Client-Side Storage in Web Applications

Markku Laine

Department of Media Technology, Aalto University
P.O. Box 15500, FI-00076 Aalto, Finland
markku.laine@aalto.fi

Abstract—In recent years, a number of sophisticated mechanisms for storing and managing data on web clients have emerged. These client-side storage mechanisms bring along several benefits, including faster websites and improved user experience. In this paper, the history of client-side storage is presented and three modern W3C-specified API proposals for persistent (session or local) data storage are introduced. In addition, significant research projects taking the advantage of the mechanisms are described.

Keywords: Client, Storage, Database, Web Application, API, JavaScript.

1 Introduction

Web applications have two options to store data: (1) on the web server or (2) on the web client (end-user's computer). Certain types of data belong on one, while others work better on the other. For instance, sensitive and fragile data should always be stored on the server, whereas static data and user preferences could be stored on the client. [16]

Figure 1 illustrates the conventional interaction model, in which all data is stored on the server-side storage (e.g., a database). When a user requests new data, an Ajax request is made and the data is retrieved from a database residing on the server. The disadvantages of this approach are: (1) unnecessary network traffic, (2) requires an Internet connection, and (3) burdens the server load. Transferring data between the server and the client, of course, takes time, and thus slows down the website, leading to poor user experience.

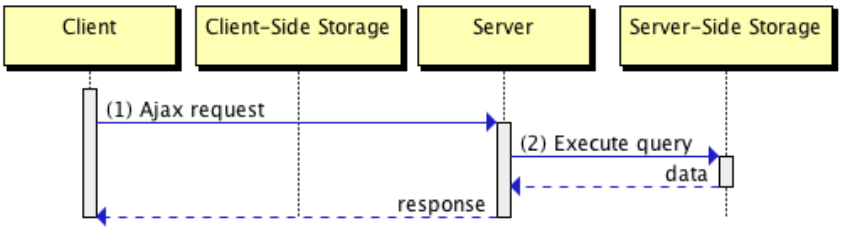

Figure 1. Conventional interaction model.

By using a client-side storage mechanism, most of the aforementioned disadvantages can be overcome. Figure 2 illustrates the modern interaction model, which includes a client-side storage. First, a web page checks whether the client-side storage already has the requested new data. If yes, then the data is immediately displayed on the user interface without communicating with the server. Otherwise, the requested new data is retrieved from a database residing on the server, and when retrieved, the data is stored in a client-side storage for future use. This approach not only makes websites faster, but it also enables offline support and reduces server load.

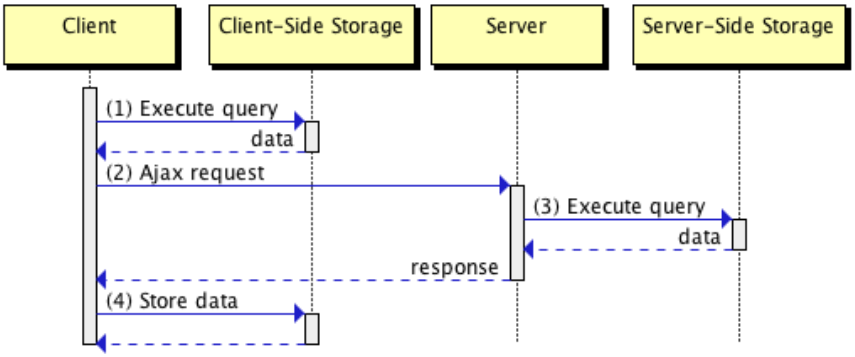

Figure 2. Modern interaction model.

HTTP cookie [3] was the first mechanism, which enabled web applications to store data on web clients. However, cookies do not meet the requirements of modern web applications. Thus, new client-side storage mechanisms were needed. First came *(Google) Gears* [7], and then influenced by it, emerged various W3C-specified API proposals [10, 11, 17]. Other useful and closely related W3C-specifications include full *Offline Web Applications* [9], *File API* [19], and *Web Workers* [12]. However, these specifications are out of the scope of this paper.

The rest of the paper is organized as follows. Section 2 examines very briefly the history of client-side storage. In Section 3, recently specified W3C API proposals for storing and manipulating persistent data on the client side are presented in detail. Next section introduces research projects, which utilize these modern client-side storage mechanisms. Finally, conclusions are drawn in Section 5.

