Scalable web application using node.js and CouchDB

Umesh Paudyal
Abstract

Scalable web application using node.JS and CouchDB

Umesh Paudyal

This report presents design and implementation of a prototype application using server side javascript programming language, node.JS and couchDB as backend database. It scales and evaluates the developed prototype application and the couchDB for their scalability and performance. The report concludes that node.JS is a suitable framework for development of scalable web servers and couchDB as a backend database, though natively not distributed and scalable, can be scaled and distributed across multiple nodes using clustering and replication mechanism.
Acknowledgements

First of all, I would like to express my gratitude to Wussap AB for providing me the opportunity to work on cutting edge technologies as my thesis work. I would like to thank my supervisor Claudijo Borovic from Wussap AB for his invaluable guidance throughout my thesis work.

I would also like to extend my sincere gratitude to Professor Olle Eriksson from Uppsala University who constantly provided guidance and reviewed my thesis throughout the thesis work.

I am also thankful to Mats Lundin for the system support throughout my thesis work. I want to extend my thanks to Dick Elfstrom who provided the place to stay for thesis work in University.

Finally, I am always indebted to my parents for their unconditional support and facing difficulties on making my education the foremost priority.
## Contents

1 Introduction .......................... 1
   1.1 Motivation .......................... 2
   1.2 Thesis Structure .................... 3

2 Related Work .......................... 4

3 Background ............................ 5
   3.1 Wussap System ........................ 5
   3.2 Javascript .......................... 6
   3.3 Node.JS ............................. 6
   3.4 QUnit ............................... 8
   3.5 JSON ............................... 9
   3.6 CouchDB ............................. 10
      3.6.1 Views ............................ 11
      3.6.2 HTTP API ........................ 11
      3.6.3 CAP theorem ........................ 11
   3.7 BigCouch ............................ 12
   3.8 Tsung ............................... 13

4 Design and Implementation .............. 14
CONTENTS

Appendices 45

A Abbreviations 46

B Tsung Configuration XML 47
   B.1 CouchDB test configuration XML 47
List of Figures

3.1 node.JS architecture stack courtesy:[Nod10] of Ryan Dahl .......................... 7
3.2 node.JS benchmark courtesy:[Nod10] of Ryan Dahl ................................. 9
3.3 CAP Theorem Courtesy:CouchDB The Definitive Guide [ALS10] ....................... 12
3.4 BigCouch System Architecture courtesy:[Sca11] of Adam Kocoloski ................. 12

4.1 Mockup representation of above use case .................................................. 16
4.2 Client Architecture .................................................................................. 16
4.3 Screenshot of Unit Test ............................................................................. 17
4.4 Server Architecture .................................................................................. 18
4.5 Client-Server Protocol ............................................................................. 19
4.6 Document in CouchDB ............................................................................. 22

5.1 One of the Horizontal scalable solution ....................................................... 24
5.2 Memcached instead of Session Manager ..................................................... 26
5.3 Reading from Memcached vs Views for 1000 requests ............................... 27
5.4 Reading from Memcached vs Views for 5000 requests ............................... 27
5.5 Clustered remote nodes ........................................................................... 30
5.6 Sharded Database .................................................................................... 31

6.1 Single CouchDB Simultaneous Users ......................................................... 34
LIST OF FIGURES

6.2 Single CouchDB transaction rate ........................................... 35
6.3 Simultaneous users with multiple nodes in a local cluster .................. 36
6.4 Transaction rate with multiple nodes in a local cluster .................... 36
6.5 Simultaneous users ........................................................... 37
6.6 Memory usage ............................................................... 37
6.7 Transaction throughput ..................................................... 37
6.8 CPU usage ................................................................. 37
List of Tables

3.1 Bigcouch Parameters .................................................. 13

4.1 Use case for User Registration ................................. 15

4.2 Protocol Message for User Registration ......................... 20
Chapter 1

Introduction

In this era of computer and internet, dynamic web is the ubiquitous source for socializing, news updates to name a few amongst many other uncountable usages. With the advent of social networking sites, huge amount of people have been visiting and spending time simultaneously on such sites. This poses the requirement that sites need to be very quick in performance, scalable and distributed despite of large number of concurrent users. Currently, we can take Facebook and Google+ as some of the examples of popular social sites to get the idea of how they need to be scalable without any degradation of performance to support large number of concurrent users and to keep the users interested on visiting sites more often. So, to avoid the user frustration on using web application requires smoothness of access and support of any number of simultaneous users without performance degradation.

In this paper, test driven development of a web application has been carried out using relatively new programming language called node.js(server side javascript technology) with CouchDB as backend database. Node.js which is suitable language for development of scalable network application has been used to develop a prototype (based on Wussap AB requirement). The same prototype has been analyzed in subsequent chapters for evaluation. The implemented Wussap prototype has been tested and evaluated for its scalability and tried to find out the possible bottlenecks.

Since almost all of the dynamic web application needs to store their data in some kind of persistent storage like database and reads/writes data from/to database most of the time. Since I/O operation in web server is one of the costliest operations, database can be one of the bottlenecks for scalability and performance of web application. So, in this thesis, database has been scaled to avoid the possible bottleneck. The database used in this thesis is document oriented database called couchDB. Though
couchDB doesn’t inherently provide the scalable solution, it is possible to make scalable with its readily available features like replication and clustering. It has been explored with the aid of an open source application, BigCouch and a load balancing software HAProxy.

1.1 Motivation

Web application today needs to be fast responsive along with support for large number of simultaneous users. The social web application are most of the times accessed by large number of simultaneous users around the globe. So such sites are always in need of a design to handle large amount of concurrent online users at the same time with performance alongside. So distributed web application with good performance and scalability is only the solution. So the goal of this thesis is to develop a prototype application and evaluate the scalability and performance of it along with the backend database. The prototype is developed in server side javascript technology(nodeJS) and couchDB(one of the nosql\(^1\) databases) as backend.

Talking about suitable technologies for the aforementioned requirements, node.js is chosen to be the best bet to provide scalable solution for web application with the asynchronous nature of programming and non-blocking code. Node.js has got peak popularity among open source community due to its promising features like non blocking I/O operation unlike traditional programming languages and event handling mechanism.

Database plays an important role on scaling of a web application because most of the web application are read/write intensive and needs to read/write data from/in persistent storage and the research in this field has suggested that database is the main bottleneck. So, with the realm of nosql databases recently, which are basically designed for web applications focussing on their performance and scalability, We chose one of the nosql databases called CouchDB. CouchDB has revolutionized the way data is stored for web applications for their better performance with schema less document oriented storage. Though the couchdb cannot offer the intended scalable and distributed solutions, it can be made scalable and distributed with the available features like clustering, sharding and replication which is addressed by author in this thesis with the aid of an open source clustering tool available for couchDB.

