NYU | TANDON SCHOOL OF ENGINEERING

Computer Science and Engineering

Distributed Web Application

Final Paper

CS-UY 394X

Version 1.0

Document ID: FP-001

REVIEW AND APPROVALS

Printed Name and Title	Function (Author, Reviewer, Approval)	Date	Signature
Warlon Zeng	Author	April 25, 2017	*Warlon Zeng*

REVISION LEVEL

Date	Revision Number	Purpose
January 19, 2017	Version 1.0	Initial Release
May 8th, 2017	Version 1.0	Initial Release

SPECIAL THANKS

To *Hui Huang*, my fellow classmate since the 11th grade of Francis Lewis High School, a valuable member of the Sharit Project software engineering team, and my coworker for BNY Mellon. He has gone far and beyond in supporting me on this project out of due friendship and respect for materialization of greater knowledge. He and I have, in addition to discussing high quality systems design information, facilitated information to pave each other's gruesome job hunting.

To *Kurt Rosenfeld*, my graduate distributed systems industry professor and NYU-Poly (2010) alumni who took me in as his student and distilled upon his class years worth of Google SRE experience. He, who discussed the very definition of databases, servers, and the computer science realm, exposed these fundamentals under the light of scalability in what is well-known in the corporate world from Google: Chubby, Bigtable, and Paxos. Kurt is an absolutely great industry professor teeming with Big 4 experience of whom I cannot learn distributed systems closer than from the source. A true educator concerned with learning through experience.

To *Phyllis Fankl*, my databases professor and advisor for this project who taught me fundamentals of MySQL databases and web applications for it. Professor Frankl is also the director of the B.S. CS program and provided gratuitous resources in my learning for system designs.

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

The purpose of this project is to apply coding abilities in transitioning theoretical knowledge into usable software. By writing software, hard work in coding will help improve competency as a programmer. This project will also prove to capitalize academic milestones, particularly in the field of software development and engineering and distributed systems. Within the NYU Tandon School of Engineering, knowledge from these courses will be noted upon: CS3083 Introduction to databases, CS4793 Computer Networking, CS4513 Software Engineering, CS4523 Design Project, and CS9223 Distributed Systems. Also a myriad of open-source technologies will be noted of.

2. SCOPE

2.1 Objectives

The final product was released on April 21, 2017. The final paper is scheduled to be released May 8, 2017.

Project Proposal	January 11, 2017
Implementation and Demonstration	April 21, 2017
Final Paper	May 8, 2017

2.2 Product Overview

The finished product will complete as much features as possible in the beta stage. The software will be fully operational as a distributed web application. Primary objectives includes means of insuring consistency and replication for high availability using modern open-source software. Secondary objectives, developing a native mobile application in either iOS or Andriod, was not completed. However, with a completed

REST API, an iOS or android application should have no problems consuming a REST API via HTTP connectivity.

This product can be accessed as a web service. Standalone access can be made possible from the client by sending http requests to the API with the appropriate networking protocols and headers.

This product can be seen as a continuation of the senior design project, Sharit, from CS-UY 4523 of Team B6 from Fall 2016. Sharit is a centralized web service using only a server built from NodeJS, PostgreSQL, and Nginx. Features of Sharit can be described from prior senior design documentation, GitHub, and online via AWS.

2.3 Project Overview

The product was completed following a code-and-fix model: rapidly writing code alongside reading API's, guides, and relevant papers/blog posts. A number of open-source softwares was examined and compared to with more established, mature software for building the architecture of the product. Architecture of high ranked Alexa sites, such as Google, Reddit, and Facebook, was studied and its architecture similarly applied to Sharit, featuring no single point of failure except the DNS.

