Linnæus University
Sweden

Computer Science, Independent Project

Scaling real-time applications

Exploring approaches for scaling synchronization in real-time applications

Author: Kristoffer Lind
Supervisor: John Häggerud, Johan Leitet, Mats Loock
Examiner: Johan Leitet
Date: 2015-06-10
Subject: Computer Science
Level: Associate's degree
Course code: 1DV42E

Abstract

The aim of this study was to explore approaches for scaling synchronization in real-time applications. It was a literary study where books, research papers, technical documentation and articles were examined. Two solutions were found that could be combined or customized into a fully tailored solution.

First solution is a backplane, an external database or service that replicate data-messages to all servers. The other solution is called channeling, where a load balancer is used to group users that share data on the same machine. Other solutions can be formed by combining or customizing these approaches. Channeling solution could group users into a subsystem that uses a backplane. Customizing is also a possibility, backplane could instead use routing to be more efficient, which could be a solution for geographic spread.

Application servers should be stateless and a Design-Implement-Deploy (D-I-D) process is recommended for resource efficiency.

Keywords

scaleout, scalable, scalability, synchronization, real-time, component, application, client, server, communication, signalr, socket.io

Contents

1 Introduction

Web usage is increasing [1] and many small start-ups meet sudden success, which results in increasing traffic load. Surviving this spike in traffic demand is essential for success. Applications that are not prepared to meet this demand might miss their chance for success as users leave or never find the service due to unavailability [2].

Scaling is the practice of allowing for different traffic loads, vertical scaling (up, down) means deciding hardware (CPU, RAM..) for individual machines, horizontal scaling (in, out) is the decision of how many machines the load should be spread across [3, Horizontal and vertical scaling].

If no preparations are made, vertical scaling will probably be the only option. Scaling up quickly becomes inefficient or impossible, as machines become increasingly expensive to upgrade with rising performance levels and upgrading indefinitely is not possible [4]. Hitting this wall with no additional options is risky for businesses. Horizontal scaling also has the advantage that cheap hardware can be used, malfunctioning units can be thrown out and replaced instead of being repaired, which can be expensive, time-consuming and possibly result in excessive downtime [4].

General ideas about scaling have not changed much, but the web has transformed, user expectations keep rising and real-time applications seem to become increasingly common [5]. In this context, a real-time application refers to an application where a synchronization component pushes data to related clients as content is updated.

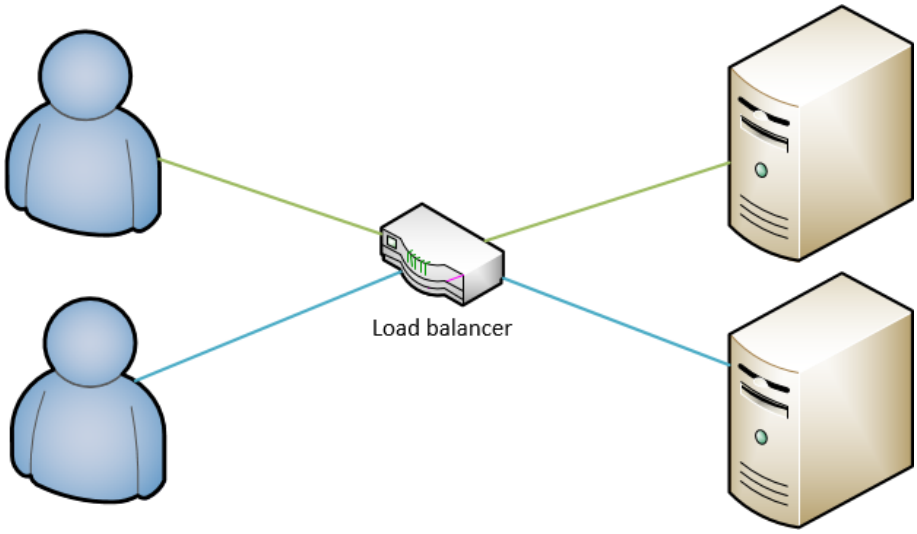

Fig. 1.1: Real-time synchronization problem

In real-time applications, scaling requires more parameters to be taken into account as data needs to be communicated between several clients, which might have been routed to different servers (see fig. 1.1)[6]. Communication between clients normally only works for clients on the same server [6]. The component responsible for this communication is the synchronization component [7].