2 Evolution

This section presents the evolution of legacy client-side storage mechanisms. The section starts by giving a very brief overview of the three popular storage mechanisms: *HTTP cookie* [3], *Flash cookie* [2], and *(Google) Gears* [7]. In the end, the features of these three mechanisms are compared against a set of evaluation criteria in order to highlight the similarities and differences of each mechanism.

2.1 HTTP Cookie

HTTP cookie[1] [3], invented by Netscape in 1994 and later standardized by IETF, is a popular mechanism that lets web servers and web clients to maintain and manage a stateful session between each other over the mostly stateless HTTP protocol. Cookies allow a small piece of simple key-value pair data (4 kB per cookie, 50 cookies per origin) to be stored in a user's web client (browser). Cookie data can be set to last over a single session (*session cookie*) or multiple sessions (*persistent cookie*). Typically, web applications use them for session management, authentication purposes, storing user preferences, and user tracking. All major browsers have native support for cookies.

Although cookies provide a simple and well-supported storing mechanism, they suffer from several problems. First, each cookie is sent back and forth with every HTTP request (via HTTP headers), which adds a lot of unnecessary overhead. Second, their storage size is way too small for modern web applications. Third, the same web application using cookies cannot be run in multiple web browser tabs simultaneously. Finally, the API is a bit clunky. [15, 16]

2.2 Flash Cookie

In 2002, Adobe introduced a feature called *Flash cookie (Local Shared Object)* [2], which is an Adobe Flash Player based mechanism to store and manage data on a user's computer. Flash cookies are exactly like HTTP cookies, except that they can store more complex data types than just plain strings and offer more storage space (defaults to 100 kB per origin). The easiest way to access Flash cookies from JavaScript is via the *ExternalInterface* object, which was introduced in Adobe Flash 8 in 2006.

In order to use Flash cookies, Adobe Flash Player 6+ plug-in needs to be installed to a web browser. Mobile browsers, however, are gradually moving away from Flash, and therefore the use of Flash cookies is no longer an option on mobile devices [1].

2.3 (Google) Gears

In 2007, Google launched *Gears*[2] [7] which offers web applications, among other features, the ability to store and manage persistent data locally on web clients, and access to it through the use of a JavaScript API and a variant of SQL. Gears comes with an embedded client-side relational database based on SQLite[3]. Compared to previous solutions, Gears' main advantage is its unlimited storage size.

In March 2011, Gears announced that there will be no new Gears releases in the future and newer browser versions will not be supported [8]. The decision was based on shifting the focus from Gears to HTML5, i.e., to help the W3C in defining and stan-

[1] Also referred to as a cookie, Web cookie, and browser cookie.
[2] Formerly Google Gears
[3] SQLite, http://www.sqlite.org/

dardizing Gears features and APIs in HTML5 and associated APIs. The Gears project still remains as open source and is available at `http://code.google.com/p/gears/`. In order to use Gears, Gears plug-in needs to be installed to a web browser. However, the plug-in works only with old and odd browser/operating system combinations, and therefore its use cannot be recommended.

Several high-level companies have used Gears as part of their products at some point, including Google's GMail and Docs as well as MySpace.

2.4 Summary

This section gave an overview of the three legacy client-side storage mechanisms: *HTTP cookie*, *Flash cookie*, and *(Google) Gears*. In Table 1, the mechanisms are compared against a set evaluation criteria to give a quick understanding and an overview of their differences.

Table 1. Comparison of legacy client-side storage mechanisms.

Client-side storage	Standardization	Main features	Supported data types	Storage space	Browser support
HTTP cookie	IETF RFC 6265, Standards Track, April 2011	simple key-value pair data	string (including string serialized JSON)	4 kB per cookie, 50 cookies per origin (recommendation)	all major browsers
Flash cookie (Local Shared Object)	proprietary browser plug-in (Adobe)	alternative to HTTP cookies, accessible via JavaScript	many (Flash data types)	100 kB per origin (can be increased)	all major browsers via Adobe Flash Player 6+ plug-in, excluding mobile browsers (Android and iOS Safari)
(Google) Gears	open-source browser plug-in (BSD License)	relational database (SQLite), uses a variant of SQL	many	unlimited (SQLite limitations)	deprecated (old browser versions via Gears plug-in)

3 Revolution: The HTML5 Approach

The W3C has specified three different API proposals for storing and manipulating persistent data on the client side: *Web Storage* [11], *Web SQL Database* [10], and *Indexed Database API* [17]. The motivation for creating the APIs was mainly fourfold: (1) standardize client-side storage access, (2) promote native browser support, (3) increase storage space, and (4) offer persistent data storage. Each API serves a specific purpose, and therefore has a different approach and advantages. However, the

common objective shared by all the APIs is to, one way or another, overcome the limitations of legacy client-side storage mechanisms. In the following subsections, these three HTML5 client-side storage mechanisms are presented along with a brief security analysis and feature comparison.

