IJCSN - International Journal of Computer Science and Network, Volume 7, Issue 5, October 2018
ISSN (Online): 2277-5420
www.IJCSN.org
Impact Factor: 1.5

RWA: Resilient Web Application Using Client-Side Processing, Database, and Web Cryptography

[1] Jebreel Alamari; [2] C. Edward Chow

[1] Computer Science Department, University of Colorado at Colorado Springs
Colorado Springs, CO 80918, United States

[2] Computer Science Department, University of Colorado at Colorado Springs
Colorado Springs, CO 80918, United States

Abstract - Building resilient web applications helps mitigate risks due to server hardware failure, cyberattacks or loss of connectivity. The advantages of web applications over native applications, such as the ease of distribution and platform independence, attract developers and businesses. Traditional web application design requires at least two separate machines, client and server, to work and be connected at any given time. Keeping client and server connected is becoming a challenging task with the recent increase of cyber-attacks [1]. With new browser capabilities, it is possible to create resilient web applications that can handle loss of connectivity or cyber-attacks on the server. Developing such applications will improve cyber resilience and help reduce disaster recovery time and cost. Existing and new APIs such as Web Cryptography and Web Storage implemented in modern browsers were explored and evaluated for the dependability in creating a resilient application design. The experiment suggests a development of a new design and implementation framework. The result application is resilient to bad Internet connections, server failures, cyber-attacks on servers, and security problems in the browser environment.

Keywords – Web Browser, Web Cryptography, IndexedDB, JavaScript.

1. Introduction

In general, resilient system is defined as the system that can withstand threats and recover in a short time with minimum cost [2]. One of the biggest issues with web applications is availability. Good availability is to have information/services available for authorized people at all time possible[3] . We have seen a lot of effort to ensure the availability over years, but it seems that all the effort was targeting at improving the availability at server level. As many Internet users are still experiencing bad connection or disruption of services, that effort does not solve the fundamental problem. The reason is simple; the issue is that those giant web enterprises only care about customers who are well connected to the Internet. Those less fortunate users are suffering as part of the reality. In this project we explore applicable solutions that can be used to design a resilient web application using existing technologies and provide a demonstration for proving its effectiveness. We focus on web applications and not native applications since web applications are easier to distribute and cheaper to maintain and update than native applications [4]. The proposed solution entails the philosophy of enabling all processing at client side with the browser. The browser is not only a piece of software that fetches static pages from servers and shows them on the client's screen. It has evolved to allow complex codes to execute, taking advantage of computation power and storage of modern personal computers. Recently, all major web browsers implemented Client-side Web storage, to allow offline applications to run. By looking at how powerful and capable web browsers are, we increase the availability of information/services and take care of users with bad access to the web. Therefore in this paper, we explore how to build a secure web app that work online, offline, and in different browsers with security taken into account, and we compare their performance with the related real life applications. The web app will mimic typical server web applications but can run standalone in the client side, since it has everything needed, database and computation. The design depends specifically on the APIs that are fully supported by browsers. We avoid using polyfills to have smaller yet consistent application. We use pure JavaScript with some JQuery [5] to ease DOM manipulation, and manage client-server interactions. The design includes efficient data synchronization between the server and the client. Also, unlike traditional username password authentication model that requires servers, the design has a different authentication model that can be performed in both online and offline mode. It is worth mentioning that this design does not follow exactly offline-first model. Offline-first application is that

application that is created to work in offline mode all the time with some minor interactions with the server for back up[6]. However, this design is a mix of online and offline. In other words, everything that can be done on the server can be done by the client side and vice versa. Nonetheless, if the server is available some, operations, like authentication, preferred to be performed on the server, but not locally.

2. Related Work

In this section we identify some of the related work and explain how our design is different.

2.1 Pouch DB

Pouch DB is a great work that was done by Dale Harvey and Nolan Lawsonand. It was first published in April 2014 [7]. They created a JavaScript library to utilize local database in different browsers and sync data to couch DB on Node.js web server. The main difference between our proposed work and theirs is that their framework is designed to work with a certain database, i.e., Couch DB. Also, web server has to be Node.js[8], which is a JavaScript web server.