\(^1\)NoSQL is a horizontally scalable database management system suitable for modern web applications that don’t require to distill information into tables(like RDBMS) but can store data into key/value, columns and document format.
1.2 Thesis Structure

The goal of this thesis is to implement a test driven prototype for wussap system using node.js and couchDB. The implemented prototype is further refactored to improve performance using caching. Finally, the couchDB database is scaled and evaluated.

Chapter 1 as an Introduction chapter defines the overview of thesis with motivation for carrying out this thesis.

Chapter 2 outlines the related work similar to this thesis work.

Chapter 3 overviews and details the background technologies that are used to carry out and needed to understand this thesis along with the company where it had been carried out.

Chapter 4 describes the design and implementation carried out to develop prototype for this thesis.

Chapter 5 outlines scalability patterns along with database scalability.

Chapter 6 incorporates testing and evaluation of the prototype and couchDB database for scalability and performance.

Chapter 7 concludes the thesis with conclusive notes and describing other possible areas of research.
In recently published article on IEEE by Authors of [GRL11] compared the scalability of the popular service domain as a comparison which was developed in blocking and non-blocking technologies including java, scala and Node.js. They conclude saying Node.js came out on top of comparison by providing accurate and timely mass message delivery.

The author of [Dan11] has pointed out in his findings (namely "The C10k problem") that supporting 10,000 inbound HTTP requests concurrently is not dependent on hardware being the only bottleneck, but rather web servers can be scaled by changing the way I/O operation is handled.

Similarly, the author of [Til10] summarizes that node.js can be used to build high performance network programs due to its evented asynchronous style of programming. It bridges the gap between javascript being used only on the client side and dependency of other platforms in server side by making a single platform usable in both environments.

Leavitt N. in his publication in IEEE as "Will NoSQL Databases Live Up To Their Promise?" based on interviews of some database experts, cites nosql databases as scalable and good performance data storage solution for applications where data precision and consistency are not major concerns (like bank transactions) compared to its scalability and performance.
Chapter 3

Background

This chapter explains about the company on which the thesis has been carried out and the technologies that have been useful in realization of this thesis.

3.1 Wussap System

[Wus10] Wussap introduced the concept of social web browsing where people can co-browse the web together and chat on the topic of similar interest. The every page user has visited is treated as a separate chat room where people can share their similar interest. Wussap AB was ranked at 7th place as a sharpest web entrepreneurs of 2010 by Internetworld, the sweden’s leading business magazine for innovation on new way of socializing the web.

The term surftrain is coined to represent the co-browsing feature where leader leads the browsing of certain web page and one or more followers can join the same surftrain. Follower’s page gets redirected to the same page leader is visiting. Once the surftrain is scheduled, it can be on the state of ongoing, departed and arrived. The participants on the particular surftrain can chat among each other along the way.

Term webchat refers that the users in different web pages can treat those pages as a separate chat room where they get to chat with people having similar interest.
3.2 Javascript

[Jik11a] Javascript is an object oriented scripting language which is also considered as a functional programming language due to its support for closures and higher order functions. Javascript has got more attention as a programming language among developers due to introduction of ajax and server side programming capability.

Javascript is the language of web browser. Javascript has been taken lightly by programmers when they need to get work done with it without taking into consideration of available features. So, its potential is not discovered to the level it is supposed to be. Ajax, jquery and finally node.JS, building blocks for modern web, are outcome of javascript. It has some good and bad parts according to its inventor like every other programming languages. The good parts are functions, loose typing, dynamic objects and expressive object literals. Javascript is not a typed language which means javascript compiler doesn’t produce any type errors. It has feature like prototypal inheritance which means one object can inherit properties from other objects without concept of classes. Object literal is way of making new objects with name/value pairs. Example of simple object literal in javascript is `var myobj = {}`. It is a simple way of storing any type of name value pairs and is a foundation for JSON which can be referenced on section 3.5 for more details.[Cro08]

3.3 Node.JS

[Jik11b] Node.JS is an asynchronous event driven I/O framework with non-blocking I/O feature which runs on top of V8 javascript engine. It is server side javascript used to develop scalable network programs like webservers. As specified in node.JS’s website[Nod10], one of the node’s goal is to make an easy way of developing scalable network programs.

As shown in figure 3.1 node.JS makes use of eventloop to handle events and queue the callbacks associated with events. V8 is high performance javascript engine written in C++ and embedded in chrome which is the core design element for node.JS. Javascript layer is allowed to access the main thread only whereas C layer is open for multithreading. Multithreaded programming approach is easy and efficient way to make use of multiple cores for concurrency but it has some pitfalls like deadlocks, accessibility of shared resources and frequent switching of processes. Whereas Event driven model of node.JS provides more scalable solution of switching between tasks making use of event notification functionalities of underlying OS like `select()`, `poll()` along with `epoll, kqueue`,
3.3. NODE.JS

Figure 3.1: node.js architecture stack courtesy[Nod10] of Ryan Dahl

kevent calls[Til10].

Most of the web application cannot achieve good performance and scalability due to unnecessary waiting time of server on I/O tasks. Traditional programming languages which works on synchronous way cannot provide good response for web application as they have to wait for an I/O task to complete before proceeding to other task. For example, the web server needs to wait for the response from database before it can proceed to next task which needs the server to allocate resources for the single task. But with event oriented and asynchronous approach of programming, node.js solves this problem of supporting large number of concurrent users with good performance without blocking all server resources on single I/O operation.

As shown in the following signature example by node.js creator, node uses event loop to handle the requested events.

```javascript
setTimeout(function(){
  console.log("world");
}, 3000);
console.log("hello");
```
3.4. **QUNIT**

Here the console prints hello first and then world after 3 seconds. But according to synchronous programming point of view, the program should have printed world first and hello afterwards. But node makes it in asynchronous way by keeping the event (in this case the timeout event) in event loop and execution continues to next line. Thus it prints hello first.

Such style of programming becomes handy when we need to do some I/O operations. Unlike traditional programming languages which incurs memory cost and performance cost by using new thread per request, Node.js simply pushes the callback handler to register the particular event in event loop and returns to the associated callback function on getting event notification of result being ready and, thus, doesn’t stop the further execution by resuming the execution on node.js runtime environment. This avoids resources being idle, thus supports concurrency which in turn provides the framework for the development of scalable architecture. Following example shows the skeleton nonblocking code for database operation.

```
getData("select * from some_table", function(results){
  //do some tasks with database results
});
readFromFile();
```

So, as shown in above example, the server resources don’t need to be idle till waiting result from database, but rather the execution is handed over to next line, for example readFromFile in above case. In this case, the callback is queued into eventloop and execution is handed over to node.js runtime environment. So, whenever server gets data from database and it is ready, then the callback from event loop is called to execute those tasks.

