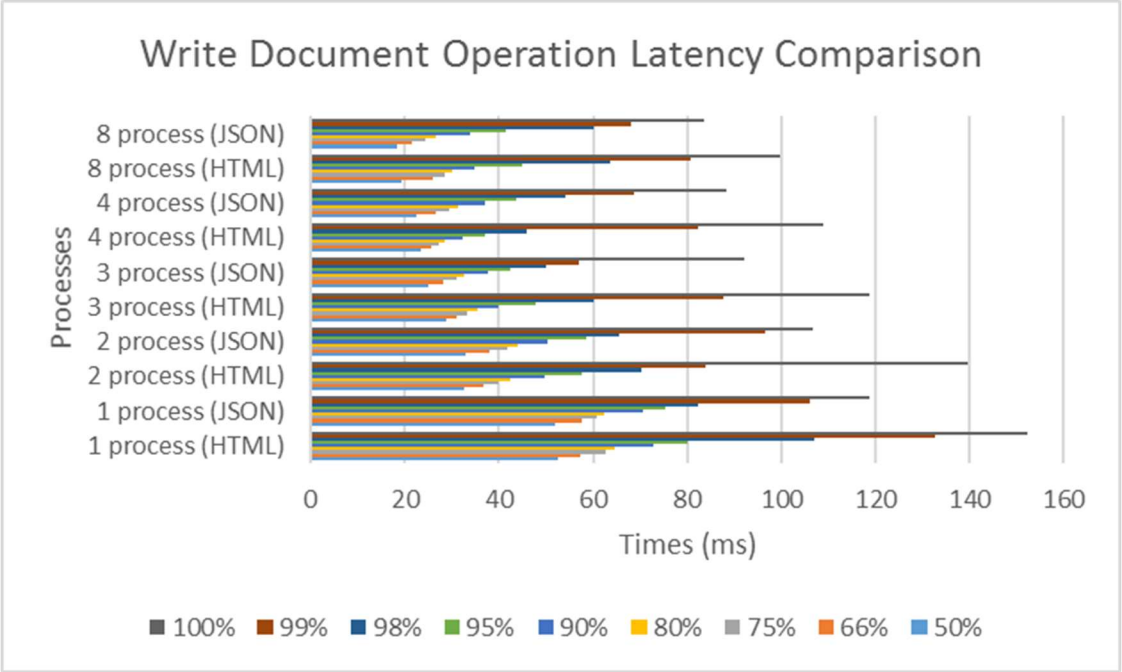

Figure 22: Latency comparison of a write operation

In Figure 22, we see the results of comparing Node.js returning HTML versus Node.js returning JSON when performing a write operation to the MongoDB database. This test was run using a concurrency level of 25 with 5000 total requests. On the x-axis, we see the times at which certain percentages of requests were completed at, and on the y-axis we see the total number of processes for the Mongo-Express and AngularJS applications. In every instance, we see that returning JSON proved to be faster than returning HTML.

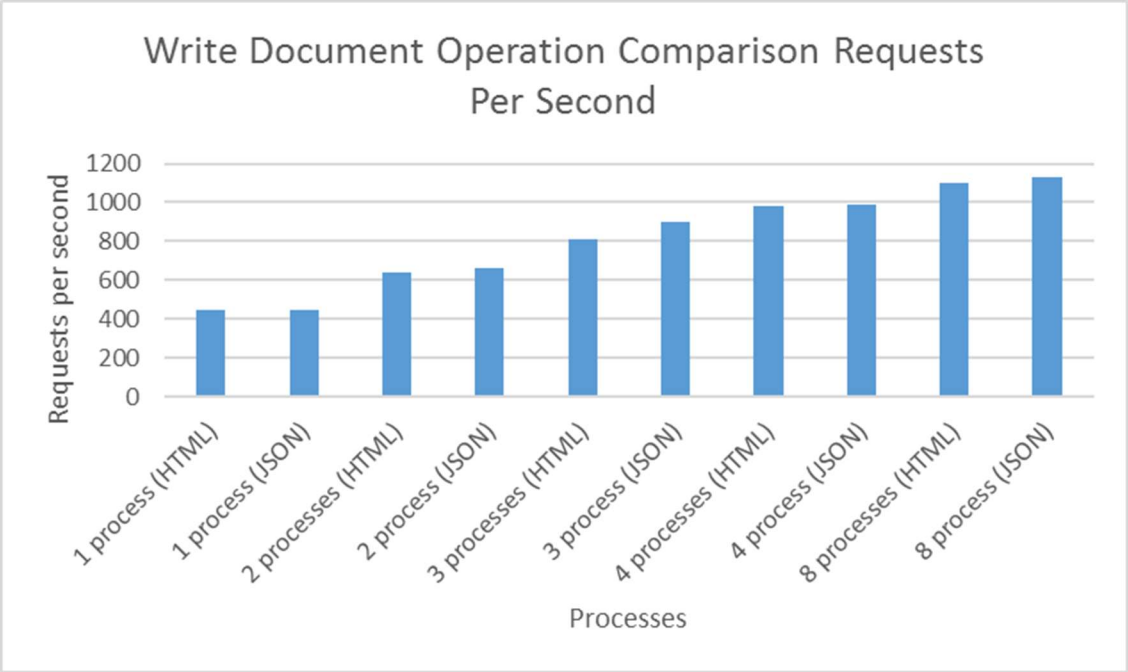

Figure 23: Requests per second comparison of a write operation

From Figure 23, we see the same trend in throughput as we do in latency: Node.js performs better when returning JSON data than when it returns HTML to the client. This test was run using a concurrency level of 25 with 5000 total requests. Along the x-axis, we see the number of processes spawned, and along the y-axis, we see the total number of requests handled per second.