2.2 xStorage

xStorage was proposed by Syed Zagham Naseem and Fiaz Majeed, faculty members at Gujrat University, Pakistan [9]. They extend the local storage in the browser to store more data. Also, in their implementation they added a synchronization feature where data in the modified local storage can be synced to the sever database. The difference between our proposal and theirs is that, we build our design on standards or recommended APIs that are or will be available in major web browsers. In addition, they do not address security issues in their proposal.

2.3 Kepler

Kepler is a Chrome extension developed by Tomas Wahlberg, Petri Paakola, Christian Wieser, Marko Laakso, and Juha Roning at University of Oulu, Finland[10]. Kepler tracks website activities in browser and displays it to the user in real time. Their goal is to increase security awareness in browsers by showing users what websites are trying to do in the browser background. Our work related to theirs through user data protection. Using their extension, users could find out about hostile web sites that try to read data from the local storage, and in our proposal we try to encrypt user data in the local storage.

3. Proposed Design for Resilient Web Application

The suggested web application design should withstand the absence of Internet connection without letting user notice. Also, it runs on online/offline mode seamlessly. Pushing data to remote or local database is designed to follow some pattern where it prevents data from being overridden or lost. It allows multiple users to use the application in the same browser from the same domain such as www.domain.com without mixing their data or letting them view each other's data. It allows a single user to use the application from different browsers or machines providing him/her with the most recent data everywhere. It has a unique authentication model that can perform authentication even if the server is not available. Finally, all major modules in the demo application are written in pure JavaScript without using any framework such as Angular.JS[11] or Ember.JS[12].

3.1 App Components

RWA preserves all components in traditional web application. It has server side code that is written in server scripting language, which is PHP in this case. However it could be written in any other server side language such as Perl or Rails. It contains two databases, server side and client side. For server side database we used MySQL[13] . The reason is because it is well supported by PHP and for its consistency as a relational database. At the client-side, we used indexedDB [14]. Similar to Mongo DB, it is a highly scalable database. But more importantly, it is a recommended standard by W3C, so it is supported in all browsers. There is another database API, namely WebSQL [15], that is still supported in Google Chrome and Safari, but it failed to become W3C standard; therefore it was deprecated in Firefox.

3.2 Database Design

Designing a database for any application is a critical, but designing two databases that are completely different is even more challenging. As mentioned above, the two databases used in this design are MySQL in the server and indexed DB in the browser. IndexedDB is a NoSQL database. It stores data in objects and each object is accessed through a unique index or key in a way that is similar to key value-pair storages. Objects may not have the same number of properties, and it is possible to add or delete object properties after the object is stored. In contrast, MySQL is a relational database in which when we create a table we must have a single value in each table's cell. When designing an SQL database, developers

have to do a lot of normalization to ensure data consistency. That results in having more tables in the database. In our implementation we made an SQL table for root objects and a table for each sub-object. We linked the sub objects' tables with the main table using foreign key constraints. To manage database operations in both sides we developed a parser to convert data from a complicated object stored in our IndexedDB database into multiple one-dimensional arrays that are suitable for our MySQL database.