3.1 Web Storage

Web Storage[4] [11], a W3C Candidate Recommendation since December 2011, is the oldest and simplest of the three client-side storage APIs. It is a direct replacement to HTTP cookies and offers more advanced capabilities for storing small-sized data (e.g., session information or user preferences) on the client side through the use of JavaScript.

The API specification defines two mechanisms for storing and managing simple key-value pair data. The first mechanism is designed for storing session-related (short-lived) data, i.e., the data disappears once the current session is terminated, such as when a web browser or its tab is closed. The second mechanism is designed for storing persistent (long-lived) data, i.e., the data remains available beyond the current session and is even available across multiple web browser tabs. For both mechanisms, the same API is used. The only difference is that the data is accessed through the *sessionStorage* and *localStorage* objects, respectively.

Listing 1 shows a simple example of the API usage. In lines 1-5, data is defined in the JSON format. Lines 7-9 demonstrate how to serialize the JSON data as string and store it persistently with the given key into a local storage. Finally, the data is retrieved from the storage and converted back to JSON (cf. lines 11-13). In addition to storing and retrieving data, the API provides means for removing and clearing data from the storage as well as for listening to storage events fired when changes occur in the storage.

```
 1: // Define data
 2: var myData = {
 3:   "code": "T-111.5502",
 4:   "name": "Seminar on Media Technology B P"
 5: };
 6: ...
 7: // Store data
 8: localStorage.setItem( "1",
 9:   JSON.stringify( myData, null, "\t" ) );
10: ...
11: // Retrieve data
12: var myResult =
13:   JSON.parse( localStorage.getItem( "1" ) );
```

Listing 1. The Web Storage syntax example for storing and retrieving data.

[4] Also referred to as DOM Storage, HTML5 Storage, Local Storage, and Offline Storage.

When compared to cookies, Web Storage offers several advantages: (1) more storage space (5 MB per origin); (2) data is stored on the client side, i.e., no need to send data back and forth over the Internet with every web request; (3) better and more secure JavaScript API (same origin policy); (4) the same web application can be run in multiple web browser tabs simultaneously; (5) no need to set an expiration date; and (6) better performance.

Currently, Web Storage has the best browser support out of the three different API proposals by the W3C. The API is supported (either fully or partially) in every major browser, including mobile browsers. Table 2 lists minimum versions needed for each browser [16, 22].

Table 2. Browser support for Web Storage.

Chrome	Firefox	IE	Opera	Safari	Android	iOS Safari
4.0+	3.5+	8.0+	10.5+	4.0+	2.1+	3.2+

3.2 Web SQL Database

As the name implies, *Web SQL Database*[5] [10] enables storing and managing persistent (long-lived) relational data in client-side relational databases through the use of JavaScript APIs and a variant of SQL. For developers already familiar with server-side relational databases and SQL statements, the use of Web SQL Database may seem like a logical choice. The biggest difference between client-side and server-side relational databases is that in client-side databases the storage space is rather limited (defaults to 5 MB per origin).

Web SQL Database has several benefits over Web Storage: (1) more complex data types, (2) more complex queries (e.g., joins), (3) support for transactions and callbacks, and (4) both synchronous and asynchronous database APIs. It even supports prepared statements to prevent SQL injections. All this, however, comes with the expense of a more complex API.