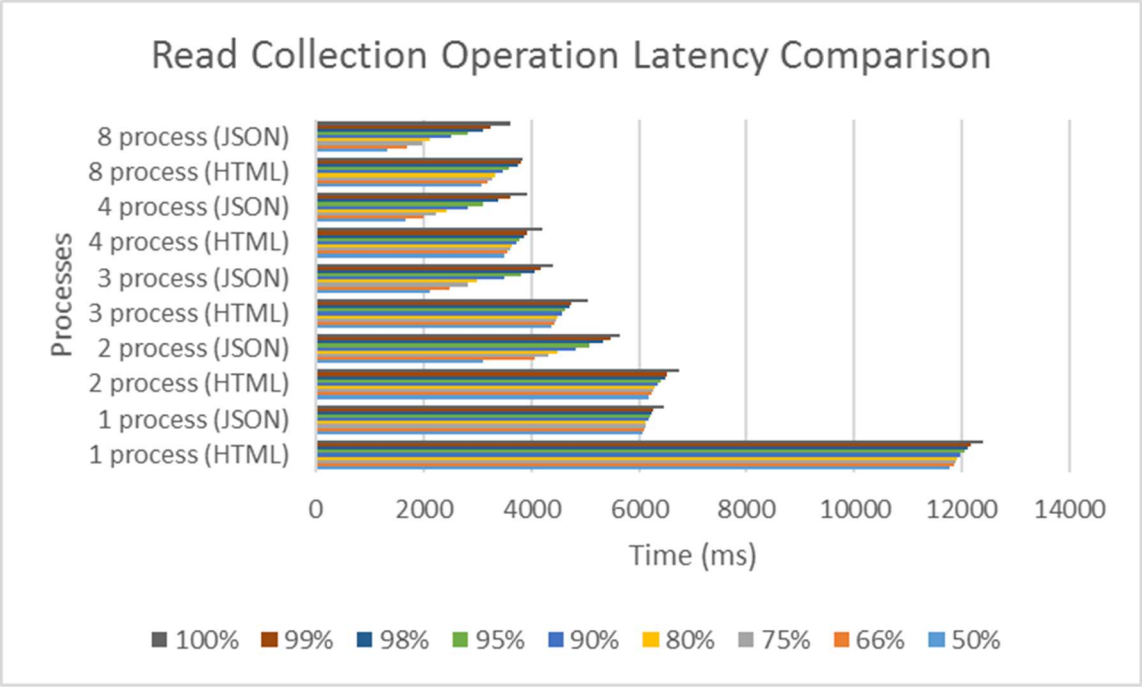

Figure 24: Latency comparison of a collection read operation

Figure 24 depicts Node.js returning HTML versus Node.js returning JSON when performing a collection read operation. This test was run using a concurrency level of 100 with 5000 total requests. In all instances, Node.js performs better when returning JSON than it does when returning HTML.

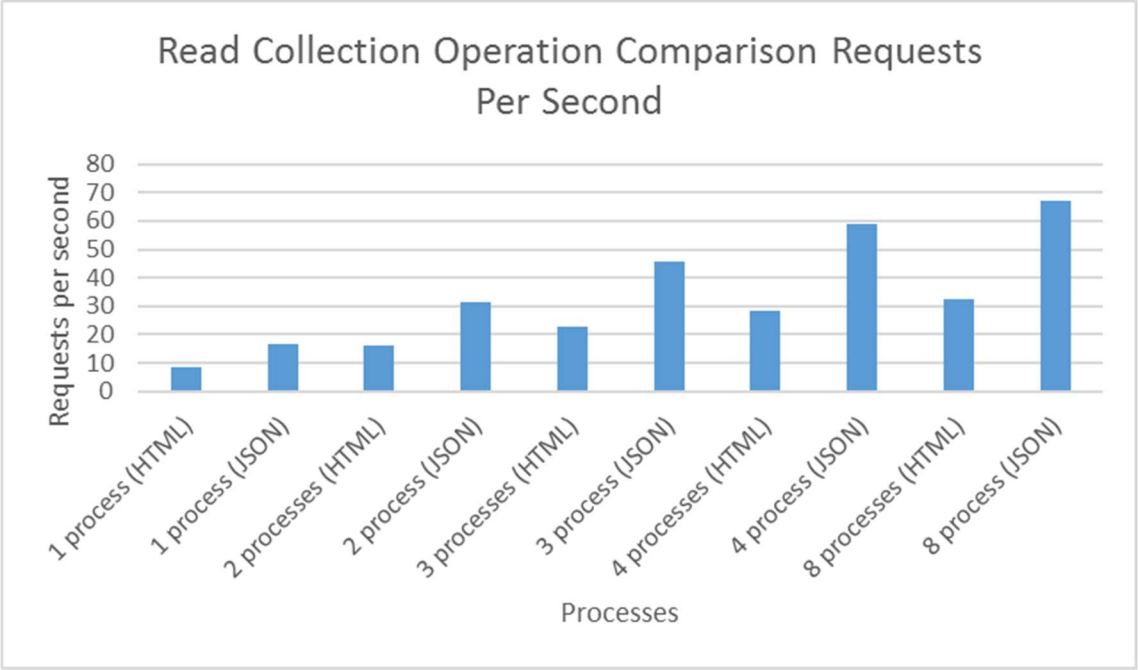

Figure 25: Requests per second comparison of a collection read operation

Figure 25 shows the throughput achieved by Node.js when returning HTML and when returning JSON when performing collection read operations. This test was run using a concurrency level of 100 with 5000 total requests. As expected, Node.js performs almost twice as better in every instance when returning JSON as opposed to HTML data.

Response Analysis

In the following section, I am going to analyze all of the steps that occur when the Node.js web server receives a read document, read collection, or a write document request. The purpose of this analysis is to see what portion of the response time each function call contributes and to which function, if any, is the primary contributor. We accomplished this via manual instrumentation by inserting console.time and console.timeEnd function calls to middleware and controller functions. These logging statements were added to every middleware function prior and to the function executed when the server receives a read document, a read collection, or a write document request.

Execution Time Percentages Across Varying Processes Comparison

Function Name	Operation	1 Process	2 Processes	3 Processes	4 Processes	8 Processes
View Collection (Angular)	Read Collection	99.95%	99.93%	99.92%	99.92%	99.92%
View Collection (Mongo-Express)	Read Collection	99.93%	99.92%	99.92%	99.92%	99.94%
Add Document (Angular)	Write Document	97.32%	96.69%	96.49%	96.26%	95.82%
Add Document (Mongo-Express)	Write Document	95.30%	95.80%	95.15%	94.39%	95.45%
Document Middleware (Angular)	Read Document	96.65%	95.53%	95.27%	96.06%	95.73%
Document Middleware (Mongo-Express)	Read Document	98.83%	98.56%	98.55%	98.43%	98.47%

Table 1: Execution time percentages across varying processes comparison

Table 1 shows the function execution time percentages for the longest-running functions when performing collection read, document read, and document write operations for both Mongo-Express and the AngularJS implementation with varying degrees of processes. As we can see, the execution time percentages were fairly consistent for each configuration regardless of total number of processes. Furthermore, the results were similar between Mongo-Express and the AngularJS implementation, as can be seen from the table.