Following figure 3.2 is taken from one of the presentation of Ryan Dahl (node.js inventor) which shows response time of simple node.js server as far better than compared to other popular web servers like thin and tornado but it is bit slower compared to nginx (another event loop based web server).

### 3.4 QUnit

QUnit is a unit testing framework for javascript and jquery based web applications. It can be used for unit testing of normal javascript code, jquery code and code written in server side
Good unit test coverage of the code during development helps easy refactoring and debugging of the code in later stages as well. It is especially useful for regression testing which involves testing of code, finding bug and testing it again after fixing.

3.5 JSON

JSON is a simple data interchange format readable to human and efficient while parsed by machine. It is language independent and merely a collection of name, value pairs. It supports the data structures that are supported in most of the modern programming languages like object, array, string, number, dictionary, lists etc as values.

Now a days, popular web sites like yahoo, google publish their contents through webservices in JSON format. For example, when we need to incorporate the real time weather information in our
3.6. COUCHDB

application from API provided by yahoo, we get it in JSON format accessible through webservice. The message format for JSON can be name/value pairs represented as dictionary, object, hashtable etc. or collection of values represented as collection like array, list etc. Examples of JSON message format could be {name:value}, {name,value}, {name:[value1, value2]} etc. and supports the nested form of json representation as well. Following example shows the JSON representation of some data and equivalent representation in XML of the same set of data.

{"username": {
"name": "umesh",
"password": "pass"
}}

Equivalent XML representation for the above JSON representation can be

<username>
  <name>umesh</name>
  <password>pass</password>
</username>

3.6 CouchDB

[Cou11a][Wiki11d] CouchDB is a schema less document oriented database written in erlang programming language. It provides the RESTful HTTP/JSON API that is accessible to other application using HTTP/JSON protocol. Other intriguing features of couchDB are MVCC and B-Tree. MVCC, the multi version concurrency control, is used in couchdb to manage concurrent access to database. Unlike locking mechanism in relational databases, couchdb uses MVCC approach to maintain different versions of documents to support concurrent requests without locks on documents[ALS10]. B-Tree invented by Rudolf Bayer and Ed McCreight in 1971, is a tree data structure for insertion, deletion, sequential access of data in logarithmic time which stores data in sorted manner[Wik11e]. Couchdb is one of the nosql databases that has revolutionized the way data is stored for web applications for better performance and scalability. It stores the data as standalone document representation unlike relational databases where tables represent data in rows and have relations in between tables.
3.6. COUCHDB

3.6.1 Views

Unlike relations between tables in relational database system that can be used to obtain the results for complex queries from multiple tables, CouchDB is designed to use views for such purpose. MAP/REDUCE functionality is used to combine and sort out the result from multiple documents. [Cou11b]There are two types of views supported in couchDB, namely temporary and permanent views.

**Permanent views** are stored in database under design documents. Permanent views are fast in comparison to temporary views. It can be queried with HTTP GET request like

http://serverip:5984/dbname/_design/designname/_view/viewname/

**Temporary views** are created on the fly when database is queried. Thus temporary views are on the slower side. It can be queried with HTTP POST request.

3.6.2 HTTP API

There are several http request methods that are supported and provided at user’s disposal to query data within couchDB database. These http request methods include GET, POST, PUT, DELETE etc. A simple example of creating database through HTTP API using curl utility is

```
curl -X PUT http://serverip:5984/dbname
```

Users can make their feasible API over this HTTP API provided by couchdb to be able to connect to their application and to abstract these HTTP API functionalities as per their needs.

3.6.3 CAP theorem

As depicted in figure 3.3, CouchDB provides availability and partition tolerance with less consistency. CouchDB trades off between availability and consistency with the expense of one of them. So, we get eventual consistency when we look for highly available database. Availability means that some version of data should always be accessible to clients. Consistency is to make sure that each users should be reading same version of data on concurrent updates on database. Partition tolerance
ensures that database can be partitioned to distribute data on multiple servers. One of the effective feature of couchDB for scaling database is its replication mechanism which is described in detail on section 5.3.

3.7 BigCouch

BigCouch can be used to create cluster of couchDB nodes and it provides partitioning of data across multiple shards. CouchDB nodes in cluster contain each of these shards and they have subset of data.

Figure 3.4 provides an overview of building blocks of bigcouch system. As shown in figure, MEM3 provides the shard mapping for each databases in cluster and also replicates the changes of registration of new nodes alongwith any changes in database as well. REXI provides parallelism in large number of RPC calls. Fabric is an OTP library application without processes for calling couchDB core APIs. Chttpd is a daemon supporting http requests to enable communication of couchdb with clients and it ultimately uses fabric to manage number of concurrent requests.
BigCouch partitions the documents and views across shards using deterministic hashing algorithm. It carries out read/write operations using quorum protocols. Some bigcouch parameters and their description are listed in table 3.1 [Hol11][Sca11].

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>Number of shards/partitions to divide the documents in database across.</td>
</tr>
<tr>
<td>N</td>
<td>Number of redundant replicas(shards) for each document. Defaults to number of nodes created.</td>
</tr>
<tr>
<td>W</td>
<td>Number for write quorum. W &lt;= N</td>
</tr>
<tr>
<td>R</td>
<td>Number of read quorum. R &lt;= N</td>
</tr>
</tbody>
</table>

Three different nodes in a local cluster are initialized with the following commands

```
./rel/dev1/bin/bigcouch
./rel/dev2/bin/bigcouch
./rel/dev3/bin/bigcouch
```

which are listening on 15984, 25984 and 35984. The requests/responses are passed through proxy server port 5984 which acts as a load balancer to divide the load across these multiple nodes.

### 3.8 Tsung

Tsung is an open source multi protocol testing tool for distributed load testing, performance testing and scalability testing of application and databases. Tsung is written in erlang, so it is inherently concurrent and fault tolerant application for carrying out tests. It has provision to define the scenarios with client, server setup along with sessions and load with different arrival phases. Inside each session we can define the urls which we want to load test on.

It supports users in distributed environment. So large number of users can be simulated. It also provides provision to define multiple servers which are used on round robin basis.

Tsung configuration includes client, server, session, arrival phase configuration. Session represents the group of requests and arrival phase represents session arrival phases with arrival rate.
Design and Implementation

With an introduction to the suitable technologies in previous chapter, this chapter is dedicated to detail out the design and implementation methodologies of the prototype application.