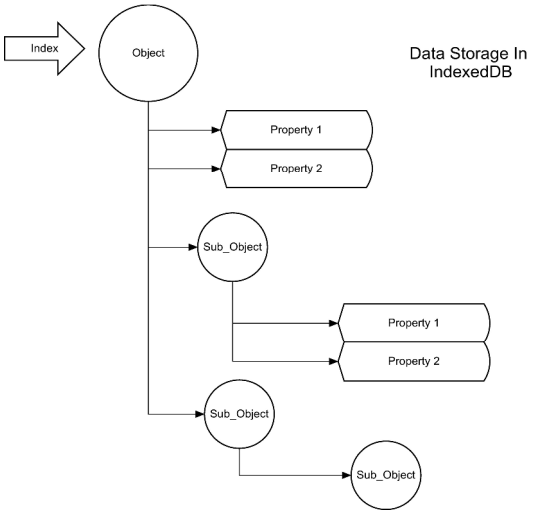

Figure 1: Complex JavaScript object stored in IndexedDB

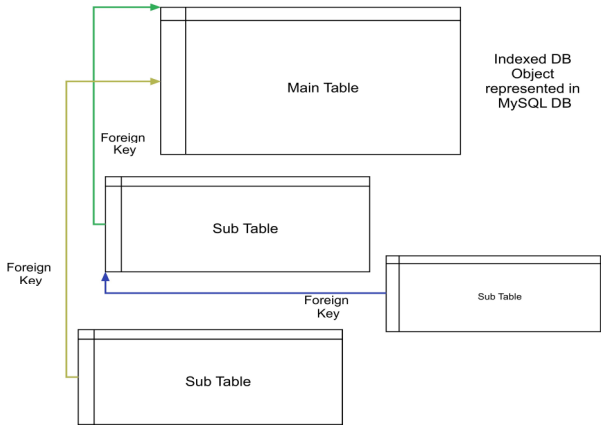

Figure 2: IndexedDB object representation in relational Database

3.3 Data synchronization

The synchronization module in this design takes care of moving data to and from the server. It is divided into two parts where some of it is implemented at client side and the rest runs on the server. The synchronization should be triggered by user interaction with the application components. Performing synchronization based on user's behavior in the application is more efficient than doing it automatically. That is because it is more likely to have data to sync when the user does some typing or button clicking. Also, synchronization should not be performed unless the server is available. Because synchronization requires querying the local database, we cannot perform it without being sure that the server is available. Unfortunately, there is no accurate way to determine whether we are online or offline in the browser. Using *navigator.onLine*, that is built in modern browsers, is not accurate because it returns true when the computer is connected to local network even if the local network has no access to the Internet. Worse than that, it is possible that the client is connected to the Internet, but the server is down. Therefore, we recommend making a lightweight AJAX request to a file stored on the server or to some server code to avoid caching the file by the browser. If the AJAX request succeeds we perform synchronization. Otherwise, the application continues to run locally without crashing.

3.3.1 Synchronization pattern

The proposed synchronization model goes through three phases:

1. FETCH Phase:

This is the first operation to be performed. If the local database is empty, it gets all the data from the remote database. If the local database is not empty and the server is available, the application passes the ID of *last-modified-record* on the server by the last time we get the data from the server. In other words, when the application gets data from the server and before storing records in the the local database, it checks their *modified-time* on the server.

Then the one that has the most recent modified time is marked as *LastDownloaded*. In the next FETCH request, we pass its ID with the request, so we only get records that have more recent modified time. Figure 3 shows steps to fetch data from the server database. We notice that when there is no connection, the application continues to work locally. However, when *online* the application fetches newer data from the server and updates the local database.

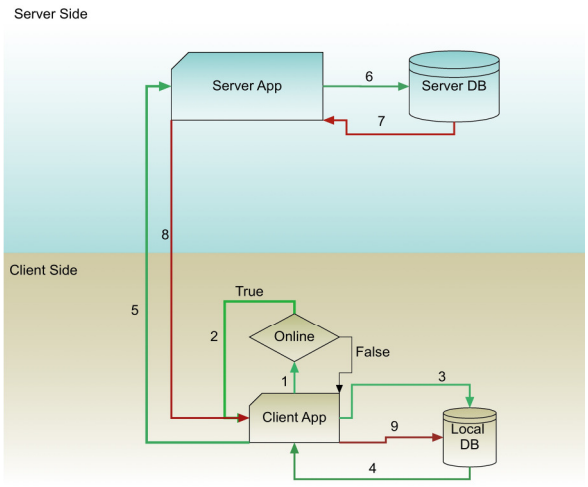

Figure 3: The steps of getting data from server database

2. GET DELETED Phase

This phase is performed right after FETCH phase. It gets deleted records from *deleted-record* table that holds deleted records on the server for synchronization purposes, in order to delete them from the local database. However, before deleting any record from the local database, the application should make sure that the record has not been modified locally, because it is possible that this record has been modified by this browser. If the record has not modified, it gets deleted.

3. POST or SET Phase:

This is the last phase to perform before the synchronization cycle ends. It gets records from the local database that are *modified, not-synced,* and *marked-for-deletion*. Then, it passes these records to the server at once. When data arrives at the server it gets parsed and separated based on its category. *Marked for deletion* records will get dropped from the database and moved to *the deleted-records*table for other browsers to see. *Modified records* go to the server database by performing update SQL command. In some applications it is likely to create new records that have the same IDs as the records in *deleted-records* table. Therefore, before we insert data to server database we make sure that none of the to be inserted records have the same ID as any record in *delete-records* table. Not doing that may cause data to be deleted from the local database when performing the next synchronization cycle. Figure 4 presents the steps to push data to the server. Only most modified and newly generated data should go to the server.