Increasing usage, missing knowledge and negligence coupled with real-time applications being harder to scale and becoming more common; an overview of how to scale real-time applications should be useful.

2 Background

2.1 Purpose

The purpose of this paper is to take a look at the options for horizontal scaling. To keep the scope reasonable it has been narrowed down to the synchronization component of a real-time application. Availability, maintainability and security will not be taken into account. Focus will be on making the component work across several servers.

2.2 Question

What are the possible options and considerations when scaling a real-time application horizontally?

2.3 Target group and definitions

Reader is expected to be a developer with some previous experience building real-time applications. Readers who match this expectation can probably skip or glance over the rest of this chapter. The rest of this chapter will contain definitions to get up to speed on a few subjects to better understand the rest of this paper.

2.3.1 Real-time libraries

SignalR and Socket.io are abstractions for communication between server and client. Several methods of transport are used to ensure a good user experience and high availability, while abstracting away all the details. SignalR also allows abstracting away messages into running methods on the opposite side; server can call client methods and client can call server methods. Both use most of the methods of transport described below. In this paper, custom communication refers to an own solution based on these methods of transport to replace SignalR or Socket.io.

2.3.2 Real-time transport solutions

- **WebSocket**, full-duplex socket connection for communication between client and server, only available in modern browsers. Does not have headers.
- **Server Sent Events**, long lasting http request where data is streamed using text/event-stream from server to client.
- **Forever Frame**, long lasting http request using iframe to incrementally render chunks of data in script blocks which are called and fetched from parent document.
- **Long Polling**, http requests where server delays response until new data exists or connection is terminated. Client makes new requests for every response or termination.

2.3.3 Sharding

Sharding is a concept from the world of databases, where tables or collections are split across systems by hash or index. It is also known as horizontal partitioning. Shard key is the key or field chosen for this index. Remapping means changing the ranges for each shard, for example due to adding additional servers.

2.3.4 Service bus

Service bus is a service that handles messages between servers.

2.3.5 Redis

Redis is an in-memory database.

2.3.6 Load balancer

Load balancers are used to spread traffic load across application servers.

3 Method

Chosen method was a literary study focusing on scalability, real-time applications and their synchronization. Associated keywords were truncated and nested with booleans forming a search string[1] that was used in OneSearch[2], ACM[3] and IEEE Xplore[4]. Results were filtered by year and quality (being peer reviewed) where available. OneSearch was chosen first as it searches several databases and therefore should have the best reach. ACM and IEEE Xplore were chosen because their focus are on IT.

General scalability and specific technologies for communication were also examined using books and their respective documentation. SignalR[5] and Socket.io[6] were chosen as specific technologies since they are popular and open-source [8][9][10][11]. Books were picked by evaluating the highest rated books on Amazon[7] when searching for scalability, SignalR and Socket.io.

Upon gaining knowledge from those resources, sharding and load balancers were also examined. NGINX[8] and HAProxy[9] seemed to be the most commonly used load balancers and NGINX was chosen as it is used by more high profile sites, seem to be rising quickly in popularity and the author found its documentation to be more readable [12]. Top books from Amazon were again chosen and documentation examined. Author strived to keep interpretation of collected content to a minimum.

3.1 Method discussion

Peer reviewed articles were chosen, books were relevant to the study, at a suitable level, among the highest rated in their subject on Amazon and their authors experts in their respective field. Documentation and source code were also sources of high validity and reliability as they were authored by the same people that built the technology. Choice of technologies was a bit biased because of previous experience with them. Picking specific technologies was also a bit problematic as solutions specific to their respective technology could be found. Care was taken to exclude concepts and solutions that were not generally applicable or had alternative solutions.

The author did not have a lot of previous knowledge in this field and the timeframe might mean that too little information could be gathered to make up for this pre-existing knowledge gap. Experiments would probably be a better way to test different ideas and make sure that they actually work. Experiments would however require more resources as quite a few different setups should be tested. Testing and measuring requires a few machines or virtual machines to be setup. This would be more time-consuming than gathering the information and require a few machines, money for virtual machines in the cloud or a machine with more cores for testing locally.