The test driven development (TDD) approach is followed to develop wussap prototype using unit test framework called qunit along with the server side javascript called node.js and document oriented nosql database called couchDB as a backend database. [Ham08] describes unit test frameworks as key element in TDD approach of development. So, Key rule of TDD is described as "Test twice Code once" in agile software development environment. The simple TDD cycle is described as test, code and refactor where a non functional (or the failing) code is produced first, then it is coded with some functionality (which must pass the test) and at last it is refactored to improve the internal design of the code without changing its external functionality. Hence TDD cycle has been extensively followed throughout the development of prototype application. QUnit, a unit testing framework for javascript code, has been useful for unit testing of each of the modules implemented.

Design architecture of client, server and database has been described in the following sections. The evented asynchronous approach of node.js is used to develop the application server which avoids the allocation of resource for single I/O task until its completion. The asynchronous approach is realized through the collection of callbacks in code which avoids blocking of code and memory overhead as there is no thread or process per request. Express[Exp11], an opensource high performance framework for node.js web application development has been used to make use of features like handling static contents, robust routing, redirection helpers, content negotiation, environment configuration etc.
4.1 Requirement Analysis

Extensive requirement analysis of wusssap system was carried out to make the basis for further design and implementation. Mockup tool was used to get the idea of requirements by drawing some figures based on the requirement elicitated with the help of use cases as shown in the following table 4.1 and figure 4.1.

<table>
<thead>
<tr>
<th>UseCaseName</th>
<th>RegisterUser</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParticipatingActors</td>
<td>NotRegistered: User</td>
</tr>
<tr>
<td>FlowofEvents</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• User clicks on &quot;register&quot; button</td>
</tr>
<tr>
<td></td>
<td>• Registration form is displayed on the client program</td>
</tr>
<tr>
<td></td>
<td>• User completes registration form by entering personal data and submits</td>
</tr>
<tr>
<td>Exitcondition</td>
<td></td>
</tr>
<tr>
<td></td>
<td>userName: The user name to register</td>
</tr>
<tr>
<td></td>
<td>userPassword: The password to register</td>
</tr>
<tr>
<td></td>
<td>statusCode: status code. 100 means success or errorCode is returned in case of error.</td>
</tr>
<tr>
<td>QualityRequirements</td>
<td>Authentication manager receives username, password and creates a new user in Database and replies to the client with status code</td>
</tr>
</tbody>
</table>

4.2 Client Side

The unit testing code for each of the modules mentioned in the figure 4.2 is written in the client side javascript following qunit test framework. The custom defined protocol is used to define the
message pattern for communication between client and server side. JSON formatted message is passed over to server using websocket based framework called socket.IO. [Soc11]Socket.IO supports several transport mechanisms like websocket, ajax long polling, JSONP polling etc. and falls back to the most applicable transport mechanism as per need.

Each modules mentioned in the above figure contains particular test setup that defines prerequisite for corresponding test cases. The test cases are defined as an asynchronous test which queues the added asynchronous tests and runs one after another.

Login/Registration module defines the username/password for the particular session of user and matches the result with response from server. Here we setup the username and password as minimum
4.3. **SERVER SIDE**

initial data required for the test setup of user registration and login. Similarly, unit test for surfroman\(^1\) module tests for creation/joining/leaving of particular surftrain. Surftrain Chat module is for sending of messages to/from users joined in a particular surftrain. Webchat test module tests for joining/leaving and sending messages in particular chat room. Here each web page visited is considered as a different chat room.

---

\(^1\)Surftrain is a term used to represent the cobrowsing feature started by the leader of the surftrain where any
Authentication manager handles the registration and logging in of the user. The registered user information is stored in couchDB database and maintained in a standalone document.

Session manager contains session for each logged in user with associated information of surfrain station (which is basically the page user is visiting). It is maintained in cache to reduce the latency and use of resources on accessing from database. Since views in couchDB are slow, the caching of data wherever possible in memory makes sense in terms of increasing the performance and scaling of application.

Surftrain manager and webchat manager is wrapped with pub/sub handler which abstracts the common functionalities of both the surftrain and website managers like joining and leaving of surftrain or webchat. These different functionalities are treated as topics of PUB/SUB architecture and are subscribed for tasks like joining, leaving, publishing of messages.

Database manager uses a customized functions on one of the javascript client for CouchDB called cradle to access the couchdb database. It includes the functionalities like initializing the database by creating it if doesn’t exist any, saving of documents, replication of database, querying the database through view and deletion of document from database.

4.4 Protocol

The protocol message is defined following the use cases according to requirement analysis of wussap system. This JSON formatted custom protocol is sent/received over socket.IO to/from node.JS numbers of followers can join and chat on the same topic of interest along the way.
4.4. PROTOCOL

Socket.IO falls back to suitable transport mechanisms as per browser environments. Web socket is one of the transport mechanisms supported by it. The message protocol is divided into two types as client initiated messages and server initiated messages. In client initiated messages, the client may get the reply from server if there is any whereas server initiated messages are originated from server directly and don’t expect any reply from client whatsoever. One of the custom protocol messages defined in JSON format looks like below.

![Client-Server Protocol Diagram]

**Figure 4.5:** Client-Server Protocol
Table 4.2: Protocol Message for User Registration

<table>
<thead>
<tr>
<th>Message</th>
<th>registerUser: {string: userName, string: userPassword}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reply</td>
<td>registerUser: int: statusCode</td>
</tr>
<tr>
<td>Arguments</td>
<td></td>
</tr>
<tr>
<td>userName:</td>
<td>The user name to register</td>
</tr>
<tr>
<td>userPassword:</td>
<td>The password to register</td>
</tr>
<tr>
<td>statusCode:</td>
<td>status code. 100 means success or errorCode is returned in case of error.</td>
</tr>
</tbody>
</table>

Description: Authentication manager receives username, password and creates a new user in Database and replies to the client with status code.

Example:

> {registerUser: {userName: "scott", password: "tiger"}}

< {registerUser:{statusCode: 100}}

4.5 Database

Database used for the prototype development is one of the document based nosql databases known as CouchDB. Some alternatives like CouchDB, MongoDB, Riak and Cassandra were taken into consideration. **MongoDB** is ruled out of further consideration because it provides only master-slave replication which doesn’t truly get distributed and scale horizontally as different slave databases contain same data as master database and slave database can be used for read operations only. So, it was not suitable for write intensive application which is our case. **Riak** is a good option for its real distributedness, fault tolerance and scalable approach but its important features like management tools, client support, masterless multisite replication(i.e. distributedness over large geographical area) come in enterprise versions and not in open source version[Ria11]. So, it was also not taken into further consideration. **Cassandra** is a database used by facebook and thus found to be suitable for social web applications. We stopped by cassandra because of its features as
4.5. DATABASE

horizontally scalable which means throughput of read and write gets increased with the addition of new nodes, decentralization as every nodes in cluster are identical and provision of fault tolerance due to its distributability which means unavailability of a single node in cluster does not affect functionality of application. But the available node.js client for communication with cassandra is not found to be stable enough and creating a custom client was going to be out of scope of this thesis work. So, it was also ruled out of further consideration. Thus, We chose couchDB as a standout backend for our application amongst other alternatives because of following advantages.

