Offline-provisioning and Synchronization of Content for Mobile Webapps

Tim A. Majchrzak[1] and Timo Hillmann[2]
[1]*University of Cologne, Köln, Germany*
[2]*University of Münster, Münster, Germany*

Keywords: Offline, Storage, Synchronization, Webapp, Mobile, Mobile Webapp, Web Application, App, Mobile Computing.

Abstract: Despite progress in the development of cross-platform frameworks, today's apps often are realized either as native apps or as Webapps. The latter are easy to implement and rely on robust and future-proof technologies such as HTML5, Cascading Style Sheets (CSS) and JavaScript. However, Webapps commonly require a stable Internet connection. Recent developments facilitate offline-enabled Webapps. If an app is able to store data locally, it should also be enabled to synchronize with a server backend. We assess current HTML5 storage capabilities and discuss their possibilities along with considerations of mobile computing synchronization strategies. Our work includes the compilation of requirements that Webapps have regarding their offline capabilities. Moreover, we discuss the status quo of offline storage and give an outlook based on a proof-of-concept implementation.

1 INTRODUCTION

The market for mobile devices has seen unprecedented growth. Less than 10 years ago – the first iPhone has been introduced in 2007 (Macedonia, 2007) – the proliferation of smartphones and tablets had not even started. Nowadays, these devices are routinely used by million peoples. They are not mere tools for entertainment but increasingly employed by businesses, too. This applies both to internal purposes and for keeping contact with customers (Majchrzak and Heitkötter, 2014). The success of mobile devices can only partly be attributed to their hardware, though. In fact, software makes them useful and versatile.

Applications for mobile devices (apps) can roughly be subdivided into native apps and Webapps (Charland and Leroux, 2011). The first make use of a device's platform and are developed using native software development kits (SDKs). The latter employ Web technologies such as HTML, CSS and JavaScript (Charland and Leroux, 2011). While native apps are bound to a platform, e.g. Android or iOS, Webapps run on any device that provides a Web browser. Despite progress in developing cross-platform solutions (Ohrt and Turau, 2012), many businesses face the choice of developing *either* natively *or* using Web technologies.

Webapps have several shortcomings. Most notably, they cannot be installed *directly* on devices if not designed as a hybrid app (Heitkötter et al., 2013). Moreover, they provide a Web-like graphical user interface (GUI) that cannot make use of platform-specific elements (e.g. *Widgets*). Much progress has been achieved with *HTML5* (2014): cf. e.g. Zorrilla, Martin, Sanchez, Tamayo, and Olaizola (2014); Karthik, Patlolla, Sorokine, White, and Myers (2014); Weeks (2014). Still, if a native look & feel is important (Majchrzak and Heitkötter, 2014), native apps or a cross-platform approach yielding native apps (cf. Heitkötter, Majchrzak, and Kuchen (2013))are preferable. However, Webapps are rather easy to develop and their interoperability is preeminent (Quilligan, 2013). They are also future-proof due to not being bound to the fast-paced change mobile platforms face[1]. In addition, development framework support is excellent (Heitkötter et al., 2013). Thus, mobile Webapps are frequently chosen for app development.

Due to the mobile nature of smartphones and tablets, a stable Internet connection is not guaranteed. There might be times without connectivity; connection quality in terms of bandwidth, latency and jitter is ever

[1]To be truly future-proof, careful decision making e.g. with regard to employed frameworks is necessary. However, in comparison the dependence on multiple technologies that have short release circles is low.

A. Majchrzak T. and Hillmann T..
Offline-provisioning and Synchronization of Content for Mobile Webapps.
DOI: 10.5220/0005423206010612
In *Proceedings of the 11th International Conference on Web Information Systems and Technologies* (WEBIST-2015), pages 601-612
ISBN: 978-989-758-106-9

changing – a fundamental challenge of mobile computing (Satyanarayanan, 1996). Thus, many apps gain from keeping (at least some) functionality even in an *offline* situation (Gonçalves and Leitão, 2009). Often more profound functionality is advisable. If data resides in apps *and* in backend systems, synchronization is an extended requirement.

Until lately, offline capabilities have been a weak spot: most Webapps simply do not work without connectivity. Recent progress has greatly improved the theoretical capabilities, though (cf. (HTM, 2014); details are covered by newly published textbooks such as those by Makzan (2014) and Felke-Morris (2014)). Therefore, these capabilities should be assessed. For this purpose we introduce strategies for offline-enabled Webapps and for their synchronization. Moreover, requirements for offline functionality are compiled and used to realize a proof-of-concept app.

This article makes several contributions. Firstly, the background of offline storage and synchronization w.r.t. mobile computing is summarized. Secondly, HTML5 storage technologies are assessed. Thirdly, requirements for Webapps with offline capabilities are compiled. Fourthly, the status quo and future developments are discussed.

This paper is structured as follows. Sections 2 and 3 introduce offline-provisioning and successively mobile synchronization. Storage in HTML5 is explained in Section 4. Section 5 presents an evaluation case study. Results are discussed in Section 6 before Section 7 draws a conclusion. Rather than discussing related work separately, we have incorporated applicable references throughout the paper and highlight implications.

2 OFFLINE-PROVISIONING

In a mobile environment, data management is generally influenced by two factors. Firstly, mobile devices tend to have limited storage capabilities – at least compared to PCs and especially servers. This storage usually cannot be extended[2] and has to be shared with several other applications as well as the operating system (respectively the platform). Secondly, Internet connectivity may be unreliable. The connection can vary in speed and disconnections may occur (cf. Marco, Gallud, Penichet, and Winckler (2013)). These aspects need special attention when designing data-heavy applications in a mobile context.

[2]Some devices allow extension with removable media such as memory cards. However, this storage cannot be used for all purposes and tends to be quiet limited in size, too.