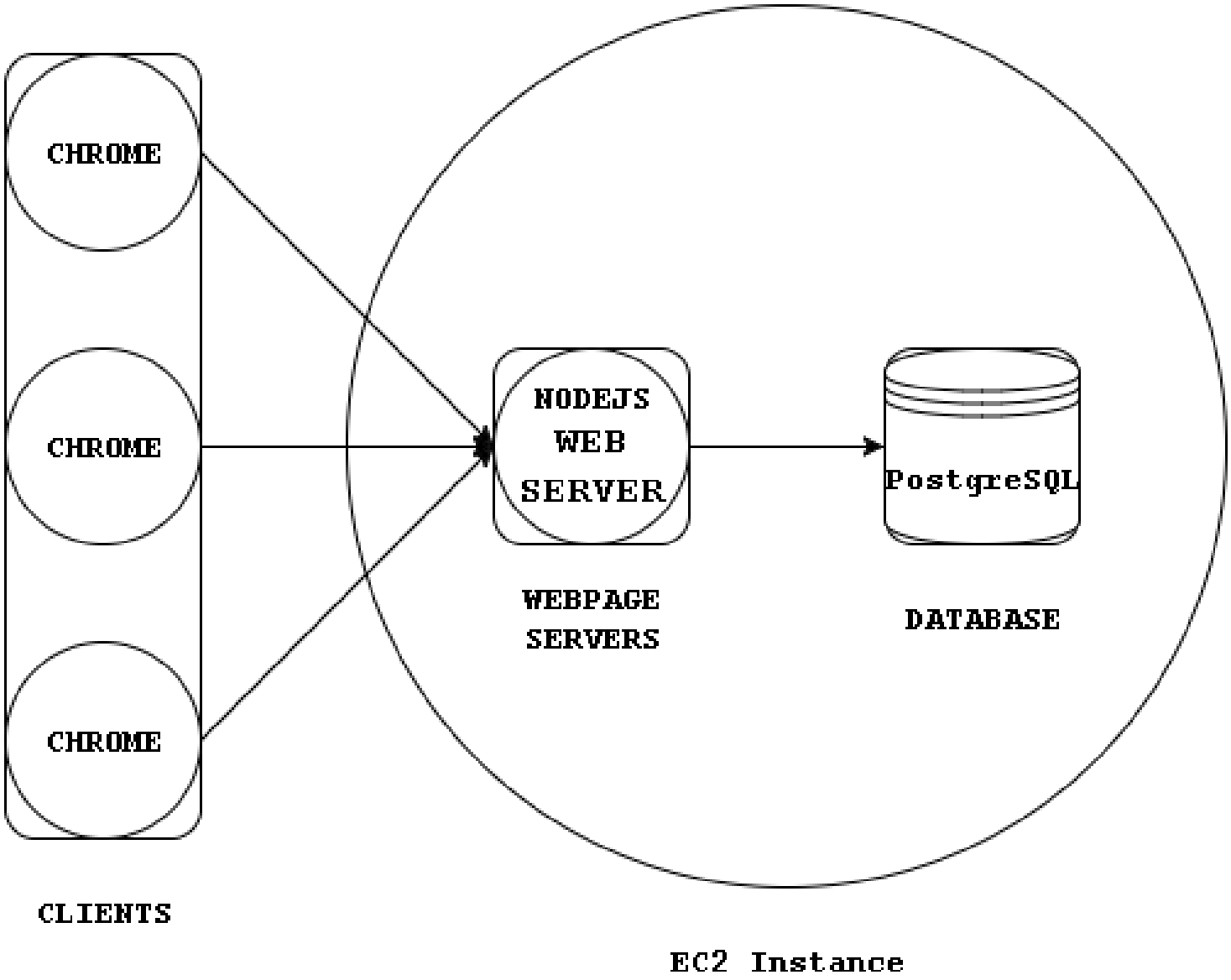
2.4 Document Overview

The project proposal describes objectives for this project to be accepted as an independent research/project for CS-UY 394X. Software requirements and prior knowledge are also noted to complete this project.

The codebase is available online directly on GitHub. Details about implementation was documented in code and on paper. Demonstration was performed on April 21.

The final paper is presented as a mix between software documentation and project paper summarizing knowledge obtained from the semester. Knowledge will be drawn from use of open-source technologies, software development and engineering, and distributed system concepts.