---

[1] ((real AND time) OR synchron*) AND (scal* OR distributed) AND (web OR application)
[2] OneSearch - EBSCO Discovery Engine, https://www.ebscohost.com/discovery
[3] ACM Digital Library, http://dl.acm.org/
[4] IEEE Xplore Digital Library, http://ieee.org/ieeexplore
[5] SignalR, http://www.asp.net/signalr
[6] Socket.io, http://socket.io
[7] Amazon, http://www.amazon.com/
[8] NGINX, http://nginx.com/
[9] HAProxy, http://www.haproxy.org/

4 Result

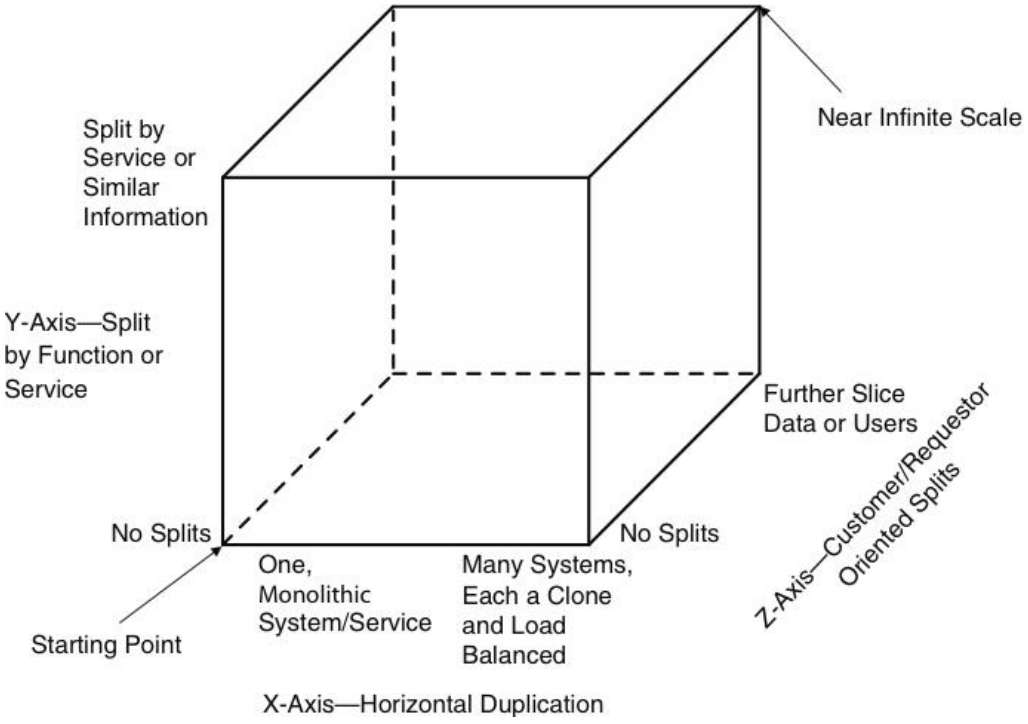

Fig. 4.1: Scaling cube, from *Scalability Rules*

General ideas for scaling out are X-, Y- and Z-axis scaling [4]. X-axis is the practice of cloning, services are duplicated to spread load [4, Rule 7]. Y-axis scaling means to split different services or splitting the application up by responsibilities [4, Rule 8]. Z-axis is splitting similar things, similar to sharding or a row-based split, services are split by something unique as an id (see fig. 4.1)[4, Rule 9]. Y-axis scaling would be inefficient, polling solutions would result in additional requests and socket solutions would increase memory usage as more connections would need to be kept alive [13]. X- and Z-axis are however viable options.

X-axis here refers to cloning the entire application server which means data-messages need to be routed or replicated between servers to enable users on different servers to share data [6][14][15]. Messages are replicated using a service bus or database and the component responsible is called a backplane. Solutions for how to do this are presented for both SignalR[16] and Socket.io[17, Scalability].

Z-axis would instead mean splitting users across servers, routing users to servers or subsystems where relevant data is shared. In this paper, this is called channeling.