Listing 2 gives an example of how to use the APIs. First, variables and data are defined (cf. lines 1-8). Next, a database is created/opened (cf. lines 10-12). Then, within a single transaction a table is created and initial data (incl. JSON as string) is stored in the database (cf. lines 14-23). Finally, the data is retrieved from the database and converted back to JSON (cf. lines 25-33). As can be seen, the amount of code is much larger and complex compared to Web Storage, but it comes with an added flexibility.

```
1: // Define variables
2: var myDatabase = null;
3: var version = "1";
4: // Define data
```

[5] Also referred to as WebDB.

```
 5: var myData = {
 6:   "code": "T-111.5502",
 7:   "name": "Seminar on Media Technology B P"
 8: };
 9: ...
10: // Create/open database
11: myDatabase = openDatabase( "myItemDB", version,
12:   "My Item Database", 2 * 1024 * 1024 );
13: ...
14: // Start transaction
15: myDatabase.transaction( function( dbTx ) {
16:   // Create table
17:   dbTx.executeSql( "CREATE TABLE IF NOT EXISTS
18:     myItems ( key TEXT UNIQUE, value TEXT )" );
19:  // Store data
20:  dbTx.executeSql( "INSERT INTO myItems VALUES
21:    ( '1', '" + JSON.stringify( myData, null, "\t" )
22:    + "' )" );
23: });
24: ...
25: // Start transaction
26: myDatabase.transaction( function( dbTx ) {
27:   // Retrieve data
28:   dbTx.executeSql( "SELECT * FROM myItems
29:     WHERE key = '1'", [], function( exeTx, exeRS ) {
30:     var myResult =
31:       JSON.parse( exeRS.rows.item( 0 ).value );
32:   });
33: });
```

Listing 2. The Web SQL Database syntax example: first a database is created/opened, after which data is stored and then retrieved.

As of 18th of November 2010, the specification work has stopped and the specification is no longer in active maintenance. The reason behind this decision was that all interested implementors have used the same SQL backend, i.e., SQLite, and therefore the standardization process cannot proceed any further. Thus, the usage of the API is no longer recommended and developers are advised to use Indexed Database API [17] for the coming web application development. In its current state, however, the API provides a viable option for storing data on mobile devices, as shown by Table 3 [21].

Table 3. Browser support for Web SQL Database.

Chrome	Firefox	IE	Opera	Safari	Android	iOS Safari
4.0+	—	—	10.5+	3.1+	2.1+	3.2+

3.3 Indexed Database API

Indexed Database API[6] or simply *IndexedDB* [17] is the newest and most sophisticated of the three client-side storage APIs. It offers a robust mechanism for storing and managing persistent (long-lived), advanced key-value pair data through the use of JavaScript APIs. Due to its advanced data management features, IndexedDB can be thought of as a combined and upgraded version of Web Storage and Web SQL Database.

IndexedDB is a NoSQL database (similar to MongoDB[7] and Apache CouchDB[8]), meaning it stores data in schemaless object stores instead of fixed-schema relational tables, as is the case in Web SQL Database [15]. The type of data that can be stored in object stores varies from primitive data types (e.g., string, array, and date) to hierarchical objects (e.g., JavaScript Object Notation, JSON). Supporting the JSON format natively is a huge advantage, because then there is no need to map the data between different data models and the problem of impedance mismatch can be completely avoided. The storage space available is also bigger than in other client-side storage options. For example, Firefox defaults to 50 MB per origin and the space can be increased upon request. Other its advantages include: (1) duplicate values for a key, (2) the utilization of indexes for efficient searches, (3) support for transactions and callbacks, and (4) both synchronous (may be removed in the future) and asynchronous database APIs. The disadvantage of the API lies in its complexity. Luckily, there are several wrappers over IndexedDB available, such as jQuery IndexedDB plugin[9].

Listing 3 demonstrates the usage of Indexed Database API as is without a wrapper. First, variables and data are defined (cf. lines 1-8). Next, a database is created/opened, after which it is initialized if needed, i.e., an object store is created and initial data (JSON) is stored in the object store (cf. lines 10-29). Finally, the data is retrieved from the object store and the database connection is closed (cf. lines 31-42).

```
1: // Define variables
2: var myDatabase = null;
3: var version = 1;
4: // Define data
5: var myData = {
```