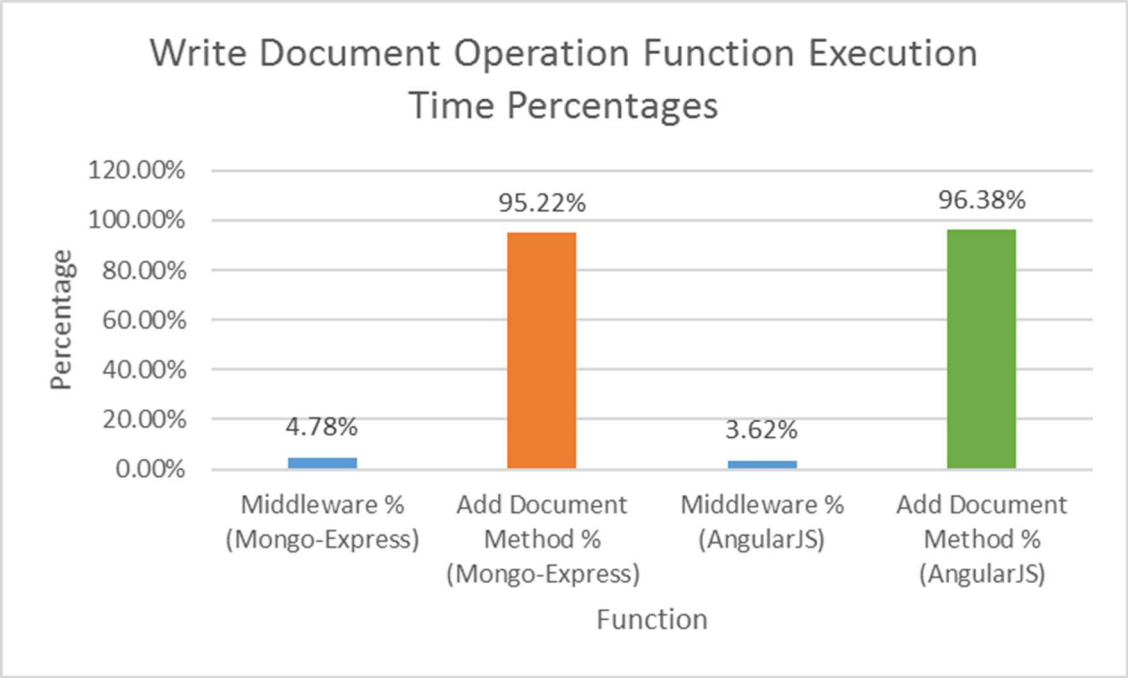

Figure 26: Write operation function execution time percentages

Figure 26 depicts the function execution time as percentages of the total response time when the Node.js web server receives a write document request. The y-axis shows the percentage of the total response time a function uses. The x-axis shows the function type, and in this chart, the we compare the add document controller method with the middleware functions run prior to the add document method. From the chart, we can see that for both Mongo-Express and the AngularJS implementation, the dominating function is the add document method, taking up approximately 96% of the total response time in both instances.

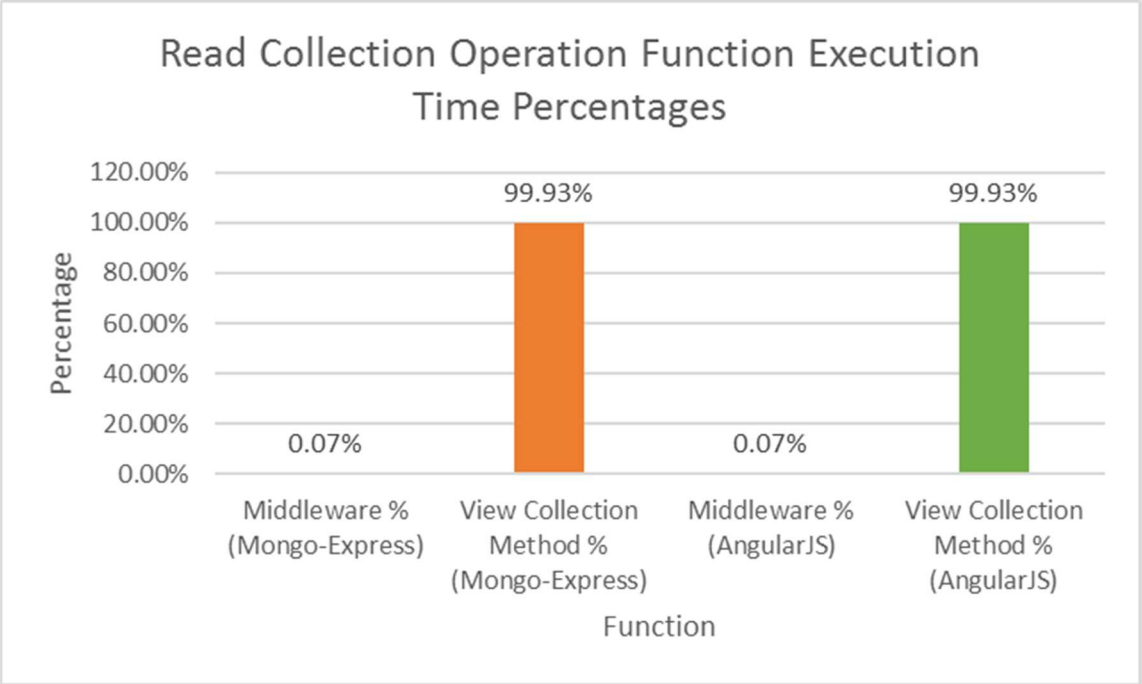

Figure 27: Collection read operation function execution time percentages

Figure 27 shows the function execution times as percentages of total response times for read collection operations. The y-axis shows the percentage of the total response time a function requires, and the function is shown across the x-axis. As can be seen from the chart, for both Mongo-Express and the AngularJS implementation, the dominating function is the view collection method, taking up 99.93% of the total response time.