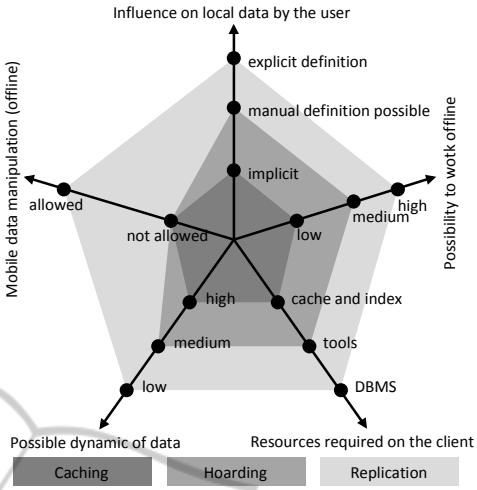

Figure 1: Classification of caching, hoarding and replication (based on the work by Höpfner et al. (2009)).

For offline-provisioning, literature differentiates between *data caching*, *hoarding* and *replication methods* (Höpfner et al., 2009). A summary of their capabilities is given in Figure 1. However, these terms are not precisely defined and sometimes used interchangeably. In also has to be noted that work on offline-capabilities in the mobile context not necessarily presents a classification of own ideas with regard to theory. This particularly concerns offline Web access as e.g. covered by Ananthanarayanan, Blagsvedt, and Toyama (2006).

A *cache* is a memory component that is used to store data with the aim of reducing access time (Vora et al., 2004). Generally, a cache has a considerably faster access time compared to other memory components. If requested data is stored in the cache, it will be served from the cache. Otherwise, the data will be served by reading from the original storage, thus making the request comparatively slower. Since cache size is very limited, usually replacement strategies are applied in order to free space for *new* data. If requested data cannot be served from the cache and there is no space left, parts of the cached data have to be removed. There are various criteria that can be evaluated when selecting data for this purpose. Client-side caching of Web content is a common approach (Marco et al., 2013), yet limited in functionality (Cannon and Wohlstadter, 2010).

Semantic caching is a special caching method in the context of mobile computing. This technique is based on the concept of *semantic locality*, i.e. subsequent requests are often semantically related (Dar et al., 1996). For instance, search engine queries are often related with regard to content. Consequently, search results are overlapping. If these results are

stored in a cache, it is possible to answer a following query partially by reading the cache, thus reducing request time.

However, caching is commonly not appropriate to store data for offline-enabled Webapps (Höpfner et al., 2009). Due to the limited capacity, data availability cannot be guaranteed. The replacement strategy determines which data will be removed and can, thereby, be a source of error. The lack of supporting local data modifications effectively makes caching read-only storage.

Hoarding methods use specially-designed analysis tools to select the hoarded data. For instance, tools could analyse usage patterns to select frequently used data. Consequently, the user does not have explicit influence on the selection. Similar to caching, hoarding techniques employ replacement policies to cope with limited storage (Vora et al., 2004). Areas of application are distributed file systems, for example. The UNIX-based *Coda File System* (2014) analyses the file accesses of its users in order to store semantically related files locally. Another variation of hoarding is used to extend database queries. The increased result set of a query is then stored locally.

In the context of offline-provisioning, hoarding methods have limited suitability. Similar to caching, hoarding does not support local data modifications (but with special approaches as e.g. proposed by Vora et al. (2004)). Consequently, hoarding methods are read-only. In contrast to caching, hoarding can store data that was not requested (but e.g. *anticipated* to be needed, also cf. *prefetching* (Gonçalves and Leitão, 2009)). However, it still makes use of replacement strategies to cope with limited storage. Thus, data availability cannot be guaranteed either.

Data replication is used to increase data availability and to reduce access time. Typically, it is employed in a client-server system. A server provides data, which a client stores locally. Subsequent requests can be served by reading the local database. Thus, requests to the server are minimized. Replication is also used in the context of databases. It enables the distribution of data and requests to multiple databases. As a result, reachability and reliability are increased.

The local copies of the data are called *replicates*. They are selected explicitly by the app or the user. Replicates are *changeable* if all three base operations (add, edit and delete) can be applied, or *unchangeable* otherwise. Synchronization between the master database and clients is necessary to ensure data consistency when replicates can be modified.

Since replacement strategies are not used for data replication, it is important to guarantee sufficient storage space availability. Similar to caching, replication is transparent (Weikum and Vossen, 2001, p. 674) to the app. In contrast to caching or hoarding, replication is fit to support offline-provisioning and can be used for offline-enabled Webapps.

3 MOBILE SYNCHRONIZATION

In many business scenarios, simple offline capabilities such as offline Web browsing (Kao et al., 2011) are not sufficient (Goncalves and Leitao, 2007). Data is not only used offline but might be changed at times without connectivity. To make the situation worse, data could be altered in its master storage (usually a backend server) concurrently.

Synchronization ensures data consistency and coherence between multiple systems. All data handled by replication methods is redundant; consequently, it is vital to keep replicates consistent, i.e. identical. Modifications have to be distributed to all copies. Usually there are a master database, which stores the original data, and replicated databases. Synchronization is possible between the master and exactly one replicant (*one-to-one* synchronization) or with multiple replicants simultaneously (*one-to-many* synchronization).

The synchronization process is initialized by one of the involved systems. With *symmetric* synchronization, all systems are capable of starting this process. If either the master database or the replicated databases are able to initiate the synchronization, the process is *asymmetric*. Synchronization can be started *periodically* or at *predefined* times. Moreover, it can be initialized *on commit*, i. e. when data was changed, or *on demand*, e.g. started explicitly by the user.

The *resolution of inconsistency* can either be based on the data sets or on the transactions (König-Ries et al., 2007). In the first case, the master database receives two copies of the affected data set. One copy was created before the data was changed, and the other one is the resulting data set after the changes. The master database itself possesses the current state of the data set. These three elements can then be used to reestablish consistency. In the latter case, the transaction itself will be passed to the master database as well.

Moreover, synchronization can be performed either *incrementally* or *completely* (König-Ries et al., 2007). Incremental synchronization is constrained to the data sets that changed since the last synchronization. Consequently, considerably less data is transferred compared to a complete synchronization. As a special form, synchronization can be limited to a subset of the data (*selective* synchronization). A possible other distinction is *hot* and *cold*, whereas hot copies have current (but small) changes, and cold copies are old

ones (Cannon and Wohlstadter, 2010).