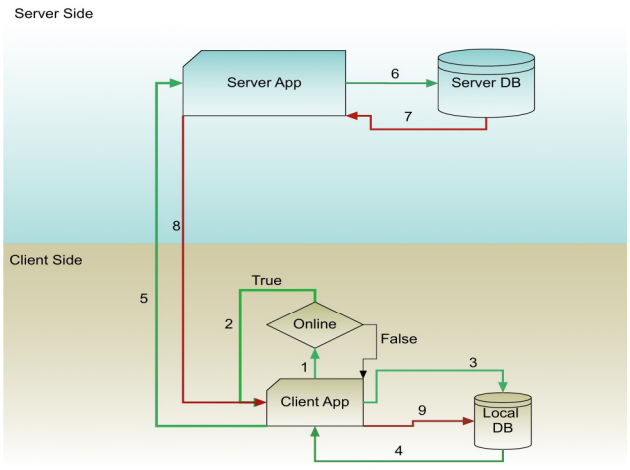

Figure 4: The steps of posting data to server database

3.4 Authentication

3.4.1 Managing Multiple Users

This design scales to allow multiple users to use the application from the same browser. Same Origin Policy[16] implemented in indexed DB API prevents viewing or accessing database from different domains. However, this policy does not provide protection in case of having multiple users using the same application (same domain) on the same browser. To tackle this problem we prevent any code that deals with the database from being executed, unless user info is present in local or server users' database. Each user has access to his data by signing in to the application using client side or server-side authentication.

3.4.2 Server-Side Authentication

Server-side authentication has a priority over client-side authentication when the application is in online mode. It follows the traditional username-password model. When the application starts, the user has to sign in if he is an existing user. Credentials will be sent to the server and based on server response the user is granted access to the application or instructed to sign up after a certain number of unsuccessful attempts. Upon successful sign in, the application checks whether the browser has a local database for the user. If so, data synchronization will be performed to update the user's local database. If there is no local database for the user, the application creates a new database and populates it the with user's data from the server database. Then, the application adds an encrypted version of the user's username to the users' database in the browser for future offline authentication.

3.4.3 Client-Side Authentication

This authentication model has less priority than online authentication, but it is useful when the server is not accessible. Native applications have been providing offline authentication for years where someone has a username and a password securely stored in an internal database. Unlike native applications, there is no secure storage in the browser. In addition, a native application source code is not easily accessible to end users who may exploit it or manage an attack based on it. In contrary, JavaScript source code is visible in the browser, so it could be modified. In this design, we propose a new design for client-side authentication based on newly implemented Web Cryptography API. When a user signs in, the application constructs a key (**see key management section 3.5.1**) then uses it to encrypt the username or username concatenated with the password. After that it checks if the result is present in local users' database. If so, the user is granted access to the application. Otherwise, access will be denied. If it happens to have a matching result in users' database, accessing the application doesn't mean the user will be able to read the content. It is because the data in the application database is encrypted using the real user's key. However, this unauthorized user can do some damage to the database by deleting or altering database content, but luckily, changes will not go to server database. The reason is that, in this design each user has his own server database that has a unique name derived from his credentials by the time of successful sign up. With every AJAX request to the server, a part of user's credentials is passed with the request. This piece of information is used to determine the right database to work with. Figure 5 depicts the local authentication mechanism.