- It is fast in performance.
- Standalone document representation.
- Assurance of availability and scalability through replication, clustering and sharding. CouchDB provides availability and partition tolerance feature out of CAP(Consistency, Availability, Partition tolerance) theorem.
- CouchDB is of and for the web as it provides HTTP API that supports common http requests/responses directly out of the box and can be run inside browser directly. It communicates through HTTP protocol and stores documents in JSON format. Since almost every language can speak HTTP and parse JSON formatted messages, it makes the database to be integrated with any platform easily and it makes the database readily available for the web[Eer10],[ALS10]

HTTP protocol that CouchDB uses is stateless which means that the connection is not kept open for longer duration. Instead http requests are short lived which makes it able to support for large concurrent connections.

- It has master-master replication mechanism, suitable for distributability and scalability of application which we could not find on our other nosql database alternatives like riak, mongodb.

Schema-less document oriented database design is quite different from that of traditional relational database system. It represents data in stand alone document as shown in figures below unlike tables with relationship in between them as commonly found in RDBMS databases.

The example of document representation in couchDB looks like as shown below: So, as shown in the above representation, documents in couchDB are just the nested representation of JSON formatted data.
4.5. DATABASE

Figure 4.6: Document in CouchDB
Scalability

Scalability is a broad term that can have many definitions specific to each application and according to the context of discussion. In ideal case, Scalability means that the system should be able to handle the growth with the increase of throughput of the system. Sometimes, the terms performance and scalability become misleading and thus creates confusion. Performance refers to response time of the application or database under test whereas scalability refers to throughput or how does application behaves under large amount of concurrent users accessing the system simultaneously. Performance and scalability should go in neck to neck while development of a web application as performance is one of the aspects of Scalability. The web server with high performance can have more support for concurrent users than that of low performance one. But it can go upto some point only, after which we have to tradeoff between these two properties. And this is all dependent on our application design. When we talk about scalability, we refer two types of scalability:

5.1 Horizontal Scalability

It means the scaling of application with the addition of more hardware to support more load. It is also referred as scaling out in other terms. Here, we expect the application to support linear growth of concurrent users without degradation of throughput of the system on addition of additional server machines.

[Wiki11c] Though it is advised that increasing the capacity of single node is costlier than scaling
5.1. HORIZONTAL SCALABILITY

with more commodity hardwares, amdahl's law\(^1\) in terms of parallelization contradicts here. For example, if 50% of our application can improve in performance due to parallelization, then with 4 processors we can get, 1.6. And if we increase upto 8 processors, then we get 1.78. So, by double number of processors, we get only 11% of performance improvement. So, horizontal scalability is not always the viable option for scaling an application.

5.1.1 Vertical Scalability

Vertical scalability means the scaling of application with the addition of hardware capacity on the same node. It is also referred as scaling up in other terms. Vertical scaling can be achieved by adding more powerful CPU, more memory and upgrading other hardwares.

As memory is getting cheaper, Computers are getting faster and capacity of harddrives are also getting increased but there is limit of these increment as well which limits the vertical scaling also at some point.

\(^1\)Amdahl's law in case of parallelization defines if P is the portion of a program that can take advantage of parallelization, then maximum performance improvement that can be achieved by N processors is given by \(1/((1-P) + P/N)\).
5.2 Scaling of Prototype application

The prototype application is further refactored and profiled to improve its performance by caching some of the data in memory instead of explicitly calling from database that involves querying from views of couchDB which is an expensive operation in couchDB[Cou11b].

5.2.1 Profiling

Profiling is used to get the runtime information of the executed code for further analysis on each of the function or modules which are slow on execution and to see the frequency of execution of same function or block of codes[Hen06]. Profiling of the prototype for a single rundown of unit test showed that the read operation from database view was called 82 times with each call consuming 1ms of time and database write operation was called for 44 times each consuming around 1ms. So, reducing unnecessary call to database for reading through views could be the alternative to improve more performance and the solution was to use caching as mentioned in section below. The profiling was carried out with the use of opensource profiler tool nodeJS inspector.

5.2.2 Caching with Memcached

Memcached is distributed high performance memory object caching system. It is used to take off the load from database by caching the query results in memory which are frequently accessed and for queries that are expensive in execution. It is a key-value store in memory to boost the performance of dynamic web application.[Mem09] Multiple memcached servers can be used to store data in distributed cache. For our prototype example as shown in figure below, a memcached server is run on 11211 port, couchDB is run on port 5984 and the node.js server runs on 8000 port in localhost machine. The memcached client in node.js is used to communicate and perform operations like get/put data objects, freeing memory with memcached server.

One of the example section of code from prototype has been used in subsequent discussions to analyze the advantage of caching over fetching from database. Caching through memcached as

```
memcached.get(username, function(error, user){//code to handle login})
```

instead of querying from database as
5.3 DATABASE SCALABILITY

Figure 5.2: Memcached instead of Session Manager

db.view(config.views.User, username, function(err, res){//code to handle login})

showed some exciting performance improvement as discussed in the following paragraph.

Following figure 5.3 shows how above approach of simple caching applied in a block of code showed improvement in performance than reading it from views. This is of course, more pronounced when data size in views are more and the query becomes complex which requires extensive mapping and reducing as shown in figure 5.4. With 5000 requests, the example code took just more than 2 seconds to complete requests using couchDB views whereas with caching, it took only about 700ms to complete the requests. It is reasonable to get more widened response time with more requests on reading from views because the increased number of records with more requests makes the querying from views more slower. So, caching data, that too, on a distributed memcached cluster helps to improve performance of application significantly by taking off unnecessary overload of database.

5.3 Database Scalability

Though unavoidable, almost every dynamic web application are obliged to store their data in a persistent storage like database which are found to be the bottlenecks on web application for performance and scalability and it comes all due to improper database design. So the underlying backend database should always scale to get a good performance and scalability of web application. Thus, the backend for our prototype application, CouchDB has been analyzed and scaled in following sections.
5.3. DATABASE SCALABILITY

Figure 5.3: Reading from Memcached vs Views for 1000 requests

Figure 5.4: Reading from Memcached vs Views for 5000 requests
5.3. DATABASE SCALABILITY

5.3.1 Scaling CouchDB

CouchDB itself doesn’t provide the scalable and distributed solution as discussed previously. Though couchDB itself is not scalable, it can be leveraged to support more users and transactions with the combination of features like replication, clustering and a load balancer (can be either software or hardware) which can also be referred from the following benchmark results. So, Replication along with clustering and load balancing constitute the method for scalability of couchDB.