3. DEPLOYMENT ARCHITECTURE DIAGRAM

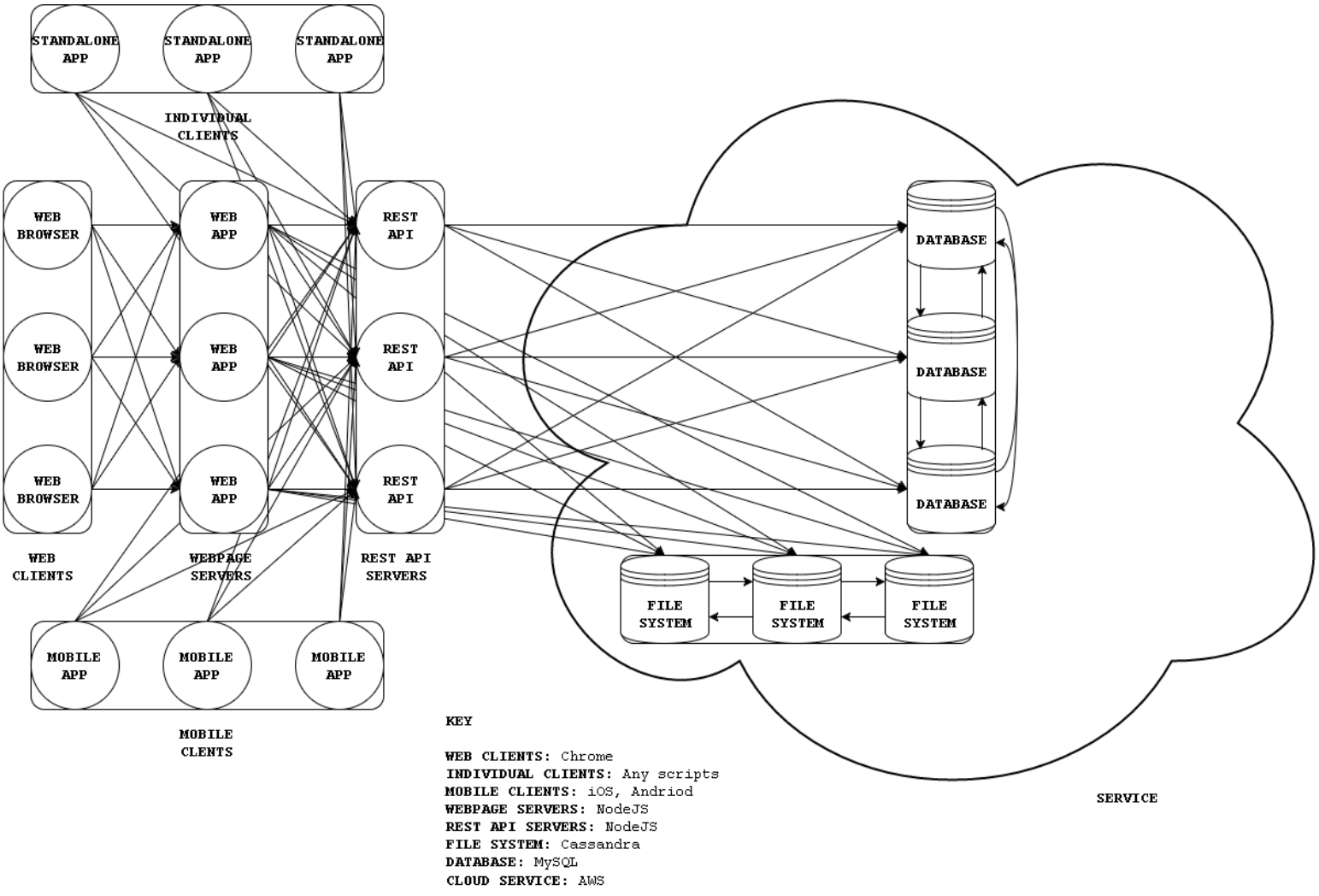


3.1 Alpha Stage Deployment Architecture Diagram

Sharit, as it is from CS4523, follows a simple architecture where clients use their web browser to connect to a webpage server, served by nginx and web application built by NodeJS. The client receives webpages served directly from the frontend with calls connecting to the PostgreSQL database. As the webpage rendering and database calls

are tied together, only HTML is served to the clients. Mobile use becomes unfavorable as native mobile do not necessarily receive HTMLs only. Native mobile applications, built in iOS or Android, have their own means to display data. In addition, all these servers are installed and deployed into a single machine, making processing and availability extremely vulnerable to one instance.

3.2 Beta Stage Deployment Architecture Diagram



In this completed architecture to restructure Sharit into Distributed Sharit, web clients can connect to any webpage server. The server will call upon any API server for features. Based on the feature request, the API will make the necessary database calls to fetch the information from the database and perform any calculations necessary.

For mobile application users, their traffic flow is similar to that of web users, only except their traffic directly calls the API. Depending on their application, mobile application developers can have their own servers to display data.

For individual clients, such as scripters, developers, and the like, standalone users can process API data according to their preferences. A common example is collecting data from the API and aggregating statistical data. Another example is running a custom built server connected to SMTP and send emails to themselves or others of a particular feed of API data.

4. FUNCTIONAL REQUIREMENTS ANALYSIS SPECIFICATIONS

4.1 Functional Descriptive Detailed Requirements

In this section, functional requirements analysis specifications details the product itself and the kind of service the web application provides. Distributed Sharit is a forum-like website designed as a mix between Reddit.com and Piazza.com, combining the democracy of popular votes, freedom of speech, and attention to education. Users can create subdomains for a particular field, create threads relevant to the field, and post comments to a thread. Threads can also contain files. If a user finds a good post, whether thread or comment, they can upvote that submission, or downvote if it is a bad post.