Lastly, it is differentiated between *synchronous* and *asynchronous* synchronization. Asynchronous – also: *lazy* (Weikum and Vossen, 2001, p. 778) – synchronization does not require permanent connectivity, which is crucial in a mobile context.

Conflicts can occur if data is changed in two databases without synchronization in between (Gaubatz et al., 2014). Two strategies exist for coping with conflicts (Weikum and Vossen, 2001, p. 129). The first is to prevent them by locking (*pessimistic*). The second applies conflict resolution (*optimistic*). However, a conflict has to be detected in order to be resolved (Lam et al., 2002). For that purpose, every data element needs a unique *identifier* (e.g. a running number of a hash value calculated from the data) and a way to *distinguish* its versions such as a version number or a time stamp.

Four different types of conflicts can be distinguished (König-Ries et al., 2007). A *deletion conflict* occurs when a replicate is deleted, whose corresponding original was changed in the meantime. Since this cannot be resolved automatically, user intervention is necessary. *Insertion conflicts* occur when a data element is created and its identifier (ID) is already assigned to another element in the master database. This can be resolved by reassigning IDs or be prevented by assigning a pool of IDs to a client (key-pool-scheme). These IDs can then be used when creating new elements. A *modification conflict* occurs when two colliding changes to a single data element are made. This might be solved by exploiting the nature of the corresponding data. For instance, the arithmetic mean can be calculated for data representing a numeric value. Lastly, a *modification-deletion conflict* occurs when a replicate is modified, whose corresponding original was deleted. This can be resolved by reversing the deletion.

4 STORAGE IN HTML5

HTML5 introduces five distinctive storage technologies, which can be used to provide Webapps with offline capabilities. These technologies have not only be designed with different aims in mind (and are based on different backgrounds of technological evolution), but actually differ w.r.t. their functionality. Unsurprisingly, their practicability varies by usage scenario. Moreover, browser support greatly differs. We, therefore, provide an overview of minimum version requirements in Table 1. Please note that some technologies are not even supported in the latest version of some browsers (denoted by "–"). As a particularity of this section, most sources refer to (usually the official) Web sites concerning the respective technology rather than scientific articles. This not only is a intentional choice of primary sources but also a necessity since scientific sources are scarce.

Table 1: Browser support of storage technologies.

	Chrome	Firefox	Safari	Internet Explorer	Android	iOS
AppCache	4.0	3.5	4.0	10.0	2.1	3.2
Web Storage	4.0	3.5	4.0	8.0	2.1	3.2
Web SQL	4.0	–	3.1	–	2.1	3.2
IndexedDB	23.0	10.0	7.1	10.0*	4.4	8.0
File API	13.0	3.6*	6.0*	10.0*	4.4*	6.1*

* limited support (cf. Deveria (2014) for an overview)

The *application cache* (AppCache) enables users to access a Webapp without using an Internet connection (Berjon et al., 2013). Since resources, such as HTML documents or images, are cached locally, the AppCache reduces server load and request time (Bidelman, 2010). A HTML document can be added explicitly by listing it in the manifest file or implicitly by adding the manifest attribute to the document. Other resources have to be listed in the manifest file in order to make use of the application cache. Furthermore, it is possible to specify resources that require a Internet connection as well as fallback pages for inaccessible ones. However, it is not possible to manually remove or update files from the cache through the API (Berjon et al., 2013). Furthermore, the cache is only updated if the manifest file is modified or users manually clear the cache in their browser (Bidelman, 2010). An example for a cache manifest file is given in Listing 1.

```
1 CACHE MANIFEST
2 # Time: Wed Sep 17 2014 14:06:47 GMT
3
4 CACHE:
5 scripts/source.min.js
6 scripts/source.min.js.map
7 css/styles.min.css
8 img/sprites.png
9 img/loading.gif
10
11 NETWORK:
12 *
```

Listing 1: Exemplary manifest file.

Due to the nature of the application cache, it works best if the application's logic and content are strictly separated. Then the cache can be used to store the Webapp's logic (Archibald, 2012). However, it is not suited to store dynamic application data or user data. The AppCache is supported by almost all current

Table 2: Comparison of storage technologies

	AppCache	**Web Storage**	**Web SQL**	**IndexedDB**	**File API**
Basic storage technology	File system	Key-value store	Relational DBMS	Document-oriented DBMS	File system
Supported data types	File	String	All SQL data types	String, Number, Object, Array, BLOB	File, Directory
Query language	*(none)*	No actual query language	SQL	Specified by the IndexedDB API	No actual query language
(Database-) queries	*(none)*	Queries possible by merely iterating over files	SQL queries can be used to full extent	Complex queries are possible	No queries possible; files can be requested directly or as part of folders
Synchronicity	Unknown	Synchronous	Asynchronous		
Memory limit	Dependent on the implementation of the Web browser				
Browser support	Broad	Broad	Mainly on mobile devices	Mainly by desktop browsers	Limited (so far)
Standardized	yes	yes	not anymore	yes	yes
Applicable for…	Application logic	Unstructured data	Complex data structures	Complex data structures; particularly when using JavaScript objects and JSON	Files

browsers.

Local storage and *session storage* are two HTML5 storage methods that are subsumed under the term *Web storage* (Hickson, 2013). While local storage stores data *persistently*, session storage stores data only for the *current session* (Hickson, 2013). If the browser or Web site is closed, all data stored using session storage will be deleted. For instance, session storage can be used to save credentials. When using Web storage, data is stored as a key-value pair consisting of strings. Consequently, data can only be accessed through the corresponding key. Web storage is sometimes regarded as an *evolution* from cookies. However, cookies are limited to four kilobytes of storage (Kristol and Montulli, 1997). In general, Web storage has no such limitation although storage capacity depends on the browser implementation. As a limitation, the server cannot access data stored using Web storage. Web storage is supported by all major browsers.

Web SQL is an API to store data in a relational DBMS (Hickson, 2010). SQL can be used to full extent to query the database. This allows complex queries with joins and sub-queries. Because relational databases are widespread, it makes sense to use them on the client, too. In most cases, the usage of Web SQL requires object-relational mapping (ORM) due to the usage of JavaScript objects (Green, 2011). Web SQL is supported by mobile browsers, mostly. Since Web SQL is no longer a standard – it was on the "W3C Recommendation track but specification work has stopped" (Hickson, 2010) –, it is unclear whether support will increase or decline.