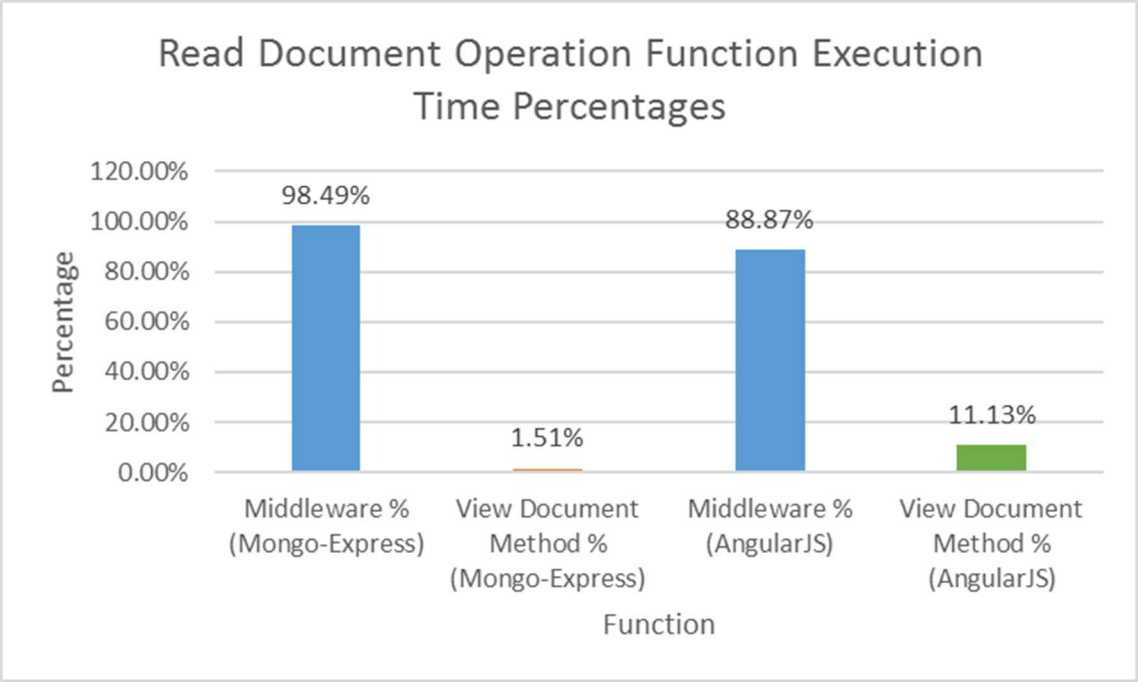

Figure 28: Document read operation function execution time percentages

Figure 28 shows the function execution times as percentages of total response times for read document operations. The y-axis shows the percentage of the total response time a function requires, and the function is shown across the x-axis. Unlike the previous charts, this chart reveals that read document operations response times are dominated by the middleware that precedes the controller function in both instances.

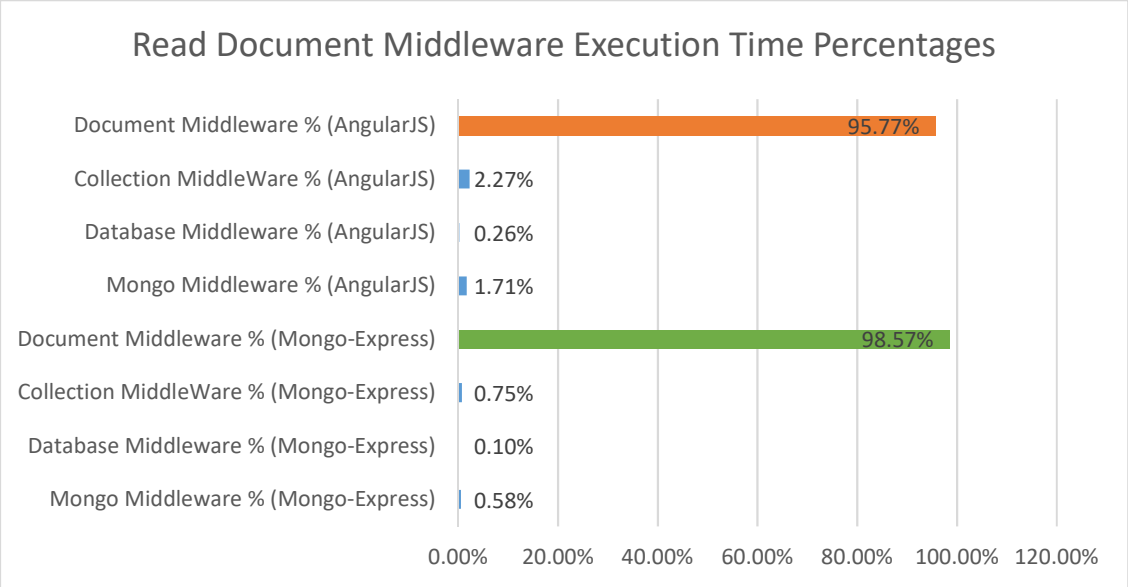

Figure 29: Document read operation middleware execution time percentages

After seeing that read document operations were dominated by the middleware rather than the view document function, we took a closer look at the distribution of time among all of the middleware functions. From Figure 29, we can see that in both instances the dominant middleware function is the document middleware function in which the document object is retrieved from the MongoDB database. This makes complete sense since the only purpose of the view document controller method is to use the document obtained from the middleware to return the object in JSON format in the AngularJS version or to return HTML in the Mongo-Express application.

In all three instances, the dominant function is the one performing the I/O operations. In the write document operation, the dominant function is the add document function which performs the actual insertion operation. In the read collection operation, the dominant function is the view collection method which obtains the collection in question and returns it. In the read document operation, it is the document middleware which performs the document retrieval, and as such, is the dominant function.

AngularJS Performance Analysis

In this section, we are going to take a look at how AngularJS's performance compares in regards to CPU and memory usage by using Mongo-Express as a baseline. We used Chrome DevTools to obtain snapshots of the heap to analyze AngularJS's memory usage. First, we will discuss how AngularJS performs when loading the homepage. Afterwards, we will see how AngularJS performs after loading a collection of one thousand elements.