This distributed version of Sharit retains most of the previously coded features established in previous software CS4513/CS4523 documents. One significant change to Distributed Sharit was the ability to view the forum-like website without requiring to register and login to an account. However, as in the previous version, delete routes are not implemented.

4.2 Component Architecture

When Sharit was first conceived, only one AWS EC2 instance was used to host the web application. The web application consisted only a webpage server built in NodeJS and a Postgresql database. The webpage server made calls to the database and rendered the results. Both Sharit and Distributed Sharit can be best described as CRUD (Create, Read, Update, Delete) apps, a specific web application service that essentially only fetches information from the database and render results.

In the making of Distributed Sharit, the PostgreSQL database was replaced with MySQL. Files are no longer kept in the primary database cluster, but kept in a separate database cluster, Cassandra, for file storage only. To the end user, this change in the component architecture does not affect any features, but makes the infrastructure highly scalable and flexible.

5. NON-FUNCTIONAL/OPERATIONAL REQUIREMENTS

5.1 Computer Resource Requirements

Distributed Sharit uses ~16 AWS EC2 free-trial commodity instances. For using Distributed Sharit as a web service, an average market phone or laptop is acceptable. The application is developed and developed in Ubuntu 16.04 LTS. For quality and speedy development, a dual monitor or a high number of monitors present are greatly preferred.

5.2 Software Resource Requirements

A number of modern open-source technologies were used to develop the web application:

	Softwares
Web Browser	Chrome
Web server	Nginx
Webpage Server	NodeJS - Express
Mobile App	Android iOS
REST API	NodeJS - Express
Cache	N/A
Database	MySQL

File System	Cassandra

The web application expects the the average user to use Google Chrome for their web browser. The webpage server is supported by Nginx, added ontop of NodeJS web application for proxying and upstreaming nodes. The REST API server, also known as the backend, is built from NodeJS with Express framework. The PostgreSQL database was replaced by MySQL, and the file storage is handled by Cassandra. Caches were once considered but are not implemented in this design.

6. SERVERS

6.1 HTTP/TCP Applications

HTTP and TCP are the primary protocols used to send information across the network. From Wikipedia, “TCP provides reliable, ordered, and error-checked delivery of a stream of octets between applications running on hosts communicating by an IP network”. While not distinctly used in Distributed Sharit, it is an underlying protocol for many world wide web and file transfer services and found in the transport layer of the OSI model. SSH, a network protocol using a standard TCP port 22, intuitively allows an user from a local machine connect to a remote machine. This becomes extremely useful for connecting to multiple EC2 instances, where each terminal of the local machine is logged into an EC2 machine and can have as many machines on one screen as the resolution of the screen allows.

HTTP is the primary protocol used to send and receive information from the webpage servers and the API. From Wikipedia, “HTTP is an application protocol for distributed, collaborative, and hypermedia information systems”. Although HTTP can be used for a number of services, it is best known for transferring enriched HTML and JSON objects. In Distributed Sharit, HTTP is used for transferring HTML with CSS and Javascript. HTTP is also used by the API server to serve structured data, called JSON, for the frontend and individuals to consume. Transferring files can also be done in HTTP, where contents of a file will be in a field of the JSON object.

6.2 UNIX Environment

Knowledge of the UNIX operating system is good, but not required for software development. The common unix utility commands used in Distributed Sharit are ls, cd,

sudo apt-get, and sudo chmod. Sudo apt-get is for downloading software and sudo chmod is for changing permissions of a file or folder. Most development softwares and their latest versions can be downloaded from the APT package manager that is commonly shared among linux distributions. Also, git commands are used often to save code progress.

7. FRONTEND REQUIREMENT SPECIFICATIONS

7.1 Javascript

Javascript in this day and age is more or less the go-to language for web development. The library ecosystem, when compared to other languages such as Python and Java, has a collection of frameworks and connectors so vast it can be described as having too much while being constantly updated and depreciated. From Wikipedia, “Javascript is a high-level, dynamic, untyped, and interpreted runtime language”. For a long time, Javascript was often used in the client side of website content production serving as client functionality for HTML DOM elements. Alongside HTML and CSS, the three form the core technologies to serve world wide web content.

A specific implementation of Javascript, NodeJS, made Javascript further significant not only in delivering web content, but full-fledged web applications. NodeJS is a javascript runtime built from Chrome’s V8 javascript engine. NodeJS, being event-driven, non-blocking I/O, has proven to be very fast for handling concurrency requests but suffers at performing CPU-intensive tasks, where multi-threaded models are faster in that regard.