[6] Also referred to as IndexedDB and WebSimpleDB API.
[7] MongoDB, http://www.mongodb.org/
[8] Apache CouchDB, http://couchdb.apache.org/
[9] jQuery IndexedDB plugin, http://nparashuram.com/jquery-indexeddb/

```
 6:   "code": "T-111.5502",
 7:   "name": "Seminar on Media Technology B P"
 8: };
 9: ...
10: // Create/open database
11: var openRequest = indexedDB.open( "MyItemDB" );
12: openRequest.onsuccess = function( event ) {
13:   myDatabase = openRequest.result;
14:   // Initialize database if needed
15:   if ( version != myDatabase.version ) {
16:     var setVersionRequest =
17:       myDatabase.setVersion( version );
18:       setVersionRequest.onsuccess =
19:         function( event ) {
20:         // Create object store
21:         var myObjectStore =
22:           myDatabase.createObjectStore( "myItems",
23:           { keyPath: "key" } );
24:         // Store data
25:         myObjectStore.add( { "key": 1,
26:           "value": myData } );
27:       };
28:   }
29: };
30: ...
31: // Start transaction
32: var myTransaction =
33:   myDatabase.transaction( [ "myItems" ] );
34: // Retrieve data
35: var myObjectStore =
36:   myTransaction.objectStore( "myItems" );
37: var getRequest = myObjectStore.get( 1 );
38: getRequest.onsuccess = function( event ) {
39:   var myResult = getRequest.result.value;
40: };
41: // Close database
42: myDatabase.close();
```

Listing 3. Indexed Database API syntax example: first a database is created/opened, after which data is stored and then retrieved. Finally, the database is closed.

Currently, browser support for Indexed Database API is rather poor, as shown by Table 4 [20]. This results from the fact that the specification is still a work in progress. However, browser support is expected to improve in the near future.

Table 4. Browser support for Indexed Database API.

Chrome	Firefox	IE	Opera	Safari	Android	iOS Safari
11.0+	4.0+	10.0+	—	—	—	—

3.4 Fallbacks and Wrappers

As we have seen in the previous subsections, browser support for these three HTML5 client-side storage mechanisms varies from relatively good to virtually non-existent. The lack of cross-browser support makes each mechanism risky to use, especially in large-scale public projects.

Luckily, there are several libraries available that help developers to obtain cross-browser support for their web applications. Paul Irish's collection of HTML5 Cross Browser Polyfills [14] is probably the best source of information for finding wrappers and fallbacks to practically any new HTML5 feature, include the aforementioned client-side storage mechanisms.

Typically, those storage libraries wrap several APIs under a single unified API: one HTML5 storage API and one or more fallbacks mechanism APIs. The API can be based on either a W3C proposed API or a custom API. When an API is used, it first tries to use a desired HTML5 client-side storage mechanism. In case the user's browser does not support the desired mechanism natively, a fallback mechanism (e.g., HTTP cookies or Gears) is used. Developing in this future-proof way means that when users upgrade their browsers, the code does not have to change and users will automatically move to the better, native experience.

3.5 Privacy and Security Concerns

There are several privacy and security concerns related to the aforementioned HTML5 client-side storage mechanisms that both developers and end users should be aware of.

Regarding end-users' privacy, the biggest threat is *user tracking*. In user tracking, a third-party advertiser could use its client-side storage area to track a user across multiple sessions, building a profile of the user's interests to allow for highly targeted advertising. Although there are a number of techniques that can be used to mitigate the risk of user tracking (e.g., expiring stored data), they do not block it altogether. [10, 11, 17] Other privacy-related concerns are the use of HTTP cookies as redundant backup (*cookie resurrection*) and vice versa as well as the possibility to store *sensitive data* (e.g., calendar appointments and health records) in either private or public web clients [17].

All HTML5 client-side storage mechanisms have a strict *same-origin policy* [4], meaning different websites (origins, to be precise) cannot access data created by another website (origin). However, this does not mean that the data stored in client-side

storage should be trusted, as a malicious user may access the data (e.g., by using Chrome's Developer Tools) and manipulate it as much as s/he wants [15]. Other security-related concerns are *DNS spoofing attacks* (preventable by using TLS and certificates); *cross-directory attacks*, i.e., the usage of client-side storage mechanisms on shared-origin websites (e.g., `http://xxxxxxx.blogspot.com`); both *SQL injection attacks* (always use prepared statements) and *JavaScript injection attacks (XSS)*; and, of course, the risk of poor-quality implementations that could potentially result in *information spoofing* [10, 11, 17].

3.6 Summary

This section gave an overview of the three HTML5 client-side storage mechanisms: *Web Storage*, *Web SQL Database*, and *Indexed Database API (IndexedDB)*. In Table 5, the mechanisms are compared against a set evaluation criteria to give a quick understanding and an overview of their differences.

