**Social Library Web App**

**Background**

The social library web app is based on the idea of real library with an additional feature of connecting likeminded book lovers with each other. The library has two sections: personal and social. A user can add new books to his personal library. Moreover, he can search and delete books from the personal library or edit them. Furthermore, books in the personal library can be viewed based on topics/metadata tags. Each book relates to a topic, therefore, when a user selects a topic then all the books under that topic are displayed to the user. In addition to personal section, the library has a social section where books from other users are recommended by the application. The recommendations are shown when a user has three books in common with another user of the library.

**Tools and Technologies**

For front-end design, we have used HTML, CSS and JavaScript with front-end library (jQuery [1]). Regarding the front-end frameworks, we have used Bootstrap [2], which is a free open-source CSS framework aimed at responsive, mobile first front-end web development.

For back-end development, we used Node.js [3] which is an open-source, event driven asynchronous I/O famous for building efficient and scalable web servers. We chose Node.js because of the two main advantages, also highlighted by [4]. First, due to familiarity with JavaScript because of its status as an accepted standard for web development. Second, due to the fact that the use of one language for both front and back end development speeds up the coding since the developers’ brain does not need to switch between different syntaxes. In addition, we have use Express.js framework [5] that is designed for building web applications and APIs.

Regarding the database choice, we opted for MongoDB [6], which is a schema-less, document-based, general purpose, distributed database build for the cloud era, focused on modern application developers. MongoDB fascinates us because it uses a JavaScript interface that completes full-stack JavaScript stack puzzle of server, browser and database layers. As a result one language can be used for all of the three layers. [4] also attributed this as a significant advantage of using MongoDB in addition to its performance and scaling.

With regards to tools for application development, we have used IntelliJ IDEA [7] and NoSQLBooster [8].

**Application Deployment**

After developing the application, it is not like we keep our computer running 24/7, therefore, we have hosted our service on a remote environment, that is, Amazon Web Services (AWS). AWS [9] aims at providing cloud computing platforms as well as APIs, on demand and metered pay as you go basis. Our online library application can now be accessed by any user of the app at any time. With AWS, we have stored and served our client side files (S3), a virtual server that can be kept on for as long as we need (EC2), and a cloud database management system (RDS). Using [REST](https://en.wikipedia.org/wiki/Representational_State_Transfer) architectural style, offerings of AWS are accessed over HTTP.

To make our application highly available, scalable, and elastic, we needed to have a high level administration mechanism for coordination between backup and front-line instances. Having a running replica of an instance that has failed recently will not help because the visitors of our library app would not know where to go and find it. For this, we have added a load balancer to the mix for monitoring health of running instances. In this way, if one instance goes dark, load balancer will automatically redirect incoming traffic towards active resources. We have used Amazon’s Elastic Load Balancing (ELB) as a load balancing tool and it will not only manage failovers but also focus on balancing traffic loads amongst multiple resources for satisfying defined performance and efficiency needs. After configuring the load balancer with addresses of all of our servers, its own network address is now the only URL that our users need to access. The users do not need to know the individual IP addresses of each of our servers.

Although the sudden loss of a server can be accommodated gracefully by the load balancers, they cannot replace the lost capacity originally provided by now-dead server. In other words, if one out of our three server crashes, the full workload will have to be managed by the remaining servers on their own. Load balancer cannot help out in this area as it is well beyond its pay scale. And regarding elasticity, load balancer keeps what you have got running nicely, but they are unable to manage change. Furthermore, as we were concerned that increased demand or unexpected server downtime may unable our application to do its job properly, therefore, we found a way of adding capacity using auto scaling. Using auto scaling, we have automated instance replacement in case of failure. Moreover, depending on the need, the number of running instances can now be increased or decreased.

**References**

[1] <https://jquery.com/>

[2] <https://getbootstrap.com/>

[3] <https://nodejs.org/en/>

[4] Mardan, Azat. (2018). Full Stack JavaScript: Learn Backbone.js, Node.js, and MongoDB. 10.1007/978-1-4842-3718-2.

[5] <https://expressjs.com/>

[6] <https://www.mongodb.com/>

[7] <https://www.jetbrains.com/idea/>

[8] <https://nosqlbooster.com/>

[1] <https://aws.amazon.com/>