4.1 Backplane solution

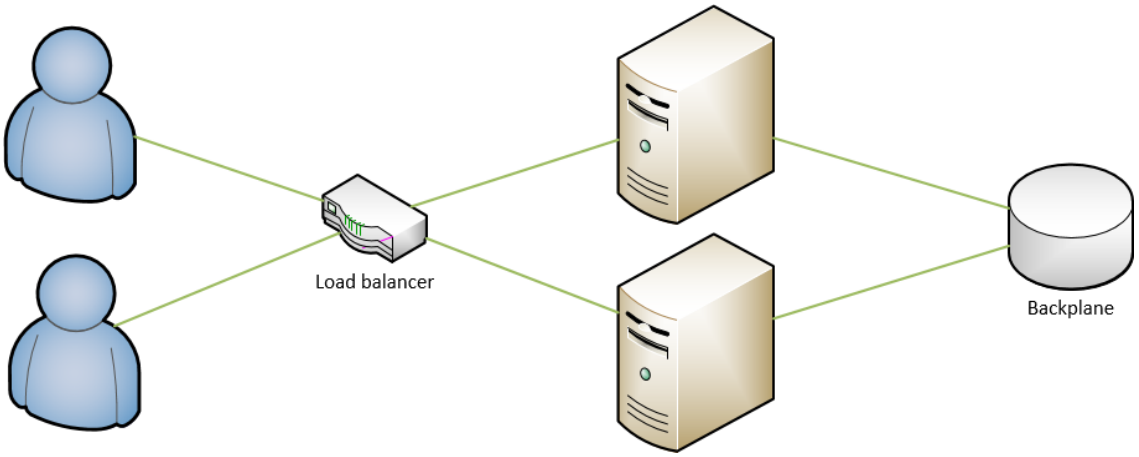

Fig. 4.2: Backplane solution

Backplane is a database or service that connects application servers by replicating data-messages (see fig. 4.2). Using a backplane is probably the easiest of presented solutions when Socket.io or SignalR is used. Implementing a Redis backplane with SignalR is done as follows. Many alternatives exist for backplane, Redis was chosen because it is the fastest solution also available for Socket.io.

- Install Redis
- Install Microsoft.AspNet.SignalR.Redis via Nuget Package Manager
- Change a SignalR dependency[10] [16]

Installed package contains logic for using a Redis backplane and is put to use by changing SignalR dependencies. Using Socket.io the procedure is similar [17].

Backplane solution does however have a set of limitations, since all data-messages from all application servers have to be handled by the backplane, it will probably become the new congestion point. Using custom communication its advantage of easy install is also lost.

[10] GlobalHost.DependencyResolver.UseRedis("server", port, "password", "AppName");

4.2 Channeling solution

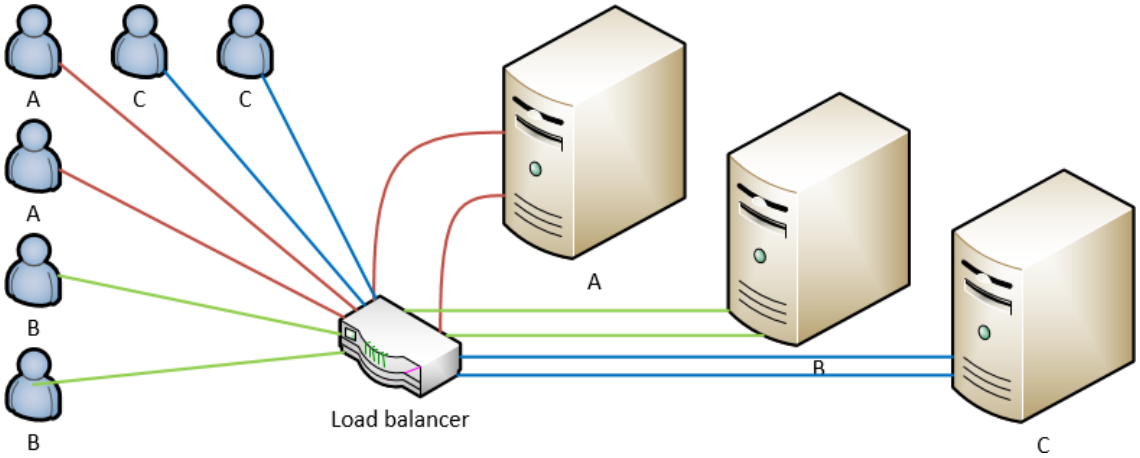

Fig 4.3: Channeling solution

Channeling is a solution reminiscent of sharding, picking a key should therefore have similarities with picking a shard key. Often this key should be the same as the most prevalent index [18], it should have high cardinality and ensure an even distribution. Key should then be hashed using a consistent hash method to avoid remapping. Most important is however trying to find natural channels where data should be shared.