Table 5. Comparison of HTML5 client-side storage mechanisms.

Client-side storage	Standardization	Main features	Supported data types	Storage space	Browser support
Web Storage	W3C, Candidate Recommendation, December 2011	simple key-value pair data, no duplicate values for a key	string (including string serialized JSON)	5 MB per origin (recommendation, can be increased)	Chrome 4.0+ Firefox 3.5+ IE 8.0+ Opera 10.5+ Safari 4.0+ Android 2.1+ iOS Safari 3.2+
Web SQL Database	W3C, Working Group Note, November 2010	relational database (SQLite), uses a variant of SQL, supports transactions and callbacks, includes synchronous and asynchronous APIs	many (SQL data types)	5 MB per origin (recommendation, can be increased)	Chrome 4.0+ Firefox — IE — Opera 10.5+ Safari 3.1+ Android 2.1+ iOS Safari 3.2+ (deprecated since November 18, 2010)
IndexedDB	W3C, Working Draft, May 2012	indexed and hierarchical key-value pair data, duplicate values for a key, indexes, supports transactions and callbacks, includes synchronous and asynchronous APIs	many (e.g., JSON, array, string, and date)	50 MB per origin (Firefox, can be increased)	Chrome 11.0+ Firefox 4.0+ IE 10.0+ Opera — Safari — Android — iOS Safari —

4 Related Research Projects

This section presents research projects that in one way or another utilize recent client-side storage mechanisms. The focus is laid especially on two research projects, *Sync Kit* [5] and *Silo* [18], but other related research projects are also briefly covered.

4.1 Sync Kit

Sync Kit [5] is a client/server toolkit for improving the performance of data-intensive websites. Sync Kit uses JavaScript and *(Google) Gears* on the client side, meaning the approach requires the installation of the Gears plug-in in order to work. Gears could be, however, replaced with a W3C-specified client-side storage mechanism to get rid of the requirement. On the server side, a Sync Kit aware web server is required. In their case, Sync Kit was implemented as a collection of scripts for the Django[10] web framework.

The idea behind Sync Kit is to use Gears to store (cache) web page templates on the client side. Templates include a JavaScript library and data endpoint definitions to access dynamic contents residing on the server. Data endpoints, on the other hand, cache database objects to Gears and keep the cached data consistent with the backend database. When a browser requests a web page for the first time, the template is returned as a response and stored in the client-side storage. Then, new data is requested via template's data endpoints. Finally, the retrieved data is added to the template, cached on the client-side storage, and the result is displayed to the user.

According to Benson *et al.* [5], their solution reduces server load by a factor of four and data transfer up to 5% compared to the traditional approach, when cache hit rates are high.

4.2 Silo

Silo [18] is a framework for deploying fast-loading web applications. Silo uses JavaScript and *Web Storage* on the client side (both standard web technologies), meaning the approach works on all modern web browsers. On the server side, a Silo-aware web server is required.

The idea behind Silo is to use Web Storage to store (cache) website's JavaScript and CSS chunks (2 kB in size) on the client side. When a browser requests a web page, the server returns a modified version of the web page containing a small JavaScript shim with a list of required chunk *ids*. Next, the shim checks the client-side storage for available chunks and informs the server of the missing chunks using their *ids*. Then, the server replies with the raw data for the missing chunks. Finally, the web page is reconstructed into its original form on the client using the JavaScript shim. Moreover, the second HTTP round trip can be completely eliminated by utilizing HTTP cookies in the initial HTTP request for sending the already available chunk *ids*.

[10] Django, https://www.djangoproject.com/

Silo was evaluated with multiple real-world websites, such as CNN, Twitter, and Wikipedia. Based on their experiments, Silo can reduce web page load times by 20-80% for web pages with large amounts of JavaScript and CSS.

4.3 Other Projects

Client-side storage mechanisms have also been used to solve other usage scenarios. Cannon and Wohlstadter [6] implemented an automated object persistence framework, which reduces the amount of work needed by developers to support offline web applications. Ijtihadie *et al.* [13] presented a prototype of an offline mobile web application for synchronizing quiz for e-learning activities, which was built upon *Offline Web Applications* and *Web Storage*.