IndexedDB is an API to store data in an object-oriented database ("*Basic concepts – IndexedDB*", 2013). The database consists of object stores, which are roughly equivalent to database tablets. The object stores contain the individual data sets. Similar to Web storage, key-value pairs are kept. However, several other data types are supported by IndexedDB, including strings, number types, and BLOBs. Furthermore, it maintains indexes, which allow complex queries. When using JavaScript objects, IndexedDB is a natural choice because these objects can be saved without manual serialization (Green, 2011). Browser support for IndexedDB had been confined to desktop browsers. Since becoming a standard, support has extended to mobile browsers.

File API consists of several programming interfaces, which allow file manipulation (Ranganathan and Sicking, 2013; Uhrhane, 2012). In addition, FileSystem API and FileWriter API have been added to File API. After getting access to a local file system, it is possible to read, create and delete files and directories. File API differentiates between persistent and temporary storage. Files that are stored in the temporary storage might be deleted by the browser in case more space is needed (Bidelman, 2011). Persistent storage has to be granted by the user. Areas of application are file uploads and handling of media files, for example. Since file operations are executed in a local sandbox file system, other data on the devices is safe from unauthorized access. While File API is supported in the recent version of most browsers, FileSystem API and FileWriter API so far only work in Chrome.

With the Quota API it will become possible to request storage memory for a Webapp from the browser (Yasuda, 2013). This should improve memory management without adding complexity.

The comparison of storage technologies is summarized in Table 2, which can also be used as a reference. Due to the differences of the technologies, it is not practical to give a one-choice-fits-all-scenarios recommendation. In fact, choice should be made after carefully waging arguments. Situation-sensitive advise is given in Section 6.2.

5 EVALUATION CASE STUDY

A case study is presented in the following. We first sketch requirements that represent a common scenario for offline provisioning. We then propose a design for the app and discuss lessons learned from the implementation.

5.1 Requirements

During the design of an offline-enabled Webapp, certain requirements have to be considered. There are basically two greatly differing perspectives: the end-user's and the developers' point of view.

First of all, the Webapp has to be accessible without connectivity. The application logic has to be saved locally. Therefore, the AppCache has to be used mandatory. Depending on the features as well as the complexity of the data management of the Webapp, a suitable offline-provisioning method (as illustrated in the prior section) has to be chosen. Transparency is an important aspect in this context. It should be irrelevant for users whether or not they are online or offline. The choice of storage technology depends on the data structure. For instance, File API can be used when developing a file system. In order to resolve inconsistencies when working with replicates, at least basic synchronization is required. For the design of the synchronization method, the aspects identified in Section 3 have to be considered. Detection of the connection status is a requirement, too (how this is possible is e.g. described by by Hales (2012)). Moreover, it is necessary to detect changes in connectivity. For example, it might be useful to initiate synchronization right when an Internet connection is established.

A change in the manifest file for the AppCache is detected after one page refresh. But at this point the app is still served from the cache. After another page refresh, the cache gets emptied and the updated app is stored in it. To avoid problems, the second refresh has to be initiated automatically.

In relation to the developer perspective, we identified three main requirements. To make use of the AppCache efficiently, the app has to be structured in a specific way. Unlike normal Web sites, an offline-enabled Webapp needs a separated application logic (cf. Section 4). Consequently, an offline-enabled Webapp cannot follow a *thin client* architecture (Goncalves and Leitao, 2007), like most Webapps do. Furthermore, the build process has to support the developer with the creation of the manifest file. Additionally, browser-specific differences of the HTML5 implementation have to be carefully considered (cf. Section 4).

5.2 Specification and Design

A *ToDo list* app was implemented for evaluation – a simple but practicable choice for an offline-capable app (Kao et al., 2012). Its main functions revolve around the management of tasks. A registered user should be able to add, edit and delete tasks. Moreover, the app should provide the flagging of tasks as done and as important. Obviously, the proposed Webapp should work offline.

In order to make the tasks accessible from multiple devices, they have to be stored in a remote database. Therefore, a client-server architecture was chosen. Usually, Webapps realize a three-tier architecture to separate user interface (*presentation*), application logic (*application*) and data storage (*data*). Moreover, Webapps generally feature a distributed presentation tier. The (Web) server provides content, which is rendered by the browser on the client. An application server and database complete the architecture. However, this architecture is inadequate for offline-enabled Webapps. If the client only consists of the presentation, business logic and data storage are inaccessible when no connection is available. Therefore, a rich client is needed (overruling the usual three-tier architecture), which provides functionality independently.

As a consequence, the client was designed as a *single-page application* (SPA) (Flanagan, 2006, p. 128). In general, a SPA is closer to a native user experience because there are no page reloads. Instead, the user navigates between different states of the Webapp. Typically, a SPA consists of views, which render the templates and manipulate the DOM (Document Object Model), and models, which interact with the (local and external) data storage (Takada, 2012). In addition, the application logic is situated in the models. Moreover, there is a routing component, which manages the different states of the app and the navigation between them. The DOM emits events such as clicking or scrolling for handling by the Webapp.

The server was designed as a *RESTful* Web service.

Since a SPA moves the application logic to the client, the server only needs to provide access to the database and, therefore, is less complex. This is also referred to as *thin server architecture* (Kovatsch et al., 2012). A Web service API is RESTful if it adheres to the architectural constraints defined by REST (Representational State Transfer) (Roth, 2009). REST simplifies the architecture, access and modification of resources, and eases offline-provisioning due to its statelessness. An additional benefit is the improved scalability, which e.g. allows load balancing (Rodriguez, 2008).