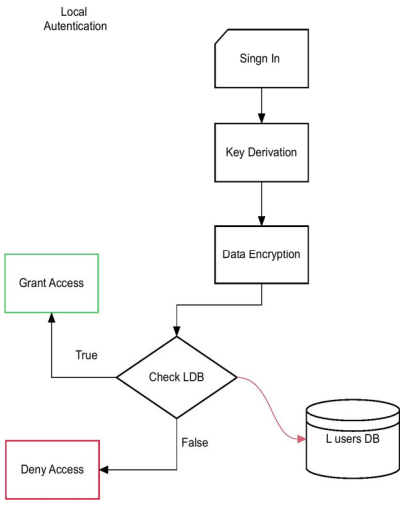

Figure 5: Local authentication design

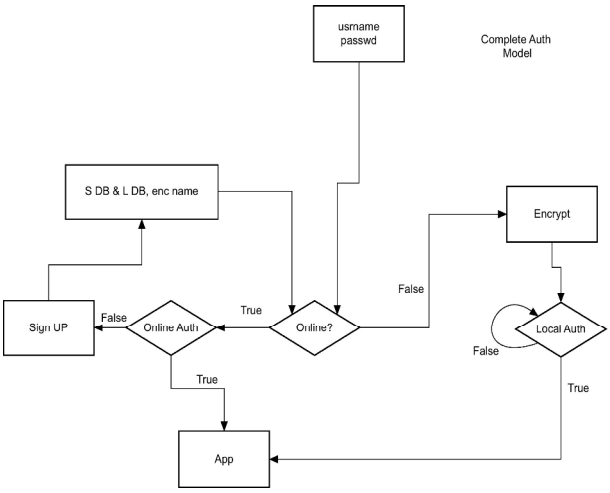

Figure 6: Online and offline authentications put together

3.5 Client Side Cryptography

With the revolutionary storage and security solutions provided by the cloud, millions of users started using it with unquestionable trust. 'Keep your data in the safety of the cloud' we hear this in a lot of commercials by cloud storage providers. However, after hacking iCloud in August 2014, a big question is asked. How safe is the cloud? The answer is, it is safe but it is not a hundred percent hack proof. Some storage services such as SpiderOak used client-side encryption where data is encrypted at the client side before storing it in the cloud [17]. Therefore, even if the cloud were compromised, decrypting data would be too expensive for attackers if not impossible. With the new HTML5 Web cryptography specification, W3C tries to allow web developers to be able to do the same, to increase the safety of users' sensitive data.The World Wide Web Consortium proposed Web Crypto In 2014 [18]. It is a JavaScript API that allows some cryptographic operations, such as encryption, decryption, and generating keys. This API can be used to perform user authentication, document signing and ensure the confidentiality and integrity of web communications. It is worth mentioning that a lot of JavaScript libraries existed way before this specification, such as CryptoJS [19]. Other JavaScript Polyfills were developed after the first draft of the spec, such as PolyCrypt[20] and NfWebCrypto [21]. In our design we prefer using Web Crypto API over other pure JavaScript libraries for multiple reasons. First, it is implemented in browser native code, which means better security. Using JavaScript to perform cryptography is risky. That is because keys or secret phrases will be available in JavaScript code and could be used by unauthorized people. Second, browser vendors now implement this API, which means better

support and maintenance. Finally, because the API is browser implemented, it is more likely to be faster than JavaScript code. Figure 7 shows Web Crypto is faster than CryptoJS in performing encryption/decryption operations.

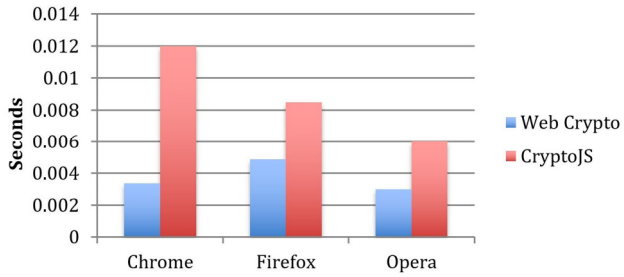

Figure 7: Web Crypto performance compared to CryptoJS (Pure JS library)

3.5.1 Web Cryptography in RWA Design

Since the design takes into account the case of multiple users using the same application from the same origin (domain), and users can see each other's data, we use this API to encrypt data in local databases. So, if curious users try to read others' databases using the browser console, they find encrypted useless data for them. Also, it is used in the signup model. When a user signed up successfully we concatenate his credentials then derive a key out of the result. After that we encrypt his username and insert it in local users' database, to perform offline authentication. In offline mode we get username and password then derive a key out of them then encrypt his username. Then we check if the result is already in the users' local database. If so, the user can access the application.