In developing the webpage servers, also known as the frontend in Distributed Sharit, all functions are, by default, **asynchronous**. Typically this is because when making a networking or system call, depending on internet quality, hops, local or VPN, etc., it may take 10-1000+ milliseconds to complete and consume results. In networking, sometimes it may take up to a minute if the handshake protocol is dragging on only to find data being transferred bits by bits or none at all. In system calls, opening, reading, and writing from a file is dependent on CPU speed and possibly available RAM. In the development side, the asynchronous execution flow can be synchronized by writing dependent code in the **callback** of the asynchronous function parameters.

This way, when asynchronous function is done in an unknown amount of time, the rest of the code can continue starting from the callback.

The REST API, also known as the backend in Distributed Sharit, is also in built in NodeJS and so the same rules of asynchronous callback applies. In the case of the backend, database calls requires a callback to process fetched results. Database calls that are independent of each other, such as static data, can be fired off asynchronously and returned altogether using a **barrier** for better performance.

The development environment can be easily set up using express-generator, which sets up middleware, error logging and the express framework.

7.2 Authorization

Authorization in Distributed Sharit is also known as a **session**. A session is a collection of developer-defined data and a cookie both unique to the connection. This means that, when assessing session data, only the session data set in the specific client will be seen. Although it is possible to view session data of another client, the session ID and access to the **storage of sessions** is required. The storage of sessions, implemented in Distributed Sharit, is a Redis database containing session objects. By default, sessions are stored in the memory of the server saving the sessions. The servers handling the sessions are the frontend. If the default option was chosen, sessions would get destroyed if the frontend servers are destroyed. To the end user, at any moment of time the user could be forcibly logged off. By using a Redis database, sessions get stored externally and independent of the webpage servers. If one web page server gets destroyed, the user will still be logged in and accessing the web service through another available webpage server.

Having the code to handle sessions in the backend will not work as intuitively as it will in the front end; the backend will not be able to remember the session data in between routes. This may be in part of networking, where each HTTP AJAX request made is unique. Individual users attempting to make use of logged services from scripts may not be able to fully utilize the POST routes, which checks if the user is authenticated already or not. The usage of tokens or API keys may be necessary to access logged services in the backend. Once logged in, a token or API key is generated, stored in the session server that the API has access to, and given to the user. The user would attach the token or API key for every feature requiring authentication, instead of authenticating the user everytime with username and password.

In implementation detail, express-session framework was used to set up sessions. Other web applications involving the use of sessions often use **passport** on top of express-session. This allows various authentication strategies such as login by

Facebook, Google, API key, oauth2, etc. A token is generated and stored as a cookie in the end user's browser until the determined expiry time. One interesting advantage in using passport over express-session is that users prefer not to create a new account unless absolutely necessary. In this regard, having the ability to login via Facebook or Google becomes ideal.

7.3 API Connectivity

The frontend servers of Distributed Sharit act as a collection of standalone HTTP clients. To provide the web application's service to the end user, the frontend makes an HTTP request to the API and renders the results accordingly. For each feature request, a route is specified in the frontend and backend servers. The user accesses the route, submits an HTTP **AJAX** (Asynchronous Javascript And XML) request that pings the API of the specific route with an attached payload, calls the database, returns the result, and the frontend renders the result.

When making a networking protocol, a few notable error responses are common in debugging: **200** for success, **304** for not modified, **404** for not found, and **500** for internal error. Response code 200 is the most ideal, which implies the route is working. Response code 304, while may display a page, may not be entirely accurate: the user may have refreshed the page and hit a cache, but in actuality the route is not rendering the updated results but a snapshot of the results cached earlier by either the user's browser or Nginx. Response code 404 is a complete error by the user, who tries to specify an undefined route in the server. Response code 500 is like 404, but the fault lies in the servers that accepted the request but failed to process the user data.

For each feature, there is usually a **GET**, **POST**, and **DELETE**, route. In the most basic terms, GET requests take in data specified in the URL of the webpage, and returns a rendered webpage. POST requests take in a payload of data from an HTTP AJAX request, and may or may not render a webpage. DELETE routes are like POST routes, but specific in deleting a resource from the server instead of creating one. In the implementation of Distributed Sharit, the routes are usually organized in parallel: a GET request will call a GET route of the same feature in the API, a POST will call a POST, etc. However, it may not necessarily be like this. Since the frontend is essentially a collection of individual HTTP clients, a GET route can make a POST HTTP AJAX request to the API. One such use case for this is sending login information that is invisible to the user but entirely unknown to the backend servers.