**Replication**  Replication in couchDB can be used for scaling of read request, write request and scaling of data across multiple nodes in a cluster. It can be carried out from readily available web interface called Futon or can be used commands as shown below from command line.

Following command is used for creating the database on couchDB that holds the replicated data.

```bash
curl -X PUT http://remote-desktop:5984/thesisdb-rep
```

Replication of database can be carried out on local node and remote node as well. Following is the replication command using curl that can be executed from command line. Here, CouchDB is provided the source and target address through http request. Then the replication of source to target database is taken care by couchDB itself.

```bash
curl -X POST http://localhost:5984/_replicate
   -H "Content-Type: application/json"
   -d '{"source":"thesisdb-test","target":"thesisdb-rep","continuous":true}'
```

CouchDB provides continuous replication which as its name implies, replicates the changes in database continuously without any need of restart of replication process again. It makes use of _changes API from couchDB to feed the changes noticed for replication. CouchDB supports unidirectional and bidirectional replication as well. It means that the source replicates the changes to target database and target database can replicate its changes to the source database as well. Thus through replication, the synchronized copies can be maintained in distributed environment. CouchDB updates the databases with changed items through replication like updated documents, deleted documents and newly added documents. The break in replication across large geographical area is unavoidable due to network problems, server crashes or any other catastrophic hazards but there is no panic in such state because couchdb triggers replication from the place where it left off
5.3. DATABASE SCALABILITY

avoiding the database inconsistency. Another important feature for distributed database is conflict management in case of same document being updated on different nodes which is managed by replication process through conflict detection and resolution. Here winning and losing versions are identified and both are maintained in different versions in database instead of removing the losing version.[ALS10]

In couchDB, there are two ways of carrying out replication process, namely, pull replication which is executed once and replicates just the current changes whereas the push replication which replicates the current plus the future changes. Push replication is triggered by the source node whereas pull replication is triggered by the target node. Filter mechanism available in javascript can be used to filter the criteria for replication to avoid replication of whole database [ALS10].

**Clustering** In database term, clustering means grouping of similar data within same cluster so that they can be accessed and served efficiently. The combination of clustering with partitioning can be used to make data distributed geographically which in turn scales the application. Clustering along with partitioning of database into shards avoids the possibility of disk space from being a bottleneck in scaling the application and also makes the application fault tolerant by keeping redundant copies over multiple nodes.

When we talk about scaling of database, we need clustering to distribute the heavy load across multiple nodes. As already pointed out in previous chapters, Clustering feature is not readily available with couchDB but there are some open source third party tools that provides such solution like couchdb-lounge, bigcouch etc. BigCouch has been used for making cluster of nodes and partitioning of database across multiple shards to make scalable, distributed and fault tolerant database. Bigcouch makes use of principles used in amazon’s distributed key-value datastore dynamo[Dyn07], as a clustering technology.

Following figure shows two remote nodes in a cluster which are capable of handling both read and write requests. These nodes are connected through distributed erlang system where we need to provide each node a long/short node name which converts a runtime system into a node[Erl05] and both nodes should share same cookie as well according to erlang security mechanism which acts as a shared secret password for communication between nodes within a cluster. Nodes in a cluster are distributed in dynamo like ring where each node is treated as masterless node. Following is some part of the configuration file of bigcouch that defines parameters for clustering:
5.3. DATABASE SCALABILITY

![Clustered remote nodes](image)

Figure 5.5: Clustered remote nodes

```
[cluster]
q=8
r=2
w=2
n=2
[httpd]
port = 5984
backlog = 512
docroot = /opt/bigcouch/share/www
bind_address = IP_ADDRESS
[httpd]
port = 5986
bind_address = IP_ADDRESS
max_connections = 2048
```

Hence, the clustering was carried out with 2 nodes and the number of shards across which database is spread is defined to be 8 (defined by parameter q). Parameter n defines number of redundant copies of document and n > 1 makes the fault tolerant cluster. w specifies the number of document copies to be saved before document is actually written. It can be less or equal to n. If n is equal to 1, it maximizes throughput because of limited reading and writing on single node only whereas w = n provides consistency as every nodes maintain the same data. Parameter r defines the identical copies that must be read before read request is ok. Provided r = 1 reduces latency because there is no need to synchronize response from multiple nodes on read request whereas r = n gives consistency on reading from every available nodes.[Sca11]

**Database sharding** Database sharding is a way of breaking down database into multiple pieces following some sort of predefined criteria. It scales the database along with increasing the throughput
5.3. DATABASE SCALABILITY

and performance of the database.[Sha09] Following futon representation of couchDB shows the

database partitioned into 8 shards in bigcouch. It uses consistent hashing for partitioning and

placing data across different shards.

![Sharded Database](image.png)

**Figure 5.6: Sharded Database**

**Load Balancing**  Load Balancer is a term used to represent hardware or software that distributes

the load across multiple machines following some sort of algorithm. Load balancer distributes the

load across multiple server machines evenly to have better use of these machines.

There are various types of load balancing softwares categorized as *layer 4* and *layer 7* load balancers

on the basis of layers of operation. Layer 4 load balancers work on TCP layer and makes use of

source, destination address and port number available in TCP packets. The commonly used load

balancing algorithm is round robin algorithm which divides the traffic evenly into different nodes.

Layer 7 load balancers work all the way up inspecting http headers also. They can be used to

balance load on the basis of url or parsing query string contents also.[Hen06]

It is one of the components required to develop scalable solution for web application. HAPProxy

has been used as a load balancing software to divide load across multiple database nodes in this

thesis work. An important section of the proxy configuration for HAPProxy load balancer looks like

following:

```
listen bccluster 0.0.0.0:5984
    balance roundrobin
    server node1 ip_address1:port1 check
    server node2 ip_address2:port2 check
```
6.3. DATABASE SCALABILITY

server node3 ip_address3:port3 check
listen bccluster_admin 0.0.0.0:5986
  balance roundrobin
server node1 ip_address1:admin_port1 check
server node2 ip_address2:admin_port2 check
server node3 ip_address3:admin_port3 check
This chapter evaluates the prototype and the database used for this thesis. The full rundown load testing includes rundown test of prototype application through database. The evaluation of couchDB database with the cluster of multiple nodes against standalone couchdb has also been realized in the following section.

6.1 Test Configuration

The hardware configuration used for the test are Dell studio laptop with Intel(R) Core(TM)2 Duo CPU T6400@2.00GHz, 4 GB RAM and HP Pavillion desktop with AMD Athlon(tm) 64 X2 Dual Core Processor, 4 GB RAM. The requests to different couchDB nodes were handled through a load balancer called HAProxy.