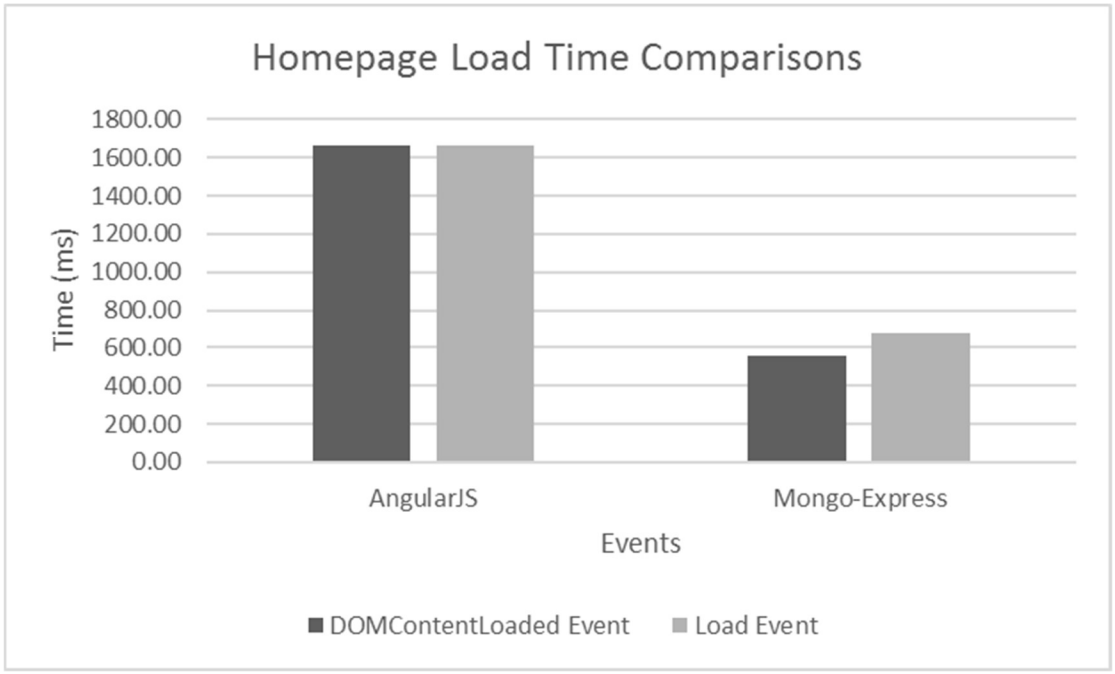

Figure 30: Homepage load time comparison

AngularJS typically took longer to load than Mongo-Express did, which was to be expected since AngularJS requires additional assets. In Figure 30, the x-axis shows AngularJS and Mongo-Express with the DOMContentLoaded event in blue and the Load event in Orange. On the y-axis, we have the total amount of time that the events took to complete in milliseconds. We see that AngularJS nearly took a full second longer than Mongo-Express before the load event was fired.

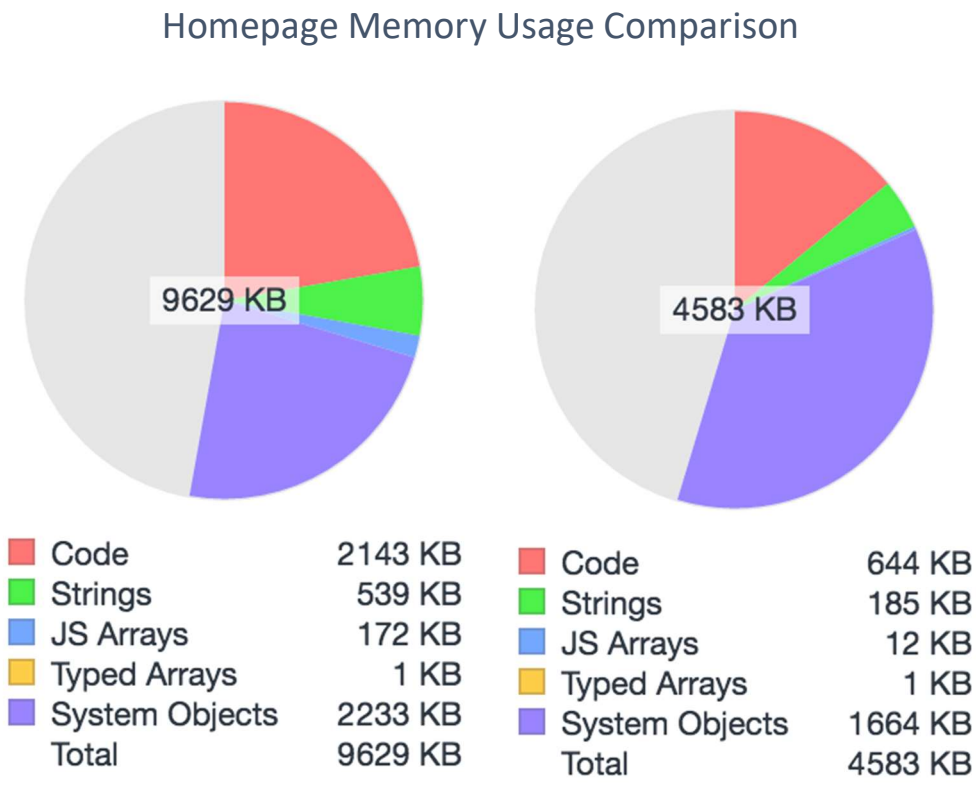

Figure 31: Homepage memory usage comparison

Figure 31 above highlights the memory usage of both AngularJS and Mongo-Express after loading the homepage. On the left is AngularJS and on the right is Mongo-Express. We see that AngularJS consumes over twice as memory as Mongo-Express, which is to be expected since AngularJS is a client-side JavaScript Framework.