Load balancer is responsible for spreading traffic across servers and its use case could be expanded to include routing logic. Users should be routed to channels where the communication of interest takes place. NGINX has an upstream consistent hash module where a key can be picked for splitting traffic [19]. Other load balancers should have similar options. A very simple script for solving this manually was also found, it is however just a starting point [20].

Channeling solution is generally applicable as it works the same regardless of how communication is implemented. Its implementation lies outside of the application server which means it can be implemented with none or very little changes to application logic. Application logic that will be needed is for switching channels and the process before a channel is joined. Scaling possibilities should be near infinite if channels can be divided into small enough pieces. Solution will however need some customization to ensure high availability. A channel key has to associated with each user to keep track of which server the user belongs to.

4.2.1 Key transport

Available choices for transportation of channel key are cookies, headers and URI. Headers are excluded as they are not available in websocket. Cookies are problematic in conjunction with cross-origin resource sharing (CORS) [21]. By default resources outside of current domain are restricted and CORS is a mechanism to allow these resources. URI is therefore the best choice and chances are that this key already exists in all requests if RESTful principles are followed. REST principles recommend semantic URIs. For example http://example.com/projects/x/users, where x would be the key.

4.3 Custom solutions

Backplane and channeling solutions might also be combined or customized to form other solutions. Higher availability and possibly more capacity per channel could be achieved by combining. Channeling solution could group users into a subsystem that uses a backplane. Putting each channel on more than one application server removes the problem of application servers being single points of failure and results in higher availability. It could also be a solution when channels have too much traffic for a single application server to handle, channel key should however be reconsidered before taking this path.

Backplane solution could also be customized into routing messages instead of replicating, which could also be expanded into layers of routing. There could be a layer for geographic routing between clouds and another layer of routing within each cloud [14].

4.4 Code preparations

When scaling horizontally state has to be moved from memory and hard drives to a location accessible by all application servers. For channeling solutions only cross-channel state is an issue. Keeping backend stateless is recommended as it will simplify scaling. Shared state also risks becoming a congestion point and results in another component where availability and risk of single point of failure has to be taken into account. In cases where it cannot be avoided, it should be moved to a database or similar. To keep backend stateless, Single Page Applications (SPA) communicating with REST APIs using JSON Web Token (JWT) authentication is a recommended approach [21]. JWT is a means of transferring claims between server and client, it consists of a header, payload and signature concatenated and encoded into an URL-safe string [21]. REST APIs are stateless [22]. Session state is contained within JWTs [21]. SPA moves all or most of application state to the client-side.

4.5 Process and deployment

A Design-Implement-Deploy(D-I-D) process is recommended [4]. Following a D-I-D process means designing for quite a lot more load than necessary(~20x), Implement for a bit more(~3x) and deploy for slightly more than necessary(~1.5x) [4]. This ensures capacity can always be met without excessive spending on scaling. Channeling solution also allows for using automatic scaling features that are available in most cloud services today, making it even more agile.

5 Discussion

Horizontal scaling will never be necessary for most applications. Depending on traffic load, synchronization load and application design, different solutions are recommended.

5.1 Backplane solution

Backplane solution should be used when traffic load is high enough to necessitate more than one application server and synchronization load is low enough to be handled by a single server. Every application server have to be able to handle all communication and the database or service used as backplane as well. This solution is suitable for applications that use SignalR or Socket.io for communication. When a custom solution for communication is used, its advantage of easy install is lost and the channeling solution might be preferable as it is more scalable. The extra steps also means this solution is slower, it results in a higher latency for synchronization data.

5.2 Channeling solution

Channeling solution should be used when more than one application server is required to handle traffic and backplane solution is not suitable; backplane solution could be too slow or backplane cannot handle the required amount of traffic load. Channels do however become single points of failure. When high availability is required a combined solution is probably preferable.

5.3 Custom solution

Combining should be used when channeling seemed suitable, but did not meet availability requirements, because channels become single points of failure which results in lower availability. Customized solutions with routing or layered routing needs to be investigated further. Routing could result in effective solutions for geographic spread and applications that cannot be divided into suitable channels.

5.4 Preparations

Even though most applications do not actually need horizontal scaling, a lot of them hope and strive for the kind of popularity that would necessitate it. Following the D-I-D process, they should pick or design a solution. Perhaps even implement it, depending on how close it seems. Premature scaling is however also dangerous as it could be very expensive. Adhere to the process to stay cost-effective.