The test configuration for TSUNG was maintained in xml file. It includes configuration to simulate users, server configuration, load configuration with different arrival phases and configuration of server requests in different sessions. Configuration detail of test xml file can be referred in Appendix B.

6.2 Benchmark Results

This section focuses on the results obtained after benchmarking the couchDB database along with benchmarking of the prototype developed for this thesis. These benchmarks are obtained as a result of load testing using a highly distributed, concurrent open source testing tool Tsung.
6.2. BENCHMARK RESULTS

6.2.1 Graphs

Following graphs are the benchmarks that show simultaneous users at particular point in time, throughput as transactions per sec, memory usage and CPU usage.

Fig 6.1 is the simultaneous users support by a single couchdb instance with the configuration on testing tool as user arrival rate of 100 users/sec for total duration of 5 minutes. The test configuration includes the transactions type of GET/POST/PUT which means that the simulated session has been divided into writing on database, reading from views and some updates as well to take realistic approach on testing of database. Following graph (Fig 6.1) is the outcome of single couchdb under load test which shows that 400 users are simultaneously connected to couchdb database out of 1000 users requested from test tools at particular point in time. Here X axis represents the duration of test intervals in seconds and Y axis represents the simultaneous users at the corresponding time during the test.

![Figure 6.1: Single CouchDB Simultaneous Users](image)

Following figure (Fig 6.2) represents the transactions rate as throughput of couchdb database on the same test configuration as mentioned before at particular instance of test duration. X axis is the representation of test intervals in seconds and Y axis represents the transaction rate at particular
instant of test duration. Here, transactions represent writing on database, reading from persistent views and updating of some documents to have realistic approach on testing of database.

Following graph(Fig. 6.3) shows that the simultaneous user support has been increased upto 10 times(i.e. 4000 simultaneous users) than that of single couchdb instance as a result of scaling of couchDB database with three nodes in a local cluster. Here also X axis is the representative of test duration in seconds and Y axis as a number of simultaneous users.

Fig. 6.4 shows transaction rate with multiple nodes in a local cluster. It shows the slightly decreased transaction rate than that of single couchDB instance with same load. The result is reasonable because multiple nodes are run in a single machine and the support of simultaneous users have been increased by 10 folds and the latency of continuous replication in between the nodes brought down the throughput a bit which is around 100 transaction per seconds compared to 140 transactions/sec of single couchDB. The notable thing here was the proxy(i.e. load balancer) became bottleneck due to temporary overloading of user requests which also contributed on decreasing the transaction rate.
6.2. BENCHMARK RESULTS

Figure 6.3: Simultaneous users with multiple nodes in a local cluster

Figure 6.4: Transaction rate with multiple nodes in a local cluster

Following figures are the results of load test with requests of 200 users/sec for 5 minutes on developed
6.2. BENCHMARK RESULTS

As shown in figure 6.5, the node.js server isn’t found to be crashed for large number of simultaneous users (around 8000) unlike C1OK\[Dan11\] issue on traditional web servers which are unable to handle large number of simultaneous users. Here Y axis is simultaneous users support at particular instant of time and X axis is the duration of test intervals in seconds.

As shown in figure 6.6, the memory usage of nodeJS application is around 300MB which is significantly low compared to memory intensive counterparts like Java application, PHP application etc. Here the Y axis represents the freememory which drops by 300 MB during test execution and gets freed after the requests is finished and the X axis represents the test duration in seconds.

Figure 6.7 shows the transaction rate as throughput of the prototype application. Here Y axis is transaction rate and X axis is the time duration in seconds. Here the transactions represent the
6.2. BENCHMARK RESULTS

request of different URLs to node.js server to carry out various functionalities like user registration, user login, surfrain creation.

Graph 6.8 shows the CPU usage around 90 percent because the web server and database server were run on the same machine. But node.js application and couchDB database are not CPU intensive if each of them run on different machines. Here Y axis is the CPU usage in percentage and X axis is the test duration in seconds.

6.2.2 Bottlenecks

When doing full rundown test including web server and database with the test configuration of user arrival rate of 200 users per second, the node.js server crashed with the message as "(node) Hit max file limit. Increase "ulimit -n"" and the error was "Error: EMFILE, Too many open files". This was an issue related with limitation imposed by Operating system on maximum number of open file descriptors. On extreme concurrent load condition, the opened file descriptors were not properly closed before other file descriptors were opened.

With slightly more concurrent load of 500 users per second, the node.js server crashed with the message as "FATAL ERROR: JS Allocation failed - process out of memory" because data had been cached for long time without freeing the cache in some intervals. Following is the code snippet that is used for caching of user information. So, the solution is to clear the old cache data on some interval of time.

```javascript
var SessionManager = (function(){
    var sessions = {};
    return {
    'create': function create(username, station, sessionId) {
        return (sessions[sessionId] = new Session(username, station, sessionId));
    },

On analyzing database tests, it was found that some default system parameters like max_connections and ERL_MAX_PORTS poses the limitation on opening large number of open files. By default, erlang sets maximum ports allowed to be opened as 1024 which limits the amount of concurrent users for couchDB below 1024. These parameters allow to open file descriptors/sockets according to hardware capacity to enhance more concurrent users support in database.
6.2. BENCHMARK RESULTS

The load balancer (i.e. haproxy) found to be the bottleneck to scale with the increase in requests. With the increase in amount of requests from client, the proxy server queued some of the requests which could not fulfill connection to couchDB on time and in turn increased the load on the proxy. So, the http response code HTTP 503 and HTTP 504 from proxy server were the results. Error 503 is server being unable to handle the request because of temporary overloading and Error 504 is defined as proxy timeout according to www.w3.org.
Chapter 7

Conclusion and Future Work

7.1 Conclusion

As a result of this thesis work, We managed to come up with a prototype application developed in node.js and couchDB. The developed prototype is further refactored and profiled to improve performance by caching frequently accessed data using memcached server. Unit testing from the start of the development phase of each of the modules gives the assurance of stability of the developed prototype. The same prototype is further evaluated by load testing to ensure the applicability of prototype under severe load conditions. So, I can summarize that node.js provides framework that can be used to create scalable and high performance web server courtesy of its evented (asynchronous) approach of programming. Such web servers provide the platform for creation of highly scalable web application on top of them.

CouchDB is a suitable choice for the web application as it provides HTTP API which is suitable protocol for any languages and can be wrapped these core functionalities accessible through API to communicate with database. Though natively not supported, CouchDB database can be made scalable and distributed with some tweaks like replication, clustering and load balancing of the database. As obtained from evaluation results, the couchDB database on addition of multiple nodes got scaled for large number of users without much degradation of throughput of the database.
7.2 Future Work

Though this thesis has covered implementation of web application using node.js and scalability of database, there are more research areas that could be realized as a future extension which are described as follows:

7.2.1 Fault tolerance

Though caching data instead of maintaining in persistent storage reduces I/O operation for server, we cannot guarantee the fault tolerance of the system in such case. So, realization of fault tolerant system with good performance could be a challenging future extension.

7.2.2 Custom API

A custom API instead of using third party API to access database could be another future work related to this thesis. So, defining custom API on top of couchDB HTTP API to connect to database from node.js gives programmer more control over own API on avoiding possible bottlenecks rather than depending on external APIs.
References


REFERENCES


Appendices
# Abbreviations

This section contains the full form for the abbreviations occurred in this document in several places.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CAP</td>
<td>Concurrency Availability Partition tolerance</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transfer Protocol</td>
</tr>
<tr>
<td>JSON</td>
<td>Java Script Object Notation</td>
</tr>
<tr>
<td>MVCC</td>
<td>Multi Version Concurrency Control</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>TDD</td>
<td>Test Driven Development</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>OTP</td>
<td>Open Telecom Platform</td>
</tr>
</tbody>
</table>
Tsung Configuration XML

B.1 CouchDB test configuration XML

Following is the configuration xml file that has been used for benchmarking of couchDB database.