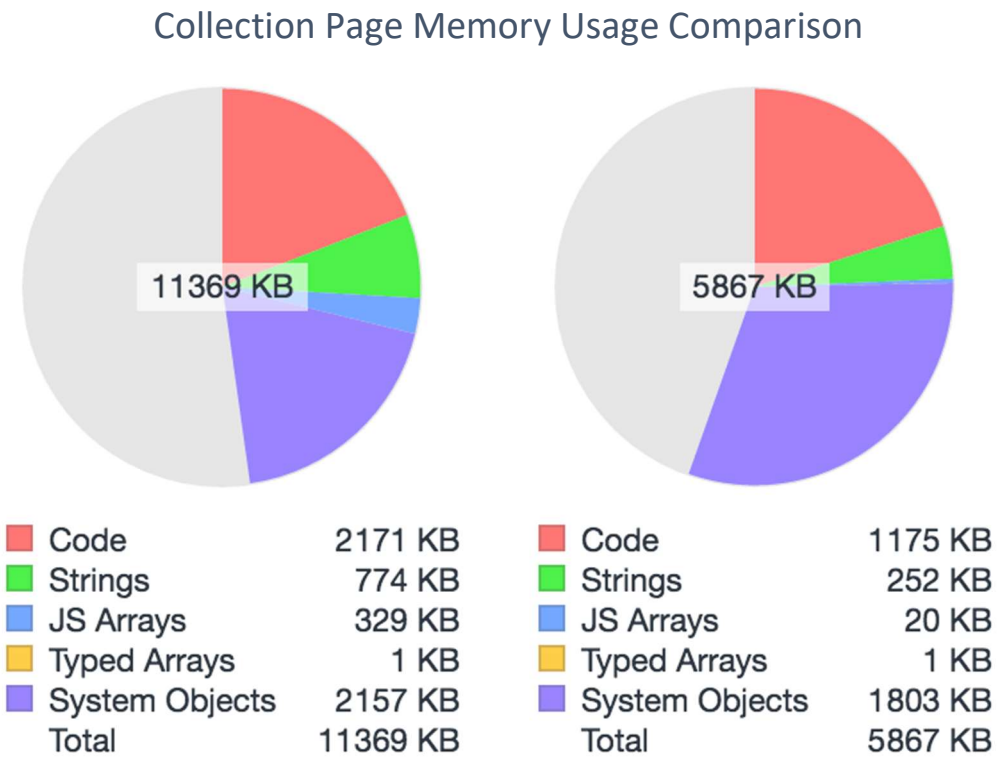

Figure 32: Collection page memory usage comparison

Figure 32 above shows the memory usage of both AngularJS on the left and Mongo-Express on the right after loading a collection with one thousand documents. Once again, we see that AngularJS consumes far more memory than Mongo-Express, but this time it only consumes approximately 93% more memory than Mongo-Express.

AngularJS Homepage Function Profile

Self Time		Total Time		Function
5159.6ms		5159.6ms		(idle)
200.8ms	43.62%	200.8ms	43.62%	(program)
0.3ms	0.06%	128.1ms	27.83%	▶ $apply
2.5ms	0.54%	109.5ms	23.77%	▶ $digest
0.3ms	0.06%	99.6ms	21.63%	requestLoaded
0.1ms	0.03%	99.3ms	21.57%	▶ completeRequest
0.4ms	0.09%	99.2ms	21.54%	▶ done
0.3ms	0.06%	89.0ms	19.33%	▶ $eval
0.1ms	0.03%	85.0ms	18.45%	completed
0.1ms	0.03%	84.8ms	18.42%	▶ ready
0.1ms	0.03%	84.1ms	18.27%	▶ fire
0ms	0%	84.1ms	18.27%	▶ fireWith
0.1ms	0.03%	83.8ms	18.21%	▶ angularInit
0ms	0%	83.8ms	18.21%	▶ (anonymous function)
0.3ms	0.06%	83.4ms	18.12%	▶ bootstrap
0.6ms	0.12%	83.1ms	18.06%	▶ doBootstrap
14.5ms	3.15%	82.7ms	17.97%	▶ invoke
0ms	0%	79.1ms	17.18%	▶ lazyCompilation
0.8ms	0.18%	77.4ms	16.82%	▶ publicLinkFn
0.8ms	0.18%	72.6ms	15.76%	▶ compositeLinkFn
1.7ms	0.36%	71.7ms	15.58%	▶ nodeLinkFn
0ms	0%	66.9ms	14.52%	▶ (anonymous function)
1.9ms	0.42%	66.9ms	14.52%	▶ processQueue
0ms	0%	65.0ms	14.13%	▶ invokeLinkFn

Figure 33: AngularJS homepage function profile

Figure 33 shows the function called along with their execution times when the homepage was loaded in the AngularJS application. From Figure 33, we see that the that the majority of the top-running functions are AngularJS functions, such as $apply and $digest.

Mongo-Express Homepage Function Profile

Self Time		Total Time ▼		Function
2895.8ms		2895.8ms		(idle)
77.5ms	65.82%	77.5ms	65.82%	(program)
0ms	0%	21.5ms	18.30%	(anonymous function)
10.6ms	9.02%	21.5ms	18.30%	▶ (anonymous function)
2.2ms	1.91%	10.9ms	9.28%	▶ (anonymous function)
1.2ms	1.02%	9.7ms	8.26%	▶ each
0.7ms	0.64%	7.8ms	6.61%	▶ (anonymous function)
4.2ms	3.56%	7.3ms	6.23%	(anonymous function)
0.1ms	0.13%	7.3ms	6.23%	completed
0.1ms	0.13%	7.2ms	6.10%	▶ ready
0.1ms	0.13%	6.6ms	5.59%	▶ each
0.1ms	0.13%	6.1ms	5.21%	▶ fireWith
0.4ms	0.38%	6.0ms	5.08%	▶ fire
0.3ms	0.25%	5.4ms	4.57%	▶ (anonymous function)
0.1ms	0.13%	4.9ms	4.19%	▶ b
0.4ms	0.38%	3.7ms	3.18%	▶ c
1.6ms	1.40%	3.6ms	3.05%	▶ (anonymous function)
0.3ms	0.25%	3.3ms	2.80%	▶ c.init
2.5ms	2.16%	2.5ms	2.16%	(garbage collector)
0.4ms	0.38%	2.4ms	2.03%	▶ jQuery
0.3ms	0.25%	2.2ms	1.91%	▶ on
0.6ms	0.51%	1.9ms	1.65%	▶ jQuery.fn.init
0.1ms	0.13%	1.8ms	1.52%	▶ (anonymous function)
0.4ms	0.38%	1.8ms	1.52%	▶ assert

Figure 34: Mongo-Express homepage function profile

In Figure 34, we see the function profile when loading the homepage in the Mongo-Express application. We can see that the longest-running function is an anonymous function with a total time of 21.5 milliseconds, a fraction of the time required by the AngularJS $apply function. When examining both charts, we see the same jQuery function: completed. However, we see that this function takes significantly longer in the AngularJS version than in Mongo-Express, requiring 85ms in AngularJS and 7.3ms in Mongo-Express. The purpose of the completed function is to remove load event listeners and to call the jQuery ready method once the DOM is ready to be manipulated. As we

saw when we analyzed the load times, AngularJS takes almost an entire second longer to load than Mongo-Express. As such, we also see the completed function is noticeably slower in the AngularJS implementation as well.