GET routes are used from the frontend to obtain rendered webpages using data from the URL. The data in the URL can be parsed directly in the express framework or tokenized using **REGEX** (if there is a "submit?=" or the like). POST requests are generally used to create new resources but may also be used for identity checking

through payload. Although DELETE routes are not implemented in Distributed Sharit, resources such as sessions objects are possible targets for deletion on logout.

7.4 File System Connectivity

File systems used in Distributed Sharit are accessed by the frontend servers. When the end user uploads a file, the contents of the file is stored as binary data in the memory of the frontend servers. From there, the binary data, an array of bytes, is sent to the file system of Distributed Sharit, Cassandra. In databases, files, a chunk of binary data, is also known as a **BLOB**. Metadata of the file, such as filename and an unique identifier of the filename, gets sent to the API which stores that information in the primary database. As the size of files get very large, amounting to an ideal maximum of ~15 MB, it is advisable to quickly store the binary data into a file system and delete the memory in the frontend servers.

7.5 Webpage Rendering

In Distributed Sharit, the job of the frontend servers is to act like a webpage server. The webpage servers' only task is to render webpages. For applications starting out, using a **templating engine** is often the first step for dependency injection into HTML. Applications may choose to service raw HTML by embedding HTTP AJAX scripts into them. However, a major downside to this is that data - sensitive and private included - that is needed to process further services, may be exposed to the public. For example, the information to access Cassandra is hidden in the servers since it is unnecessary and potentially dangerous for consumers to know how to interface with the file storage outside valid service methods.

Frontend servers can also provide other services, such as streaming file downloads, chat services, and in the case of games, online multiplayer. Distributed Sharit allows downloading a file by buffering and streaming binary data. In real production systems, and some amateur-learning web applications, frontend frameworks such as Angular 2 and React, developed by Google and Facebook respectively, emerge as popular javascript open-source frontend web application frameworks. Both of these frameworks solves dependency injection, web rendering, and also conforms to a modern web rendering convention: the **MVC architecture**.

8. BACKEND REQUIREMENTS SPECIFICATION

8.1 APIs

Web APIs are distinctly distinguished as belonging in the backend of any production web application. APIs are also the primary interface to service the web application. Developers develop the API in accordance to the web application's business logic. REST APIs in particular accepts standardized HTML requests with the appropriate headers in a specified route such as GET, POST, DELETE, etc. After processing the HTML request, the API sends back a JSON object. APIs can also send back other primitive data structures, such as a single string, or advanced data structures such as a buffer stream. In the past, before JSON emerged as the standard choice for data exchange, XML was often the most used data representation partnered with the SOAP API architecture.

In regards to writing and documenting an web API, there are a variety of open-source frameworks that simplify some repetitive work. For javascript, **ExpressJS** is often the most used for writing APIs as the framework is designed to be unopinionated and flexible. **Swagger**, the most popular and another javascript framework to develop web APIs, requires a definition file to create the skeleton for the API routes. In terms of added benefits, Swagger provides a friendly web interface to quickly validate API routes. If developing the API in Java, **Java Spring** is one of many frameworks to develop the API in. **Apache Thrift**, originally developed by Facebook, creates APIs in variety of languages - C++, Java, NodeJS - to be interfaced internally by the application and not necessarily by HTTP (not a web API).

8.2 Databases

The primary database which stores most of the web application's relational data is MySQL. The database used in Sharit was PostgreSQL, and fit the needs for production systems requiring relational databases. PostgreSQL, compared to MySQL, can organize tables under "domains", which keeps tables more organized and hierarchized. While being very efficient and considered the norm for production databases, PostgreSQL was swapped out for MySQL for replication capabilities.

MySQL has built-in replication strategies that, for Distributed Sharit needs, uses Master-Slave replication.

Master-Slave replication can be best explained by the Google Whitepaper for the **Chubby** Lock service. In a cluster of servers, one act as the master and the others as the slaves. The master propagates writes to its database, its commit log, and then to other slaves and their databases. The API, which connects to the database, reads from the slaves and writes to the master. If one slaves dies, the API can read from any other slave. However, if the master dies, another slave will have to become master. The process of electing a slave to become the new master is called **leader election** algorithm, which takes about ~30 seconds to complete for MySQL auto-failovers. The slave with the most up-to-date commit log is the slave that is likely to get the most votes from other slaves and subsequently becomes the new master. However, there are rare occurrences where two masters can be elected. If latency between slaves cannot be be determined, a majority of servers did not receive the most up-to-date commit log, etc., the state of having two masters elected is a called "**Split Brain**". MySQL does not have auto-failover built-in, but can be acquired through additional open-source software. PostgreSQL replication abilities are supplemented by the Citrus company's open-source and proprietary support.