5 Conclusion

In this paper, distinct approaches for storing and managing data on the client side were covered. For a long time, simple *HTTP cookies* were the only considerable option and they were mainly used for session management due to their limited storage space. The emergence of *(Google) Gears* revolutionized the area with its "unlimited" storage space and rich set of features, which also paved the way towards standardizing client-side storage mechanisms.

Currently, the W3C provides three API proposals for persistent (session or local) data storage: *Web Storage*, *Web SQL Database* (deprecated), and *Indexed Database API (IndexedDB)*. Web Storage offers a simple key-value pair data storage mechanism supported by all major browsers, whereas IndexedDB includes a number of advanced features but is still a work in progress. Both mechanisms clearly have a future, as their scopes and benefits are distinct as well as they can be used to complement each other. Recent studies have also shown how the usage of client-side storage mechanisms can significantly improve both the performance and user experience of (mobile) web applications.

References

1. Adobe. *An Update on Flash Player and Android*. June 2012. https://blogs.adobe.com/flashplayer/2012/06/flash-player-and-android-update.html.
2. Adobe. *What are Local Shared Objects?* 2012. http://www.adobe.com/security/flashplayer/articles/lso/.
3. Barth, A. (eds.). *HTTP State Management Mechanism*. IETF RFC 6265 (Standards Track), April 2011. http://tools.ietf.org/html/rfc6265.
4. Barth, A. (eds.). *The Web Origin Concept*. IETF RFC 6454 (Standards Track), December 2011. http://www.ietf.org/rfc/rfc6454.txt.
5. Benson, E., Marcus, A., Karger, D., and Madden, S. Sync Kit: A Persistent Client-Side Database Caching Toolkit for Data Intensive Websites. In *Proceedings of the 19th Interna-*

tional Conference on World Wide Web (WWW '10), pages 121-130. ACM, 2010. DOI: 10.1145/1772690.1772704.

6. Cannon, B. and Wohlstadter, E. Automated Object Persistence for JavaScript. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*, pages 191-200. ACM, 2010. DOI: 10.1145/1772690.1772711.
7. Google. *Gears — Improving Your Web Browser*. 2010. http://code.google.com/p/gears/.
8. Google. *Stopping the Gears*. March 2011. http://gearsblog.blogspot.fi/2011/03/stopping-gears.html.
9. Hickson, I. (eds.). *Offline Web Applications — HTML5*. March 2012. http://www.w3.org/TR/html5/offline.html#offline.
10. Hickson, I. (eds.). *Web SQL Database*. W3C Working Group Note, November 2010. http://www.w3.org/TR/webdatabase/.
11. Hickson, I. (eds.). *Web Storage*. W3C Candidate Recommendation, December 2011. http://www.w3.org/TR/webstorage/.
12. Hickson, I. (eds.). *Web Workers*. W3C Candidate Recommendation, May 2012. http://www.w3.org/TR/workers/.
13. Ijtihadie, R.M., Chisaki, Y., Usagawa, T., Cahyo, H.B., and Affandi, A. Offline Web Application and Quiz Synchronization for e-Learning Activity for Mobile Browser. In *Proceedings of the 2010 IEEE Region 10 Conference (TENCON '10)*, pages 2402-2405. IEEE, 2010. DOI: 10.1109/TENCON.2010.5685899.
14. Irish, P. *HTML5 Cross Browser Polyfills*. July 2012. https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills.
15. Kessin, Z. *Programming HTML5 Applications: Building Powerful Cross-Platform Environments in JavaScript*. O'Reilly Media, 2011. ISBN: 978-1-449-39908-5.
16. MacDonald, M. *HTML5: The Missing Manual (1st ed.)*. O'Reilly Media, 2011. ISBN: 978-1-449-30239-9.
17. Mehta, N., Sicking, J., Graff, E., Popescu, A., and Orlow, J. (eds.). *Indexed Database API*. W3C Working Draft, May 2012. http://www.w3.org/TR/IndexedDB/.
18. Mickens, J. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads. In *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps '10)*, pages 99-110. USENIX, 2010.
19. Ranganathan, A. and Sicking, J. (eds.). *File API*. W3C Working Draft, July 2012. http://www.w3.org/TR/FileAPI/.
20. When can I use… *IndexedDB*. http://caniuse.com/#search=IndexedDB.
21. When can I use… *Web SQL Database*. http://caniuse.com/#search=Web%20SQL%20Database.
22. When can I use… *Web Storage*. http://caniuse.com/#search=Web%20Storage.