AJAX (Asynchronous JavaScript And XML) is used for the communication between client and server. AJAX uses XMLHttpRequests (XHR) to send and retrieve data from the server asynchronously. These requests are commonly used in synchronization contexts (cf. with Cannon and Wohlstadter (2010)). Despite the name, AJAX is not limited to XML. In fact, *JSON* is often used as the data exchange format. When using JSON, it is sometimes referred to as *AJAJ* (Hurth, 2010). In contrast to XML, JSON is less complex, more compact, and easier to use. Consequently, JSON is suited for Webapps. JSON supports JavaScript objects, arrays, strings, numbers, booleans and `null` values. Furthermore, it is possible to nest objects and arrays. Listing 2 shows an example for a JSON representation of a task.

```
{
 "_id":      "a01mVP9RL9rA",
 "title":    "Example",
 "descr":    "This is an example",
 "dueTo":    1379059200000,
 "done":     true,
 "favorite":  false,
 "_creator":  "Ip6OHG4q3OGw",
 "_created":  1379035100000,
 "_modified": 1379042200000,
 "_deleted":  false
}
```

Listing 2: JSON representation of a task.

5.3 Implementation and Outcome

Two screenshots in Figure 2 illustrate the example. In order to support creating, editing and deleting tasks offline, a replication and a synchronization method were implemented.

IndexedDB was chosen as the storage technology, since it works well with JavaScript objects and JSON. Furthermore, the server uses a compatible database; no serialization or transformation of the data was necessary. In order to expand the support of IndexedDB to mobile browsers, a *polyfill* (Sharp, 2010) based on Web SQL was used (namely indexeddb.shim.js (Narasimhan, 2014)). The structure of a task can be

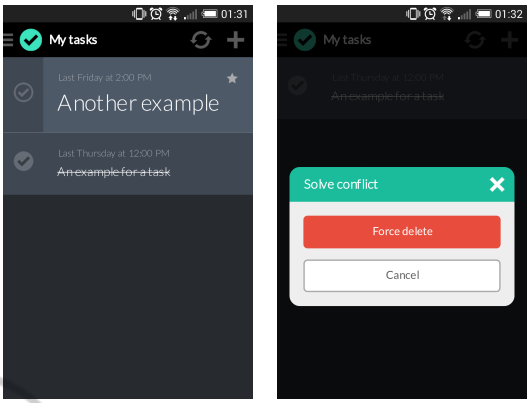

Figure 2: Example Application.

seen in Listing 2. In accordance with the demand for transparency (cf. Section 2), it is irrelevant to the app whether it stores tasks in the local or remote database.

The class `Storage` implements the replication method. Every task belongs to a collection, which has its own instance of `Storage`. Each instance has its own database connection, which is established on instantiation. The class provides methods to read, add, edit and delete data. It accesses IndexedDB through a third-party library (db.js by Powell (2012)), which provides convenient functionality for IndexedDB usage.

The synchronization method uses an approach based on data sets (cf. Section 3). Consequently, two copies of the data will be transferred to the server: the unmodified data set and the resulting data set. However, when adding or deleting tasks, it is not necessary to transmit the two copies. Due to the small data size, complete (and, obviously, asynchronous) synchronization was chosen. Synchronization is initiated after every operation (add, edit, delete) or on demand. Additionally, it starts whenever a connection becomes available, and when the app is started.

The class `StorageManager` implements the synchronization. Each instance of `Storage` has an instance of `StorageManager` assigned to it. This class provides methods to pull and push changes from and to the server. Therefore, it creates AJAX requests. After starting the synchronization, all tasks are pulled from the server and checked for changes and deletions. Changes are applied on the local database before local changes are pushed to the server. A *dirty* flag is used to identify locally modified tasks. New and deleted tasks are identified similarly. In order to differentiate the versions of a task, a time stamp is employed.

A custom build process simplifies deployment. The workflow supports the automatic creation of the manifest file, compression and concatenation of assets (e.g. CSS and JavaScript files), and error checking. As Webapps usually consists of a multitude of files, concate-

nation is useful to reduce their number; compression saves storage space on devices.

The developed app is able to manage tasks of a registered user across multiple devices. All changes are transferred to the server once connectivity is given. In order to avoid insertion conflicts, a local ID is assigned to a new task. After synchronization, the affected task is deleted locally. Then a new task is created with an server-assigned ID. Where possible, conflicts are resolved automatically. Modification conflicts are detected by comparing the old and new version of the task and resolved by using the newer one based on attributes. Modification-delete conflicts are detected through a *delete* flag and resolved by revoking the deletion. Deletion conflicts have to be resolved manually by users. They can decide whether to force delete the task or cancel the deletion.

6 DISCUSSION

While our article has laid out the foundation for offline provisioning and synchronization, there is much room for discussion. To address the fast progress that underlies the topic, we first highlight insights. We then propose recommendations. Moreover, we have a look at limitations and present the outlook of our work.

6.1 Insights

There has been interest in offline Web *access* even before smartphones became popular, e.g. by by Song, Yu, Wang, and Nie (2006). With the widespread usage of Webapps, not only have offline capabilities become a necessity but the requirements considering *what* has to be possible in offline scenarios have become diverse. The possibilities introduced in Section 2 will facilitate a new generation of Web applications. Despite – or even due to – the availability of different technologies, the choice of the *right* approach is not straightforward. In fact, it has to be made based on the specific requirements (see next Section).

Both the theoretical assessment and the case study have shown that it is possible to create powerful Webapps with offline-capabilities using HTML5. Implementing apps is manageable if an adequate technology has been chosen. However, a shift in architecture is needed: while design patterns – especially the model-view-controller distinction – remain, some past lessons for the design of Webapps do not apply any more. This is a consequence from requiring application logic on the backend and might be confusing at least for unexperienced developers.

Whereas the (rather general) requirements for our case study should be generalizable, it has to be concluded that many aspects of offline-capabilities reflect in diverse, fine-grained requirements. While there are some common necessities, no set of requirements can be attributed to all situations.

Any app that allows modifications of local data while being offline needs some form of synchronization. The concepts as sketched in Section 3 are rather easy to understand and their application is straightforward if using a RESTful (or other Web service-based) approach. In many cases, these simple forms do not suffice, though. This particularly applies to situations in which concurrent changes are made and users cannot intuitively decide which data set to keep if queried during synchronization. Moreover, fully automated synchronization is desirable in many situations. The insights provided in this article help to understand this problem but do not close the research gap. In fact, future work need to put emphasis onto improved mechanisms and eventually *better* tools for synchronization.