In Google, Chubby manages millions of locks and is used by Google Bigtable, the **NoSQL** database that uses Chubby for its atomic row-level operations. The Google Whitepaper for Google Bigtable influenced many NoSQL databases including Apache Cassandra, originally developed by Facebook, and Amazon Dynamo. Cassandra follows a similar internal architecture as Google Bigtable, where writes update the the commit log and memtable and reads looks up a merged view of the memtable and SSTables.

In the development side, Cassandra's nodes are all equal; each node can accept writes and reads and can propagates its changes to a set number of other nodes. Large tables are partitioned into smaller tables, scattered across many nodes, and the partitioning key is used to efficiently search smaller tables. Horizontal partitioning, also known as **sharding**, is spreading data row-level data to different tables and is usually seen in practice for geographical locations, i.e., East and West servers. Sharding can be decided at application level or database level ("automatic partitioning"). When serving database queries, Cassandra uses a query language similar to that of SQL, CQL, where any data node is capable of serving the client API. Being a NoSQL database, Cassandra is extremely scalable in serving data readily but at the cost of ACID compliant capabilities such as joins and relational data modeling.

Since databases are often the core component of services, whether web or other software applications, choosing between relational and NoSQL databases for projected scalability is often spoken upon. Relational databases offer powerful SQL which

empowers many queries of a data model and in turn is easier to supply future feature requests. For databases in companies starting out and even surviving for decades, relational databases are still used for the core services and for data requiring a relational model. Services that do not require such strict consistency and needs scalability can be replaced with NoSQL; i.e., SQL and NoSQL databases in one application. My professor, Kurt, of Distributed Systems said that the best way for companies requiring NoSQL and not spend so much money changing databases is to build it in NoSQL in its early stages. This perhaps may be viable to companies whose services are more or less defined and set in stone forever.

In recent years, NoSQL was proven to be the scalability solution to many relational databases. One most recent upgrade to databases in the past few years is the notion of in-memory databases, in which most, if not all data is stored in the memory instead of the disk. One notable drawback is if all nodes of a cluster goes down, then all data for that cluster may be lost. While MySQL uses innodb engine for its database handling, the in-memory database cluster version, MySQL NDB Cluster, uses ndb engine and achieves better performances: <1 second for auto-failover, auto partitioning, preconfigured high availability, etc. MySQL NDB Cluster and Cassandra falls in the availability and partition sides of the **CAP Theorem**, which states a distributed system cannot have 100% of consistency and availability.

8.3 File systems

File systems manage file storages in an application. While the operating system stores the file into the hard drive and/or in-memory, there can be many implementations to a go about creating or using a storage for files. When picking out a file storage to store file objects, Cassandra was chosen for its ability to replicate data easily. With no relations other than file data and an unique identifier, Cassandra works well as a database for file storage. However, in a larger scale production system, a proprietary file system or open-source HDFS are ideal for storing files. Facebook, Google, and Amazon developed haystack, GFS, and S3 for their file storage respectively.

HDFS, for example, is built for distributed purposes with **map-reduce** in mind by reducing a file into blocks, replicating them, and spreading it across nodes. Reading from a large sized file becomes remarkably faster by parallelizing I/O to read the blocks of the file from several nodes at once.

8.4 Caching

Caches, and can also be known as proxies, stores results in a storage for a set amount of time and updates stored results whenever needed. Typically caches are in

their separate servers but are used by either the frontend or backend servers. Database results can be cached by the backend into the cache, as a key-value pair, by setting the string of the SQL as the key and the results of the SQL at the time as the value. Popular open-source implementations of caches include Redis and Memcached.

Caching heavy traffic service requests can be useful for saving a massive number of calls to the database. However, in a distributed system, caches are ought to be updated everytime a write to the database is made. This, in turn, mitigates the usefulness of caches if it needs to be updated everytime. In real world production systems, services that do not require strict consistency should not have its cache instantly updated, but updated after a period of time. Some examples of this is Google Youtube's viewer count for popular release videos and the voting scores of Reddit posts.

8.5 Machine Learning

Applications that managed to acquire **Big Data** can specialize its services and plan for feature requests that are relatively long or impossible to achieve without big data. In most applications, a set of rules is simply not feasible for identifying, for example, the face of a person in a photo or the sentiment of a piece of text. By using big data, data of similarity can be inferred upon by statistics and can be applied to predict the course of incoming data. Such feature requests is Google's personalization for its search engine, Amazon's recommended items for consumers shopping retail, and recommendations for movies, art, etc. Using another company's big data may not actually work, as a company's audience is seasoned within its community and applying principles from another community can be faulty in both model training, assumption, and accuracy.