AngularJS Collection Page Function Profile

Self Time		Total Time ▼		Function
8286.3ms		8286.3ms		(idle)
264.2ms	42.77%	264.2ms	42.77%	(program)
0.4ms	0.07%	214.4ms	34.70%	▶ $apply
3.9ms	0.63%	198.3ms	32.10%	▶ $digest
1.1ms	0.17%	189.7ms	30.70%	requestLoaded
0.1ms	0.02%	188.5ms	30.51%	▶ completeRequest
0ms	0%	188.3ms	30.48%	▶ done
0ms	0%	134.0ms	21.69%	▶ lazyCompilation
1.8ms	0.28%	130.1ms	21.06%	▶ publicLinkFn
3.9ms	0.63%	112.4ms	18.19%	▶ nodeLinkFn
0.4ms	0.07%	110.8ms	17.93%	▶ $eval
2.0ms	0.33%	107.0ms	17.32%	▶ compositeLinkFn
0.3ms	0.04%	104.6ms	16.93%	▶ controllersBoundTransclude
0.8ms	0.13%	103.2ms	16.71%	▶ boundTranscludeFn
0.1ms	0.02%	99.0ms	16.03%	▶ invokeLinkFn
2.7ms	0.44%	98.9ms	16.01%	▶ ⚠ (anonymous function)
0ms	0%	86.7ms	14.04%	▶ (anonymous function)
1.1ms	0.17%	86.7ms	14.04%	▶ processQueue
0.1ms	0.02%	74.6ms	12.07%	completed
0.3ms	0.04%	74.4ms	12.05%	▶ ready
13.8ms	2.23%	74.4ms	12.05%	▶ invoke
0.1ms	0.02%	73.4ms	11.87%	▶ fire
0ms	0%	73.4ms	11.87%	▶ fireWith
0.1ms	0.02%	73.0ms	11.81%	▶ (anonymous function)

Figure 35: AngularJS collection page function profile

Figure 35 shows the function profile for the AngularJS application when loading a collection of one thousand elements. Once again, we see that the longest-running functions are AngularJS's $apply and $digest functions, requiring 214.4ms and 198.3ms, respectively.

Mongo-Express Collection Page Function Profile

Self Time		Total Time ▼		Function
3785.1ms		3785.1ms		(idle)
781.4ms	82.31%	781.4ms	82.31%	(program)
1.7ms	0.18%	82.9ms	8.73%	(anonymous function)
0.3ms	0.03%	77.8ms	8.20%	▶ Vd
77.6ms	8.17%	77.6ms	8.17%	▶ Oc
0.1ms	0.01%	24.8ms	2.62%	completed
0.1ms	0.01%	24.7ms	2.60%	▶ ready
0.1ms	0.01%	23.4ms	2.47%	▶ fireWith
0.8ms	0.09%	23.3ms	2.45%	▶ fire
10.3ms	1.09%	20.6ms	2.17%	▶ (anonymous function)
0ms	0%	20.6ms	2.17%	(anonymous function)
0.1ms	0.01%	19.8ms	2.09%	(anonymous function)
3.9ms	0.41%	19.4ms	2.04%	▶ a.fromTextArea
0.8ms	0.09%	15.5ms	1.63%	▶ (anonymous function)
0.8ms	0.09%	15.5ms	1.63%	▶ a
0.4ms	0.04%	12.1ms	1.28%	▶ jQuery
5.2ms	0.54%	11.9ms	1.25%	▶ (anonymous function)
1.1ms	0.12%	11.7ms	1.23%	▶ jQuery.fn.init
0.7ms	0.07%	10.6ms	1.12%	▶ find
2.2ms	0.24%	10.3ms	1.09%	▶ (anonymous function)
6.1ms	0.65%	9.9ms	1.04%	▶ Sizzle
9.3ms	0.98%	9.3ms	0.98%	▶ ud
1.8ms	0.19%	9.2ms	0.97%	▶ each
5.6ms	0.59%	7.0ms	0.73%	(anonymous function)

Figure 36: Mongo-Express collection page function profile

In Figure 36, we see the function profile when loading the same collection of one thousand elements in Mongo-Express. We see that the longest-running functions are a far cry from the longest-running functions in the analogous AngularJS function profile. Once again, we see that the completed method takes exponentially longer in the AngularJS implementation than it does in the Mongo-Express implementation.

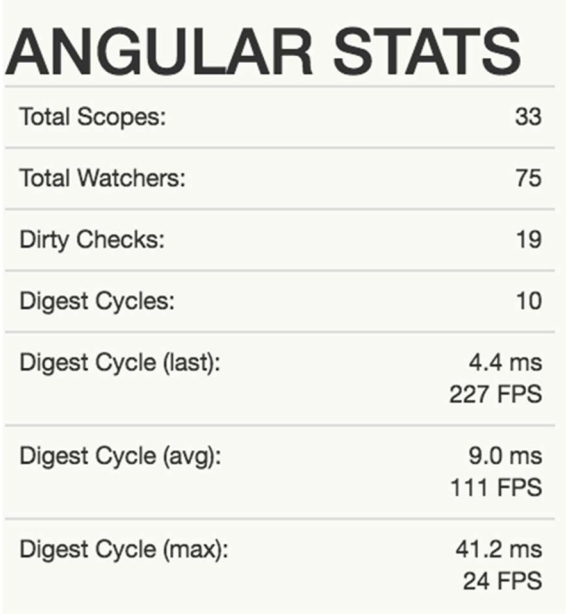

Figure 37: AngularJS homepage statistics

Figure 37 shows the performance statistics of AngularJS when loading the homepage. We see AngularJS creating 33 scopes, watching 75 objects, and requiring an average of 9 milliseconds per digest cycle.

Figure 38: AngularJS collection page statistics

In Figure 38, we see the performance statistics of AngularJS when loading a collection of 1000 documents. The performance impacts of using AngularJS are even more pronounced when loading a collection. We see the number of scopes has increased to 273, the number of watchers has increased to 539, and the average digest cycle has increased to 13.8 milliseconds. However, these numbers are still within what Google considers acceptable for AngularJS. They have deemed that slow AngularJS applications to be those that either have more than 2000 watchers or those that take more than 25 milliseconds for digest cycles [42].

Conclusion

The main thrust of this thesis is to see whether Node.js as a web server has any distinguishable performance differences when serving HTML as opposed to JSON data. We decided that the best way to achieve this was to port and existing application, Mongo-Express, to a full MEAN stack implementation while attempting to have the addition of AngularJS be the only difference between the two. As such, we followed best practices in creating out AngularJS application along with using proven tools and standards.