6.2 Recommendations

The insights from our case study implementation in combination with a theoretical assessment allow for drawing recommendations.

Regarding the basis paradigms, caching is suited for read-only offline apps that merely need to maintain a current set of data. An example is an app that reads news feeds. Hoarding is applicable if apps are more sophisticated but still only read data. This e.g. applies to apps that provide access to Wikis or online journals; they could hoard articles that match the user's interest and that are linked by already read articles. Replication is needed in any case of truly working with data (i.e. also modifying it) without permanent connectivity.

Moreover, advice for development can be given. Firstly, a manifest file as required for the AppCache should be created (automatically) during the built process. Maintaining the manifest file manually is cumbersome and error-prone. Secondly, the efficiency of the AppCache is bound to the structure of the app. It has to be recommended to separate application data from application logic. This calls for the compilation of offline-app development best practices, as already argued earlier. Thirdly, programmers should use developers' community advice where it is already available. This e.g. applies to the selection of frameworks for single-page apps (*TodoMVC*, 2014).Assessed, specifically scientific sources remain to be scarce.

Even though HTML5 in theory is a technology to bridge all platform gaps, it has become evident that browser support needs to be taken into consideration.

Therefore, when making a choice the intended platforms should be kept in mind. Judging from basic support only, AppCache and WebStorage form the "smallest common divisors" but at the same time are not always useful (e.g. since WebStorage has no query language). Web SQL could be chosen for Webapps that are mainly used on mobile devices; IndexedDB would be the choice for PC applications. These considerations are only rough recommendations, though: app usage could change and also performance has to suffice. At the same time, polyfills might be used to extend browser support by providing emulated functionality (Sharp, 2010). Our recommendation, therefore, is to wage arguments if more than one technology seems appropriate, *and* if the app is intended for longer use or if the storage component of it is not easily interchangeable.

An additional recommendation is to consider how synchronization and offline-capabilities are used in other scenarios. For some applications, mature solutions exist; the underlying concepts might be applicable for usage in mobile Webapps. One closely related example is *Google Docs* (2015). It offers quite advanced offline functionality. In general, synchronization for Web-based collaborative tools (Fraser, 2009) might be worth a closer look.

Another source of advise might be work on database replication (Wiesmann et al., 2000a; Wiesmann et al., 2000b). Even thought replicating databases is different to synchronizing apps, the basic principles apply to both. Depending on what kind of synchronization is desired, particularly domain-specific applications of replication should be assessed. Details are out of scope of this paper; for examples check the work by by Groothuyse, Sivasubramanian, and Pierre (2007), Amza, Cox, and Zwaenepoel (2003) and Lin, Kemme, Patino-Martinez, and Jimenez-Peris (2006).

6.3 Limitations, Outlook and Future Work

Two kinds of *limitations* have to be assessed: the limitations of the presented work, and the boundaries inherent to Webapp offline usage and synchronization. Our work is naturally limited in that it is a snapshot. However, this limitation is inherent to most papers on mobile devices and apps due to the fast pace of progress. The key to making a contribution is to provide applicable knowledge (which we do in Sections 2 to 5) *and* a generalization (Section 6). Our case study and the recommendations are limited due to their qualitative character. To back our work, additional case study-like experimental work is required. Advise should then be verified quantitatively (e.g. in a field study), which is a task for future work.

Offline-enabled Webapps are much less limited than they used to be. The technology is quite potent. Nevertheless, improvements are desirable particularly with regard to support for developers in designing synchronization mechanisms (Agrawal et al., 2013), e.g. with novel frameworks. The greatest shortcoming, however, seems to be the lack of experience: developers will need more guidance on how to choose technology and on how to apply it. Eventually, the collecting of best practices and the design of patterns drive further technological improvements.

Giving an outlook that goes beyond speculation is no easy task. There is no reason to believe that technological progress will decelerate. Moreover, a widespread use of offline technology in new Webapps is likely. How the distinct technologies evolve and whether browser support of technologies will improve is hard to estimate, though.

From a scientific point of view, it will be needed to examine existing approaches that combine mobility with offline usage. This for example is relevant when deploying mobile devices in regions with scarce connectivity (Shrestha et al., 2011). It has to be assessed whether HTML5 Webapps with offline capabilities are a proper choice in the scenarios already described in the literature or if there are obstacles that have yet to be overcome. Moreover, there are situations in which simple offline capabilities and common synchronization strategies are not sufficient. In these cases, sophisticated approaches are required, which are beyond the scope of what HTML5 seems to be suited for. An quite recent example is (near) real-time access of health data in a mobile environment (Lomotey and Deters, 2014).

As also given as a recommendation, future work could try to transfer knowledge from other applications of synchronization to what is required for mobile Webapps. This particularly applies to advanced database replication.

Another field that should be assessed is security of storage technologies (West and Pulimood, 2012). Security is named a prevailing topic by practitioners (Majchrzak and Heitkötter, 2014) and it poses many research challenges. Storing data locally that used to reside on backends introduces new risks. Think of stolen smartphones and compromised devices that contain confidential corporate data. In addition, storage technology itself might pose security risks.

With this article, our work on offline-provisioning for Webapps is finished. However, we will likely revive our efforts once there has been technological progress *or* typical usage has moved beyond today's scope. Offline-provisioning however will stay on our

agenda for it also is important for native apps and in cross-platform scenarios. While the native case is quite well understood and covered by textbooks (such as by Mednieks, Dornin, Meike, and Nakamura (2012) and Conway, Hillegass, and Keur (2014)), offline capabilities can become challenging for cross-platform frameworks. For those built atop or by employing Web technology (such as the popular PhoneGap a.k.a *Apache Cordova* (2014)), the findings of this article commonly apply. For those that have an (probably proprietary) technology stack of their own, the situation is not as simple. Particularly if frameworks yield native code, the unified approach that comes with HTML5 is lost. Rather, an abstraction from platform-depended capabilities has to be found. The latter is a challenge we will tackle in the future.

Additionally, we intend to more closely investigate synchronization. This article provides the technological background, but synchronization has both implications for the proper usage of technology and for business processes.