Here are some critical issues we solve with web cryptography

I Key Management

Web Cryptography API does not provide reliable secure storage for keys generated by the application. Even if it has keyWrap and keyUnwrap functions to protect secret keys, the wrapping keys are browser specific and cannot be accessed by the user or the application. That means if the browser crashes or gets uninstalled, wrapped secret keys will not be retrieved; therefore data encrypted with them will be lost. One solution is to store keys on the server and retrieve them when they are needed. However, doing that does not follow the design of RWA that must work in online and offline mode. Another way of solving this problem is through allowing the user to export his key and upload it to the application when signing in. This solution follows the design and the application is accessible even without connection. However, such a solution does not make a good user experience. Because the user has to keep his keys available at any given time so he can use the application when needed, that is not convenient. A better solution for this problem is to find a way to construct a key out of username and password. Therefore, user does not need to remember his key or store it in a file in an insecure place. Luckily, Web Crypto API can help facilitate this solution, but not in a direct way. One of the methods available in the API is deriveKey. Unfortunately, this method cannot be used in this design because in order to derive a key using this method, we need to pass a key to it, which means this key should be stored in a secure storage that is not available in browser. However, Web Crypto has other methods that could help when used together. By observing genrateKey, exportKey, digest, and importKey methods in the API, our idea of constructing a key out of the user's credentials has three steps. (1) Hashing user's credentials, or some of it, using digest SHA-512 method (2) Using the result of digest method to make a key using importKey method (3) The result of importKey is a key object that can be used for cryptographic operations. Figure 8 shows key generation steps. Key is generated with every encryption/decryption operation.

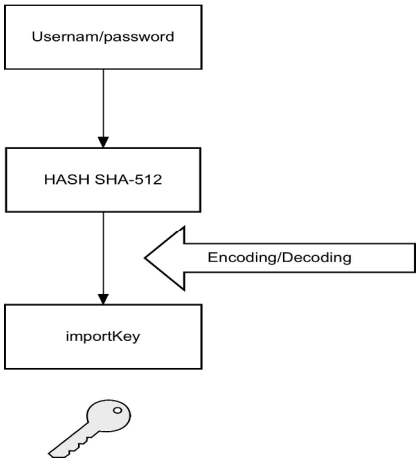

Figure 8: Key generation mechanism

II Encryption and Decryption

Data in local databases that have the same origin can be seen from the browser console. Therefore, we need to encrypt data inside the database, so even if other users see it, it will be in an encrypted form. Data encryption/decryption will be performed using the user's key that gets generated every time user interact with his database. It is worth mentioning that data is decrypted only when it needs to show up in the user graphical interface. In

other words hidden elements on the page have no data until they show up on the page. Figure 9 shows the time taken to perform encryption and decryption in studied browsers.

4. Performance evaluation

We created a test case for this design by building a demo web application. We created two versions of the same application. The first one follows traditional design where there is no local database. However, the other version is built with RWA features.

Figure 9 shows the average time the traditional application takes to access data on the server compared to the time required to access data locally. We notice that accessing data locally is almost 10 times faster than accessing remote data even though the test was performed with a high-speed Internet connection. This may not affect user experience if they have a fast connection, but that is not the case all the times. Therefore, in the case of bad connection this design can significantly improve the performance of the web application which leads to a better user experience.

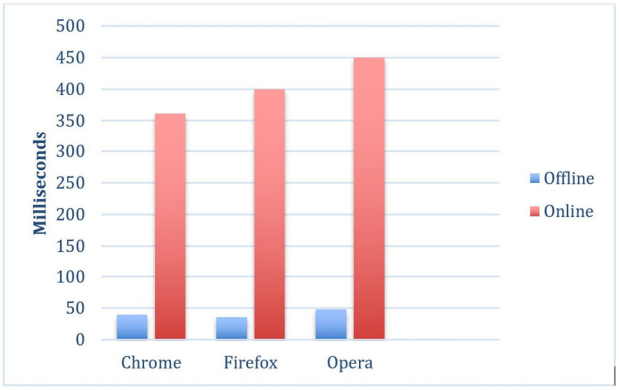

Figure 9: Traditional Web App vs RAW

As mentioned in the design, the client application must connect to the back-end for synchronization purposes. Figure 10 shows the average time taken by RWA application to perform different synchronization operations in different web browsers. As shown in the graph, the time taken to test connectivity is less than 50ms in case the user is offline, or the server is not available. It turns out performing this test can improve the overall performance of RWA applications because if the server is not available, synchronization module does not have to run. Being connected to the Internet and the server is available, the test for connectivity could take as much time as regular server requests.

It is worth mentioning that all synchronization operations are *asynchronous*, therefore, they should not affect user experience or the performance of other application components.