```xml
<?xml version="1.0"?>
<!DOCTYPE tsung SYSTEM "/usr/share/tsung/tsung-1.0.dtd">
<tsung loglevel="notice" version="1.0">
   <!-- Distributed Clients -->
   <clients>
      <client host="client-hostname" weight="1" maxusers="10000" cpu="2" />
   </clients>
   <!-- Server setup -->
   <servers>
      <server host="proxy-hostname" port="5984" type="tcp" />
   </servers>
   <!-- Load Setup -->
   <load>
      <arrivalphase phase="1" duration="5" unit="minute">
         <users arrivalrate="70" unit="second"/>
      </users>
   </load>
</tsung>
```
\[\text{Sessions setup}\]

\[
\begin{aligned}
\text{<session name="post" probability="50" type="ts_http">} \\
\text{<setdynvars sourcetype="random_string" length="10">} \\
\text{<var name="username"/>} \\
\text{</setdynvars>} \\
\text{<setdynvars sourcetype="random_string" length="8">} \\
\text{<var name="password"/>} \\
\text{</setdynvars>} \\
\text{<request subst="true">} \\
\text{<match do="abort" when="nomatch">201 Created</match>} \\
\text{<dyn_variable name="regid" jsonpath="$.id/"/>} \\
\text{<dyn_variable name="regrev" jsonpath="$.rev"/>} \\
\text{<http}> \\
\text{method="POST"} \\
\text{url="/couchscale"} \\
\text{content_type="application/json"} \\
\text{contents="} \\
\text{&quot;user&quot;:[} \\
\text{&quot;%%_username%%&quot;,} \\
\text{&quot;%%_password%%&quot;};} \\
\text{</http>}
\end{aligned}
\]
<?xml version="1.0" encoding="UTF-8"?>

<session name="put_get" probability="10" type="ts_http">
  <thinktime value="10" random="true"/>
  <setdynvars sourcetype="random_string" length="32">
    <var name="id"/>
  </setdynvars>
  <setdynvars sourcetype="random_string" length="10">
    <var name="username"/>
  </setdynvars>
  <setdynvars sourcetype="random_string" length="8">
    <var name="password"/>
  </setdynvars>
  <request subst="true">
    <http>
      <method>POST</method>
      <url>/couchscale</url>
      <content_type>application/json</content_type>
      <version>1.1</version>
      <contents>
        {
          "title":"my surftrain",
          "Type":"Surftrain",
          "currentStation":"http://www.umesh.com"
        }
      </contents>
    </http>
  </request>
</session>

<session name="get" probability="10" type="ts_http">
  <thinktime value="10" random="true"/>
  <setdynvars sourcetype="random_string" length="32">
    <var name="id"/>
  </setdynvars>
  <setdynvars sourcetype="random_string" length="10">
    <var name="username"/>
  </setdynvars>
  <setdynvars sourcetype="random_string" length="8">
    <var name="password"/>
  </setdynvars>
  <request subst="true">
    <http>
      <method>GET</method>
      <url>/couchscale/
          id="{{id}}"
          username="{{username}}"
          password="{{password}}"
      </url>
      <content_type>application/json</content_type>
      <version>1.1</version>
      <contents/>
    </http>
  </request>
</session>

<session name="put_get" probability="10" type="ts_http">
  <thinktime value="10" random="true"/>
  <setdynvars sourcetype="random_string" length="32">
    <var name="id"/>
  </setdynvars>
  <setdynvars sourcetype="random_string" length="10">
    <var name="username"/>
  </setdynvars>
  <setdynvars sourcetype="random_string" length="8">
    <var name="password"/>
  </setdynvars>
  <request subst="true">
    <http>
      <method>PUT</method>
      <url>/couchscale/
          id="{{id}}"
          username="{{username}}"
          password="{{password}}"
      </url>
      <content_type>application/json</content_type>
      <version>1.1</version>
      <contents>
        {
          "title":"my surftrain",
          "Type":"Surftrain",
          "currentStation":"http://www.umesh.com"
        }
      </contents>
    </http>
  </request>
</session>
<match do="abort" when="nomatch">201 Created</match>
<dyn_variable name="rev" jsonpath="$.rev"/>

<http
  method="PUT"
  url="/couchscale/%%_id%%"
  content_type="application/json"
  contents="{
    "user": ["%%_username%%", "%%_password%%"]"
  }">
  <http_header name="Accept" value="application/json"/>
</http>

<thinktime value="10" random="true"/>

<request subst="true">
  <match do="abort" when="nomatch">304 Not Modified</match>
  <http method="GET" url="/couchscale/%%_regid%%">
    <http_header name="If-None-Match" value=""%%_regrev%%"/>
    <http_header name="Accept" value="application/json"/>
  </http>
</request>

</session>

<session name="read_from_view" probability="40" type="ts_http">
  <thinktime value="10" random="true"/>
  <request>
    <http
      method="GET"
      url="/couchscale/_design/Admin/_view/Surftrain?reduce=false&amp;skip=0&amp;limit=10"
    >
    <http_header name="Accept" value="application/json"/>
  </request>
</session>
B.1. COUCHDB TEST CONFIGURATION XML

</http>
</request>
<thinktime value="10" random="true"/>
<request>
  <http
    method="GET"
    url="/couchscale/_design/Admin/_view/Surftrain?
    reduce=false&amp;skip=0&amp;limit=10&amp;stale=ok"
  >
    <http_header name="Accept" value="application/json"/>
  </http>
</request>
</session>
</sessions>
</tsung>