7 CONCLUSION

In this paper, we presented the status quo of offline-provisioning and synchronization of content for mobile Webapps. Besides providing an overview, which can be used as a reference, we sketched a scenario and discussed insights. Moreover, we illustrated situation-specific recommendations.

The outlook is blurry to some degree but promising. The increasing interest of businesses in mobile Computing (Majchrzak and Heitkötter, 2014) will likely foster sophisticated usage of the available technology and drive improvements – and probably new developments. The same applies to synchronization, which has significance for many apps. Therefore, offline technologies for Webapps and synchronization will be topics of both research and practitioners' interest for the near time to come.

REFERENCES

Agrawal, N., Aranya, A., and Ungureanu, C. (2013). Mobile data sync in a blink. In *Proc. HotStorage '13*, HotStorage'13, Berkeley, CA, USA. USENIX Association.

Amza, C., Cox, A. L., and Zwaenepoel, W. (2003). Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Proc. Middleware '03*, pages 282–304, New York, NY, USA. Springer.

Ananthanarayanan, G., Blagsvedt, S., and Toyama, K. (2006). Oweb: A framework for offline Web browsing. In *Proc. LA-WEB '06*, pages 15–24, Washington, DC, USA. IEEE CS.

Apache Cordova. (2014). Retrieved from http://cordova.apache.org/

Archibald, J. (2012). "Application Cache is a Douchebag". Retrieved from http://alistapart.com/article/application-cache-is-a-douchebag

"*Basic concepts – IndexedDB*". (2013). Retrieved from https://developer.mozilla.org/en-US/docs/IndexedDB/Basic_Concepts_Behind_IndexedDB

Berjon, R., Faulkner, S., Leithead, T., Doyle Navara, E., O'Connor, E., Pfeiffer, S., and Hickson, I. (2013). Offline Web applications. Retrieved from http://www.w3.org/html/wg/drafts/html/master/browsers.html#offline

Bidelman, E. (2010). A Beginner's Guide to Using the Application Cache. Retrieved from http://html5rocks.com/en/tutorials/appcache/beginner/

Bidelman, E. (2011). Exploring the FileSystem APIs. Retrieved from http://html5rocks.com/en/tutorials/file/filesystem/

Cannon, B. and Wohlstadter, E. (2010). Automated object persistence for JavaScript. In *Proc. WWW '10*, pages 191–200, New York, NY, USA. ACM.

Charland, A. and Leroux, B. (2011). Mobile application development: Web vs. native. *Commun. ACM*, 54:49–53.

Coda File System. (2014). Retrieved from http://www.coda.cs.cmu.edu/

Conway, J., Hillegass, A., and Keur, C. (2014). *iOS Programming*. Big Nerd Ranch, 4th edition.

Dar, S., Franklin, M. J., Jónsson, B. T., Srivastava, D., and Tan, M. (1996). Semantic data caching and replacement. In *Proc. VLDB '96*, pages 330–341, San Francisco, CA, USA. Morgan Kaufmann.

Deveria, A. (2014). Can I use _ ? Retrieved from http://caniuse.com/

Felke-Morris, T. (2014). *Web Development and Design Foundations with HTML5*. Addison-Wesley Professional, 7th edition.

Flanagan, D. (2006). *JavaScript*. O'Reilly, 5 edition.

Fraser, N. (2009). Differential synchronization. In *Proc. 2009 ACM DocEng'09*, pages 13–20, New York, NY, USA. Retrieved from http://neil.fraser.name/writing/sync/eng047-fraser.pdf

Gaubatz, P., Hummer, W., Zdun, U., and Strembeck, M. (2014). Enforcing entailment constraints in offline editing scenarios for real-time collaborative Web documents. In *Proc. SAC '14*, pages 735–742, New York, NY, USA. ACM.

Goncalves, E. and Leitao, A. M. (2007). Offline execution in workflow-enabled Web applications. In *Proc. QUATIC '07*, pages 204–207, Washington, DC, USA. IEEE CS.

Gonçalves, E. E. M. and Leitão, A. M. (2009). Using common lisp to prototype offline work in web applications for rich domains. In *Proc. 6th Europ. Lisp Workshop*, pages 18–27, New York, NY, USA. ACM.

Google Docs. (2015). Retrieved from https://docs.google.com/

Green, I. (2011). Migrating your WebSQL DB to IndexedDB. Retrieved from http://html5rocks.com/en/tutorials/webdatabase/websql-indexeddb/

Groothuyse, T., Sivasubramanian, S., and Pierre, G. (2007). Globetp: Template-based database replication for scalable Web applications. In *Proc. WWW '07*, WWW '07, pages 301–310, New York, NY, USA. ACM.

Hales, W. (2012). *HTML5 and JavaScript Web Apps*. O'Reilly.

Heitkötter, H., Hanschke, S., and Majchrzak, T. A. (2013). Evaluating cross-platform development approaches for mobile applications. In *LNBIP*, volume 140, pages 120–138. Springer.

Heitkötter, H., Majchrzak, T. A., and Kuchen, H. (2013). Cross-platform model-driven development of mobile applications with MD2. In *Proc. SAC '13*, pages 526–533.

Heitkötter, H., Majchrzak, T. A., Ruland, B., and Weber, T. (2013). Evaluating frameworks for creating mobile Web apps. In *Proc. WEBIST 13*, pages 209–221. SciTePress.

Hickson, I. (2010). Web SQL Database. Retrieved from http://www.w3.org/TR/webdatabase/

Hickson, I. (2013). Web Storage. Retrieved from http://www.w3.org/TR/webstorage/

Höpfner, H., Mansour, E., and Nicklas, D. (2009). Review of Data Management Mechanisms on Mobile Devices. *it - Information Technology*, 51(2):79–84.

HTML5. (2014). Retrieved from http://www.w3.org/TR/html5/.

Hurth, D. (2010). JSON Beats XML, or Ajaj vs Ajax. Retrieved from http://www.ajaxonomy.com/2010/xml/json-beats-xml-or-ajaj-vs-ajax

Kao, Y.-W., Chow, T.-H., and Yuan, S.-M. (2011). Offline Web browsing for mobile devices. *J. Web Eng.*, 10(1):21–47.