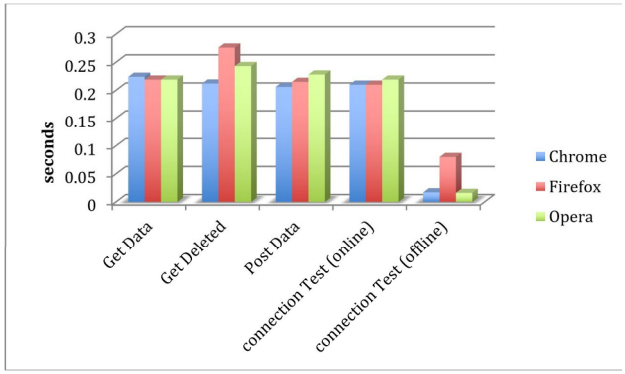

Figure 10: RAW Synchronization

Figure 11 shows the time taken by RWA key generation scheme. Using the built in importKey with previously generated key is faster than RAW key generation method. As mentioned, Web Crypto does not provide a way to store keys, so we had to work around the limitation and generate the key on the fly based on a hashed value of a string derived from the user's credentials. This string should be deleted if the page is closed or if the user is not interacting with the application.

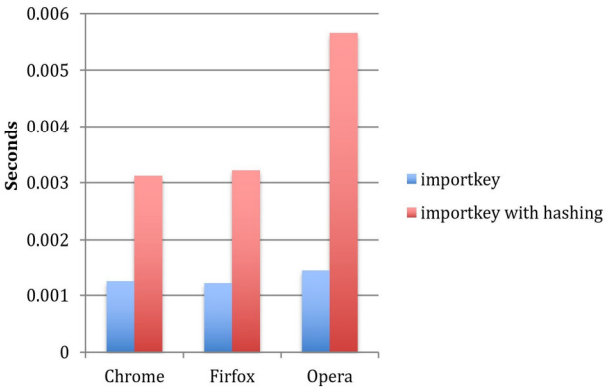

Figure 11: RAW Key Generation

5. Advantages and drawbacks

5.0.1 Advantages

1. *Better availability:* Having web applications that can run in the absence of back-end code, can be beneficial for website owners and their users. Owners or service

providers can reduce the cost of upgrading their infrastructure to handle increasing number of requests. Users with a bad Internet connection, specifically developing countries, can enjoy low cost or free web applications. For businesses, browser base systems can use this design to utilize office computers. Instead of using PCs at work as an output of computation on server, we could allow those computers to do some processing and reduce the load on servers.

2. *Better security:* In terms of security, users should be in charge of protecting their data. By building web applications that allow client cryptography, users could become more confident about their data security. In other words, back-end providers will know nothing about users' data because data is encrypted in the back-end database. Also, using client-side encryption mitigate the risk of using http without SSL.

5.0.2 Drawbacks

1. *Browser errors:* It could be one of the biggest threats to this design model. Browser error could be in different forms. For example, clearing cached files or wiping out local database. In Chrome and Opera if a user cleared browsing data Indexed DBs get deleted. Also, uninstalling the browser will cause all its data to be wiped out. Browser error can be devastating if user is not frequently connected to the Internet. Unfortunately, this design does not include offline data sharing between browsers because it is not feasible due to Operating Systems restrictions [22]. It is not possible even in computers that run browser OS, such as Chrome OS [23].
2. *Browser restrictions:* It could make developing such design more challenging. Based on my observation I found that JavaScript control over browser is limited. It is true we can refresh a page from JavaScript code but we cannot prevent user from refreshing the page using refresh button in the browser. That could be a big issue for application data that are still in variables. To solve this issue developers can use persistent local storage as a back up for variable's data and retrieve them after page refresh. However, that could cause performance degradation. In addition, running the browser in private or incognito mode prevents the application from reading local databases created in non-private mode. That means if a user decides to switch to private browsing, previously created databases are not accessible.

6. Future Work

We plan to extend this research by adding more features to the design. One of the features that could be extremely useful is to develop a tool to test connection speed between client and server in real time. This tool helps provide more efficient synchronization algorithms. Because by knowing how fast the connection is we can make decision about the way we send data to the server. For example if the connection is slow we could divide data into small chunks and send them in a sequence. On the other hand, if we have fast connection we can send data at once. I will extend recovery plan when data at client side accessed or modified by the wrong person. This is partially supported in my design by having separate database on the server for each client. However, if a wrong person accessed the application and modified data then the real user accessed the application after the wrong one, data on server will be corrupted as well. Therefore, I will extend the recovery plan by detecting unauthorized request to server and take an action based on that. One of possible solutions is to cause client application to drop local database when detecting malicious activity.