We anticipated that Node.js would handle serving JSON better than HTML due to no longer having to create the HTML on the server. Furthermore, we also felt that Node.js serving JSON would beat HTML because responses would be smaller as a result of offloading the majority of the work to the client side. From extensive testing, we saw that, on average in all cases, Node.js performed significantly better when serving JSON data instead of HTML data.

We hypothesized that the client would see an increased workload from including AngularJS. Through using Chrome DevTools to analyze memory usage, CPU usage, and network performance, we indeed saw that AngularJS had significant impacts on the client; AngularJS both required more memory, had longer-running functions than Mongo-Express, and required additional time to load. We looked at how memory was allocated on the heap both after loading the homepage and loading a data-heavy page: a collection page. We also examined the function profiles for the same instances.

If a team of developers wishes to begin a new project or port an existing project using AngularJS, there are tangible benefits and drawbacks as discussed above. However, there are a few additional considerations to take in to account. First, AngularJS does have

a steep learning curve to properly use; while it is fast to build a simple application, using more advanced features, such as custom directives, will require a significant investment. Second, AngularJS does provide the benefit of being able to further separate concerns and allows you to bring the MVC design pattern to client-side development. Third, AngularJS will most likely continue enjoy support as it created by Google and has already established foothold. Fourth, AngularJS 2.0 has experienced a paradigm change, greatly shifting the way code is written from Angular 1.x. As such, supporting applications written in both 1.x and 2.0 will require significantly more effort.

References

1. https://github.com/mongo-express/mongo-express. Accessed 27 Feb 2016
2. https://www.phpmyadmin.net/. Accessed 27 Feb 2016
3. https://www.apachefriends.org. Accessed 27 Feb 2016
4. https://nodejs.org/. Accessed 27 Feb 2016
5. https://github.com/nodejs/node-v0.x-archive/wiki/Projects,-Applications,-and-Companies-Using-Node/. Accessed 27 Feb 2016
6. https://github.com/search?q=stars:%3E1&s=stars&type=Repositories/. Accessed 27 Feb 2016
7. Chaniotis, Ioannis K., Kyriakos-ioannis D. Kyriakou, and Nikolaos D. Tselikas. "7. Is Node.js a Viable Option for Building Modern Web Applications? A Performance Evaluation Study." Computing 2015. Print.
8. https://docs.angularjs.org/guide/introduction/. Accessed 27 Feb 2016
9. https://www.madewithangular.com/. Accessed 27 Feb 2016
10. Padhy, Rabi Prasad, Manas Ranjan Patra, and Suresh Chandra Satapathy. "RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Database's." International Journal Of Advanced Engineering Sciences And Technologies 11.1 (2011): 15-30. Web.
11. https://www.facebook.com/notes/facebook-engineering/cassandra-a-structured-storage-system-on-a-p2p-network/24413138919/. Accessed 27 Feb 2016
12. https://www.mongodb.com/who-uses-mongodb/. Accessed 27 Feb 2016
13. http://blog.mongodb.org/post/5545198613/mongodb-live-at-craigslist. Accessed 27 Feb 2016
14. Nance, Cory; Losser, Travis; Iype, Reenu; and Harmon, Gary, "NOSQL vs RDBMS - Why There Is Room For Both" (2013). SAIS 2013 Proceedings. Paper 27.
15. Okman, L., Gal-Oz, N., Gonen, Y., Gudes, E., and Abramov, J. (2011) Security issues in NoSQL databases trust, security and privacy in computing and communications (TrustCom), IEEE 10th International Conference, 541-547, 16- 18.
16. https://www.mongodb.com/what-is-mongodb. Accessed 27 Feb 2016
17. Wei-Ping, Zhu, Li Ming-Xin, and Chen Huan. "Using MongoDB to Implement Textbook Management System Instead of MySQL." 2011 IEEE 3rd International Conference on Communication Software and Networks (2011). Web.
18. https://docs.mongodb.org/ecosystem/tools/administration-interfaces. Accessed 27 Feb 2016
19. https://github.com/mrvautin/adminMongo. Accessed 27 Feb 2016
20. http://siliconangle.com/blog/2013/04/01/the-birth-of-node-where-did-it-come-from-creator-ryan-dahl-shares-the-history/. Accessed 10 Mar 2016
21. https://docs.angularjs.org/guide/databinding. Accessed 10 Mar 2016
22. https://www.facebook.com/notes/facebook-engineering/cassandra-a-structured-storage-system-on-a-p2p-network/24413138919/. Accessed 09 Nov 2016
23. R. Hashemian, D. Krishnamurthy, and M. Arlitt. Overcoming Web Server Benchmarking Challenges in the Multi-core Era. In Proceeding of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), April 2012
24. http://pm2.keymetrics.io/. Accessed 16 Nov 2016
25. https://developers.google.com/web/tools/chrome-devtools/. Accessed 16 Nov 2016
26. https://www.npmjs.com/. Accessed 12 Dec 2016
27. https://docs.angularjs.org/guide/compiler. Accessed 12 Dec 2016
28. https://www.mongodb.com/faq?jmp=footer. Accessed 12 Dec 2016
29. http://gruntjs.com/. Accessed 12 Dec 2016
30. http://gulpjs.com/. Accessed 12 Dec 2016
31. https://github.com/gulpjs/gulp/issues/540/. Accessed 12 Dec 2016
32. https://bower.io/docs/about/. Accessed 12 Dec 2016
33. http://www.protractortest.org/#/. Accessed 12 Dec 2016
34. https://docs.angularjs.org/guide/e2e-testing. Accessed 12 Dec 2016
35. https://testing.googleblog.com/2012/11/testacular-spectacular-test-runner-for.html/. Accessed 12 Dec 2016
36. http://www.funnyant.com/angularjs-ui-router/. Accessed 12 Dec 2016
37. https://www.joyent.com/blog/node-js-on-the-road-sf-node-js-at-paypal/. Accessed 12 Dec 2016
38. http://docs.libuv.org/en/v1.x/threadpool.html/. Accessed 12 Dec 2016
39. https://blogs.msdn.microsoft.com/wesdyer/2007/12/22/continuation-passing-style/. Accessed 15 Feb 2017
40. https://www.sciencedaily.com/releases/2013/05/130522085217.htm. Accessed 15 Feb 2017
41. https://www.ibm.com/blogs/cloud-computing/2014/04/explain-vertical-horizontal-scaling-cloud/. Accessed 15 Feb 2017
42. https://github.com/joeLepper/ng-conf/. Accessed 15 Feb 2017