Machine Learning feature requests are typically implemented in their own servers with access to the API or database. The most common languages are Python, Java, and R, with Python leading in production systems using open source TensorFlow, Scikit-Learn, Numpy, and Pandas. Although a great deal of machine learning knowledge is needed to create training models, popular training techniques such as SVM, decision trees, and even neural networks simplifies the process to create training models. Data must be cleaned everytime it is collected from the API or database.

9. NETWORK ORGANIZATION

9.1 Cloud Hosting

Creating a cloud network requires a reasonable knowledge depth in networks, virtualization, operating systems (Unix) and open-source softwares such as OpenStack and VMWare. Not only that, a number of criterias needs to be considered, i.e., projected workload, permissions, VLAN/VPN configurations, etc. For Distributed Sharit, AWS (Amazon Web Services) was leveraged due to its friendly interface and cost-effectiveness for replication and general purpose needs.

At this time, AWS offers a variety of cloud services servicing its public cloud to customers. Of these services, EC2 (Elastic Compute Cloud) instances and Route 53 are used. AWS EC2 instances are essentially consumer-grade virtual servers on demand and are specifically designed for handling service requests such as REST API calls or computation, i.e., machine learning. Databases and file storages should be installed in servers with high storage capabilities, in what is known within Amazon as S3. File storages that are distributed geographically are also known as **CDN** (Content Delivery Network) and are meant to service media or arbitrary large files from a server that is closest to the client.

In Distributed Sharit, EC2's are used for all its services and its connectivity configured in Route 53 and Nginx web server. For each EC2, MySQL, Cassandra, and NodeJS were installed if it was a webserver or a database. Also each individual instance requires its security group, more simply known as the firewall, configured to accept certain protocols from certain ip addresses. In a production environment, exposing these ports to allow requests can be potentially dangerous but for testing purposes connectivity can be easily verified by accepting all requests - more frequently known as "0.0.0.0".

9.2 DNS routing

DNS (Domain Name System) is part of IPS (Internet Protocol Suite) that is relevant to the OSI model and is a hierarchical decentralized service that relates domain names to ip addresses. DNS lies on the application layer, and perform its tasks by acting as a server that answers queries for a domain from DNS records in its database. Specifying an alphabetical named address queries the DNS server and redirects the

connection to the appropriate ip addresses specified under the name in the DNS records.

In Distributed Sharit, the frontend and backend servers' IP addresses are listed under the domain name and the subdomain of the domain, respectively. In this example, "distributed-sharit" and "api.distributed-sharit" are is the domain and subdomain and each lists 3 different ip addresses that routes to any server readily available in a round-robin fashion. Other strategies to distribute load across servers from the same DNS can be configured with a load balancer, and such methods can be by graphical, latency, or weighted, etc. Databases typically do not require a DNS, however CDNs may use a DNS if getting and storing files was written with a web server. Reddit.com., for example, redirects Reddit stored media to i.redd.it with the appropriate hash for the file.

10. REFERENCE DOCUMENTS

10.1 SOFTWARE DOCUMENTATION

Team A6 System Requirements Specification, Version 2.0, March 23, 2016.
Team A6 System Analysis Specification, Version 1.0, April 18, 2016.
Team B6 Requirements Analysis Specification, Version 1.0, October 4, 2016.
Team B6 Software Design Description, Version 2.0, December 15, 2016.
Project Proposal, Version 1.0, January 11, 2017.

10.2 GITHUB REPOSITORIES

https://github.com/rasagle/Sharit, Version 1.0, Alpha Stage, Minimal Viable Product.

https://github.com/WarlonZeng/Distributed-Sharit, Version 2.0, Beta Stage, Minimal Viable Product

10.3 GOOGLE DRIVE

https://drive.google.com/open?id=0B0aXclHx5n42VFlOdURvSGRRQms, All Original Sharit Software Documentations, 2016.

https://drive.google.com/open?id=0B8SEHfpAjaYCYXlfWkdCUWZxZGc, Distributed Application Software Documentations, 2017.

10.4 ONLINE SERVICE

http://sharit.warloncs.net/, or http://ec2-52-91-21-93.compute-1.amazonaws.com/, Version 1.0, Alpha Stage, Minimal Viable Product.

http://distributed-sharit.warloncs.net, Version 2.0, Beta Stage, Minimal Viable Product

http://api.distributed-sharit.warloncs.net, Version 1.0, Beta Stage, Minimal Viable Product, API.