7. Conclusion

In this research we built a resilient web application design that can handle multiple threat scenarios, such as connection loss, cyber attacks on the back-end machines, or hardware failure at server level. It also includes an efficient synchronization pattern to transfer data securely between client database and server database in about 20 milliseconds in average, taking into account the differences between the databases in the way they store data. Additionally, the design has a unique authentication model that can work either in online or offline mode. Also, in this design we put the user in charge of his/her data security by encrypting it at rest, either in the browser or on the server. Last, we came up with a mechanism to generate cryptographic keys from user credentials using Web Crypto API in reasonable time.

Therefore, based on the results of this experiment, we think that it is possible to create robust web applications that can handle a variety of threat scenarios, such as having server down for maintenance or because of cyber attack. In addition, users in the developing world could enjoy free or low cost web applications without the need to connect to servers all the time. Additionally, Utilizing process and data storages at client side could lead to reduce Internet cost by reducing network traffic. Finally, web base games powered by pure JavaScript or with WebGL API can be

taken into another dimension by storing players' achievements in local database and keep server database for back up.

References

[1] "Cyber security: global incidents 2015 statistic."[Online]. Available: https://www.statista.com/statistics/387857/number-cyber-security-incidents-worldwide/

[2] T. F. S. LIVRAMENTO, E. A. Q. DE OLIVEIRA, M. S. RODRIGUES, and M. B. MORAES, "Scientific production analysis of resilient enterprises," in *Conference Proceedings in International Association for Management of Technology-IAMOT*, 2015, pp. 2253–2264.

[3] Z. Yang, Y. Gou, Y. Zhu, and H. Zheng, "Availability modeling and simulation of satellite navigation system based on integration of pdop and reliability maintainability supportability," in *China Satellite Navigation Conference*. Springer, 2018, pp. 241–256.

[4] W. West and S. M. Pulimood, "Analysis of privacy and security in html5 web storage," *Journal of Computing Sciences in Colleges*, vol. 27, no. 3, pp. 80–87, 2012.

[5] "Jquery," https://jquery.com/.

[6] Offlinefirst, "offlinefirst." [Online]. Available: http://offlinefirst.org/

[7] J. Justin and J. Jude, "Go offline," in *Learn Ionic 2*. Springer, 2017, pp. 79–97.

[8] N. Foundation, "Node.js." [Online]. Available: https://nodejs.org/

[9] S. Z. Naseem and F. Majeed, "Extending html5 local storage to save more data; efficiently and in more structured way," in *Digital Information Management (ICDIM), 2013 Eighth International Conference on*. IEEE, 2013, pp. 337–340.

[10] T. Wahlberg, P. Paakkola, C. Wieser, M. Laakso, and J. R¨oning, "Kepler–raising browser security awareness," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 435–440.

[11] "Superheroic javascript mvw framework." [Online]. Available: https://angularjs.org/

[12] "Ember.js: Homepage." [Online]. Available: https://www.emberjs.com/

[13] [Online]. Available: https://www.mysql.com/

[14] "Indexed database api 2.0." [Online]. Available: https://www.w3.org/TR/IndexedDB-2/

[15] "Web sql database." [Online]. Available: https://www.w3.org/TR/webdatabase/

[16] "Same origin policy." [Online]. Available: https://www.w3.org/Security/wiki/Same Origin Policy

[17] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Keys to the cloud: formal analysis and concrete attacks on encrypted web storage," in *International Conference on Principles of Security and Trust*. Springer, 2013, pp. 126–146.

[18] "Web crypotography." [Online]. Available: https://www.w3.org/TR/WebCryptoAPI/

[19] "crypto.js." [Online]. Available: https://code.google.com/archive/p/crypto-js/

[20] "polycrypt.js a web crypto polyfill." [Online]. Available: http://polycrypt.net/

[21] "Nfwebcrypto." [Online]. Available: https://github.com/Netflix/NfWebCrypto

[22] C. Reis, A. Barth, and C. Pizano, "Browser security: lessons from google chrome," Communications of the ACM, vol. 52, no. 8, pp. 45–49, 2009.

[23] S. Adee, "Chrome the conqueror," IEEE Spectrum, vol. 47, no. 1, 2010.