5.5 Further research

Further research should be done for custom solutions. DNS solutions should be explored. Experiments should be done to test implementations by setting up systems with machines or virtual machines, trying out different theories and measuring spread of load, maintainability, reliability and performance. Results for applications will probably also vary a lot depending on the actual application. Experiments should therefore be done for several applications and even then, further research will probably have to be

done for individual systems, especially for customized solutions. Further research should also be done for geographic spread and how communication should be solved there.

References

[1] “Number of worldwide internet users from 2000 to 2014 (in millions)”, Statista, [Online]. Available: http://www.statista.com/statistics/273018/number-of-internet-users-worldwide/. [Accessed: April 15, 2015]

[2] “Breaking News: Downtime Kills Small Businesses”, Paradigm Network Solutions, [Online]. Available: http://www.ittoronto.com/breaking-news-downtime-kills-small-businesses/. [Accessed: April 23, 2015]

[3] “Scalability”, Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Scalability. [Accessed: May 11, 2015]

[4] M. L. Abbott and M. T. Fischer, *Scalability Rules*, Place of publication: Addison-Wesley Professional, 2011

[5] “The top 10 realtime web apps”, Creative bloq, [Online]. Available: http://www.creativebloq.com/app-design/top-10-realtime-web-apps-5133752. [Accessed: May 21, 2015]

[6] “Introduction to Scaleout in SignalR”, ASP.NET, [Online]. Available: http://www.asp.net/signalr/overview/performance/scaleout-in-signalr. [Accessed: April 20, 2015]

[7] “Synchronization (computer science)”, Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Synchronization_%28computer_science%29. [Accessed: April 23, 2015]

[8] “Posts containing ‘signalr’ - Stack Overflow”, Stack Overflow, [Online]. Available: http://stackoverflow.com/search?q=signalr. [Accessed: May 21, 2015]

[9] “Posts containing ‘socket.io’ - Stack Overflow”, Stack Overflow, [Online]. Available: http://stackoverflow.com/search?q=socket.io. [Accessed: May 21, 2015]

[10] “SignalR/SignalR”, Github, [Online]. Available: https://github.com/SignalR/SignalR. [Accessed: May 21, 2015]

[11] “Automattic/socket.io”, Github, [Online]. Available: https://github.com/Automattic/socket.io. [Accessed: May 21, 2015]

[12] “Usage of web servers broken down by ranking”, W3Techs, [Online]. Available: http://w3techs.com/technologies/cross/web_server/ranking. [Accessed: May 11, 2015]

[13] “NGINX WebSocket Performance”, NGINX, [Online]. Available: http://nginx.com/blog/nginx-websockets-performance/. [Accessed: May 18, 2015]

[14] B. Solomon et al, “Distributed clouds for collaborative applications”, Collaboration Technologies and Systems (CTS), 2012 International Conference, Denver, CA, pp. 218-225.

[15] R. Sureswaran et al, “Scalable and Reliable Multi Session Document Sharing System”, Information and Communication Technologies: From Theory to Applications, 2004. Proceedings. 2004 International Conference, pp. 613-614.

[16] “SignalR Scaleout with Redis”, ASP.NET, [Online]. Available: http://www.asp.net/signalr/overview/performance/scaleout-with-redis. [Accessed: April 20, 2015]

[17] “Introducing Socket.io 1.0”, Socket.io, [Online]. Available: http://socket.io/blog/introducing-socket-io-1-0/. [Accessed: April 20, 2015]

[18] K. Chodorow, MongoDB: The Definitive Guide, Place of publication: O'Reilly, 2nd Edition, 2013

[19] “HttpUpstreamConsistentHash”, NGINX, [Online]. Available: http://wiki.nginx.org/HttpUpstreamConsistentHash. [Accessed: May 11, 2015]

[20] “Sharded balancing with nginx and perl”, Github, [Online]. Available: https://gist.github.com/perusio/2154289. [Accessed: April 20, 2015]

[21] “Cookies vs Tokens. Getting auth right with Angular.JS”, Auth0, [Online]. Available: https://auth0.com/blog/2014/01/07/angularjs-authentication-with-cookies-vs-token/. [Accessed: May 18, 2015]

[22] “Representational state transfer”, Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Representational_state_transfer. [Accessed: May 18, 201