Kao, Y.-W., Lin, C., Yang, K.-A., and Yuan, S.-M. (2012). A Web-based, offline-able, and personalized runtime environment for executing applications on mobile devices. *Comput. Stand. Interf.*, 34(1):212–224.

Karthik, R., Patlolla, D. R., Sorokine, A., White, D. A., and Myers, A. T. (2014). Building a secure and feature-rich mobile mapping service app using HTML5: Challenges and best practices. In *Proc. of the 12th ACM Int. Symp. on MobiWac '14*, MobiWac '14, pages 115–118, New York, NY, USA. ACM.

König-Ries, B., Türker, C., and Höpfner, H. (2007). Informationsnutzung und -verarbeitung mit mobilen Geräten – Verfügbarkeit und Konsistenz. *Datenbank-Spektrum*, 7(23):45–53.

Kovatsch, M., Mayer, S., and Ostermaier, B. (2012). Moving application logic from the firmware to the cloud. In *Proc. IMIS '12*, pages 751–756. IEEE.

Kristol, D. and Montulli, L. (1997). Request for comments 2109 – HTTP state management mechanism. Retrieved from https://www.ietf.org/rfc/rfc2109.txt

Lam, F., Lam, N., and Wong, R. (2002). Efficient synchronization for mobile XML data. In *Proc. CIKM '02*, pages 153–160, New York, NY, USA. ACM.

Lin, Y., Kemme, B., Patino-Martinez, M., and Jimenez-Peris, R. (2006). Applying database replication to multi-player online games. In *Proc. NetGames '06*, New York, NY, USA. ACM.

Lomotey, R. K. and Deters, R. (2014). Mobile-based medical data accessibility in mhealth. In *Proc. MOBILECLOUD '14*, pages 91–100, Washington, DC, USA. IEEE CS.

Macedonia, M. (2007). iPhones Target the Tech Elite. *Computer*, 40:94–95.

Majchrzak, T. A. and Heitkötter, H. (2014). Status quo and best practices of app development in regional companies. In *LNBIP*, volume 189, pages 189–206. Springer.

Makzan (2014). *HTML5 Game Development Hotshot*. Packt Publishing.

Marco, F., Gallud, J., Penichet, V., and Winckler, M. (2013). A model-based approach for supporting offline interaction with web sites resilient to interruptions. In *Current Trends in Web Engineering*, volume 8295 of *LNCS*, pages 156–171. Springer, Heidelberg, Germany.

Mednieks, Z., Dornin, L., Meike, G. B., and Nakamura, M. (2012). *Programming Android*. O'Reilly.

Narasimhan, P. (2014). IndexedDB polyfill over WebSql. Retrieved from http://nparashuram.com/IndexedDBShim/

Ohrt, J. and Turau, V. (2012). Cross-platform development tools for smartphone applications. *IEEE Computer*, 45(9):72–79.

Powell, A. (2012). db.js. Retrieved from http://aaronpowell.github.io/db.js/

Quilligan, A. (2013). HTML5 Vs. Native Mobile Apps: Myths and Misconceptions. Retrieved from http://forbes.com/sites/ciocentral/2013/01/23/html5-vs-native-mobile-apps-myths-and-misconceptions/

Ranganathan, A. and Sicking, J. (2013). File API.

Rodriguez, A. (2008). RESTful Web services: The basics. Retrieved from http://www.ibm.com/developerworks/webservices/library/ws-restful/

Roth, G. (2009). RESTful HTTP in practice. Retrieved from http://infoq.com/articles/designing-restful-http-apps-roth

Satyanarayanan, M. (1996). Fundamental challenges in mobile computing. In *Proc. PODC '96*, pages 1–7, New York, NY, USA. ACM.

Sharp, R. (2010). What is a Polyfill? Retrieved from http://remysharp.com/2010/10/08/what-is-a-polyfill

Shrestha, S., Moore, J., and Nocera, J. A. (2011). Open-source platform: Exploring the opportunities for offline mobile learning. In *Proc. MobileHCI '11*, pages 653–658, New York, NY, USA. ACM.

Song, J., Yu, G., Wang, D., and Nie, T. (2006). Offline web client: Approach, design and implementation based on web system. In *Proc. WISE '06*, pages 308–314, Berlin, Heidelberg, Germany. Springer.

Takada, M. (2012). Modern Web applications: an overview. Retrieved from http://singlepageappbook.com/goal.html

TodoMVC. (2014). Retrieved from http://todomvc.com/

Uhrhane, E. (2012). File API: Directories and System. Retrieved from http://www.w3.org/TR/file-system-api/

Vora, A., Tari, Z., and Bertok, P. (2004). An hoarding approach for supporting disconnected write operations in mobile environments. In *Proc. SRDS '04*, pages 276–288, Washington, DC, USA. IEEE CS.

Weeks, M. (2014). Creating a web-based, 2-D action game in JavaScript with HTML5. In *Proceedings of the 2014 ACM Southeast Regional Conference*, ACM SE '14, pages 7:1–7:6, New York, NY, USA. ACM.

Weikum, G. and Vossen, G. (2001). *Transactional Information Systems*. Morgan Kaufmann, San Francisco, CA, USA.

West, W. and Pulimood, S. M. (2012). Analysis of privacy and security in HTML5 web storage. *J. Comput. Sci. Coll.*, 27(3):80–87.

Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., and Alonso, G. (2000a). Understanding replication in databases and distributed systems. In *Proc. of the 20th ICDCS*, Washington, DC, USA. IEEE CS.

Wiesmann, M., Schiper, A., Pedone, F., Kemme, B., and Alonso, G. (2000b). Database replication techniques: A three parameter classification. In *Proc. of the 19th IEEE SRDS*, Washington, DC, USA. IEEE CS.

Yasuda, K. (2013). Quota Management API. Retrieved from http://www.w3.org/TR/quota-api/

Zorrilla, M., Martin, A., Sanchez, J. R., Tamayo, I. n., and Olaizola, I. G. (2014). HTML5-based system for interoperable 3D digital home applications. *Multimedia Tools Appl.*, 71(2):533–553.